JOpera
A toolkit for efficient visual composition of web services

JOpera: a Toolkit for Efficient Visual Composition of Web Services

Cesare Pautasso, Gustavo Alonso
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,alonso}@inf.ethz.ch

December 15, 2003

Abstract

1 Introduction

Although they may not solve all interoperability problems, Web service technologies show great promise in reducing the complexity of integrating heterogeneous software components over the Internet. They provide standard protocols for invoking (SOAP [44]), describing (WSDL [45]), and discovering services (UDDI [35]) published on the Internet in a platform, programming language and vendor independent manner. A most natural evolution of these technologies concerns the ability to compose complex Web services from basic ones [12]. Especially in E-Business scenarios, the standardization efforts to integrate Web services into business processes have produced many proposals [15, 25, 26, 30, 40, 46, 47]. However, none of these is yet well established in practice [42], although the Business Process Execution Language for Web Services (BPEL4WS [25]) specification seems to be ahead at the moment.

The standardization efforts behind Web services, and the increasing number of aspects that are being formalized, open up the possibility of reducing the development cost and complexity of large distributed information systems. In particular, a visual approach to Web service composition may very well be a suitable complement to such existing XML-based composition standards. Using a visual programming language may help to bridge the different standards and will certainly help to make Web services much more designer-friendly. In such a system, the order of invocations of services, their data exchanges and the necessary failure handling behavior could all be specified

Part of this work is supported 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).

with a simple visual syntax. However, having a visual programming language for Web service composition is not enough. The visual programming environment needs also a set of tools for efficient, scalable and reliable execution of such composite applications.

In this paper we present the JOpera system, a visual programming environment and execution engine for Web service composition [36]. JOpera is currently a prototype being used to explore several different aspects of software composition. First of all, JOpera brings visual programming to Web services composition [37]. To do so, we have defined the syntax and semantics of the JOpera Visual Composition Language (JVCL), in which services are composed into processes, defined by a visual notation based on data and control flow graphs. Furthermore, we show how our language can be mapped to BPEL and vice-versa. Second, in discussing the usability features of the JOpera development environment, the paper illustrates our approach to improve the visual scalability of the language [9] by using nesting constructs, multiple views and automatic support for graph layout and diagram navigation. These topics have been typically neglected in previous work on workflow systems. However, given the complexity of real business processes [10], they are key to the success of such visual environments. Third, regarding the efficient and scalable execution of composite Web services, we describe 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. By switching between different implementations of these abstractions, different quality of service levels can be achieved in terms of both scalability and reliability. Fourth, within this flexible architecture, we show that replication can be applied to any system component to increase the overall system's throughput. An important feature of JOpera is that, unlike most workflow systems, it achieves high performance execution by compiling process descriptions into executable code, rather than interpreting them with a generic navigation algorithm. In this paper, we validate our approach with an extensive set of experiments, which systematically compare the performance of different configurations.

This paper is organized as follows. In the first part we focus on the JOpera Visual Composition Language, starting from Section 2, where we state our assumptions about Web Services and about our composition model. In Section 3 we define the syntax and semantics of the JVCL language, and in Section 4 we briefly discuss how to map JVCL to BPEL4WS and vice-versa. A short description of the JOpera visual development environment (Section 5) is followed by an example (Section 6) of a process written in JVCL. In the second part of the paper, we concentrate on the JOpera Process Execution Kernel. First, in Section 7 we present its architecture, describing how the system executes the processes written in the JVCL language. In Section 8, we list some of the configurations into which the JOpera kernel can be deployed. To compare them, we conducted a systematic performance evaluation and in Section 9 we present the most important results. Section 10 covers related work and Section 11 concludes the paper.

2 Web services composition

Before describing in detail our visual language, we present our model for building applications by composing different Web services. We use the model to both motivate the language's design and give an overview of its major features.

2.1 Web services

Web services technologies provide open standards for interaction among heterogeneous applications running on different platforms across the Internet. XML based mechanisms have been standardized for describing service interfaces (WSDL [45]), for publishing and discovering services (UDDI [35]) and for invoking them over different communication protocols (SOAP [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, which may need to be invoked following a certain interaction pattern. To denote these ideas various terms have been proposed: choreography (WSCI [46]), orchestration (BPEL [25]), automation (XLANG [40]), coordination (WS-Coordination [26]), collaboration (BPSS [15]), and conversation (WSCL [47]).

In our case, we prefer the term composition, since we are interested in 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, so that we can integrate 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 scheduling system, or a request to perform a grid computation. 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 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.

We fully agree with such a view, and have also designed the JOpera system to support components than can be accessed through a variety of protocols, including, but not limited to Web service compliant ones. This way, the developer of a composite application is not limited to use components accessible only through the SOAP protocol and, in case of components supporting more protocols, the system can use the most appropriate one in terms of performance, security and reliability.

Keeping this in mind, for simplicity in the exposition, in the rest of the paper we will however focus on components modeling individual calls to Web services and use the terms program, component and service as synonyms.

2.2 Composition through processes

To model the composition of independent but related software components we use the notion of process. A *process* is composed of a set of tasks, which can represent either a service invocation (activities) or a call to other processes (sub-processes). All the information necessary to instantiate and execute a task is derived at runtime to support a form of *late binding*, where the actual implementation of a service is located at the latest possible moment based on constraints imposed on the task.

In general, a task involves the execution of an operation which may require some input data and may produce some output results. Similarly, to exchange data with other processes or with the user, a process has input and output parameters. To describe the connections between the input and output parameters of its tasks, a process includes a data flow graph. From the data flow graph of a process, it is possible to derive its control flow graph defining the partial order of execution of the tasks. Similar to data-driven data flow languages, a task cannot be started until all of its data dependencies are satisfied [22]. Unlike in traditional data flow models, we include an explicit description of the control flow of a process. This is useful to get 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 we will describe later, the development environment enforces the appropriate editing constraints to keep the two graphs synchronized.

Finally, the various services and processes to be composed as tasks of a process are chosen from a library of existing, reusable components. JOpera provides a set of tools to manage this component library. For example, it is possible to look up external services in UDDI registries and to automatically import their interfaces. This is done by translating the corresponding WSDL descriptions into the 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. In the case of services offering more than one operation, if one is provided, the WSCL [47] description of the conversation is automatically mapped to a process, defining the basic sequence of invocations and information exchanges between the various operations.

3 JOpera Visual Composition Language

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

Figure 1: Activity and SubProcess

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 sub-processes have a double border to indicate nesting.

3.1 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 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 (Figure 2). These edges are not removable, since there cannot exist a parameter box without its task.

To improve readability, the input and output parameters of a process are displayed linked to two separate boxes. The two gray shapes in Figure 2 represent the input and the output interface of a process to which the corresponding parameters are attached. The input parameters can be initialized by the user starting the process, or receive their data from the calling process. The output parameters can be read as soon as the process has finished its execution.

Data parameters may contain values of any data type encoded as 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 way, connections between parameters of mis-matching data types will be rejected.

Activities representing a call to a Web service have input and output parameters. In this case, each input parameter corresponds to a *part* of the SOAP request message, while each output parameter is extracted from the SOAP response. Thus, it is possible to model in detail the information exchanged with the Web service.

3.1.1 Data bindings

Data flow relationships between the parameters define how the 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. Furthermore, as shown in Figure 2, constant values can be bound to 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

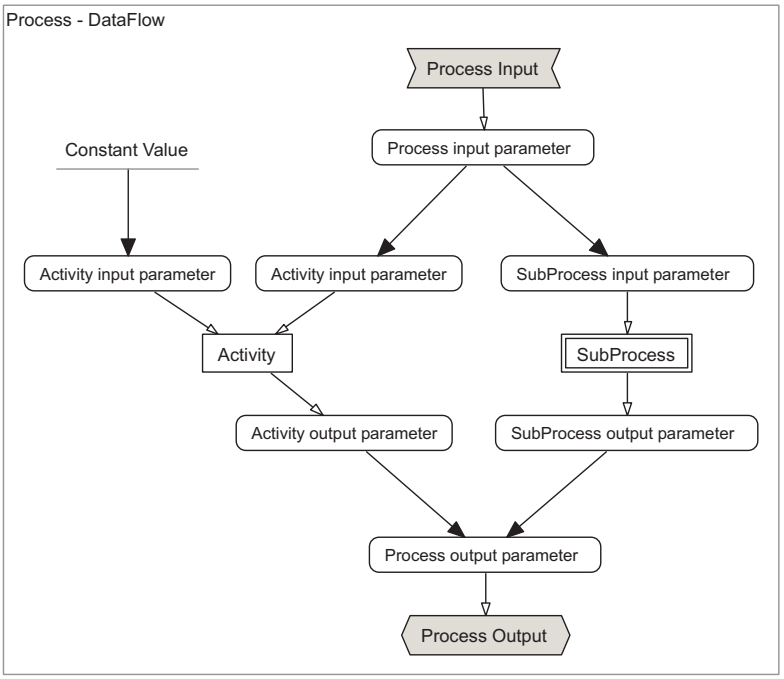


Figure 2: Sample data flow syntax

useful in 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, which prevent the user from drawing invalid bindings and explain with an error message why an edge is not allowed. For example, data always flows from output to input parameters of tasks. In the case of processes, input parameters of processes can only be connected to input parameters of tasks, and output parameters of processes may receive data only from output parameters of tasks. Furthermore, the same constant can be connected to multiple input parameters, but an input parameters 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.

3.1.2 System parameters

In addition to the user defined data flow parameters, each task has a set of system parameters and properties which can be used for a variety of purposes. They contain metadata about the execution of the process and are generated automatically by the runtime environment. The same visual syntax applies to both system and user data flow

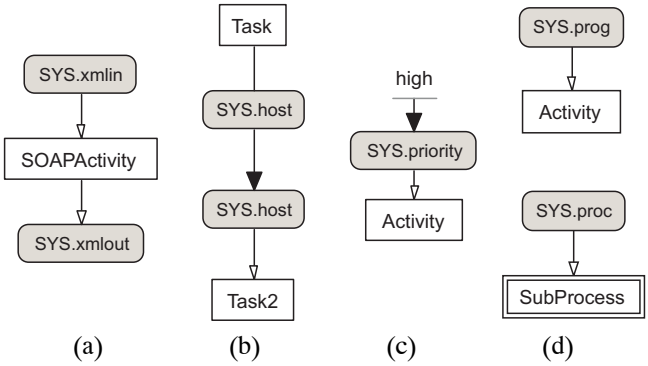


Figure 3: Example of system parameters

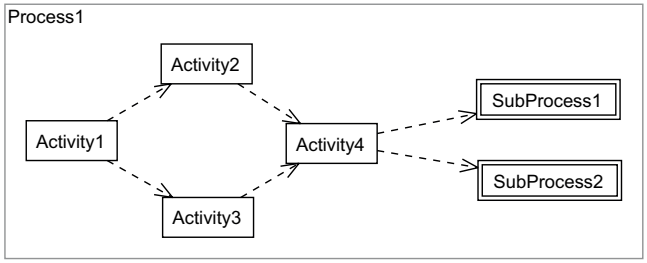


Figure 4: Sample control flow syntax

parameters, with the only difference that the former are colored in gray and their name always begins with the `SYS` prefix. Connections between user and system parameters are supported.

Figure 3 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 (3.a). To specify additional scheduling constraints the `host` and `priority` system parameters can be used. The `host` parameter may be used, for example, when composing a stateful conversation out of a set of operations belonging to the same 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 (3.b). The `priority` system parameter may be used to manually raise (or lower) the scheduling priority of critical tasks with respect to the other tasks of the process (3.c). 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 sub-process) is done dynamically based on the value of the `prog` (or `proc`) system parameter (3.d).

3.2 Control flow

Control flow defines the partial execution order between the components 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: finished (by default), failed (when an error occurs), or aborted (after an user kills the task). Figures 4 and 5 show examples of control flow graphs.

By definition, a data flow connection between two tasks implies a control flow dependency. The reason is that 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. Furthermore, extra control flow dependencies can be introduced directly in the control flow graph to model constraints that are not explicit in the data flow.

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 deleting a control flow dependency all of the corresponding data flow bindings are removed. The user may be notified with optional warning messages of the consequences of these actions, which otherwise are carried out in a transparent manner.

If there is more than one incoming control flow edge to a node C, the 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 both tasks A and B have 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.

In order to model alternative execution paths, a start condition may be associated to each node. This is a boolean expression referencing the value of some 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.

As shown in Figure 5, failure handling behavior is specified in the control flow graph by using connectors which fire on failure of a task. An exception handling task may be added to a process by drawing such connection from one or more tasks to it. With start conditions 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 is possible to retry its execution after the exception handler has finished.

3.3 Iteration

Supporting iteration in a language based on the data flow paradigm requires to introduce some auxiliary construct [32]. In our approach we rely on two constructs with a different degree of generality. First, we introduce a special data flow connector used to perform either sequential or parallel operations on lists. Second, we have been experimenting with explicit arbitrary loops in the control flow graph.

List-based loops can be used to repeat the same operation on a given set of values.

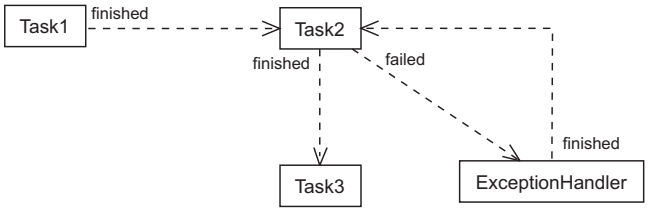


Figure 5: Control flow with exception handler

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, we introduce a pair of special data flow connectors, called *split* and *merge*. Like in other graph rewriting schemes [7], the overall effect at runtime is to replicate a set of nodes for each value of the input parameter list (Figure 11). In the case of a sequential split connector, the appropriate control flow dependencies between each task of the sequence are automatically inserted when the loop is unrolled. If the tasks produce output, the merge connector can be used to conveniently concatenate it into a single parameter when the execution of all replicas has completed.

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.

Another possible way of modeling repeated behaviour is through recursion. In the simplest case, this can be achieved with a sub-process referring to itself. This way, the tasks composing the process will be repeated as long as the condition associated to the sub-process holds true.

4 BPEL Mapping

In this section we show to what extent it is possible to map our visual composition language to the Business Process Execution Language for Web Services (BPEL [25]), an emerging XML-based specification for Web service composition, and viceversa. The main idea is to use the JOpera platform for visually composing Web services into processes, which can be later 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.

4.1 Mapping to BPEL

The components of a JVCL process can be accessed using various mechanisms, which are not limited to those compatible with SOAP/WSDL. In order to keep the mapping feasible we will assume that either all tasks of a process represent Web service invocations or that Web service wrappers for the other classes of components can be readily

provided. Such 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 which contains a service link corresponding to the activity's program and also a BPEL *invoke* activity is prepared. For each of the JVCL sub-processes a *link* to the JOpera systems where the process is accessible, or, alternatively, to make the final BPEL document self contained, a new *scope* is added to the BPEL *process*.

Control Flow In general, given the arbitrary topology of control flow connections in JVCL it will not always be possible to reduce it 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 case of control flow dependencies used to model exception handling, specific BPEL constructs can be employed. In case of loops in the JVCL control flow graph, they can be detected and mapped to a BPEL *while* block.

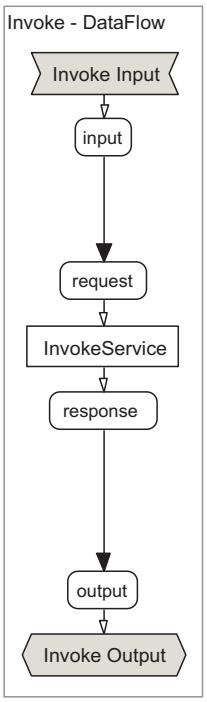
Data Flow and Variables In BPEL the flow of information between the services is not explicitly modeled with a data flow graph as in JVCL. Instead global variables are used as temporary storage for the messages exchanged by the services and XPath expressions are used to refer to individual data elements of the messages. To map the data flow graph of a JVCL process, a BPEL *variable* to store the request/response messages of each service is created and for each data flow connections in the JVCL graph a BPEL *assign* activity is inserted before and after the service invocation represented by the BPEL *invoke* activity. This assignment activity contains the XPath expression used to access the individual JVCL parameter (or message parts). As an alternative, a BPEL *variable* for each JVCL data flow parameter can be added. An example of a basic data flow mapping is shown in Figure 6.

There are a few constructs in JOpera that 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 meta-data about the process and its tasks within the process itself, and can be used to specify the late binding of a service interface, cannot be mapped to standard BPEL expressions.

Although the concept of *correlation* isn't part of JVCL, a BPEL process translating a JVCL process can contain a simple default correlation structure. For example, it could refer to the input data of the process, if this is enough to uniquely identify the process.

4.2 Mapping from BPEL

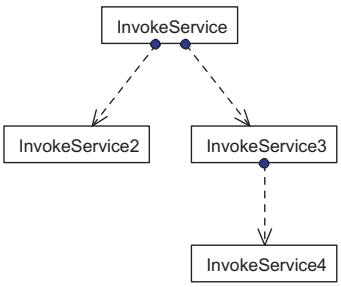
Control Flow As the BPEL control flow is expressed using both a block and graph structure, it is always possible to map this to a pure graph based model (Figure 7). This




```
<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 6: 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.




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

Figure 7: 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.

way, BPEL constructs such as *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*, *catch all* activities) can be mapped to JVCL by introducing rule-based exception handlers and an ad-hoc task in the BPEL library which 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 and arbitrary *assign* activities in BPEL. To do so there are two main possibilities: either an *assign* activity can be mapped to a direct data flow connection between a pair of JVCL parameters or it is necessary to add an explicit task, which runs the XPath expression contained in the *assign* activity and transforms the input into the output parameter accordingly.

Messaging BPEL *invoke* activities can be directly mapped to JVCL activities with a reference to a program representing the corresponding service invocation. Furthermore, mapping BPEL activities such as *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 which corresponds to the *onAlarm*/*onMessage*. This task terminates its execution on receipt of a message, or on 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*. In order to support events happening more than once, if necessary, a loop can be used. Such loop can immediately reactivate the task waiting on for the Alarm or, depending on the actual BPEL semantic, it may link the end of the action block back to the onAlarm task, so that it is reactivated after the event handler has terminated.

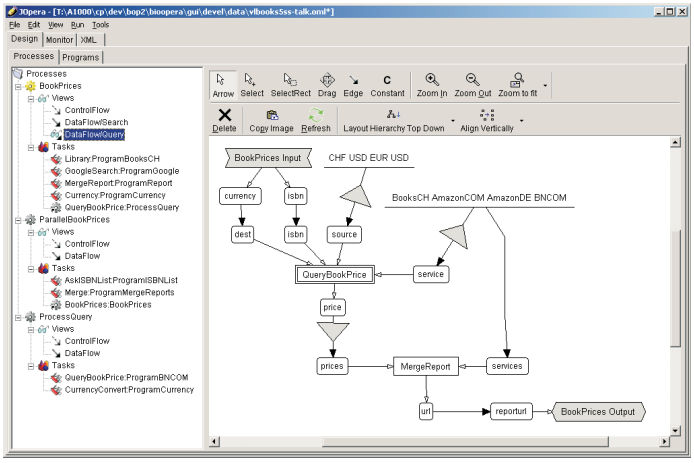


Figure 8: JOpera process developer

Again, at this stage, given the ID-based strategy to identify processes in JVCL it is not possible to map an arbitrary BPEL *correlation* structure to our language.

5 Visual development environment

The JOpera visual process development environment provides an integrated toolkit to manage the whole lifecycle of a process (Figure 8). 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 them into a process. Then, the data flow graph needs to be specified. This operation is partially automatic, since 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. To get an overview over the order of execution of the tasks and add additional constraints, the user may view and edit the control flow graph anytime. The editor automatically keeps the two graphs synchronized.

Once all of the services have been connected, the process may be compiled and uploaded to a JOpera runtime environment for execution. While monitoring a running process the user may watch its progress indicated by the color of the task boxes, and click on the parameters to inspect their content. The user may interact with a running process or its tasks, and abort, pause, continue, and restart them at will. More than one copy of a process may be run concurrently. Once a process has completed its execution, the user may access the content of all parameters as well as measurements about the execution time of each task, until the process is explicitly deleted from the system.

5.1 Visual scalability

One of the advantages of using a visual programming language is that the data and control flow of a process can be specified directly by drawing graphs. In practice, however, some manual effort may be required in order to obtain a readable diagram, even for small sized graphs. Thanks to the automatic layout facilities built into the development environment, the amount of work necessary to re-arrange the graph layout is significantly reduced. We have adapted several hierarchical layout algorithms to take into account the syntactial relationships between the graph elements. Furthermore, these algorithms are intended to be used incrementally in order to preserve the user's mental map of the process [31].

Although the automatic layout features already improve the user's productivity, better support is required to visualize realistic graphs having a large number of elements. Therefore our development environment provides the user with other features that increase the scalability of the visual language. First of all, thanks to the sub-process construct, parts of the graph may be collapsed into single nodes and the user may easily navigate back and forth between the various levels of nesting. This allows the user to design processes following both a top-down progressive refinement and a bottom-up aggregation approach. Second, the environment provides the ability to create and work with multiple views over the same data flow graph. In this case, the user may easily extract a subset of the data flow graph, for example, to analyze the data flowing through a particular task, or to focus on the tasks receiving data from a certain parameter. This way, the user may interactively navigate through a complex data flow graph and is always presented with an uncluttered view over the relevant information. The development environment also allows the user to edit the data flow graph from any of the views by enforcing the required consistency constraints. For example, when deleting a redundant data flow connection which is present in more than one view, the user will be warned about it and may decide to remove the connection from all views.

6 Example

As an example, we discuss a process used to compare the prices of books sold at various Internet stores. This process receives as input an ISBN number and returns as output an URL for a report containing the price comparison for the book. Since stores at different countries return prices in their own currency, the user may supply the currency to be used in the report as optional input parameter. The process contains the necessary steps 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 5 results returned by a web search engine looking for the author and the title of the book. All tasks of the example processes involve performing calls to actual Web services.

6.1 Process BookPrices

Figure 9 shows the control flow graph for the price comparison process. The process is composed of 3 activities (`Library`, `GoogleSearch`, `MergeReport`) and 1

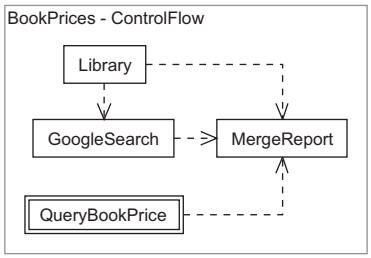


Figure 9: Control flow graph of the BookPrices process

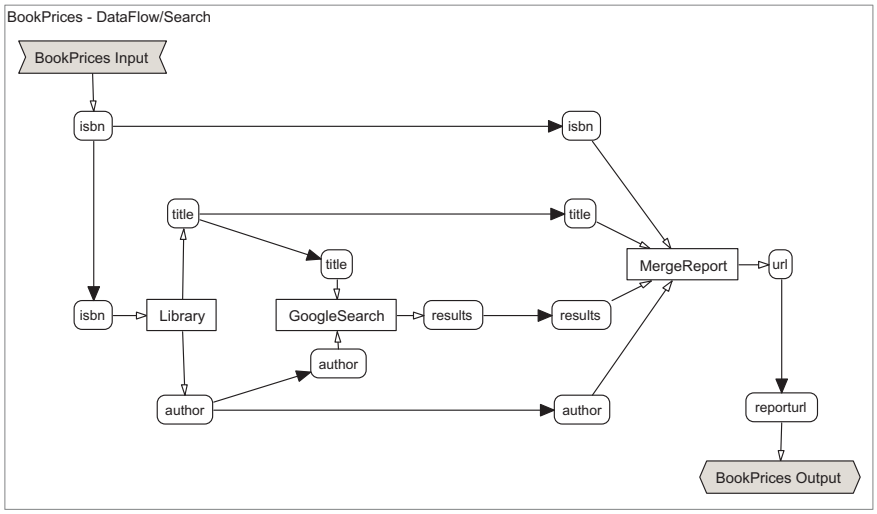


Figure 10: First data flow view of the BookPrices process

sub-process (`QueryBookPrice`). As its name suggests, `QueryBookPrice` involves contacting a book store to inquire about the price of a certain book identified by its ISBN. While this happens, the `Library` activity retrieves the author and title of the book. When the library query finishes, the web search is started and when all of the previous tasks are finished the report is generated.

The data flow graph of this Process has been partitioned into two different views to enhance its readability. Figure 10 shows one view with data parameters and bindings of the `Library`, `GoogleSearch`, and `MergeReport` activities. While the second view in Figure 11 shows the data flowing through the `QueryBookPrice` sub-process.

The first view (Figure 10) shows one of the input parameters of the process (`isbn`) passed both to the `Library` and `MergeReport` activities. 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 will run a web

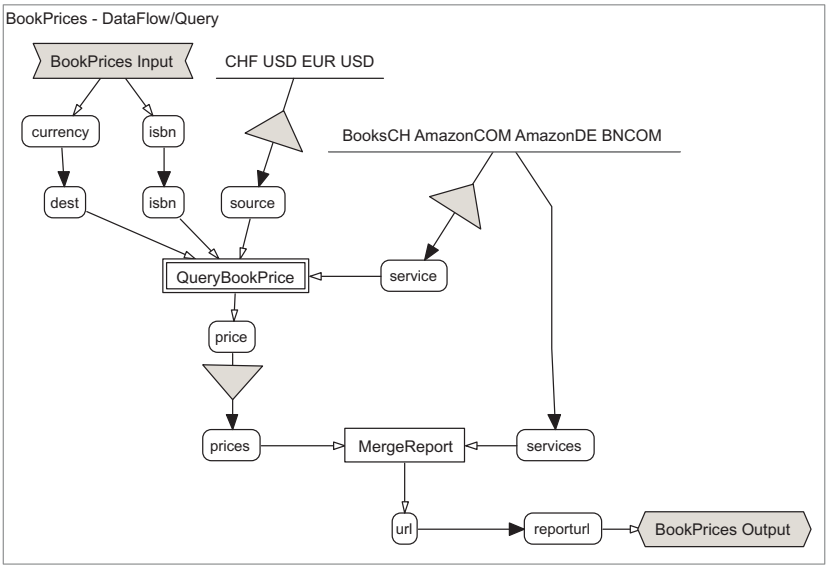


Figure 11: Second data flow view of the BookPrices process with parallel split and merge operations

search using them as keywords and return the top 5 `results`. The `MergeReport` activity receives the `title`, the web search `results`, the `author` and `isbn` of the book, it uses it 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 is shown in the view of Figure 11, 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` sub-process, 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` sub-process will be executed for each element found in these two input parameters. In the example, the `service` and `source` parameter are bound to constants with a list of four strings, which contain service identifiers (`BooksCH, AmazonCOM, AmazonDE, BNCOM`) and the corresponding currency identifiers (`CHF, USD, EUR, USD`). The prices returned by the parallel instances of the `processQuery` sub-process 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.

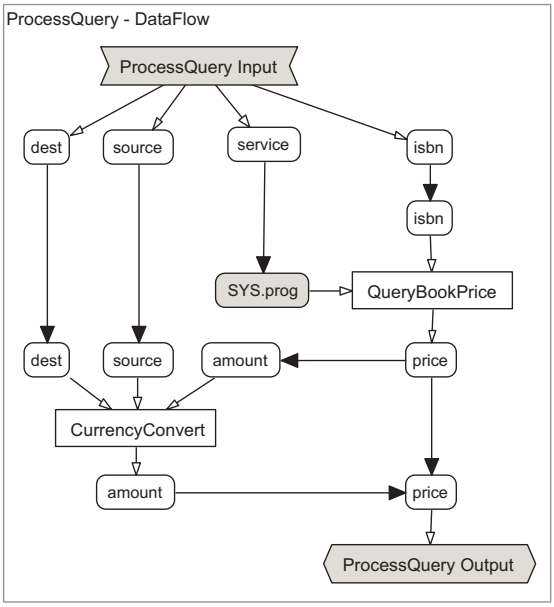


Figure 12: Main data flow view of the query process

6.2 Process QueryBookPrice

The `QueryBookPrice` process is called from within the `BookPrices` process. It contacts two Web services in order to inquire for the book's price and to convert it to the desired currency. Figure 12 shows its data flow graph. This process contains two activities: `QueryBookPrice, CurrencyConvert`. The input and output parameters of the process match the ones of the `processQuery` sub-process. 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. It should be noted 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.

7 JOpera Process Execution Kernel

In this section we present the architecture of the JOpera Process Execution Kernel. The main purpose of such kernel is to provide an execution platform for the JOpera Visual Composition Language which can be tailored to different levels of performance.

Figure 13 depicts the relationship between the JOpera Visual Development Envi-

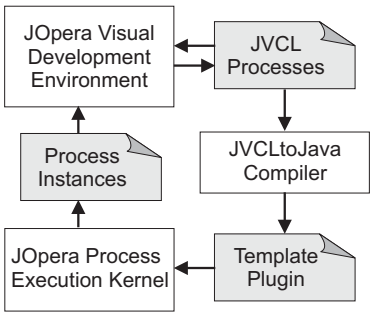


Figure 13: Overview of the JOpera Visual Development Environment and the JOpera Kernel

ronment and the Process Execution Kernel. The processes defined in the JVCL language are created and edited using the development environment. As we will explain 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 the progress of their processes.

Before describing in detail the architecture of the JOpera Kernel, we would like to give some background on how it is possible to execute the description of a process written in JVCL using the so-called navigation algorithm.

7.1 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. To do so, the navigation procedure interprets the information of a directed acyclic 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 state change of a task occurs, 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 if they are ready to be started. This way, after every task state change it is possible to determine the set of tasks to be started next.

This approach is very similar to mapping the process description to a set of Event, Condition, Action (ECA) rules. State changes of tasks trigger events, which will cause the evaluation 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 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

for running the task. The second type groups operations that access or modify the state information of a process. 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.

7.1.1 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, to implement our process navigation algorithm we build on the idea of mapping the process description to a program embodying the specific rules corresponding to the process description, generated using an ordinary programming language. This way, we can use the language's compiler to produce executable code which then can 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 our case Java [1], which then is compiled one more time. This way we can map the process model to standard language constructs, which can be efficiently executed. Moreover, during the code generation it is possible to analyze the structure of the process and perform optimizations.

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

7.2 Architecture

The core infrastructure necessary to run the processes written in JVCL is depicted in Figure 14. The kernel of our 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 itself. It should be noted that our navigation algorithm is independent of the actual implementation of these basic facilities. Before we discuss how to parallelize the navigation algorithm, we would like to give some more information about the main system components.

[1] In our prototype we choose Java because of its portability, the maturity of the available tools and its ease in dealing with dynamically loadable code. Nevertheless, other programming languages could be also used.

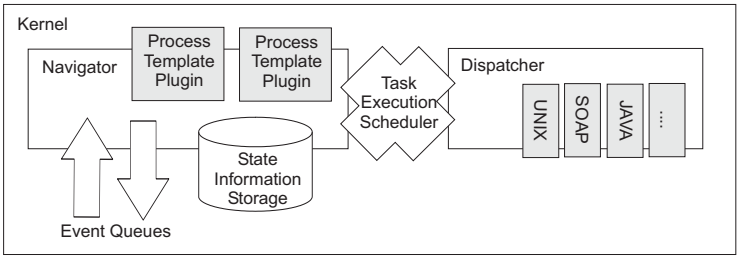


Figure 14: Architecture of a Monolithic Kernel

7.2.1 Navigator

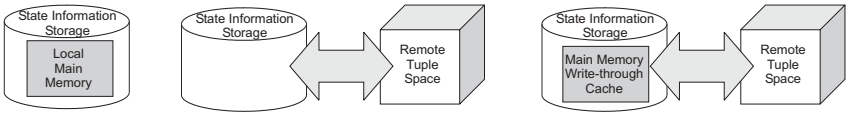
The navigator is the kernel component responsible for handling incoming process events, which are generally triggered by changes in the state of tasks or represent user requests. When such events occur, for example when the dispatcher has finished executing a task, the navigator runs the algorithm for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

7.2.2 Task Execution Scheduler

This component couples the navigator, generating task execution requests, with the dispatcher, which manages the actual task execution. In a distributed kernel (Figure 16), the scheduler receives task execution requests from a number of navigators and forwards them to a set of dispatchers. This is a key component concerning the scalability of the system, as its throughput limits the rate at which tasks can be executed.

7.2.3 Dispatcher

If the navigator is in charge of deciding what tasks should be started next, the dispatcher is the component which 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 which triggers it. This way, 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 to execute jobs containing Unix programs, SOAP [44] requests, Java method calls, as well as sub-process invocations. Once the job's execution has completed, the dispatcher sends an event encapsulating its results to the navigator.



(a) Volatile Storage (b) Remote Persistent Storage (c) Remote Persistent Storage with Cache

Figure 15: State Information Storage Implementations for the monolithic kernel

7.2.4 State Information Storage

This is the component responsible for storing the 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 has been 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 which are grouped into four boxes (or logical namespaces): system, input, output and whiteboard.

The main advantages of this approach are summarized in the following arguments. First of all, since the information in the key is neutral with respect to the physical location of the data, it becomes possible to transparently move the data to exploit locality and even replicate it among different physical locations to improve its 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, since this low level data representation is mostly independent from the data and metadata that needs to be stored [1]. Finally, as shown in Figure 15 the data layer can be implemented with a wide variety of mechanisms. These range from centralized memory based data structures (such as a hash map), to traditional forms of persistent storage (such as network file systems, or relational databases), or distributed storage systems (such as Linda-like tuple spaces [11] like TSpaces [29] or JavaSpaces [16])

7.2.5 Event Queues

The various kernel components communicate by exchanging event notifications managed by the event queues. 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 has finished its execution. Navigators also exchange events, for example, when a sub-process has completed its execution and navigation over the calling process, managed by a different navigator, needs to be triggered. The priority of these three classes of events can be adjusted.

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

First, as a reference, we used a single tuple space server to which all kernel components connect and exchange events by writing and taking tuples. As expected, this centralized event queue quickly becomes a bottleneck if all events sent by multiple dispatchers to a set of replicated navigators need to go through it. Therefore, in a second design, we chose to use a multi-layered approach, by distributing the event queue across all navigator components, with the following heuristic in mind: the navigator responsible for handling the incoming events should be kept as close as possible to the events themselves. This way, the dispatchers may directly send the "task-finished" events to the appropriate navigator. Events which are 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 are queued centrally and the corresponding "start-process" events may be retrieved by idle navigators. To further reduce the communication overhead, events generated by a navigator which can be processed by the same navigator are kept locally and do not need to be sent over the network.

7.3 Parallel Navigation

We chose to parallelize the navigation algorithm based on the observation that each process instance is a fully independent entity. In particular, 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 perform navigation on different, independent process instances in parallel. As a consequence, the navigation algorithm presented here doesn't need to be changed, since 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, issues such as load balancing and fault tolerance should be dealt with. In our current prototype we support 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, since each navigational step is executed atomically within a transaction, and if the state information can be stored remotely and persistently, recovery from failures occurring in the navigator becomes completely transparent. In fact, when a dynamic load balancing strategy is used, upon detection of the failure, the process instances belonging to a failed navigator can be immediately assigned to another one.

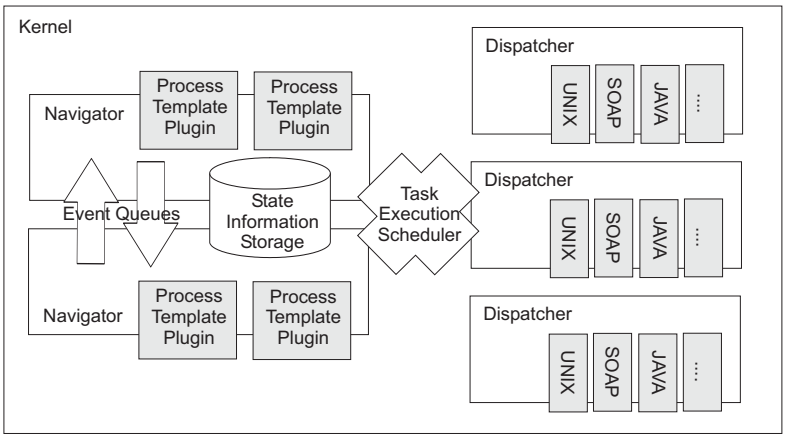


Figure 16: Architecture of a Distributed Kernel

8 Deployment Scenarios

In this section we present some of the configurations, listed in Table 1, in which our flexible kernel architecture can be deployed and discuss the main advantages and disadvantages regarding their performance. Not only flexibility is important for performance reasons, but also to have a system which can be adapted to different requirements, deployed into several environments to match a specific workload target. For example, our architecture can be deployed as a light-weight process simulation engine, embedded into a process development tool, or it can be used as a reliable service orchestration platform inside an application server, and can also scale to handle very large workloads, with a cluster based configuration.

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, the event queues and the task execution services are implemented using the appropriate main

	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

memory data structures. Since all data is kept in main memory, this configuration trades recoverability from failures with very fast access to the state information. Given its centralized nature, such an architecture doesn't 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 called 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, at the cost of a larger overhead, as the results already shown in Figure 17 indicate.

The limitations of these centralized configurations concern all five main system components: the Navigator, the Dispatcher, the Task Execution Scheduler, the State Information Storage and the Event Queues. If one is replicated in order to improve its throughput, very soon another component becomes 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 higher rate, putting a higher burden on the event queues. Thus, while scaling up the system and configuring it with replicated components (Figure 16), care must be taken to keep the system well balanced.

The first replicated configuration we present 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. As the task execution capacity of the system increases, it is to be expected that the system may be capable of handling a larger workload. As the measurements show, this is only true when the task duration is long enough, e.g. longer than 10 seconds. For tasks lasting a shorter time, the actual bottleneck lies in the navigator component.

This problem is addressed by the configurations (e) and (f), where also the navigator component is replicated, keeping the number of dispatchers and the corresponding task execution capacity constant. As our 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, we also tested a configuration (e) having the storage of the state information 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.

9 Measurements

The goal of the experiments is to analyze the performance of a significant subset of the deployment and configuration options described above. First of all we attempt to point out the scalability limits of a centralized system, where all the data storage, event and job scheduling services are implemented using the local main memory. Then we add external persistent storage for the state information to determine what is the cost of adding persistence to the system. Then we replicate the dispatcher and navigator

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

components, and observe the changes to the system's throughput.

9.1 Hardware Setup

The hardware and software setup for the experiments is as follows: the navigator and dispatcher kernel components were running on a cluster of dual Pentium-III 1000Mhz PCs with 1024 MB of RAM using Java 1.4.1 running on Linux v2.4.17. The three tuple space servers dedicated to state information storage, task execution scheduling and event communication were running each on separate dual Athlon 1.5Ghz 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].

9.2 Workload description

The behavior of the system is affected by the properties of the workload, which are defined by the control variables listed in Table 2. The number of processes indicates how large is the batch of concurrent processes to be executed. The size of a process is the number of tasks composing it. We used three different process sizes: 1, 10 and 100 tasks. Larger processes require more storage space and generate a higher number of jobs and events. The duration of the tasks affects the navigator's throughput, since 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) may be free to process other events or have to 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. We have been testing our system with a variety of control flow graphs. In the case of ten tasks we used two topologies, one sequential, where the tasks are executed sequentially and a parallel one, where all tasks are executed concurrently. The same parallel topology has also been used with the larger process, composed of 100 tasks. In the case of a large process, we also tested a more complex control flow graph modeling a matrix-like computation.

9.3 Measured variables

First of all, we are interested in measuring the user perceived effect of the different configurations. This effect is measured by how long a process takes to complete. More precisely, we computed the average wall-clock time over all the concurrent processes of a certain batch.

Second, for every experiment, we recorded the batch execution time, this is 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 can be used to compute the average throughput of the system, defined as the number of processed tasks per second. This value indicates the overall speed of the system in performing the operations (navigation, scheduling, running and results gathering) required to execute the tasks.

Third, in order to observe the system's internal behavior we instrumented the state information storage services to measure the time necessary to create the image of a new process instance. In our experience, this critical step is a potential performance bottleneck, since it is not possible to perform navigation until an instance has been created. We expected process instantiation to be expensive since, 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.

9.4 Results

9.4.1 Reliability Overhead

The limitations of centralized architectures can be illustrated by analyzing the performance of a centralized process support system. 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 it has been often observed [28, 39], both centralization and persistence generate a significant overhead in process support systems under heavy workload. In Figure 17 we quantify the user perceived behavior of a centralized system while running four different types of processes.

As the results show, the system's response time, i.e., 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, in the worst case the response time grows about 200 times when the workload size is increased thousand-fold. The actual performance degradation depends both on the type and size of the processes and on the specific properties of the system's configuration (Figure 15). 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 component to store the processes' state information, can lead to response times up to 50% shorter than the time required to perform navigation over persistent state.

As an attempt to combine the benefits of both configurations, we added a write-through cache located 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.

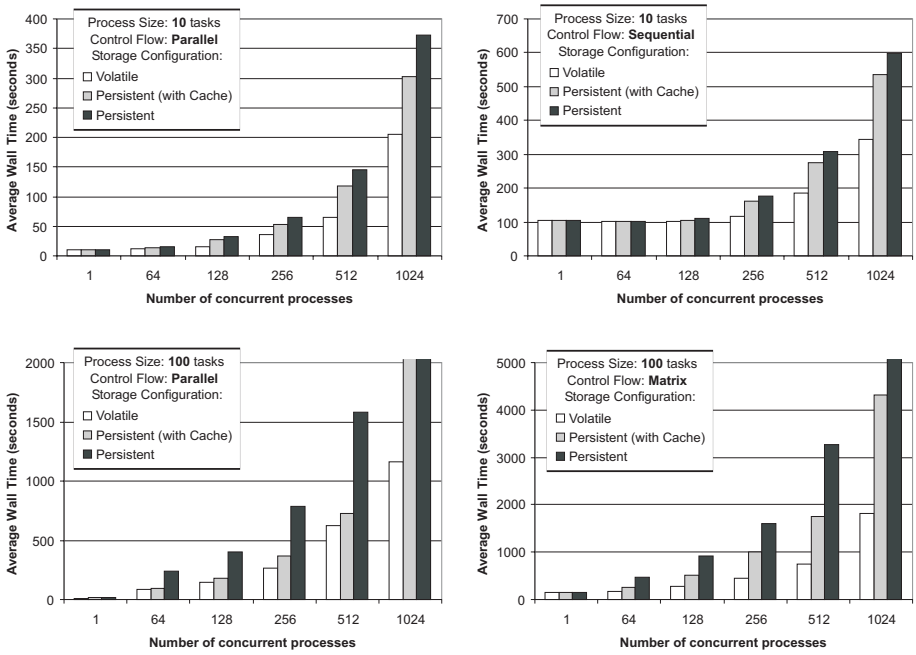


Figure 17: Performance degradation of a centralized process support system under increasingly large workloads

9.4.2 Monolithic kernel

In addition to the results already presented in Figure 17 concerning the degradation of the response time of a centralized system under increasingly large workloads, we would like to display the corresponding throughput's degradation in Figure 18. This set of measurements has been performed with a monolithic kernel configured to use volatile storage and up to 64 threads for local task execution, i.e. its execution capacity is limited to 64 concurrent tasks.

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

9.4.3 Process Instantiation

Figure 19 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 we expected, the instantiation time grows linearly with the process size: the higher number of tasks, the more information about them needs to be written to the data repository. The Figure also contains two interesting results. Not only is the

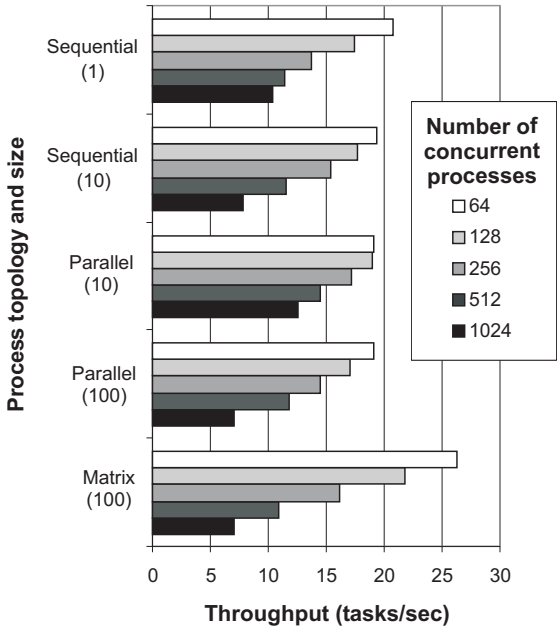


Figure 18: Throughput degradation of a centralized process support system under increasingly large workloads

instantiation time using persistent storage more than one order of magnitude longer than the time with volatile storage, but also, the volatile storage scales well with the number of navigators, since the process instantiation time remains constant. On the other hand, the performance of the centralized repository degrades as more and more navigators store in it data about their new processes. As it has been often suggested [28, 39], replicating the persistent storage would alleviate this problem. In all cases the instantiation time remains well below the 1 second boundary.

9.4.4 Scalable Process Navigation

Figure 20 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) In the case of persistent storage, for all workload sizes the throughput peaks at 12 navigators at about 140 tasks/second. This is a significant improvement with respect to a centralized system, especially considering that the throughput does not degrade as more and more processes run concurrently. In (b) the throughput actually improves as the workload size increases, indicating that, in the case of volatile storage, the performance of the replicated navigator does not saturate. Although the absolute throughput reaches about 350 tasks/second, this value is also obtained with 12 navigators, as the centralized task execution scheduler is the limiting factor of this configuration.

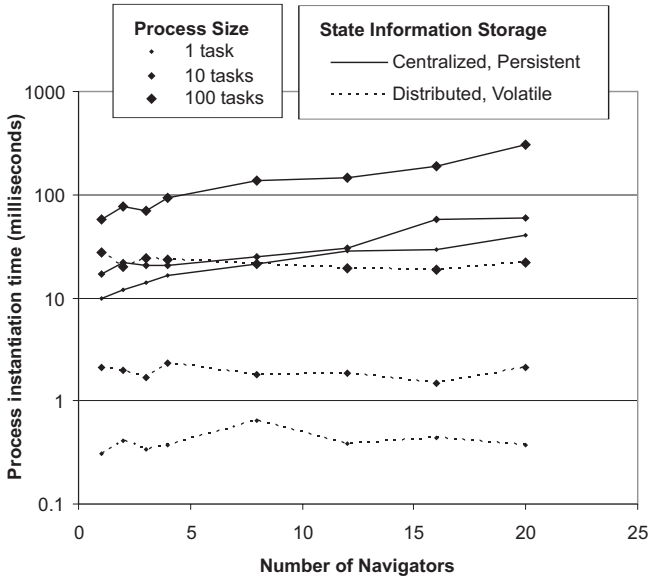


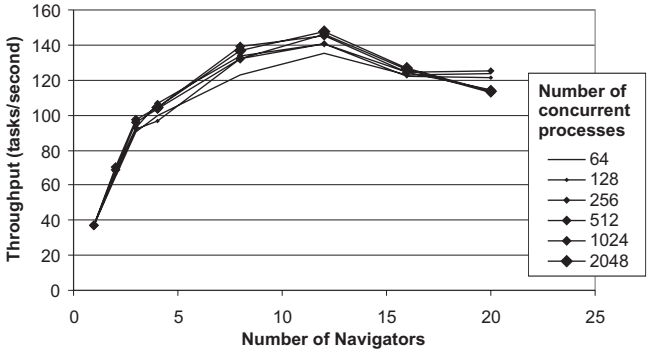
Figure 19: Scalability of the process instantiation

Figure 21 shows the system response time with up to 2048 concurrent processes of 10 sequential tasks run in the same settings as Figure 20. It can be observed that for small workloads, as the number of navigators increases the batch execution time approximates the time necessary to run only 1 process, which is close to 10 or 100 seconds, depending on the duration of the tasks. For larger workloads, the response time still grows linearly with the workload size, although the rate of increase can be controlled by changing the number of navigators. Using volatile storage the system scales well up to 20 navigators. Although the absolute response time is twice as high, the penalty of adding persistent storage is acceptable, as it shows good scalability up to 16 navigators accessing the same centralized data repository.

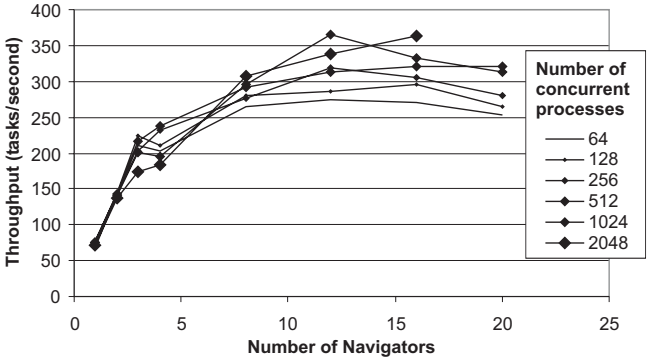
10 Related Work

The idea of developing large scale applications by composing coarse grained, reusable component modules has been pioneered by [48]. A formal model for software based on traditional CORBA, EJB and COM components has been developed in [14], while an overview over established component based visual languages can be found in [33]. A good argument on the need for a composition "glue" language, different from traditional programming languages has been presented in [17].

A similar, two-step approach has been proposed in the parallel computing domain [8]. In this case, sequential procedures are first written in Fortran or C, and afterward they are composed into a parallel structure using a control flow based graphical



(a) Processes of 10 parallel tasks with persistent storage



(b) Processes of 10 parallel tasks with volatile storage

Figure 20: Scalable navigation: average throughput of the system using an increasingly large number of parallel navigators

notation, where the data flow is derived implicitly by matching parameter names [7].

In the past, there have been many contributions concerning the problem of extending data flow languages with iteration constructs. A survey can be found in [32]. An example of iteration through vector operators and conditional switches is [3].

There is a wide range of commercial products and research projects dedicated to process management systems [18], especially regarding process modeling languages, with emphasis on flexibility [43] and transactional properties [38].

Many different graphical formalisms have been used as a modeling tool in the workflow community. Examples include State Charts, used in the the Mentor project [50], or Petri Nets [41] and variations such as Object Coordination Nets (OCoN) [49]. However, there is still a lack for a well established visual standard for process modeling.

Relatively less work can be found about 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

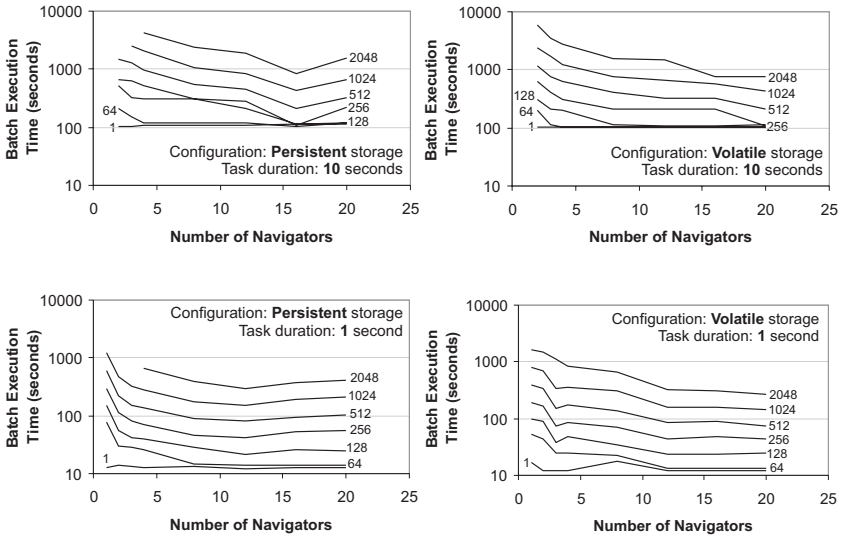


Figure 21: Scalable navigation: Execution time of batches processes having 10 tasks and sequential control flow topology as a function of 4 variables: 1) the number of navigators (X-Axis), the workload size (Z-Axis), the duration of the tasks and the system configuration (volatile or persistent storage)

all of these approaches have 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 been also proposed in the EVE project [19]. The exchange of event notifications plays an important role in our approach, however, in our experience, ECA rules are only a useful intermediate representation to bridge the gap between graph based models, which can be more readily understood by the user designing the process, and the corresponding executable code.

The theme of enhancing the system's fault tolerance and scalability through replication at the database layer has been pioneered by [28]. Also in the MOBILE project [21], in order to replicate the process execution layer, a scalable strategy for distributing the process data among separate databases has been proposed [39]. Although we compare the performance of a centralized, persistent repository with a distributed, volatile implementation, in this paper we do not pursue replicated storage any further.

Decentralization has been pursued by the MENTOR project [50], where process definitions are analyzed and automatically partitioned among distributed execution sites in order to avoid the bottleneck of a centralized engine [34]. This approach fits well with the requirements of workflows spanning across multiple organizations. However, it is possible for one execution site to 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 quality of service guarantees become interesting options. In [27] a

cluster-based workflow management system has been presented focusing on a quantitative comparison of two different load balancing strategies. In [5] simulations are used to study how different workloads influence the load of the network and thus, the scalability of the workflow engines in the context of several distributed architectures. In [20] extensive simulations are used to validate a composition model with quality of service guarantees based on service overlay networks.

To the best of our knowledge, the BPEL4WS [25] specification is currently supported by two implementations. Both execution engines are meant to be deployed inside an application server. The Collaxa BPEL Server [13] is the most advanced as it comes with a graphical process designer and debugger. The visual notation employed has a very close mapping to the underlying BPEL document. This has the advantage that a BPEL document doesn't 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 second implementation is the Business Process Execution Language for Web Services Java Run Time (BPWS4J [23]) from IBM, which also includes an editor with minimal visual support.

11 Conclusion

E-Commerce applications composed of Web services are one of the most complex distributed application that can be built today, as they potentially involve the interaction and exchange 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.

In this paper we have 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 user in rapidly building processes from a library of existing component services and in monitoring their execution. We have not only developed an integrated set of tools for component library management, automatic layout of graphs, static type checking, process compilation, execution profiling, analysis and optimization, but have also successfully tried the system with computer science students developing small application integration projects.

In addition to our visual composition language, in this paper we also presented a novel architecture for a process support system kernel, which can be used to execute processes written in such language. The main innovation of this architecture consists of the ability to transparently tailor the system's performance to different quality of service 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 different properties regarding performance, scalability and reliability. In particular, we determine the cost of reliability by comparing navigation performed over volatile and persistent state. In this setting we also study the effect of caching. We also show that

the system's 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 our extensive set of process management tools, we are in the process of completing the integration of the new kernel in the existing BioOpera API [6]. We are also evaluating other possible implementations of the data storage layer, for example using JDBC with a relational database, and we 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 variation of the system's workload and investigating the feasibility of using peer to peer technologies for distributed storage and event propagation.

References

[1] R. Agrawal, A. Somani, and Y. Xu. Storage and Querying of E-Commerce Data. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB 2001)*, pages 149–158, Roma, Italy, 2001.

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

[3] M. Auguston and A. Delgado. Iterative constructs in the visual data flow language. In G. Tortora, editor, *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL97)*, pages 152–159, Capri, Italy, September 1997.

[4] T. Baeyens. Java business process management. `http://www.jbpm.org/`.

[5] T. Bauer and P. Dadam. 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, editors, *Proceedings of the 2nd IFCIS International Conference on Cooperative Information Systems (CoopIS'97)*, pages 99–108, Kiawah Island, South Carolina, USA, June 1997.

[6] W. Bausch, C. Pautasso, and G. Alonso. Programming for dependability in a service based grid. In S. Lee, S. Sekiguschi, S. Matsuoka, and M. Sato, editors, *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid03)*, pages 164–171, Tokyo, Japan, May 2003.

[7] A. Beguelin, J. J. Dongarra, A. Geist, R. Manchek, K. Moore, R. Wade, and V. S. Sunderam. HeNCE: Graphical development tools for network-based concurrent computing. In R. Voigt, J. Saltz, and L. O'Connor, editors, *Proceedings of the 1992 Scalable High performance Computing Conference (SHPCC-92)*, pages 129–136, Williamsburg, Virginia, April 1992.

[8] J. C. Browne, S. I. Hyder, J. Dongarra, K. Moore, and P. Newton. Visual programming and debugging for parallel computing. *IEEE parallel and distributed technology: systems and applications*, 3(1):75–83, Spring 1995.

[9] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, and P. van Zee. Scaling up visual programming languages. *Computer*, 28(3):45–54, March 1995.

[10] C. Bussler. *B2B Integration. Concepts and Architecture*. Springer, 2002.

[11] N. Carriero and D. Gelernter. *How to write parallel programs*. MIT Press, 1990.

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

[13] Collaxa. BPEL Server. `http://www.collaxa.com`.

[14] P. T. Cox and B. Song. A formal model for component-based software. In S. Levialdi, editor, *Proceedings of the 2001 IEEE International Symposium on Human-Centric Computing Languages and Environments (HCC 2001)*, pages 304–311, Stresa, Italy, September 2001.

[15] ebXML. *ebXML Business Process Specification Schema (BPSS) 1.01*, 2001. `http://www.ebxml.org/specs/ebBPSS.pdf`.

[16] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces: principles, patterns and practice*. Addison Wesley, 1999.

[17] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[18] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An overview of workflow management: From process modelling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, April 1995.

[19] A. Geppert and D. Tombros. Event-based distributed workflow execution with eve. Technical Report 96.05, Dept. of Computer Science, University of Zurich, 1998.

[20] X. Gu, K. Nahrstedt, R. N. Chang, and C. Ward. QoS-assured service composition in managed service overlay networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 194–201, 2003.

[21] P. Heinl and H. Schuster. Towards a highly scaleable architecture for workflow management systems. In *Proceedings of the 7th International Workshop on Database and Expert Systems Applications*, pages 439–444, 1996.

[22] D. D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing*, 3(1):69–101, 1992.

[23] IBM. BPEL4WS Java Runtime. `http://www.alphaworks.ibm.com/tech/bpws4j`.

[24] IBM. *TSpaces*. `http://www.almaden.ibm.com/cs/Tspaces/`.

[25] IBM, Microsoft, and BEA Systems. *Business Process Execution Language for Web Services (BPEL4WS) 1.0*, 2002. `http://www.ibm.com/developerworks/library/ws-bpel`.

[26] IBM, Microsoft, and BEA Systems. *Web Services Coordination (WS-Coordination)*, 2002. `http://www.ibm.com/developerworks/library/ws-coor`.

[27] L. jie Jin, F. Casati, M. Sayal, and M.-C. Shan. Load balancing in distributed workflow management system. In *Proceedings of the ACM Symposium on Applied Computing*, pages 522–530, 2001.

[28] M. Kamath, G. Alonso, R. Guenthoer, and C. Mohan. Providing high availability in very large workflow management systems. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Proceedings of the 5th International Conference on Advances in Database Technology (EDBT'96)*, Volume 1057 of *Lecture Notes in Computer Science*, pages 427–442, Avignon, France, March 1996. Springer.

[29] T. J. Lehman, A. Cozzi, Y. Xiong, J. Gottschalk, V. Vasudevan, S. Landis, P. Davis, B. Khavar, and P. Bowman. Hitting the distributed computing sweet spot with TSpaces. *Computer Networks*, 35(4):457–472, March 2001.

[30] F. Leymann. *Web Services Flow Language (WSFL 1.0)*. IBM, 2001.

[31] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[32] M. Mosconi and M. Porta. Iteration constructs in data-flow visual programming languages. *Computer Languages*, 26(2–4):67–104, July 2000.

[33] M. Muench and A. Schuerr. Leaving the visual language ghetto. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 148–155, 1999.

[34] P. Muth, D. Wodtke, J. Weissenfels, A. Dittrich, and G. Weikum. From Centralized Workflow Specification to Distributed Workflow Execution. *Journal of Intelligent Information Systems*, 10(2):159–184, 1998.

[35] Oasis. *Universal Description, Discovery and Integration of Web Services (UDDI) Version 3.0*, 2002. `http://uddi.org/pubs/uddi_v3.htm`.

[36] C. Pautasso. JOpera: Process Support for Web Services. `http://www.iks.ethz.ch/jopera`.

[37] C. Pautasso and G. Alonso. Visual Composition of Web Services. In *Proceedings of the 2003 IEEE International Symposium on Human-Centric Computing Languages and Environments (HCC 2003)*, pages 92–99, Auckland, New Zealand, 2003.

[38] H. Schuldt, G. Alonso, C. Beeri, and H.-J. Schek. Atomicity and isolation for transactional processes. *ACM Transactions on Database Systems (TODS)*, 27(1):63–116, 2002.

[39] H. Schuster and P. Heinl. A workflow data distribution strategy for scalable workflow management systems. In *Proceedings of the 1997 ACM symposium on Applied computing*, pages 174–176, 1997.

[40] S. Thatte. *XLANG: Web Services for Business Process Design*. Microsoft, 2001.

[41] W. M. P. van der Aalst. The application of petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[42] W. M. P. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18(1):72–85, 2003.

[43] W. M. P. van der Aalst and P. J. S. Berens. Beyond workflow management: product-driven case handling. In *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, pages 42–51, 2001.

[44] W3C. *Simple Object Access Protocol (SOAP) 1.1*, 2000. `http://www.w3.org/TR/SOAP`.

[45] W3C. *Web Services Definition Language (WSDL) 1.1*, 2001. `http://www.w3.org/TR/wsdl`.

[46] W3C. *Web Services Choreography Interface (WSCI) 1.0*, 2002. `http://www.w3.org/TR/wsci`.

[47] W3C. *Web Services Conversation Language (WSCL) 1.0*, 2002. `http://www.w3.org/TR/wscl10`.

[48] G. Wiederhold, P. Wegner, and S. Ceri. Towards megaprogramming: A paradigm for component-based programming. *Communications of the ACM*, 35(11):89–99, 1992.

[49] G. Wirtz, M. Weske, and H. Giese. Extending uml with workflow modeling capabilities. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS-2000)*, Volume 1901 of *Lecture Notes in Computer Science*, pages 30–41, Eilat, Israel, September 2000. Springer.

[50] D. Wodtke, J. Weissenfels, G. Weikum, and A. Kotz-Dittrich. The mentor project: Steps toward enterprise-wide workflow management. In S. Y. W. Su, editor, *Proceedings of the 12th International Conference on Data Engineering (ICDE 1996)*, pages 556–565, New Orleans, Louisiana, February 1996.