Visual Composition of Web Services

Cesare Pautasso, Gustavo Alonso
Department of Computer Science
Swiss Federal Institute of Technology (ETHZ)
ETH Zentrum, 8092 Zürich, Switzerland
{pautasso,alonso}@inf.ethz.ch

Abstract

Composing Web services into a coherent application can be a tedious and error prone task when using traditional textual scripting languages. As an alternative, complex interactions patterns and data exchanges between different Web services can be effectively modeled using a visual language. In this paper we discuss the requirements of such an application scenario and we present the design of the BioOpera Flow Language. This visual composition language has been fully implemented in a development environment for Web service composition with usability features emphasizing rapid development and visual scalability.

1 Introduction

Although they may not solve all application integration problems, Web service technologies show great promise in reducing the complexity of interconnecting heterogeneous software components across the Internet. Yet, Web services can realize their full potential only through the ability to compose complex services out of agglomerations of basic services [8]. Especially in E-Business scenarios, the standardization efforts that integrate Web services into business processes have recently produced many non visual, XML oriented proposals [10, 14, 15, 16, 21, 26, 27]. However, none of these is yet well established in practice [23].

As we suggest in this paper, a visual approach may very well be a natural complement to such composition standards. The order of execution of services, the data exchanges between them and the necessary failure handling behaviour can all be specified with a simple visual syntax. In this paper we present in depth our visual composition language called BioOpera Flow Language (BFL). The language is intended to be used as a generic glue language [11] for coordinating collections of coarse-grained software components. Nevertheless, its development and runtime environment has been tailored to specifically support the composition of Web services.

With this, the paper makes several significant contributions. First, it proposes the application of visual programming to the domain of Web services composition. Second, it introduces the syntax and semantics of the BFL visual composition language, which is expressive enough to be applied to realistic settings. Third, in discussing the usability features of the development environment, the paper illustrates our approach to improve the visual scalability of the language [7] by using nesting constructs, multiple views and automatic support for graph layout and diagram navigation.

To prove the feasibility of the approach, an editor, a compiler, and other monitoring and debugging tools for the BioOpera Flow Language have been developed within the BioOpera system [4], a middleware infrastructure for coordinating distributed processes with applications in bioinformatics [1] as well as in cluster and grid computing [3]. In extending the BioOpera system with a visual language our main goal was to provide an intuitive graphical syntax for the purpose of rapidly developing, executing and monitoring distributed applications composed out of a library of Web services. To avoid misinterpretation problems [12] we reduced the number of *ad hoc* constructs and extensions to a minimum, keeping the balance between the need for expressive features and the underlying semantics and constraints imposed by the BioOpera runtime platform.

This paper is structured as follows. First, Section 2 introduces the Web services composition application domain. Section 3 presents the syntax and semantics of the BFL visual language. Section 4 describes the most innovative features of our integrated development environment. In Section 5 we prove the feasibility of the approach with some examples. Section 6 covers related work and Section 7 concludes the paper.

2 Web services composition

Before describing in detail the visual language, we present our model for building applications by composing different Web services. This way, we 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 [25]), for publishing and discovering services (UDDI [20]) and for invoking them using different communication protocols (SOAP [24]).

Once it is possible to interact with individual services, the ability to compose and describe relationships between basic services becomes important. 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 [26], orchestration [14], automation [21], coordination [15], collaboration [10], and conversation [27].

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: thanks to the flexible runtime architecture of BioOpera 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 job submitted to a cluster scheduling system, or a request to perform a grid computation. In the rest of the paper we will focus on components modeling individual calls to a Web service 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. Like in data-driven data flow languages a task cannot be started until all of its data dependencies are satisfied [13]. 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. BioOpera 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 BFL 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 [27] description of the conversation is automatically mapped to a process, defining the basic sequence of invocations and information exchanges between the various operations.

3 BioOpera Flow 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 BFL language used to 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.

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 (Fig-

ure 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 useful in case of a loop or when the corresponding control flow merges from two or more alternative execution paths. The BioOpera 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

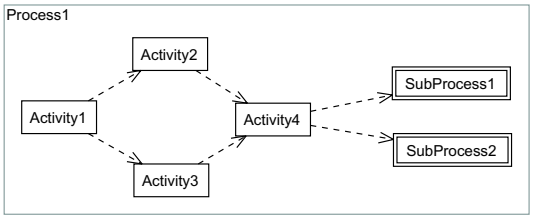

Figure 1. Sample control flow syntax

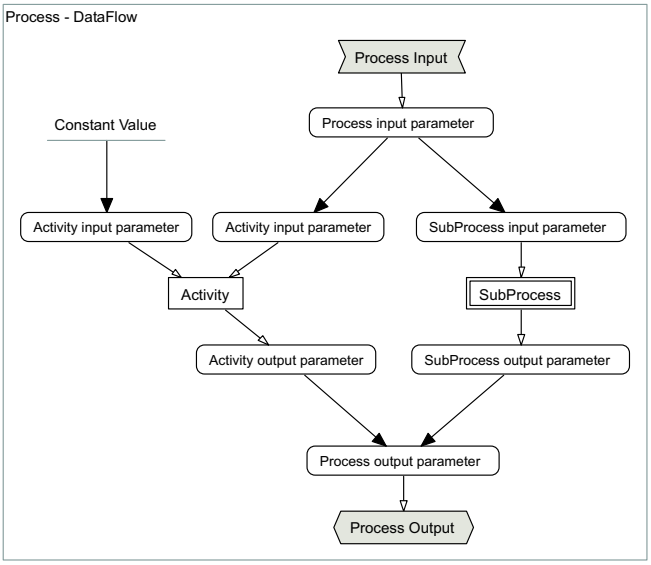

Figure 2. Sample data flow syntax

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 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

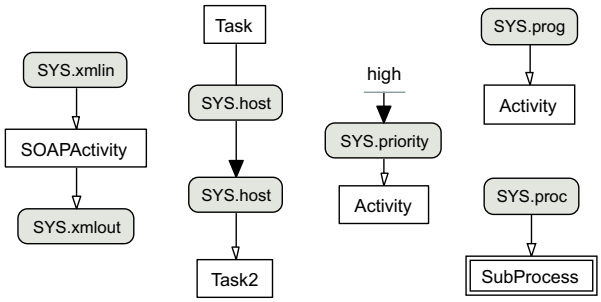

Figure 3. Example of system parameters

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 1 and 4 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

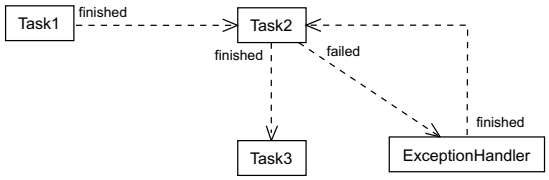

Figure 4. Control flow with exception handler

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 4, 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 [18]. 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. 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 rewrit-

ing schemes [5], the overall effect at runtime is to replicate a set of nodes for each value of the input parameter list (Figure 8). 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.

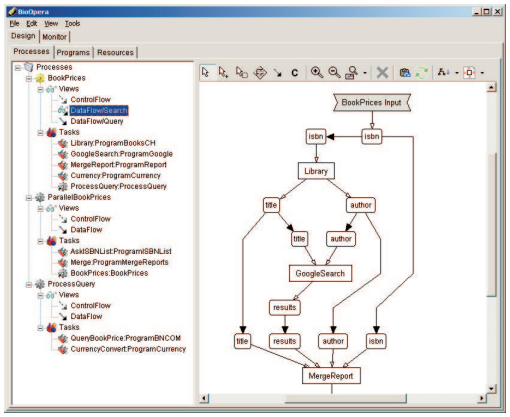

Figure 5. BioOpera process developer

4 Visual development environment

The BioOpera visual process development environment provides an integrated toolkit to manage the whole lifecycle of a process (Figure 5). 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 bind parameters with matching names. 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 BioOpera runtime environment for execution[1]. 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.

4.1 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 [17].

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.

[1] Please refer to [3] and [4] for more information on the process execution and monitoring features of BioOpera.

5 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.

5.1 Process BookPrices

Figure 6 shows the control flow graph for the price comparison process. The process is composed of 3 activities (`Library`, `GoogleSearch`, `MergeReport`) and 1 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 7 shows one view with data parameters and bindings of the `Library`, `GoogleSearch`, and `MergeReport` activities. While the second view in Figure 8 shows the data flowing through the `QueryBookPrice` sub-process.

The first view (Figure 7) 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

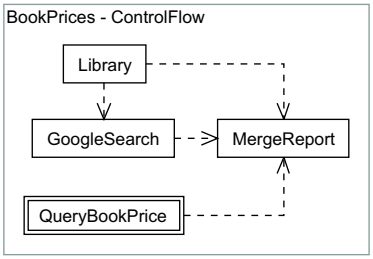

Figure 6. Control flow graph of the BookPrices process

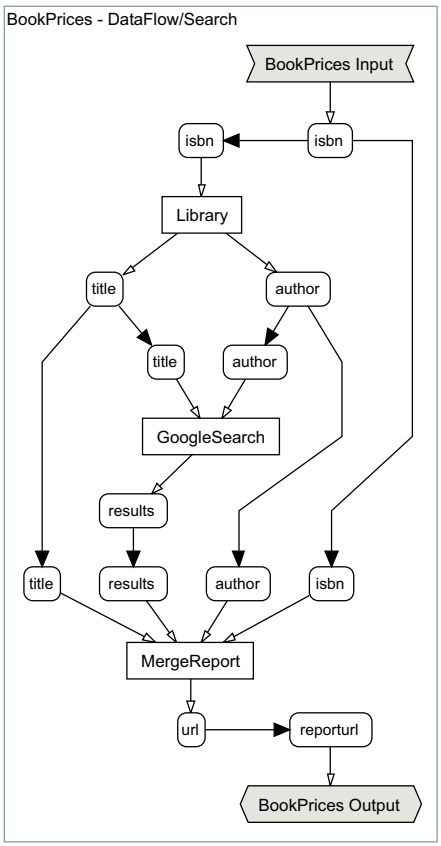

Figure 7. First data flow view of the BookPrices process

the `GoogleSearch` activity, which will run a web 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 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` 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 iden-

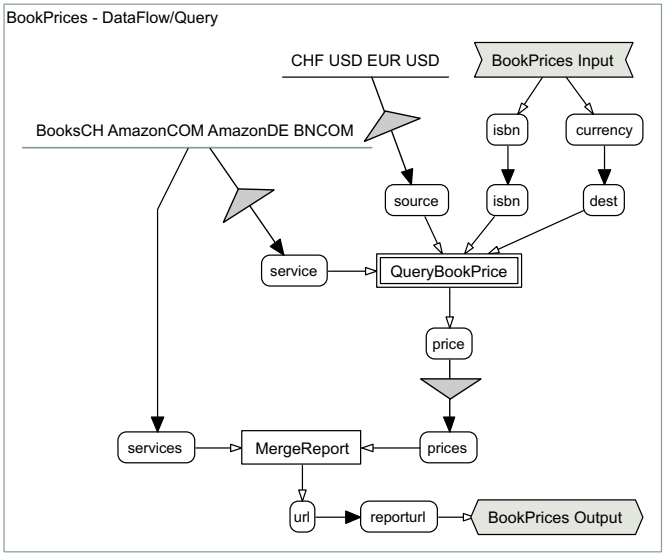

Figure 8. Second data flow view of the BookPrices process with parallel split and merge operations

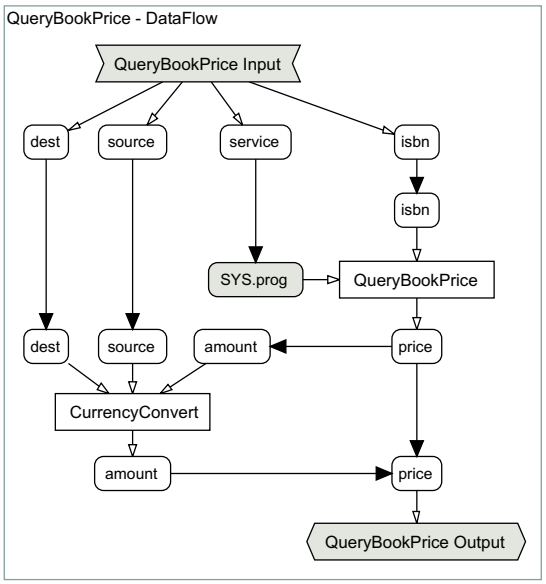

Figure 9. Main data flow view of the query process

tifiers (`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.

5.2 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 9 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.

6 Related Work

The idea of developing large scale applications by composing coarse grained, reusable component modules has been pioneered by [28]. A formal model for software based on traditional CORBA, EJB and COM components has been developed in [9], while an overview over established component based visual languages can be found in [19]. A good argument on the need for a composition "glue" language, different from traditional programming languages has been presented in [11].

A similar, two-step approach has been proposed in the parallel computing domain [6]. 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 notation, where the data flow is derived implicitly by matching parameter names [5].

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 [18]. An example of iteration through vector operators and conditional switches is [2].

Finally, graphical formalisms have also been used as a modeling tool in the workflow community. Examples include State Charts, used in the the Mentor project [30] to achieve distributed execution of the various workflow steps, or Petri Nets [22] and variations such as Object Coordination Nets (OCoN) [29].

7 Conclusion

This paper presents the BioOpera Flow 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 BioOpera 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 both computer science students developing small application integration projects and bioinformaticians working on large scale computations based on Web service technologies [3].

Acknowledgments Part of this work is supported by grants from the *Hasler Foundation* (DISC Project No. 1820).

References

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

[2] M. Auguston and A. Delgado. Iterative Constructs in the Visual Data Flow Language. In *Proc. of the IEEE Symposium on Visual Languages*, pages 152–159, 1997.

[3] W. Bausch, C. Pautasso, and G. Alonso. Programming for Dependability in a Service Based Grid. In *the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid03)*, Tokyo, Japan, 2003.

[4] W. Bausch, C. Pautasso, R. Schaeppi, and G. Alonso. BioOpera: Cluster-aware computing. In *Proc. of the 4th IEEE International Conference on Cluster Computing (Cluster2002)*, Chicago, IL, 2002.

[5] 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 *Scalable High performance Computing Conference (SHPCC-92)*, pages 129–136, 1992.

[6] 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, 1995.

[7] 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, 1995.

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

[9] P. T. Cox and B. Song. A formal model for component-based software. In *Proc. of the IEEE Symposium on Human Centric Computing*, pages 304–311, 2001.

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

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

[12] C. A. Gurr. Effective diagrammatic communication: Syntactic, semantic and pragmatic issues. *Journal of Visual Languages and Computing*, 10(4), 317–342 1999.

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

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

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

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

[17] 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.

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

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

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

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

[22] 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.

[23] 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.

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

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

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

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

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

[29] G. Wirtz, M. Weske, and H. Giese. Extending UML with Workflow Modeling Capabilities. In *7th International Conference on Cooperative Information Systems (CoopIS-2000)*, Filat, Istrael, 2000.

[30] D. Wodtke, J. Weissenfels, G. Weikum, and A. Kotz-Dittrich. The Mentor Project: Steps Toward Enterprise-Wide Workflow Management. In *Proc. of the 12th International Conference on Data Engineering*, pages 556–565, Feb. 1996.