Published in final edited form as:
Concurr Comput. 2010 June 25; 22(9): 1098–1117. doi:10.1002/cpe.1547.

A Comparison of Using Taverna and BPEL in Building Scientific Workflows: the case of caGrid

Wei Tan[1], **Paolo Missier**[2], **Ian Foster**[1,3], **Ravi Madduri**[1,3], and **Carole Goble**[2]

Wei Tan: wtan@mcs.anl.gov; Paolo Missier: pmissier@cs.man.ac.uk; Ian Foster: foster@anl.gov; Ravi Madduri: madduri@mcs.anl.gov; Carole Goble: carole.goble@manchester.ac.uk

[1]Computation Institute, University of Chicago and Argonne National Laboratory, Chicago, IL, USA

[2]School of Computer Science, University of Manchester, Manchester, U.K

[3]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL

Abstract

With the emergence of "service oriented science," the need arises to orchestrate multiple services to facilitate scientific investigation—that is, to create "science workflows." We present here our findings in providing a workflow solution for the caGrid service-based grid infrastructure. We choose BPEL and Taverna as candidates, and compare their usability in the lifecycle of a scientific workflow, including workflow composition, execution, and result analysis. Our experience shows that BPEL as an imperative language offers a comprehensive set of modeling primitives for workflows of all flavors; while Taverna offers a dataflow model and a more compact set of primitives that facilitates dataflow modeling and pipelined execution. We hope that this comparison study not only helps researchers select a language or tool that meets their specific needs, but also offers some insight on how a workflow language and tool can fulfill the requirement of the scientific community.

Keywords

scientific workflow; functional programming; Taverna; BPEL; caGrid

1 Introduction

E-science practice increasingly relies upon service-oriented architectures (SOA) to access data and software [1]. Given the proliferation of Web services, service-oriented science [2] is emerging as an important paradigm for facilitating scientific investigation. Users inevitably want to compose calls to multiple services to define more advanced functionality —in other words, to create what are commonly, if perhaps incorrectly, called workflows [3].

The cancer Biomedical Informatics Grid (caBIG) [4], sponsored by the US National Cancer Institute, is an example of a service oriented science initiative that requires workflow tools. caBIG has deployed a service-based software infrastructure, caGrid [5], based on Globus Toolkit [6], which enables the secure and reliable sharing of data and analytical resources as services. By this means it helps cancer researchers leverage and contribute to caBIG resources, accelerating biomedical research in a Web-scale, multi-institutional environment. As the number of caGrid services grows, the need emerges for service composition capabilities. For example, a user may want to specify that data should be extracted from

multiple data services, pass that data to a series of analytical services for processing, and deposit the results of the analysis in another data service.

Academia and industry have developed many languages, tools and systems for scientific workflow [3], which serve diverse application domains, address e-science problems at different scale (of computational complexity and data volume, for example), and expect different levels of IT expertise from users. In this landscape, caGrid users are cancer scientists with limited IT expertise, who are nevertheless ready to use a compact and expressive language to describe their data processing routines. caGrid users also welcome an easy-to-use tool suite to cover the full data processing lifecycle, from selecting services, to composing them in a given sequence, to analyzing the results. Through a comprehensive survey of workflow tools [7], caGrid selected Taverna [8] and BPEL [9] as candidate solutions for adoption by its users. These systems were shortlisted as they represent typical scientific modeling and business process modeling styles, respectively, while at the same time providing models for the orchestration of Web services, a key requirement for exploiting the heavily Web-service based caGrid infrastructure.

Recognizing that designers of scientific applications are primarily interested in modeling workflow patterns that are typical of their areas, in this paper we compare and contrast these two systems, with the goal of providing scientific users community insights into their similarities and differences. This analysis is based upon our experience working with both BPEL and Taverna on behalf of the caGrid scientific community. We focus mainly on the design aspects of scientific workflows, namely modeling style and support for specific design patterns, building upon our experience in modeling and orchestrating scientific workflows in caGrid.

The comparison is aimed at providing practitioners, i.e., workflow designers and users, with useful elements for selecting appropriate tools to achieve their scientific goals. Thus, while the paper does not include a formal comparison between two programming languages, it does make a comparison between two distinct workflow modeling styles, namely functional and data-driven (Taverna) vs. imperative and control-based (BPEL). With *functional and data-driven*, or *dataflow*, we refer to a design style where services are viewed as functions, and their composition is specified by data dependencies between outputs of one service and inputs to another. In contrast, *imperative and control-based* refers to a workflow design in which services are primitive execution blocks, and service composition is achieved using control primitives (primarily sequence, conditional branching, and loops).

While the Taverna system is a complete implementation of a workflow design language and execution model, the BPEL standard only covers the language specification along with its semantics. As our goal is not to focus on any specific BPEL implementation, and given these differences in scope, our analysis is limited primarily to language elements, while our discussion on operational properties such as reliability, fault tolerance, performance optimizations, etc. is limited to what can be inferred from the language semantics, rather than based upon an experimental comparison.

A typical workflow lifecycle can be described as a sequence of steps, primarily (a) discovery of data/analytical services, (b) composition of those services into a workflow, (c) execution of workflows, and (d) analysis of results through provenance metadata collection and query (see Section 2). We use this lifecycle as a guideline in our analysis, as all of these steps, with the possible exception of (a), can be used to highlight differences between workflow systems.

In the remainder of this paper we first review the landscape of current workflow models. Next, we examine the lifecycle and features of scientific workflows, and present a caGrid

use case. Then, we compare Taverna and BPEL from three perspectives: service composition, workflow execution, and workflow results analysis. Finally, we draw conclusions.

2 Why BPEL and Taverna

A distinction can be made between workflow languages that are designed to address business management processing problems, from those that address scientific data and analysis processing. This distinction arises from the different nature of business and scientific applications. Business workflows present the routines inside or between enterprises, and therefore comprehensive control-flows embodying business rules, process integrity (including transaction and audit) and human involvement are the major concerns. Scientific workflows present the experiments conducted by scientists and therefore, to quickly process data and schedule the computation appropriately are the major concerns.

Languages like WS-BPEL belong to the first category. WS-BPEL (Web Service-Business Process Execution Language or BPEL for short) is an XML-based specification that describes the behavior of a business process that is composed of Web services, where the workflow itself is also exposed as a Web service. Although originally designed for business workflows, BPEL has recently enjoyed some adoption by the scientific community, mainly because of the appeal of the SOA paradigm in this space [10,11].

Systems like Taverna, Kepler [12], Triana [13], and Askalon [14] belong to the second category. They all offer a visual notation for Web service composition, with a corresponding serialization format for portability and interpretation. The caGrid project selected Taverna from this category because of its integration with Web service technology, support for third party plug-ins, and broad user base within the bioinformatics/biomedicine community.

A number of other systems, notably Swift [15] and Pegasus [16], support the composition, or scripting, of executable programs rather than Web Services, and the execution of those scripts on large-scale parallel and distributed computing systems. These parallel programming systems are also sometimes referred to as workflow systems, but as they do not support the composition of Web services, and further do not support a graphical programming model, we do not consider them further here.

Developed in the UK by the myGrid consortium (http://www.mygrid.org.uk), Taverna is an open-source workbench for the design and execution of scientific workflows. Aimed primarily at the life sciences community, its main goal is to make the design and execution of workflows accessible to bioinformaticians who are not necessarily experts in Web services and programming. A Taverna workflow consists of a graph of *activities*, which represent Web services or other executable components. Activities receive their inputs on input ports, and produce outputs on output ports. An arc in the graph connects an output port of one activity to an input port of another, and thus it specifies a data dependency. Workflows are specified using a visual language, and executed according to a dataflow computation model [17]. Taverna also provides a plug-in architecture so that additional applications, such as secure Web Services, can be populated to it. The caGrid plug-in recently implemented by members of our group [18] is an example.

Although BPEL and Taverna share some common modeling elements, these come as a result of different design motivations, which account for different programming styles. BPEL is designed to orchestrate interactions among business partners that expose their services using Web service interfaces, and to facilitate their interoperability. In addition, transactional properties are viewed as critical to the robustness of business transactions,

notably atomicity, i.e., the ability for the system to roll back to a consistent state following transaction failure.

The main motivation for the design of Taverna, on the other hand, is the support for data-intensive, scientific dataflows, which allows users to orchestrate the invocation of arbitrary Web services, scripts, and other sources of process logic while imposing minimal programming requirements on the users.

Taverna and BPEL have a historical connection. The original Taverna enactor, FreeFluo, supported both a WSFL [19] subset and for Scufl. Then WSFL joined with XLANG [20] to create BPEL 1.0. Specifically, in this paper we will compare BPEL 2.0 [9] and the workflow specification used by Taverna 2.0.

3. Lifecycle of Scientific Workflows and an Example Workflow

To define the scope of the comparative study, we first discuss the scientific workflow lifecycle. This lifecycle, illustrated in Fig. 1, involves four stages: discovery of data/analytical services, composition of services into a workflow, workflow execution, and analysis of results. Analysis of results can itself lead to further workflow problems.

1. *Discovery of data/analytical services*. In a Web-scale system like caGrid, data/analytical services are developed, owned and maintained by different institutions. Usually the URL of these services is not well known. Moreover, the scientific community is too autonomous to share a common terminology, so domain knowledge is needed to determine the semantics of services whose syntax is already known. We acknowledge that the discovery of services is a challenge for both business and scientific workflows. However, since the issue of service discovery is orthogonal to workflow languages, it is beyond the scope of this paper.
2. *Composition of services into a workflow*. After individual services are found, the next step is to compose them into a workflow. This step involves the addition of data and control dependencies between services; it may also involve data transformations, if the output of one service is not in the format required for another service's input. Composition can be performed by using a general-purpose programming language like Java, a proprietary scripting notation like Scufl (used by Taverna), or a graphical user interface (GUI). A GUI usually depends on a script format for persistent storage.
3. *Execution of workflows*. A workflow definition is sent to an engine for execution. The engine invokes the individual services in some order, providing input and retrieving output of services during the ordered execution. As we can see later, in a functional language like Taverna, the invocation order of the services is driven by the data, rather than by control as with an imperative language, and therefore cannot be determined statically.
4. *Analysis of results*. Scientific workflow is for the purpose of exploratory research, and therefore, the intermediate results generated by component services, as well as the final results yield by the workflow, are of great value and deserve to be analyzed carefully. Scientific researches are usually undertaken in an iterative manner so the analysis results often initiate another round of workflow modeling/execution.

The following features of the scientific workflows that we wish to create in caGrid can also be seen as challenges encountered when providing a scientific workflow solution in a more general sense (the numbers refer to the lifecycle steps ofFig. 1):

- (2) Data-flow oriented. Data is considered to be a first-class citizen in scientific workflows, because scientific workflows are mostly pipelines of parallel data processing. In a dataflow, tasks and links represent data processing and data transport, respectively. It is important that concurrent execution of independent tasks be easily modeled at build time and enforced at run time. In Section 4 you can see that this is a non-trivial issue.
- (2&3) Large scale. Scientific workflows often contain many tasks, involve large data sets, and require intensive computation. The language should make it easy to model such complex workflows.
- (4) Data analysis and provenance is an important step and the workflow execution can be in an iterative manner.

In the rest of this paper we highlight differences between BPEL and Taverna from the point of view of their impact on the users' experience in the scientific workflow lifecycle. The discussion is organized according to the lifecycle model of Fig. 1 (except the discovery stage).

We use an example caGrid workflow to guide the comparison. This workflow is about the querying of semantic data in cancer research. The use of a standardized meta model and semantic annotation to enable the formal description and harmonized use of data is a primary feature that moves caGrid beyond an elementary Web services infrastructure.

This CaDSR workflow relates to the Cancer Data Standards Registry and Repository (CaDSR). In Fig. 2, boxes represent data items and arrows represent service operations. A user wants to query description logic concepts that relate to a particular context, for example, “caCore.” First, the user queries all projects related to context “caCore”, using operation *findProjects*; second, they search UML classes within each project, using *findClassesInProject*; third, they use project and UML class information to query the semantic metadata, using *findSemanticMetadataForClass*; and finally, they use the semantic metadata for a class to retrieve all the description logic concept used to describe this class, with operation *searchDescLogicConcept*. This simple workflow is well suited for our comparison since 1) it is a typical dataflow in which execution is driven by data availability, and 2) intermediate data contains multiple items so parallel and pipelined execution needs to be explored. In Section 4 we will use segments of this workflow to contrast the different modeling style of BPEL and Taverna. We have implemented the complete workflow using both Taverna and BPEL.

4 Workflow Modeling and Execution

We contrast the design and execution models of BPEL and Taverna that pertain to phases (2) and (3) of the workflow lifecycle: the composition of services into workflows, and workflow execution. Taverna offers a set of visual notations to design workflows, and so do the different implementations of BPEL editors. The major differences between BPEL and Taverna are in the modeling style, which are imperative and control-driven, vs. functional and data-driven, respectively. The consequences of these differences, on the design and execution models, include the following:

- Scheduling and parallelism: in data-driven workflows, the execution engine hides details of task scheduling, and manages the available parallelism amongst tasks (sec. 4.2)
- Multiple task activation: in Taverna, the same task can be activated multiple times, and this occurs implicitly, through iterations over input collections. In contrast, BPEL requires the use of explicit control loops (sec. 4.3)

- Pipelined execution and support for data streaming: a data-driven model makes it simple to support stream-based input, i.e., to handle input of unknown size (sec. 4.4)

4.1 Summary of design and execution model differences

In BPEL, workflows specify service orchestration using control primitives, much as in imperative programming languages. In contrast, workflow modeling in Taverna is data-centric, i.e., the workflow graph describes data dependencies. Correspondingly, a workflow is executed according to a dataflow model, whereby tasks are activated when data, rather than control, reaches them. Dataflow computation has the advantage, from the user's perspective, that system-level issues such as task scheduling and parallelization are completely transparent to the designer, as parallelism is specified implicitly through data dependencies.

A second difference relates to how multiple task activation is handled. In BPEL, designers use control loop primitives to specify the repeated invocation of a task, subject to some condition (see sec. 4.3). This structure is reflected in the presence of loops in the workflow graph. In contrast, since a Taverna workflow specifies data dependencies, the graph must be acyclic, as circular dependencies are prohibited. Thus, there is no primitive for explicit specification of multiple invocations of a task. However, when data consists of collection values (i.e., nested lists), the dataflow model admits the execution of a task on each element of an input list, independently from the others. This mechanism results in multiple activations of a task, a behavior that we refer to as *implicit iterations of tasks over lists*. These iterations are implicit because they are, once again, driven by the structure of the data, rather than being specified explicitly by the workflow designer. Furthermore, since the iterations are independent of one another, they are easily executed in parallel. As we will see in Sec. 4.5, this mechanism forms the basis for a pipelined execution of multiple parallel streams of data, a feature of the Taverna 2 execution engine.

Table 1 summarizes some of the differences between the two systems. We articulate these differences in more detail in the rest of this section.

4.2 Data -driven vs. control-driven modeling

A data-driven workflow, or dataflow, is described by a graph where nodes represent activities that can be executed on input provided along the incoming arcs, and whose output is forwarded to other activities through outgoing arcs. In this dataflow, the order in which the activities are executed is determined primarily by the order in which the data appears on the various inputs. Any activity for which the input data is available can be scheduled for execution. Scheduling in Taverna is simple and greedy: activities are executed as soon as possible, as long as a new execution thread can be started. (A limit on the number of threads can be defined on the scheduler.) Thus, the scheduler can manage all parallelization of activity execution, based on the data dependencies and without the need for explicit user directives. Also, the order of execution of two independent activities may be different for different executions of the same workflow, even on the same input, due to the possible variations in execution speeds of other activities.

In contrast, a procedural workflow language like BPEL includes the explicit definition of the control flow that determines the order of execution of the activities. In particular, parallel execution of independent activities must be specified explicitly.

A comprehensive analysis on the differences and relative merits of control-driven and data-driven execution is beyond the scope of this paper. See Shields [21] for more in-depth

discussion. In the rest of this section we focus on the specific differences between Taverna and BPEL, summarized in Table 1.

Implicit vs. explicit definition of data—In Taverna, each activity has a set of input and output ports, each with an associated data type. Data travels from an output port of one activity to the input port(s) for of one or more downstream activities. In order words, Taverna variables are either the input or the output of some activity, they are only *locally* visible to the associated activity and data links, and they are *immutable*. These properties guarantee that no variable is referenced unexpectedly and that there is no access conflict in parallel execution. No other data structure specification is allowed besides the port types, and interaction among activities is defined entirely by the arcs in the dataflow graph. As a specific case of input variables, Taverna provides a special built-in activity, called an *XML splitter*, which automatically pulls apart an input XML message whose structure is defined in a WSDL interface, providing the message components as individual outputs, so that their values can later be more easily accessed by other user-defined activities. An example of its use is provided below.

In contrast, BPEL requires the explicit definition of variables to hold data structures that are meant to be shared among activities. In other words, BPEL's variables are *global* as well as *mutable*. Also, complex variables must be explicitly initialized prior to their first use (i.e., by means of the <copy> construct – see Section 8.4.2 of the WS-BPEL Specification [9]), otherwise an exception will be raised. Although BPEL's requirement for explicit data definition takes additional effort, it also provides flexibility. For example, in BPEL you can easily define data that controls the overall flow but is not the input/output of any activities, whereas in Taverna the notion of data is inevitably linked to the input or output of some activity.

Since BPEL uses globally shared variables while Taverna uses locally visible ones, the two models exhibit different data passing patterns [22]. In BPEL, there is no explicit data passing. Instead, the <assign> activity is used to manipulate globally shared variables. In contrast, data is passed explicitly in Taverna along data links. Thus, each activity that wants to use an output variable must add a data link from its associating port. See Fig. 3 for a simple example. Suppose the output of service A is exactly the input of service B. With BPEL you need to add two <invoke> and an <assign> to copy the output of A to input of B. Moreover, a <sequence> is needed to ensure the correct execution order. In Taverna you also need to add two activities to invoke service A and B, respectively. However in Taverna the assignment of the output of A to input of B is achieved by adding a data link to the associated port.

In this regard, BPEL imposes a heavier burden on the programmer than Taverna does, in particular to ensure correct variables initialization. Moreover, in a dataflow where parallel execution is intensive, users must be careful when using BPEL since multiple <assign> activities could affect the same variable concurrently; this problem can be hard to debug, since data manipulation is hidden inside <assign> activities, rather than being expressed explicitly as data links.

The imperative vs. functional difference in modeling style also influences the way BPEL and Taverna deal with fault. As we have mentioned in Section 2, transactional properties are viewed as critical to the robustness of business processing and therefore BPEL provides a rich set of primitives to keep the process itself as well as participant services in a consistent state upon process failure. In BPEL, faults can be raised from either Web service invocations or thrown explicitly by the process itself (e.g., by detecting that the business data is not in a consistent state). Faults are caught by *fault handlers* and processed there with

business-relevant execution logic. Usually a fault handler needs to compensate for the work that has already been accomplished to keep the business logic in a consistent state. The ability to compensate is crucial because usually not all service invocations inside the business process participate in the same ACID transaction and therefore it is important to eliminate the effect of executed tasks. For example, an already accomplished task might have sent an email to the customer regarding some business decision. In this case, another email to cancel the decision may be the right compensation task.

Taverna embodies a functional programming style, which assumes the side-effect-freeness of involving functions (i.e., activities). With this assumption, Taverna handles failures in a simple and greedy way, i.e., it asks the failed activities to retry (or try a substitute) and meanwhile pushes partial results through the workflow so that some branches may still proceed. This assumption is reasonable because a large majority of Web services (for example, most caGrid services) are stateless and side-effect free, being concerned simply with either data querying or data processing If such services fail to yield a result, Taverna can simply retry them or resort to a substitute. Taverna will require enhancement if it is to handle failed stateful services, such as services that create resources or update databases.

4.3 Implicit vs. explicit iteration on data

In parallel computing, SPMD (Single Process, Multiple Data) [23] refers to the concurrent execution of multiple instances of one program, usually, but not always, operating on different data. SPMD is common in *embarrassing parallel* applications [24], where there is no communication among parallel execution branches. Taverna embraces this computation paradigm through implicit iteration of processors over elements of a list.

Each port in a Taverna activity has a type, which is either a simple type value, for example string 's', or a list, possibly nested, of simple type values, denoted l(s), l(l(s)), etc. Furthermore, lists in Taverna are homogeneous, that is, all the values are of the same simple type, and simple values always appear at the same level within the list. Thus, a port type is completely determined by a pair (*t*, *d*) where *t* is the simple type and *d* is the list depth.

As part of normal processing, it may be the case that an input port of type (*t*, *d*) receives a value of type (*t*, *d'*) with *d* != *d*, i.e., the values are of the correct type, but at different depth than expected. Thus, an activity that outputs a value of type l(*s*), for example, could be connected to an activity with an input port of type *s*. This is not considered as a type error, and indeed, accounting for this possibility is an important feature of the Taverna model, in that it allows composing services that produce lists, with services that are designed to accept simple values. To see why this is the case, consider again our running example CaDSR, where the main operations *findProjects*, *findClassesInProject* and *findSemanticMedatadaForClass* are highlighted in Fig. 4 and Fig. 5. What is important here is the one-many relationships between projects and UML classes that describe the project, and between those classes and their semantic metadata descriptions (expressed as a set of logic axioms).

These relationships are reflected in the operations' signatures: *findProjects* returns a list of projects, *findClassesInProject* takes a single project and returns a list of relevant UML classes for it, and *findSemanticMedatadaForClass* takes a UML class and retrieves its associated logic axioms. In the workflow, however, *findClassesInProject* takes an input list of projects, rather than a single project, as it is connected to *findProjects*.[1] By accepting a list as input to *findClassesInProject*, Taverna makes it possible to seamlessly compose the two operations, although *findClassesInProject* is designed to handle one input project at the time (note, however, that any type mismatch where t! = t', for example string – number, is considered a type error).

The natural semantics that makes this behavior possible involves iterating over the input list of projects, executing operation *findClassesInProject* independently on each element if the list. This behavior is consistent with Taverna's functional programming model [17][25], whereby the application of a function *f* with a formal argument of type *t*, to an actual parameter *x* of type list(*t*), is interpreted as (map *f x*). In practice, a depth difference *d*'-*d*>0 is an indication that an implicit iteration over the input takes place. Additionally, when depth differences appear simultaneously on multiple input ports of an activitiy, that activity is executed repeatedly on each element of the *cross-product* (i.e., a Cartesian product) of those input lists. When the input lists have the same length, users may optionally indicate that the inputs for each iteration are to be taken from a *dot product* as opposed to the cross product.

From this description, it should be clear that BPEL and Taverna differ substantially in the way they support repeated invocation of a service. The implicit iteration feature is commonly used in Taverna scientific workflows, and it is designed to simplify service composition, by assuming that it will manage individual data items, while the execution engine takes care of managing input collections. In this model, the termination condition for the loop is also implicit, i.e., the end of the input list. BPEL, on the other hand, requires tasks to be surrounded by a more general loop operator <*forEach*>, with an explicit termination condition associated to it. The two cases are shown in Fig. 4, where the left and right parts are Taverna and BPEL representation, respectively. In the left part, the output of *findProjects* is fed to *findClassesInProject*. Note that the process of adding the intermediate XML splitter is facilitated by the Taverna workflow design environment, so that users need not be aware of the specific structure of the XML documents, and no explicit programming is needed to customize the splitter.

In the right side of Fig. 4, we show how an explicit <*ForEach*> operator is added to a BPEL specification for the same task, and configured to iterate on the array of project names. Furthermore, after each invocation of *findClassesInProject*, result data need to be collected and merged into the final results set. Note that, unlike with Taverna, BPEL requires knowledge of the exact type of *findProjects*'s output at workflow design time.

Fig. 5 shows a more complex example that makes use of the cross-product feature discussed earlier. In this case, activity *assignInput* has two ports, one expects a single project (a string) and the other a single UML class. By configuring the iteration strategy to be a cross-product, this activity receives each combination of pairs <project, class> and feeds it to the *findSemanticMedatadaForClass* activity. The intermediate XML splitters take care of adapting the inputs and outputs to the structure required by each of the interfaces. Note in particular that *assignInput* constructs a new XML document, of the format required by the WSDL interface for *findSemanticMetadataForClass*, using the projects and class values extracted from the previous steps. Achieving the same cross-product effect in BPEL requires the use of two nested <*ForEach*> loops, as shown in Fig. 5(b).[2]

From these two examples one can see that BPEL handles the iteration in an imperative way, a <*ForEach*> construct and the iteration method (a counter, an array or an expression) is to be configured. It quickly becomes verbose and hard to understand and maintain, i.e., when a

[1]The function of *extractProjectArray* is to extract these elements from the XML document, and collate them into a list, which is then passed on to *findClassesInProject*. This is an example of an "XML splitter", a utility processor mentioned earlier. The use of this type of mediators is common in all cases where the complex XML documents produced by a Web services need to be taken apart, as the Taverna type system does not include the notion of structured record, or tuple.

[2]As an aside, note that feeding all possible <project, class> pairs to findSemanticMetadata… is not very efficient, as only valid pairs, i.e., of classes that are associated to the project, will result in associated metadata. A variant of the cross product operator has been recently added to Taverna to avoid this "blind" and rather inefficient computation. Its discussion, however, is beyond the scope of this paper.

cross product involves more than two lists and exposes too many implementation details to the end users (and thus error-prone). In contrast, the Taverna implicit iteration semantics does not require any configuration: the iteration is specified when an output containing a collection of items is connected to an input which consumes a single item of the same type. This leaves the complexity to the workflow engine

Although Taverna handles the most common iteration patterns more conveniently, BPEL as an imperative language offers more flexibility in modeling advanced iteration strategies, such as a combination of lists other than using dot/cross product, a completion condition in the "at least N out of M" form, and break/continue during iteration. For example, BPEL can handle the following problem: an activity receives two lists of inputs, needs a special kind of dot-product iteration over them, with a special "correlation" mechanism (like, classes and projects with the same developer should be combined.)

4.4 Data pipelining and support for data streams

The most recent release of the Taverna execution engine provides support for the pipelined execution of activities along a path in the workflow. This feature is strictly connected with the implicit iteration mechanism just discussed, and is aimed at improving efficient execution over large data collections. It is therefore interesting to analyze how a similar mechanism could be supported by a BPEL engine.

In Taverna, iterations over items in a list may be executed concurrently if sufficient resources are available to support parallel execution threads. When this happens, the elements of the result list may be produced in arbitrary order, due to different speeds of each of the threads. Eventually, the entire output list is complete, and in normal operations, the engine then activates each of the downstream activities that consume the list.

In many cases, however, the consumer activities may begin to process individual elements of the result without the need to wait for the entire list to be complete, i.e., when consumers are also subject to implicit iteration. This is the case in our earlier running example, where *findSemanticMedatadaForClass* processes one project at a time. In this case a new thread can be started as soon as one new project, say p_i, is produced by *findClassesInProject*. As a result, the computation of the set of classes C_i associated to p_i can be pushed down along the path without having to wait for slower threads in upstream activities to complete. In general, partial lists are systematically pushed along the arcs as soon their elements become available[3]. This all happens automatically in Taverna, without the need for explicit user directives.

This mechanism is exploited to support continuous input data, i.e., streams of discrete input tokens of unpredictable length. In this case, successful processing of a stream relies on the ability of the individual activities to compute on segments of the stream, without waiting for its completion. This is indeed a realistic requirement is commonly satisfied in applications such as process monitoring, or processing of sensor data, for example. Note that this does not prevent activities from blocking the computation, whenever they need to wait for more input to arrive, for example when sensor data processing requires a whole window of set size to be available.

In case of BPEL, since there is no implicit iteration, so a data item generated by one activity IS immediately forwarded to the subsequent one by default. As we have pointed out in the previous sub section, in BPEL users have fine control over behavior of iteration and this

[3]This mechanism is generalized to the case of lists that are trees of depth>1, by stipulating that an entire sub-tree can be forwarded to the next processor, as soon as all of its leaves have been computed.

control is also applicable to pipelined execution. In BPEL, pipelined execution can be achieved by nested <*forEach*> constructs. You can also control the degree of pipeline by changing the order of nested <*forEach*>.

See Fig. 6 to see the different levels of pipelined execution between four operations: *findProjects*, *findClassesInProject*, *findSemanticMetadataForClass*, and *searchDescLogicConcept*. To keep the figures simple we omit all <assign> activities after each <invoke> for result collection. In each case *findProjects* is first invoked and a list of Project is returned. In fully pipelined situation shown in Fig. 6 (a), two nested <forEach> (*ForEach Class* and *ForEach SemanticMetadata*) are within a <forEach> (*ForEach Project*). In this case, for each Project, a parallel execution branch is created and all the classes in this Project are obtained. Once a classes list pertaining to a Project is obtained, a new <forEach> (*ForEach Class*) is initiated without waiting for the classes of other projects to return. Similarly, once a SemanticMetadata list pertaining to a Class is obtained, a new <forEach> (*ForEach SemanticMetadata*) is initiated without synchronizing with other concurrent threads. Fig. 6 (b) differed from (a) in that the execution is pipelined until it reaches to the *searchDesclogicConcet* step, i.e., all the *SemanticMetadata* are items are collected and synchronized before *searchDesclogicConcet* starts. It is a partially pipelined execution. Fig. 6 (c) illustrates a non pipelined situation. The process is made up of three isolated <forEach> constructs and the iterations on each step have to be fully completed before the subsequent step of iterations begins.

In most circumstances the fully pipelined mode is the desired case. To achieve that with Taverna you simply connect all activities with necessary shims. With BPEL you need to add consecutively nested <forEach> constructs. Obviously Taverna offers a more compact representation in this case. However, when users want more control on the pipeline (partially of non pipelined cases), BPEL offers more flexibility. Again we have to point out that, pipelined execution is highly parallel and BPEL users should pay attention to concurrent data access to avoid undesired results.

In BPEL, at the beginning of <*forEach*>, a deterministic iteration range has to be specified. Therefore, it is not applicable to the case of streamlined execution in which data arrives at a non-deterministic manner. A more natural way to achieve streaming in BPEL is to initiate one process instance upon the arrival of a new piece of data. However one or more data collection services have to be deployed together with the BPEL workflow in this circumstance.

Fig. 7 and Fig. 8 show the completed Taverna and BPEL version for the caGrid workflow given in Fig. 1, respectively. In both Fig. 7 and 8 there are four activities (i.e., *findProject*, *findClassesInProject*, *findSemanticMetadataForClass*, and *searchDescLogicConcept*) that represent caGrid services. In Fig. 7 there are a few “shim” activities for data transformation between caGrid activities. In Fig. 8 there are some <assign> activities for data manipulation and <forEach> activities to achieve iteration and fully-pipelined execution (which is implicitly achieved in the Taverna version).

5 Analysis of Workflow Results

The final phase of the workflow lifecycle, namely results analysis, is increasingly perceived to be of great importance both within the e-science and the business processing management communities. The *provenance* of a piece of data produced by a process is a complete account of how that piece of data was computed, starting from user input and taking into account intermediate results produced by the activities involved in the computation. Prototype implementations of provenance management systems are available for both Taverna, with applications to scientific data, and BPEL, with applications to use cases in

business management [26]. The systems differ both in their purpose and architecture. The main reason for collecting scientific data provenance is to provide evidence in support of an experimental result, while the main use for provenance in BPEL is to check regulatory compliance [26], i.e., adherence to a set of business rules designed to ensure that certain steps in the process comply with laws and regulations.

The main technical difference is in the way provenance is collected and in the nature of the associated *provenance graphs*, i.e., the data dependency graphs computed from the observation of one workflow execution, are constructed.

On the one hand, dataflow models like Taverna have an advantage in that, intuitively, the provenance graph is just an unfolding of the workflow graph, itself a specification of data dependencies, obtained by replicating the nodes that correspond to multiple processor executions. In BPEL, on the other hand, the collection of provenance metadata involves the explicit definition of suitable control points, i.e., points in the execution that are relevant for the analysis of the business rules for which provenance analysis is designed. Furthermore, after collecting data at control points, it is still necessary to infer data dependencies from the execution order of the services involved in the workflow, a process known as *correlation* of the provenance data.

Both systems face a problem of *granularity*: the level of detail at which provenance can be traced depends on the units of information that the workflow engine can observe during execution. There are two aspects to granularity. Firstly, when dealing with Web Services, the atomic unit of information that flows through an activity is an XML document, for instance “purchase order” for a business process, or an XML-formatted description of a protein in the case of a scientific process. The black-box nature of the Web services that produce and consume these documents limits the ability to track its individual elements. For instance, consider a service that takes a purchase requisition request document as input, and returns a purchase order document. While it is likely that specific elements within the purchase order depend on only some of the input document elements, this fine-grained dependency is hidden within the service logic: from the point of view of provenance, the service is a black box, because the nature of the data transformation they implement is not exposed through the WSDL interface. Thus, the only data dependency that can be safely used in provenance tracking is that the entire purchase order depends on the entire purchase requisition request.

Of course, when Web services expose message types that are simple types, for example simple strings, then provenance can be traced at the level of these types. Business applications, however, are more likely to expose complex types that are part of articulated information exchanges. Thus, in general the black-box nature of the service limits the degree of precision with which provenance of the output can be tracked: the granularity of traceable provenance is that of entire XML documents, rather than that of their composing elements. An important example of this fine-grained data manipulation is the “packing” and “unpacking” of complex XML data, something that can be achieved automatically using XML splitters, as mentioned in Sec. 4.2 and 4.3. In some cases, this may enable provenance tracking through the internal element of XML documents, for instance it may be possible to trace the *originator* element of a purchase order back to some specific workflow input, at a stage in the process prior to its use as part of the order.

In our preliminary experiments on provenance tracking in Taverna, performed within the myGrid team, we have been able to achieve high precision in many practical cases, namely when simple values are composed into collections or into complex XML messages in a way that is visible to the engine, i.e., by means of dedicated packing and unpacking activities.

The second aspect where granularity issues emerge is in the management of collection data. Scientific data is quite naturally collection-oriented, as we have seen in our running example, where the association between a Project and a set of UML classes holds. In this case it is important to be able to track the provenance of each collection element individually as much as possible. Indeed, a collection-level statement such as "the collection of the concepts depends on the collection of input projects," is as trivially true as it is uninteresting. Instead, we must be able to determine that the presence of a specific concept in the output is due to a specific project being present in the original input. It is easy to see that (a) fine granularity can be preserved when implicit iteration over collections is involved, because in this case each element in the input collection is mapped exactly to one element in the output collection; and (b) conversely, activities that transform entire collections to new collections (i.e., without any iteration being exposed to the workflow engine) destroy the fine granularity property.

6 Conclusion and future work

Although the design and implementation of workflow systems for scientific purposes has been a subject of considerable research, there is a lack of in-depth analysis on the common data processing patterns in the scientific domain, and how well existing languages can support those patterns. We have attempted here to answer these questions based on our practical experience in modeling and orchestrating scientific workflows in caGrid. In this analysis, we consider the scientific workflow lifecycle, from service composition to workflow execution and result analysis.

From our experience in using both Taverna and BPEL as the candidate solutions for caGrid workflows, we have the following conclusions:

1. BPEL offers a comprehensive set of primitives to model workflows of all flavors (control-flow oriented, data-flow oriented, event-driven, etc), with full feature (process logic, data manipulation, event and message processing, fault handling, etc). As an imperative language, BPEL offers fine control and close communication to workflow engines. BPEL is capable of handling dataflows, although in some cases the modeling experience is cumbersome and tedious.
2. Taverna provides a compact set of primitives that eases the modeling of dataflows. Its functional programming framework makes many routine tasks invisible to the programmer so that users can focus on specifying "what to do" instead of "how to achieve it." The workflow engine handles routine tasks such as variable initialization, parallel execution through implicit iteration, and pipelined execution.

We believe that no reasonable Taverna developer would argue against the superiority of BPEL for certain tasks, and vice versa. Correspondingly, we believe different tasks will result in demand for different types of workflows.

We also suggest a promising *multi-stage modeling approach* in adapting BPEL to scientific workflow, retaining its strength in orchestrating multi-party interactions and adding simplicity through a functional programming style. Such an approach will allow scientists to specify their algorithms in terms of a dataflow model, which is subsequently transformed into a standard BPEL model automatically through a macro-expansion procedure. Then, this standard BPEL can be orchestrated by a BPEL-compliant engine. Similar approaches have already been adopted by existing research efforts but not applied specifically to dataflow problems [11,27].

The analysis presented in this paper is based on our understanding of caGrid's requirements for a workflow language, but we believe the conclusion is also applicable to other areas in

data intensive and exploratory science. We hope that our work not only helps researchers choose a tool that meets their needs, but also provides some insight on how a workflow language can help with the business of science.

Acknowledgments

We thank Prof. David De Roure from the School of Electronics and Computer Science, University of Southampton, UK for his insightful comments. We also thank Mr. Stian Soiland-Reyes at University of Manchester for his great help in using and extending Taverna and Mr. Dinanath Sulakhe at University of Chicago for co-working on caGrid project. This project has been funded in part with Federal funds from the National Cancer Institute, National Institutes of Health, under Contract No. N01-CO-12400.

Reference

1. Erl, T. Service-oriented architecture: concepts, technology, and design. Upper Saddle River, NJ, USA: Prentice Hall PTR; 2005.
2. Foster I. Service-Oriented Science. Science. 2005; vol. 308:814–817. [PubMed: 15879208]
3. Taylor, IJ.; Deelman, E.; Gannon, DB.; Shields, M. Workflows for e-Science: Scientific Workflows for Grids. Springer-Verlag; 2007.
4. Buetow KH. Cyberinfrastructure: Empowering a "Third Way" in Biomedical Research. Science. 2005; vol. 308:821–824. [PubMed: 15879210]
5. Saltz J, Oster S, Hastings S, Langella S, Kurc T, Sanchez W, Kher M, Manisundaram A, Shanbhag K, Covitz P. caGrid: design and implementation of the core architecture of the cancer biomedical informatics grid. Bioinformatics. 2006; vol. 22:1910–1916. [PubMed: 16766552]
6. Foster I. Globus Toolkit Version 4: Software for Service-Oriented Systems. Journal of Computer Science and Technology. 2006; vol. 21:513–520.
7. c. I. W. W. Group. Tool Reviews. 2007
8. Oinn T, Greenwood M, Addis M, Alpdemir MN, Ferris J, Glover K, Goble C, Goderis A, Hull D, Marvin D, Li P, Lord P, Pocock MR, Senger M, Stevens R, Wipat A, Wroe C. Taverna: lessons in creating a workflow environment for the life sciences. Concurrency and Computation: Practice & Experience. 2006; vol. 18:1067–1100.
9. OASIS. Web Services Business Process Execution Language Version 2.0. 2007
10. Slominski, A. Adapting BPEL to Scientific Workflows. In: Taylor, IJ.; Deelman, E.; Gannon, DB.; Shields, M., editors. Workflows for E-science: Scientific Workflows for Grids. Springer Press; 2007. p. 212-230.
11. Wassermann, B.; Emmerich, W.; Butchart, B.; Cameron, N.; Chen, L.; Patel, J. Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling. In: Taylor, IJ.; Deelman, E.; Gannon, DB.; Shields, M., editors. Workflows for E-science: Scientific Workflows for Grids. Springer-Verlag: 2007. p. 428-449.
12. Ludäscher B, Altintas I, Berkley C, Higgins D, Jaeger-Frank E, Jones M, Lee E, Tao J, Zhao Y. Scientific Workflow Management and the Kepler System. Concurrency and Computation: Practice & Experience. 2005
13. Taylor, I.; Shields, M.; Wang, I.; Harrison, A. The Triana Workflow Environment: Architecture and Applications. In: Taylor, IJ.; Deelman, E.; Gannon, DB.; Shields, M., editors. Workflows for E-science: Scientific Workflows for Grids. Springer: Guildford; 2007. p. 300-319.
14. Fahringer T, Jugravu A, Pllana S, Prodan R, Seragiotto C, Truong L. ASKALON: a tool set for cluster and Grid computing. Concurrency and Computation: Practice & Experience. vol. 17; 2005:143–169.
15. Zhao, Y.; Hategan, M.; Clifford, B.; Foster, I.; von Laszewski, G.; Raicu, I.; Stef-Praun, T.; Wilde, M. Swift: Fast, Reliable, Loosely Coupled Parallel Computation; 2007 IEEE Congress on Services; 2007. p. 199-206.
16. Deelman E. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Scientific Programming. 2005; vol. 13:219–237.

17. Turi, D.; Missier, P.; Roure, DD.; Goble, C.; Oinn, T. Taverna Workflows: Syntax and Semantics. 3rd e-Science Conference; Bangalore, India. 2007.
18. Tan W, Foster I, Madduri R. Combining the Power of Taverna and caGrid: Scientific Workflows that Enable Web-Scale Collaboration. IEEE Internet Computing. 2008; vol. 12:61–68.
19. Leymann F. WSFL -- Web Services Flow Language. IBM Software Group, Whitepaper. 2001
20. Thatte S. XLANG: Web services for business process design. 2001
21. Shields, M. Control- Versus Data-Driven Workflows. In: Taylor, IJ.; Deelman, E.; Gannon, DB.; Shields, M., editors. Workflows for E-science: Scientific Workflows for Grids. Springer-Verlag; 2007. p. 167-173.
22. Russell, N.; Hofstede, AHMt; Edmond, D.; Aalst, WMPvd. QUT Technical report, FIT-TR-2004-01. Brisbane: Queensland University of Technology; 2004. Workflow Data Patterns.
23. Darema, F. The SPMD Model: Past, Present and Future; Proc. of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface; 2001.
24. Foster, I. Designing and building parallel programs: concepts and tools for parallel software engineering. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 1995.
25. Hidders, J.; Sroka, J. Towards a Calculus for Collection-Oriented Scientific Workflows with Side Effects; Proc. 16th International Conference on Cooperative Information Systems; 2008. p. 374-391.
26. Curbera, F.; Doganata, Y.; Martens, A.; Mukhi, NK.; Slominski, A. Business Provenance---A Technology to Increase Traceability of End-to-End Operations. OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems; 2008. p. 100-119.
27. Tan, W.; Fong, L.; Bobroff, N. BPEL4Job: A Fault-handling Design for Job Flow Management. International Conference on Service Oriented Computing (ICSOC); Vienna, Austria. 2007.

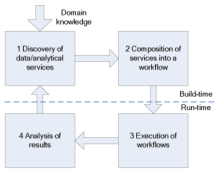

Fig. 1.
Lifecycle of a scientific workflow

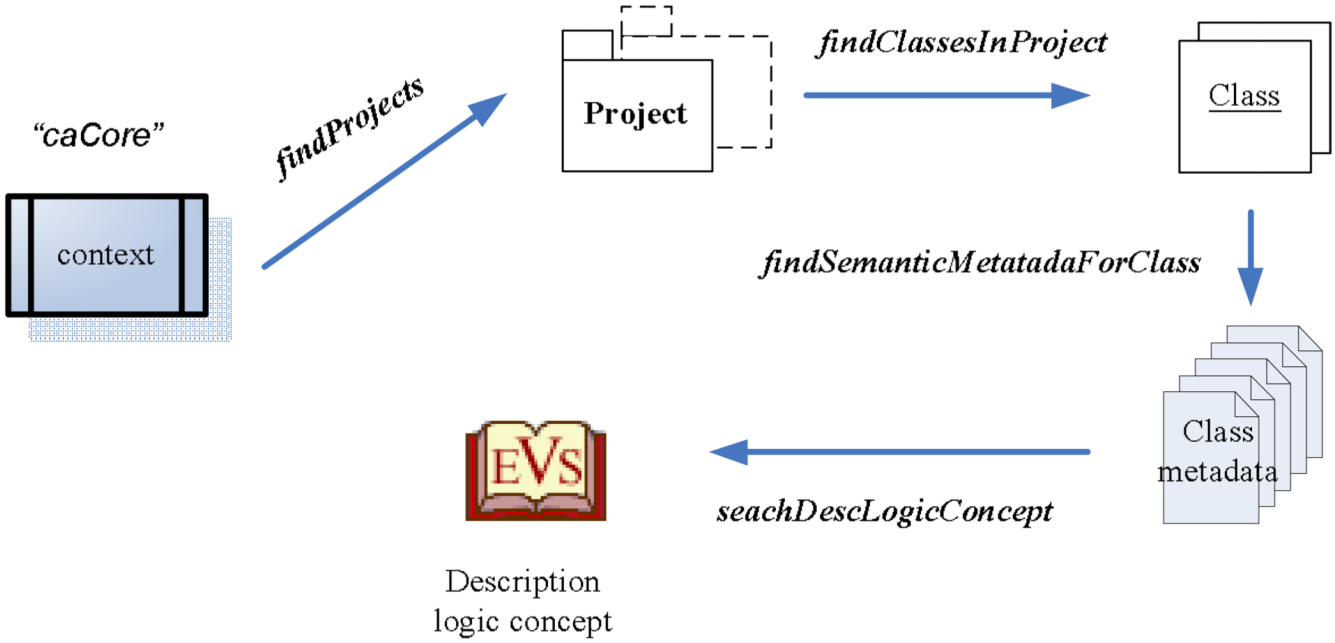

Fig. 2.
The caGrid CaDSR workflow

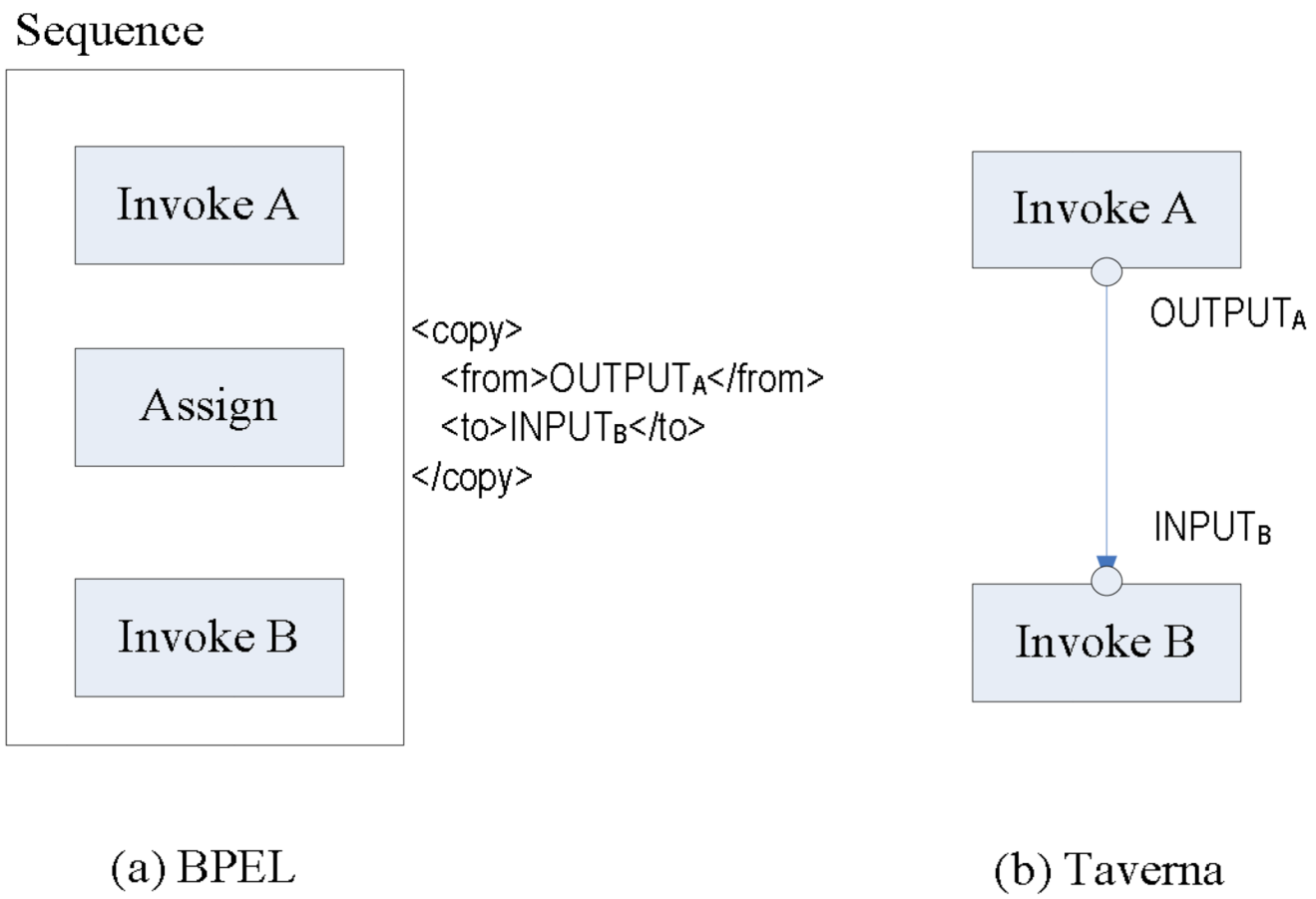

Fig. 3.
Comparison of data passing patterns of BPEL and Taverna

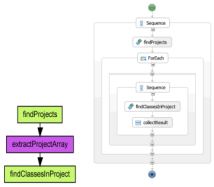

Fig. 4.
Implicit iteration: case 1

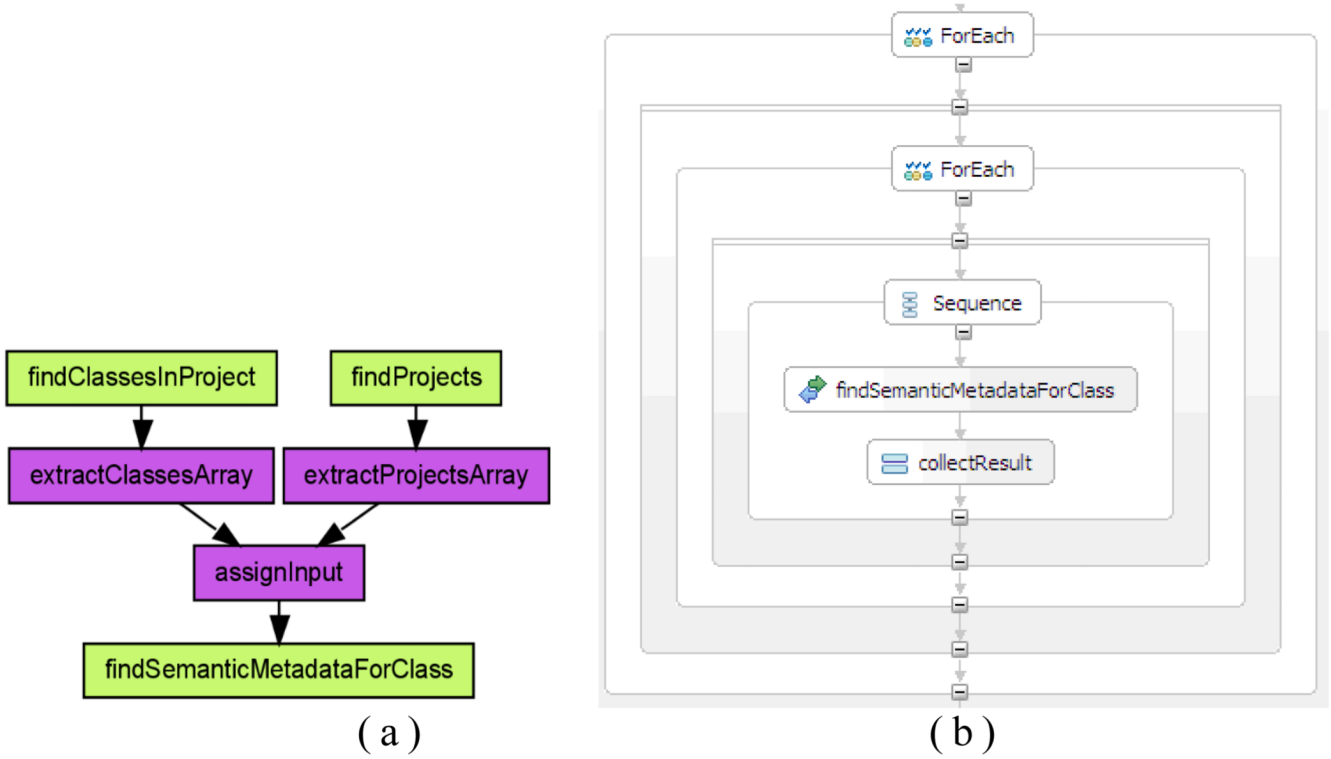

Fig. 5.
Implicit iteration: case 2

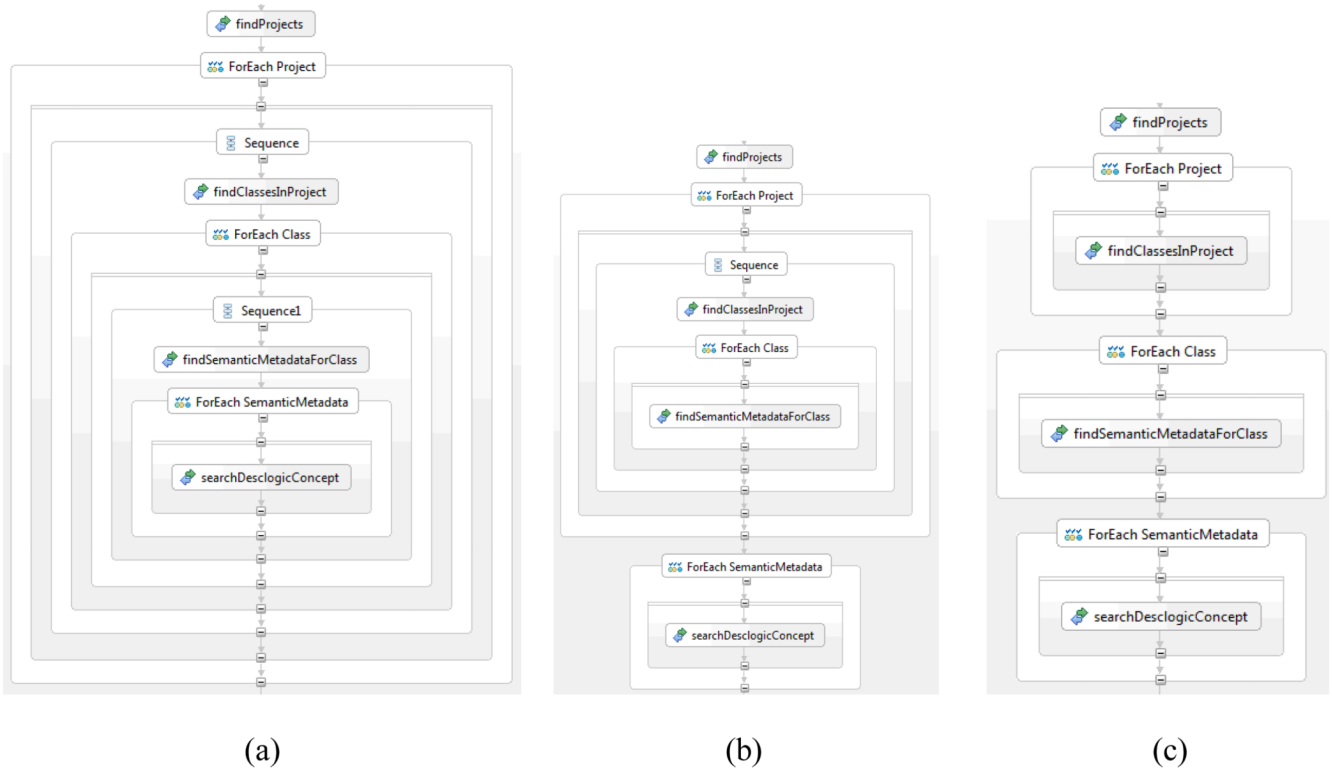

Fig. 6.
Different degrees of pipelined execution in BPEL (a) fully-pipelined (b) partially-pipelined and (c) non-pipelined

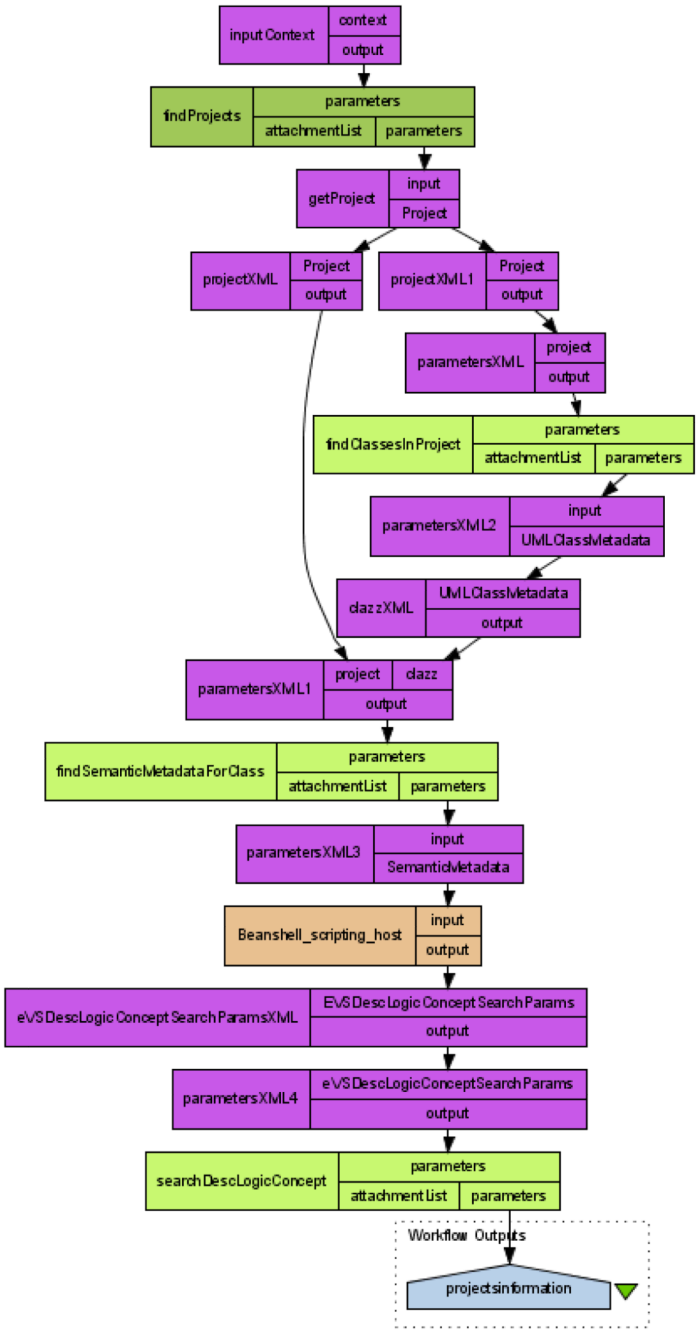

Fig. 7.
Taverna version for the caGrid workflow in Fig. 1

Fig. 8.
BPEL version for the caGrid workflow in Fig. 1

Table 1

Comparison of BPEL and Taverna w.r.t. modeling style

	BPEL	**Taverna**
Driving factor in design	1) Interoperability of business processes; 2) Interaction of multiple partners in a given sequence; 3) Recoverability	1) Extensible invocation scheme 2) Straight through processing of data; 3) Reproducibility;
Activities in model	Basic and structure activities	Activities as data processing units with in/output ports
Semantics of links	Transfer of control	Transfer of data
Data definition	Explicitly defined (global variables)	Implicit defined (activities' input/output)
Data initialization	Complex data type need to be explicitly initialized	Implicit
Control logic	Full-fledged: sequence, conditional, parallel, event-triggered, etc	Limited: sequential, parallel and conditional
Parallel execution	Explicitly defined using <flow> or <ForEach> operators	Implicit
Fault handling	Throw-catch, compensation handler	Retry and substitution