A Framework for Advanced Modularization and Data Flow in Workflow Systems

Niels Joncheere, Dirk Deridder, Ragnhild Van Der Straeten,
and Viviane Jonckers

System and Software Engineering Lab (SSEL)
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
{njonchee,dderidde,rvdstrae,vejoncke}@vub.ac.be

Abstract. Workflows have become a popular technique for describing processes in many different application domains, including Computer Aided Engineering (CAE). State-of-the-art workflow languages lack the necessary modularization techniques and data flow capabilities to express processes in a way that facilitates their design, evolution and reuse. In this paper, we aim to tackle this problem by presenting a conceptual framework for advanced modularization and data flow in workflow systems, which is independent of specific modeling approaches and technologies.

1 Introduction

Workflows have long been a popular technique for describing processes in a number of application domains, such as business process management and web service orchestration. More recently, they have started to be applied in scientific computing and Computer Aided Engineering (CAE). Workflow languages for each of these application domains have been developed.

As processes become more complex, mechanisms are needed to manage this complexity in order to facilitate the processes' design, evolution and reuse. Traditionally, workflow languages tackle this problem by allowing to decompose workflows into separate modules such as sub-workflows. However, these modules are often strongly tied to the workflow in which they are used, resulting in limited reusability. In addition, most approaches do not support modularizing concerns that crosscut a workflow (such as authentication, transaction management, and logging), and approaches that do [1–3] are limited in their expressiveness.

In scientific computing and CAE, the volume, complexity, and heterogeneity of data in processes is much larger than in other application domains. This means that workflows in these domains contain much more data flow: they deal with data transfer from persistent storage to the resources that will process the data, partition data in preparation of parallel processing, and handle transformation of data when different processing steps require different data formats. This *data perspective* [4] is insufficiently supported by current workflow approaches.

The goal of our approach is to improve *separation of concerns* [5] in workflow languages by tackling both the lack of modularization of the main concern and the lack of modularization of *crosscutting concerns* [6] using a single, general workflow construct. More specifically, we revalue the sub-workflow construct as a powerful means for workflow modularization. Based on our collaboration with industrial partners in Computer Aided Engineering, we also aim to provide better support for the data perspective.

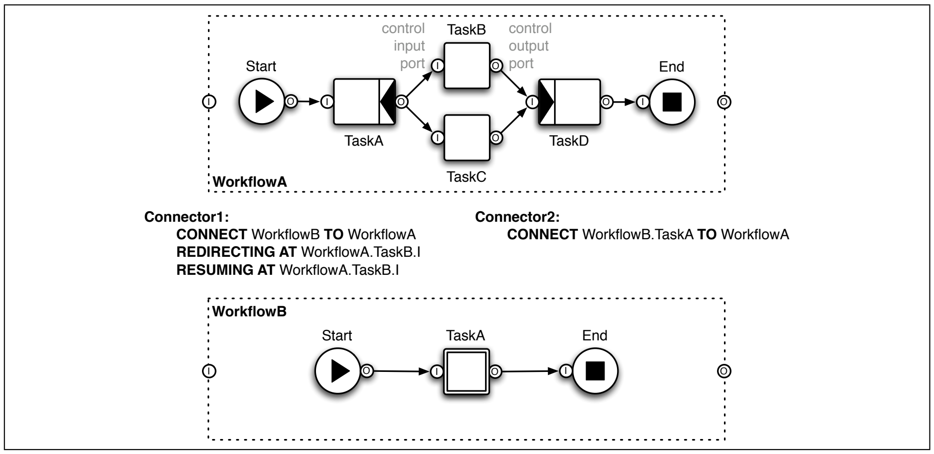
This paper presents our conceptual framework for advanced modularization and data flow in workflow systems. Section 2 describes our modularization mechanism, and Section 3 describes our data flow mechanism. Section 4 presents related work, and Section 5 states our conclusions.

2 Modularization Mechanism

Traditionally, sub-workflows are used to decompose the main concern of a workflow into smaller modules, thus facilitating evolution and reuse of these modules. The main workflow contains a *composite task* that specifies — at design time — which sub-workflow should be invoked. When the workflow is enacted, the sub-workflow will be executed. Although this mechanism is a good means for managing workflow complexity, it is not always present in popular workflow languages such as BPEL [7].

While our approach supports this basic mechanism, it improves on it by allowing sub-workflows to be attached to main workflows in a way which inverts the flow of control: it allows specifying at which points in a main workflow a sub-workflow should commence and cease execution, without explicitly specifying this in the main workflow. Thus, we facilitate adding concerns that were not considered when the main workflow was designed, and facilitate comprehending, maintaining, reusing, and removing concerns that are specified using such sub-workflows. For example, this mechanism can be used to invoke an authentication sub-workflow before each invocation of a certain service, even though the workflow that invokes this service was not designed with authentication in mind. This inversion of control is similar to traditional aspect-oriented techniques [8], but unlike aspects, sub-workflows are not separate language constructs introduced solely for the sake of encapsulating crosscutting concerns. Such a *symmetrical* [9, 10] approach reduces the number of language constructs, and is expected to facilitate the adoption of our approach in industrial environments.

In order to allow specifying such symmetrical workflow compositions, we introduce the notion of **control ports**. Control ports are the entry and exit points of a task's control flow; each task has exactly one *control input port* and one *control output port*. A workflow's control flow perspective is specified by connecting the control output port of the workflow's start event to the control input port of a task, and connecting the control output ports of all tasks to the control input ports of other tasks or end events. Just like tasks, workflows have exactly one control input and output port; these are the entry and exit points of the workflows' control flow. We visualize control ports by extending the YAWL [11] notation with small circles at the sides of tasks and workflows. Control input ports are marked with the letter I, while control output ports are marked with the letter O. The upper part of Figure 1 shows a workflow called `WorkflowA`, which has a control input port (at its left side) and a control output port (at its right side). Additionally, all elements of the workflow, such as `TaskB`, have control ports as well, which are connected in order to specify the workflow's control flow perspective.



```
Connector1:
    CONNECT WorkflowB TO WorkflowA
    REDIRECTING AT WorkflowA.TaskB.I
    RESUMING AT WorkflowA.TaskB.I

Connector2:
    CONNECT WorkflowB.TaskA TO WorkflowA
```

Fig. 1. Using a connector for symmetrical workflow composition (left) and for connecting a task to a workflow (right)

Analogous to the way they are employed in component based software engineering [12] and aspect-oriented programming [10], we introduce **connectors** to connect workflows to each other. In Figure 1, `Connector1` specifies that `WorkflowB` needs to be connected to `WorkflowA`. The connector also specifies that, when `WorkflowA` is enacted, its control flow should be *redirected* to `WorkflowB` when it reaches the control input port of `TaskB`. If control flow should be *split*, one should use the SPLITTING keyword instead of the REDIRECTING keyword. Additionally, the connector specifies that, when the execution of `WorkflowB` has finished, control flow should resume at the control input port of `TaskB`. The net result of this connector is that `WorkflowB` will be executed *before* `TaskA`.

In our example, control flow is redirected and resumed at the same control port in `WorkflowA`. However, this need not be the case. For example, if the connector would specify that `WorkflowB` should resume at the control output port of `TaskB`, the net result of the connector would be that `WorkflowB` is executed *instead of* `TaskB`. In an initial phase, our system will support all the workflows' control ports as resuming points in order to maximize connector expressivity. However, some resuming points may yield undesirable workflow behavior (such as infinite loops). Therefore, future work will be directed at producing safe connector patterns.

The traditional composite task construct is still available in our approach, but we do not require the composite task to be hard-wired to a concrete workflow at design time: a connector can be used to wire the composite task to a concrete workflow. This reduces the coupling between a workflow that contains a composite task and the concrete workflows that will implement this composite task. In Figure 1, `Connector2` specifies that `TaskA` in `WorkflowB` should be connected to `WorkflowA`. When `WorkflowB`'s control flow reaches `TaskA`, `WorkflowA` will be executed, and when its execution is finished, the control flow will continue with the remainder of `WorkflowB`. In fact, such a connection can be made even if `TaskA` is a regular task instead of a composite task. In that case, `WorkflowA` would be executed instead of `TaskA`.

Connectors are specified separate from the workflows they connect. This reduces the coupling between sub-workflows and main workflows, and improves the reusability of sub-workflows by making them independent of the context to which they might be applied. This also means that a workflow can assume the role of sub-workflow in one composition, while assuming the role of main workflow in another. Figure 1 illustrates this: in `Connector1`, `WorkflowA` assumes the role of main workflow, and in `Connector2`, `WorkflowB` assumes the role of main workflow. Of course, it would not make sense to have both connectors in the same workflow composition, as this would yield an infinite loop.

3 Data Perspective

Most state-of-the-art workflow languages are not designed with the data perspective in mind. Data is typically accessible by groups of tasks using some basic scoping mechanism, or is simply passed along with the control flow. The data and control flow perspectives are tightly integrated, and their independence is concealed. The concepts modeled in these perspectives, as well as their independence get blurred and distorted. Existing workflow research [13] has recognized the need to uphold this independence. Therefore, we improve on existing languages by specifying data flow and control flow separately.

A first concept we introduce to this end is **data ports**. Each task can have an arbitrary number of *data input ports* and *data output ports*, which represent the input and output parameters of a task, respectively. Each of a workflow's ports has a unique name, and specifies the types of data transfer that it supports, which are either pass-by-value, pass-by-reference, or streaming. Depending on the application domain, a data port can specify the type of its data (for example using XML Schema). Analogous to tasks, workflows have an arbitrary number of data input and output ports as well.

Secondly, we introduce a first-class **data flow** construct, which is visualized as a special kind of arrow that connects the data output port of one task to the data input port of another. The basic case of the construct specifies *data transfer* between two tasks. The data flow specifies the type of transfer that will actually be used (pass-by-value, pass-by-reference, or streaming — which should be compatible with the connected data ports), and depending on this type, specifies where data should be stored intermediately. For example, the data flow can specify that a data output port's data should be passed by reference to a certain data input port, while storing the actual data in a certain database. A more advanced case of the data flow construct specifies *data transformation*: instead of simply transferring data, the user can specify the way in which data should be transformed during its transfer. Using these two cases of our first-class data flow construct, one can express all data transfer patterns defined in [4].

The upper part of Figure 2 provides an example using a workflow named `WorkflowA`. Data ports are visualized by extending the YAWL notation with small rectangles at the sides of tasks and workflows. Data input ports are marked with the letter I, while data output ports are marked with the letter O. `WorkflowA` contains a data transformation named `Transformation` between the data output port of `TaskA` and one of the data input ports of `TaskC`, and an anonymous data transfer between the data output port of `TaskB` and the other data input port of `TaskC`. The data flows are visualized by extending the YAWL notation with thick arrows; the transformation is differentiated from the transfer by the square in the middle of its arrow.

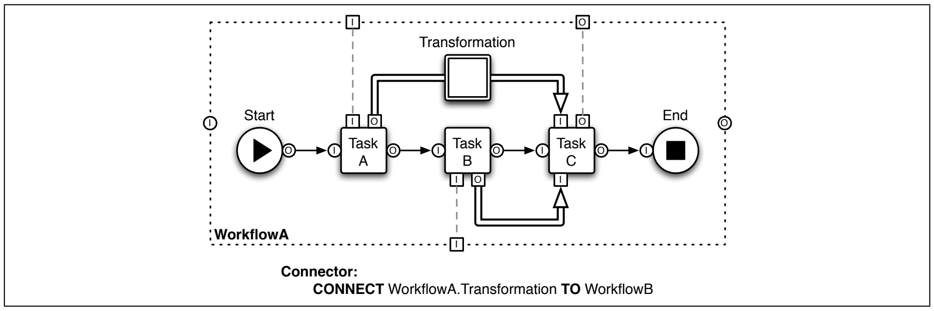


Fig. 2. Data transfer and transformation

By default, the workflow's tasks' unconnected data ports are implicitly exposed as the workflow's data ports. This is shown in the example by the dashed lines, which are not part of the notation. If a more advanced mapping between the workflow's tasks' data ports and the workflow's data ports is desired, it can be specified in the workflow, but this is not visualized using the notation.

The data transformation construct greatly simplifies CAE workflows, as these typically contain a large amount of data transformation logic. Depending on the application domain, a specific engine for our language might support a number of built-in transformation strategies, such as XSLT transformations for XML data, but in general, the user can specify data transformation strategies by using the workflow language itself: because data transformations are first-class, data transformations can be composites, and can be connected to a workflow using a connector. Figure 2 illustrates this scenario by showing a connector that links `Transformation` to a workflow named `WorkflowB`, which is not shown. `WorkflowB` can be any workflow that has exactly one data input port and one data output port. When `Transformation` is executed, its incoming data will be sent to `WorkflowB`'s data input port, and its outgoing data will be retrieved from `WorkflowB`'s data output port.

4 Related Work

The Abstract Grid Workflow Language (AGWL) [14] is an interesting approach which also identifies the need for an powerful, separate data perspective. However, the approach does not address the requirement of separation of concerns.

Data flow languages [15] have long been available as a paradigm for expressing computations according to its data perspective. However, existing workflow research [13, 14] has shown that having only a single workflow perspective is insufficient. Nevertheless, the data perspective of our approach can be seen as a coarse grained data flow language where tasks are the macro actors.

In the modeling community, UML activity diagrams [16] have been developed as a means of expressing workflows. Data flow can be modeled separate from control flow using pins, and there is a distinction between streaming and non-streaming data transfer. However, the approach does not consider separation of concerns or the specific needs of data-intensive applications.

5 Conclusions

In this paper, we propose a conceptual framework for advanced modularization and data flow in workflow systems. We describe a workflow language which introduces four language elements: control ports, data ports, data flow, and connectors. A workflow's data flow is specified separate from its control flow by connecting tasks' data ports using a first-class data flow construct. Connectors allow expressing that a task in one workflow should be executed by another workflow, in a way that minimizes dependencies between these workflows and thus facilitates their independent evolution and reuse. Additionally, connectors allow connecting data flow constructs to workflows. The inversion-of-control connector allows augmenting a workflow with concerns that were not considered when it was designed, and again facilitates independent evolution and reuse of these workflows.

Many of the concepts we introduce are not present in current workflow approaches. In particular, our inversion-of-control mechanism is significantly more powerful than existing aspect-oriented approaches for workflows [1–3]. Future work will be directed at providing a proof-of-concept implementation for our approach.

Acknowledgements

The research presented in this paper is funded by the Research Foundation – Flanders through the DyBroWS project.

References

1. Charfi, A., Mezini, M.: Aspect-oriented web service composition with AO4BPEL. Lecture Notes in Computer Science **3250** (2004) 168–182
2. Courbis, C., Finkelstein, A.: Towards aspect weaving applications. In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005). St. Louis, MO, USA, ACM Press (2005)
3. Braem, M., Verlaenen, K., Joncheere, N., Vanderperren, W., Van Der Straeten, R., Truyen, E., Joosen, W., Jonckers, V.: Isolating process-level concerns using Padus. Lecture Notes in Computer Science **4102** (2006) 113–128
4. Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow data patterns. QUT Technical Report FIT-TR-2004-01, Queensland University of Technology, Brisbane, Australia (2004)
5. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM **15**(12) (1972) 1053–1058
6. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. Lecture Notes in Computer Science **1241** (1997) 220–242
7. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services: Version 1.1. `http://www.ibm.com/developerworks/library/specification/ws-bpel/` (2003)
8. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. Lecture Notes in Computer Science **2072** (2001) 327–354
9. Tarr, P., Ossher, H., Harrison, W., Stanley M. Sutton, J.: N degrees of separation: Multi-dimensional separation of concerns. In: Proceedings of the 21st International Conference on Software Engineering (ICSE 1999). Los Angeles, CA, USA, IEEE Computer Society (1999) 107–119
10. Suvée, D., De Fraine, B., Vanderperren, W.: A symmetric and unified approach towards combining aspect-oriented and component-based software development. Lecture Notes in Computer Science **4063** (2006) 114–122
11. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language. Information Systems **30**(4) (2005) 245–275
12. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Upper Saddle River, NJ, USA (1996)
13. Axenath, B., Kindler, E., Rubin, V.: AMFIBIA: A meta-model for the integration of business process modelling aspects. International Journal of Business Process Integration and Management **2**(2) (2007) 120–131
14. Fahringer, T., Pllana, S., Villazon, A.: AGWL: Abstract Grid Workflow Language. Lecture Notes in Computer Science **3038** (2004) 42–49
15. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Computing Surveys **36**(1) (2004) 1–34
16. Object Management Group: UML superstructure, version 2.1.2. `http://www.omg.org/spec/UML/2.1.2/` (2007)