An Environment for Modeling Workflow Components

Colin Atkinson and Dietmar Stoll

Lehrstuhl für Softwaretechnik
University of Mannheim
68131 Mannheim
{atkinson, stoll}@informatik.uni-mannheim.de

Abstract: An important goal of workflow engines is to simplify the way in which the interaction of workflows and software components (or services) is described and implemented. The vision of the AristaFlow project is to support a "plug and play" approach in which workflow designers can describe interactions with components simply by "dragging" them from a repository and "dropping" them into appropriate points of a new workflow. However, to support such an approach in a practical and dependable way it is necessary to have semantically rich descriptions of components (or services) which can be used to perform automated compatibility checks and can be easily understood by human workflow designers. This, in turn, requires a modeling environment which supports multiple views on components and allows these to be easily generated and navigated around. In this paper we describe the Integrated Development Environment (IDE) developed in the AristaFlow project to support these requirements. After outlining the characteristics of the "plug and play" workflow development model, the paper describes one of the main innovations within the IDE –the multi-dimensional navigation over views.

1 Introduction

An important goal of workflow engines is to simplify the way in which the interaction of processes and software components (or services) is described and implemented [DR04, Ac04]. The AristaFlow project's vision of how to achieve this is based on the "plug and play" notion popularized on the desktop, in which workflow designers can describe interactions with components simply by "dragging" them from a repository and "dropping" them into the desired points of a new workflow [Da05]. However, the ability to define new workflows in such a simple and straightforward way is only advantageous if there is a high likelihood that the resulting processes are well-formed, correct and reliable. In other words, to make the "plug and play" metaphor work in practical workflow scenarios it is essential that components are used in the "correct way", and the possibility for run-time errors is significantly reduced at design time. In short, there should be few if any "surprises" at run-time. If workflows defined by the "plug and play" metaphor are highly unreliable or unpredictable this approach will not be used in practice.

In order to support this goal, components must be described in a way that –

1. has well defined semantics so that their properties are machine-readable and can be used to automatically check workflow-component compatibility, correctness and reliability.

2. is easy for humans to understand, so that workflow designers can easily comprehend components' properties and decide which components to use where and in what way.

Only component description approaches that fulfill both of these requirements provide the required foundation for the "plug and play" development and adaptation of workflows. In addition, of course, workflow components must be developed using the best available practices and subject to rigorous validation and quality assurance activities (e.g. inspection and testing). So called "Semantic" approaches for describing components/services, such as OWL-S [Ow04] or WSMO [Ws05] score highly on the first requirement since they utilize a description logic based language such as OWL to describe component semantics in a rigorous and machine-accessible way. However, since they are optimized for reasoning efficiency rather than human readability they are difficult to use.

Model-based representations of components based on languages such as the UML score much more highly on the second requirement, but score less well on the first requirement. This is because the semantics of some of the UML diagrams is somewhat vague, and it is unclear what combination of diagrams should be used to fully document a component and what information each diagram should contain. Indeed, the views supported by the current set of UML diagrams do not allow all the necessary information to be described and/or do not present it in an appropriate way. Moreover, there are no predefined relationships between the UML diagram types, so there is no built-in way of determining whether different views of a component are consistent with one another.

Nevertheless, modeling languages such as the UML provide a much more suitable foundation for describing workflow components in a way that supports the plug and play paradigm than OWL based approaches. By using OCL to tighten the semantics of models and adding additional view types optimized for workflows it is possible to overcome these problems and attain a component/service representation approach which fulfills both criteria outline above. However, to make this viable in practice it is necessary to define suitable consistency rules between views and provide a pragmatic metaphor for creating and navigating around them. In addition, the approach must be integrated within a practical software engineering environment that allows components to be designed, implemented and tested using traditional development techniques.

This paper describes the approach to component modeling and development within the AristaFlow project and the integrated development environment (IDE) created to support it. Although these are optimized for the description of workflow components (e.g. by views and editors especially tailored to the requirements of workflow developers and administrators), they are useful for general component modeling as well. In the next chapter we describe the overall life-cycle of components, and describe how they fit into

the workflow definition and execution process. The following two sections then describe the main innovations in the AristaFlow approach. Chapter 3 describes the AristaFlow IDE's strategy for integrating the various kinds of diagrams types and view types needed to fully describe workflow components and for ensuring that they stay consistent. Chapter 4 describes the IDE's innovative strategy for organizing the different views and supporting navigation around them. Chapter 5 then presents some implementation details of the IDE, and chapter 6 concludes with some final remarks.

2 Component-Oriented Development of Workflows

The AristaFlow project aims to cover the whole lifecycle of components from their initial development to their use in workflow management systems (Figure 1). In this lifecycle there are three main human roles: the component developer, the workflow administrator and the workflow developer.

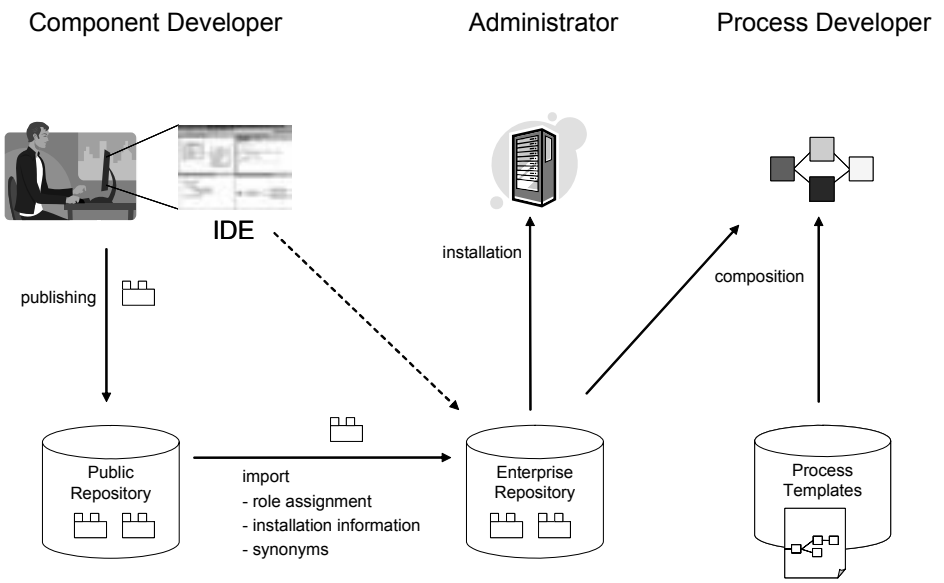

Figure 1: Workflow Component Lifecycle

The component developer models and implements components and then publishes them in a public repository. A workflow administrator of an enterprise then browses the repository or searches it by using various criteria as defined in the component description. Once they have been found, suitable components can be imported into a private enterprise repository – the so called deployment store – where they are installed and ready to be executed. The administrator can then add information like role assignments and deployment information. In addition, taxonomical information can be adjusted or added, e.g. synonyms and (enterprise wide) unique identifiers for parameters,

and these can later be automatically "wired" to data elements by the workflow management system.

A process developer then uses a process template editor to combine components and process templates into executable processes. Figure 2 shows a screenshot of the ADEPT2 editor developed in the AristaFlow project [Ar07]. In this example, an application function ("Amazon Item Search") has been chosen from the activity repository and dragged into the process graph. As the input parameter is not yet assigned to another node in the process graph, the problem window shows an error message. A process template can only be released to the ADEPT2 runtime system (for later instantiation) if it contains no errors. Similar checks take place at runtime, when ad-hoc changes or process schema evolutions are made [RD03, Ri04]. The system only allows changes if they do not lead to inconsistencies. An extensive description of the issues and requirements involved in modern workflow process modeling tools and workflow management systems such as ADEPT2 is given in [Da05].

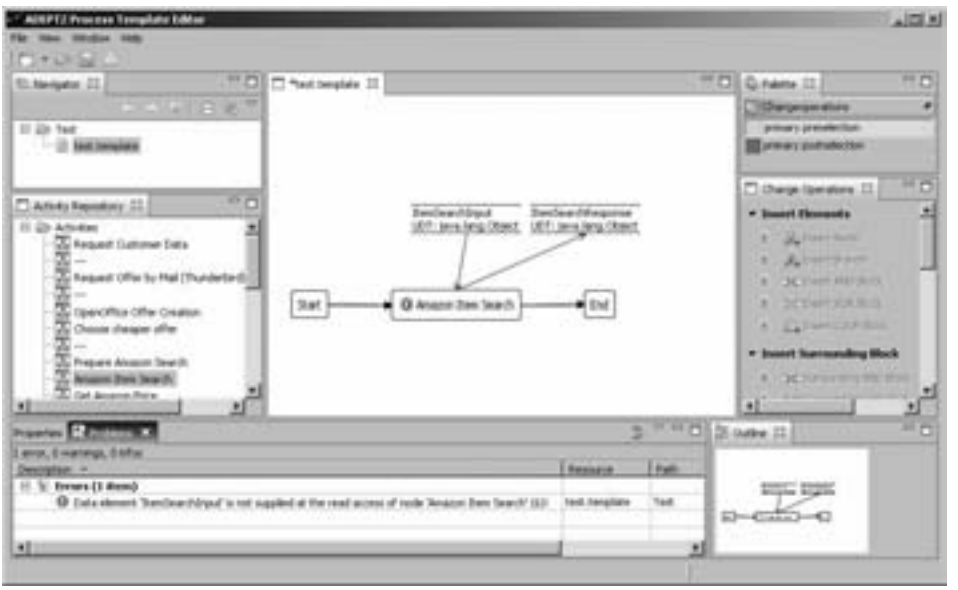

Figure 2: AristaFlow Workflow Editor

2.1 Integrated Development Environment

Since components/services are software applications in their own right, a component development IDE needs to support the full range of development activities including code development, testing and debugging. However, when building a component IDE it is clearly undesirable and impractical to redesign or redevelop the rich range of capabilities that modern IDEs provide. The AristaFlow IDE is therefore built upon an existing, well known and extensible development environment - namely the Eclipse environment.

Taking this goal into account, the component modeling approach and IDE were developed to fulfill the following main requirements –

1. to fit seamlessly and with minimum impact on top of the Eclipse environment, giving developers access to the full range of native Eclipse functionality and existing Eclipse plug-ins,

2. to provide a concise, well-defined and human friendly representation of components which workflow designers can easily understand and use to select and employ components in their workflows,

3. to provide suitable, machine readable descriptions of component properties which automated checkers (e.g. the workflow editor, the workflow execution engine) can use to verify the suitable use of components within workflows, and

4. to support the packaging (exporting) of components in a way that can easily be imported by the repository and the workflow tools.

Given requirements (1) and (3), it makes sense for the AristaFlow IDE to exploit as much of the information in regular software development artifacts as possible and translate it automatically into formats that can be understood by workflow management systems. For example, activity templates, which are required for process modeling, can to a large extent be automatically generated from source code. If the parameters of operations of different vendors are related (e.g. by two parameters that represent an account number), a mapping between parameters and unique identifiers is defined to enable the automatic association of data elements to parameters. It is also possible to associate a component or single operation with one or more taxonomies. The taxonomy editor of the IDE allows custom taxonomies to be imported or created.

When a component developer specifies the behaviour of a component, for example, as a state chart whose transitions correspond to operation calls or as regular expressions, protocol checkers can be employed. These make sure that a process modeler can plug operations into a workflow schema only in a way that obeys the constraints specified by the component modeler. Similarly, constraints (i.e. invariants and pre- and post conditions on operations) can be used for runtime checks, for example, by arranging for the workflow execution engine to check preconditions before an operation is actually invoked. The code for checking the preconditions can be generated by the IDE and delivered with the component. This makes it possible to catch violated preconditions that could lead to expensive and difficult-to-trace runtime errors and to generate a response understandable to a workflow administrator.

The more information that is available to the execution environment, the more checks can be made. Many of the checks are done at modeling time on the process schemata, ensuring incorrect schemata are not allowed to be executed. In cases where modeling time checks are impossible or too complex, checks at runtime are applied. These can prevent unwanted behaviour of components, for example, by checking whether the preconditions of an operation have been fulfilled by the component developer before it is called.

3 View-Based Component Modeling

Although the basic idea of capturing software artifacts from numerous inter-related viewpoints has been around for some time (e.g. [NKF03]), there are no widely used tools that provide clean and inherent support for this approach. With the success of the agile development movement most applications are still developed using source code as the single view of software objects, and although the use of multi-view notations such as the UML has grown in popularity, the selection of which views to use and how views should be related is invariably left to the user. In particular, there are no widely used component modeling tools that provide users with the flexibility to define new view types and generate/access new view instances on demand while at the same time systematically enforcing and checking consistency rules.

The problem is that during the development of a component, users need to generate and work on all kinds of views ranging from UML diagrams to code fragments, and as the number of views increases, navigation becomes more tedious and maintaining consistency between them becomes increasingly difficult. This is particular so when the relationships and consistency rules between different types of views are defined and checked on a pairwise basis as is usually the case today.

An ideal solution to this problem would be for every view of the IDE to be generated from, and to work on, a single underlying model and for changes made to individual views to be synchronized directly with this model [AS07]. In this way, the consistency between the editors and the model is automatically ensured, as long as each individual change to a view is checked for validity against this model. This "on demand" generation of views is schematically depicted in Figure 3. The single underlying model should be an instance of a metamodel which contains the minimum set of concepts necessary to store the required information.

However, building the whole IDE in this way, although possible in the long term, is incompatible with requirement (1) in the short term. Thus, in the AristaFlow project a hybrid solution was developed in which several underlying model formats coexist. In the long term these will be merged into one representation and all views will be generated (by model transformation) from this single representation.

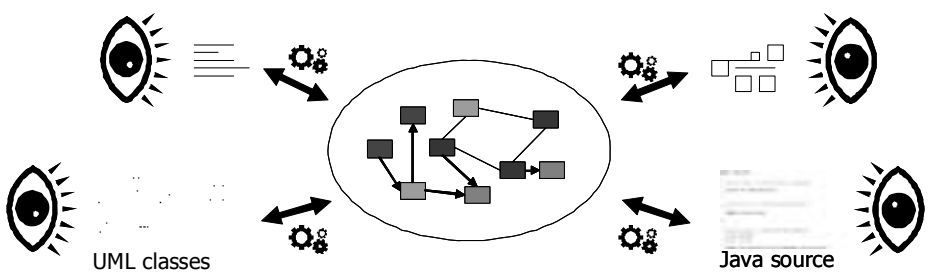

Figure 3: On demand View Generation

3.1 View Types

Although advanced users need to have the ability to define their own view types and to describe how they relate to the existing views, most users will be content with using the existing view types defined by the recommended modeling approach. A key question, therefore, is what set of basic views users should employ to describe components. AristaFlow's basic approach to component modeling is based on the KobrA approach [At02], which defines a systematic approach to the UML based representation of components. This is organized around the notion of **projections**, which define the kind of information conveyed in a view, and **abstraction levels**, which define the level of platform independence represented by a view and whether the information conveyed is black box or white box.

There are currently three levels of abstraction supported in the IDE: specification, realization and implementation. The most abstract level is the specification which provides a black box view of the component. It describes all externally visible properties of a component and thus serves as its requirements specification. The realization of a component describes the design of the component and provides a white box view of its internal algorithms and subcomponents. Source code and test cases are the most platform specific representation, and capture the implementation of the component using the chosen programming language.

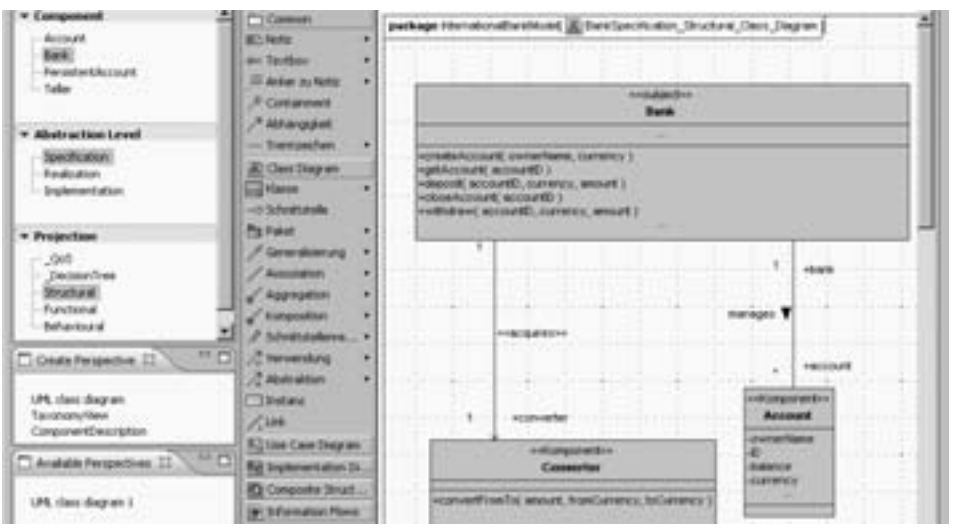

Figure 4: Structural view of a Component Specification

KobrA also defines three fundamental projections: the structural, functional and behavioural projection [At02]. The structural projection includes classes and associations manipulated by the component as well as other structural information like taxonomical information and source code. Operations of a component and their interaction with other artifacts are modeled in the functional projection, e.g. by means of operation specifications and UML interaction diagrams. Finally, the behavioural projection focuses

on the behaviour of the component and its operations, as manifest by UML state charts and UML activity diagrams.

The basic principle behind KobrA is that a component should be viewable and describable at both the specification and realization levels of abstraction from all three "projection" perspectives. Thus, the specification of a component can be viewed from a structural, functional and behavioural viewpoint, and the realization of a component can be viewed from a structural, functional and behavioural viewpoint. Figure 4, for example, is a screenshot from the IDE which shows the structural view of the specification of a Bank component. This takes the form of a UML class diagram which shows only externally visible properties of the component and its environment.

Figure 5, on the other hand, is a screenshot of the IDE which shows an element of the functional view of the Bank specification. This is a so called "operation specification" for an operation of the Bank (the withdraw operation), and defines the effects of the operation in terms of OCL pre and post conditions. It is only one element of the

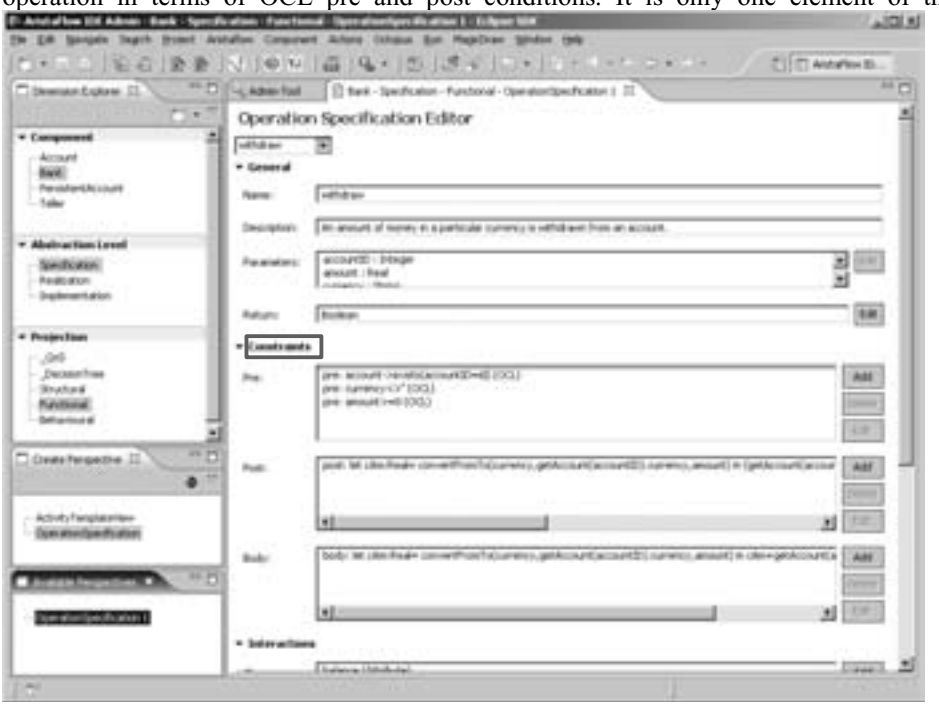

Figure 5: Functional View of a Component Specification

functional view of the Bank specification, because such a specification is needed for each operation. Similarly, the behavioural view of a component specification consists of a UML state diagram depicting the externally visible state and transitions of the component. However, this is not shown here for space reasons.

These views are general views on components defined in the KobrA method. With one exception they all employ a UML diagram. The exception is the operation specification which is a form based view that uses the OCL. However, to support requirement (3) above, additional views are supported in the IDE in order to provide information directly needed for workflow compatibility checking. For example, Figure 6 below shows a view which is used to define the classification of a component within a standardized business taxonomy (like UNSPSC [Un07] or NAICS [Na07]). This provides information which is directly used by the repository to support the cataloguing and organized browsing of components.

Figure 6: Component Classification View

Figure 7 shows an even more detailed view which allows users to add extra information needed explicitly by the ADEPT workflow editor and execution system.

Additional views can also obviously be added for different purposes. For example, it would be easily possible to add higher-level business oriented views such as those defined by [Tu02].

Of course, to support the integration of legacy components and not overwhelm developers with features they are not familiar with, a developer is not forced to use all the possibilities of the development environment. Only the use of a minimal set of artifacts for the component repository is mandatory, e.g. information about the executable operations of a component.

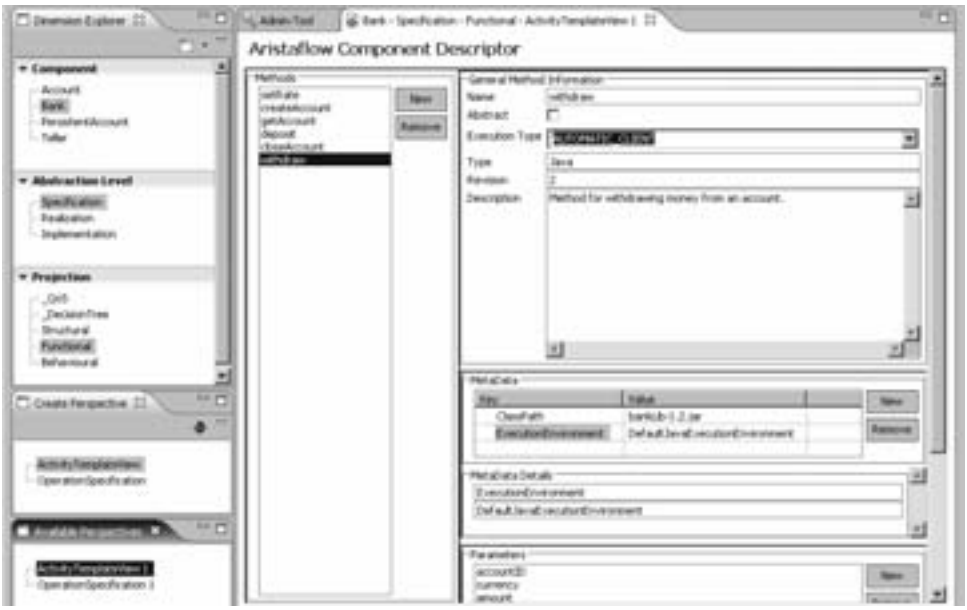

Figure 7: Component Descriptor

4 Dimension-based Navigation

Supporting a fundamentally view-based way of creating and manipulating components of the form described in the previous section can greatly simplify the task of developing components and assessing whether they are suitable for use in workflows. Different stakeholders can view a component using diagram types and notations which best meet their needs and expertise, and specialized views can be generated for specific purposes. However, the downside to a view-based approach is that the number of views can quickly explode. As a result, the benefit gained by the simplicity and clarity of individual views can be outweighed by the extra complexity and overhead involved in organizing and navigating around a large number of different views. This problem is particularly acute in environments which use different, third party editors to generate and manage views, since a user must then become acquainted with and navigate around different, heterogeneous artifact trees.

An effective view-based IDE should therefore provide a simple, integrated approach for managing and navigating around the various views supported by the system. To meet this need the AristaFlow IDE employs a new navigation metaphor based on the notion of independent, orthogonal development dimensions. This is motivated by the "orthographic projection" paradigm used in mechanical and physical engineering to create detailed drawings of physical objects, and exploits the fact that the different projections and abstract levels used in KobrA to define the different views are essentially

orthogonal and hence can be selected independently. This is no accident, since the KobrA method explicitly recognizes the existence of three fundamental and orthogonal development dimensions. However, the use of these as a navigation metaphor is original in the AristaFlow IDE. This is why we use the name "orthographic modeling" to characterize the representation approach supported by the IDE.

In principle, there is no limit to the number of dimensions that can be supported. However, in the current version of the IDE there are three dimensions: an abstraction level dimension which represents the abstraction level discussed in the previous section and has three distinct choices (specification, realization and implementation), a projection dimension which represents the projection type discussed in the previous section and also has three distinct choices (structural, functional and behavioural), and a component dimension, which represents the component which is being worked on or viewed. This has as many choices as there are components in the system – one for each component.

The organization of the views around the notion of three orthogonal dimensions can be visualized in terms of a cube, as illustrated in Figure 8. Each view corresponds to a cell in the cube, which represents a particular choice for each of the independent dimensions. Users are thus able select particular views by navigating around the cube and selecting specific cells corresponding to specific choices of component, abstraction level and projection.

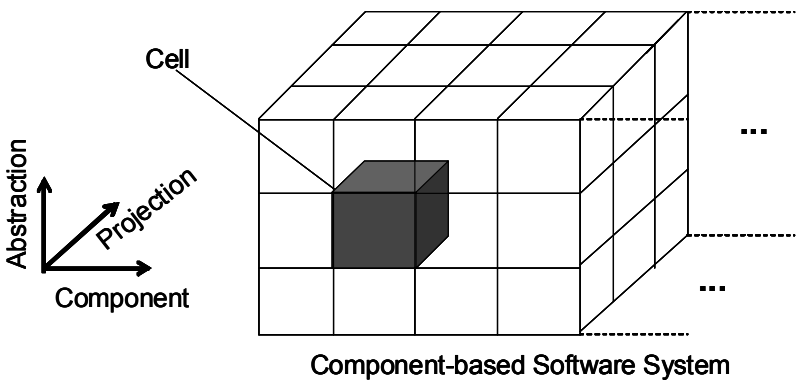

Figure 8: Cube metaphor

Figure 4 to 7 show how this dimension-based navigation is actually supported in the current IDE. The left hand side of each of these diagrams shows the navigation area which contains a selection panel for each of the three dimensions, each showing the currently selected option for each dimension. Thus, the navigation area on the left hand side of Figure 4 shows that the displayed UML diagram actually occupies the cell corresponding to the Bank option of the component dimension, the specification option of the abstraction-level dimension and the structural option of the projection dimension.

In other words, it shows a structural view of the specification of the Bank component. Similarly, the navigation area on the left hand side of Figure 5 shows that the displayed operation specification occupies the cell corresponding to the Bank option of the component dimension, the specification option of the abstraction level dimension and the functional option of the projection dimension. In other words, it shows a functional view of the specification of the Bank component. Obviously, by selecting different combinations of choices from each dimension, users can navigate to different views.

Usually, one cell is associated with exactly one editor, e.g. a UML class diagram is associated with the UML tool MagicDraw. However, if greater flexibility is desired, a cell can be mapped to multiple editors, for example, when there are alternative tools available for UML class diagrams. This is the role of the bottom selection panel. It identifies which specific representation or rendering of a view is desired.

5 Configuration of the IDE

As mentioned above, to create a practical prototype IDE within the original AristaFlow project a number of existing editors and tools were integrated under the view-based metaphor just described. In this section we briefly explain what tools were integrated and what role they play.

5.1 Editor overview

Projection / Abstraction	Structural	Functional	Behavioural
Specification	UML Class Diagram Taxonomy Component Descript.	Operation Specification Activity Template	UML State Chart Regular Expression
Realization	UML Class Diagram	UML Communication Diagram	UML Activity Diagram
Implementation	Source Code	-	-

Figure 9: Overview of editors for workflow component modeling

As an elegant and widely available UML diagramming tool, MagicDraw was chosen for the following Perspectives: Class Diagram, State Chart, Communication Diagram, and Activity Diagram. The Class Diagram editor is used for both the black box view (*Structural – Specification*) and the white box view (*Structural – Realization*) of a component. The behaviour of a component is modeled with a UML State Chart. In the Structural Realization, the UML class diagram from the Structural Specification is refined. The Functional Realization shows by the means of a UML communication

diagram with which components the function interacts. The Behavioural Realization focuses on the decomposition of functions by exposing the internal logic with a UML activity diagram.
The Operation Specification Editor focuses on pre- and post conditions of single operations and their syntactical correctness. Especially suited for the component repository are the Component Description Editor, the Taxonomy Editor, the Activity Template Editor and the Regular Expression Editor (whose description can be used for checks of allowable method call sequences at run-time).

After implementing a component, all artifacts can be packaged and saved in a single file in the Reusable Asset Specification (RAS) format. The RAS is an OMG standard specifying the structure, contents and description for reusable software components/assets [Ras05]. Thus, a RAS can be populated with all the artifacts generated during the component development process including models, requirement specifications and tests as well as the final source code. This allows the component to be imported into arbitrary development environments (i.e. any IDE that supports the RAS) for further development and maintenance when the need arises.

6 Conclusion

The goal of the AristaFlow project was to develop and prototypically implement a platform to support the whole lifecycle of flexible, process-aware information systems - from the modeling and implementation of suitable components through process composition in a plug and play like fashion up to flexible and adaptive process enactment. An important part of this goal was to minimize errors at runtime by using advanced component development and process composition methods. The IDE and component repository described in this paper were developed to support the component modeling element of this concept.

The developed IDE makes two major contributions to the state of the art – the first is a novel approach for the dynamic generation of views on demand, and the second is a novel approach for allowing users to organize and navigate around the different views. While the IDE was specifically developed for the AristaFlow project, its view, navigation and component representation concepts are useful for other software development approaches as well.

Acknowledgements**:** This work was largely performed as part of the AristaFlow project under the support of the State of Baden-Württemberg.

References

[Ac04] H. Acker, C. Atkinson, P. Dadam, S. Rinderle, M. Reichert: *Aspekte der komponentenorientierten Entwicklung adaptiver prozessorientierter Unternehmenssoftware*. In: K. Turowski (Hrsg.): Architekturen, Komponenten,

Anwendungen - Proc. 1. Verbundtagung AKA 2004, Augsburg, Dezember 2004. LNI P-57, 2004, S. 7-24

[At02] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, J. Zettel: Component-Based Product Line Engineering with UML. Addison-Wesley Publishing Company, 2002

[Ar07] The AristaFlow project, http://www.aristaflow.de, visited Nov 2007

[AS07] C. Atkinson, D. Stoll: *Orthographic Modelling Environment*, Fundamental Approaches to Software Engineering (FASE'08), Budapest (Hungary), 29 March - 6 April, 2008, submitted

[Da05] P. Dadam, M. Reichert, S. Rinderle, C. Atkinson: *Auf dem Weg zu prozessorientierten Informationssystemen der nächsten Generation - Herausforderungen und Lösungskonzepte*. In: D. Spath, K. Haasis, D. Klumpp (Hrsg.): Aktuelle Trends in der Softwareforschung - Tagungsband zum doIT Software-Forschungstag 2005, Karlsruhe, Juni 2005. Schriftenreihe zum doIT Software-Forschungstag, Band 3, MFG Stiftung, 2005, S. 47-67

[DR04] P. Dadam, M. Reichert: *ADEPT - Prozess-Management-Technologie der nächsten Generation*. In: D. Spath, K. Haasis (Eds.): Aktuelle Trends in der Softwareforschung - Tagungsband zum doIT Software-Forschungstag 2003, IRB Verlag Stuttgart 2004, S. 27-43

[Na07] *The North American Industry Classification System* (NAICS), http://www.census.gov/epcd/www/naics.html, visited May 2007

[NKF03] B. Nuseibeh, A. Finkelstein and J. Kramer, "ViewPoints: meaningful relationships are difficult," International Conference on Software Engineering (ICSE 2003), Portland, Oregon, 2003

[Ow04] The OWL Services Coalition, OWL-S 1.1 Release, http://www.daml.org/services/owl-s/1.1/, visited Nov 2007

[Ri05] S. Rinderle: Schema Evolution in Process Management Systems. Dissertation, Universität Ulm, Fakultät für Informatik, Dezember 2004

[RD03] S. Rinderle, P. Dadam: *Schemaevolution in Workflow-Management-Systemen* ("Aktuelles Schlagwort"). Informatik-Spektrum, Band 26, Heft 1, Februar 2003, S. 17-19

[Ras05] *Object Management Group, Reusable Asset Specification, version 2.2*. http://www.omg.org/technology/documents/formal/ras.htm, Nov 2005, visited May 2007

[Tu02] K. Turowski (Editor) et al., *Standardized Specification of Business Components*, Memorandum of the working group 5.10.3 Component Oriented Business Application System, February 2002, http://www.wi2.info/downl/gi-files/MEMO/Memorandum-english-final-included.pdf, visited Nov 2007

[Un07] *The United Nations Standard Products and Services Code (UNSPSC)*, http://www.unspsc.org/, visited May 2007

[Ws05] Web Service Modeling Ontology (WSMO), W3C Member Submission, http://www.w3.org/Submission/2005/SUBM-WSMO-20050603/, visited Nov 2007