Distributed and Parallel Databases manuscript No.
(will be inserted by the editor)

Enhancing Traceability of Persistent Data Access Flows in Process-Driven SOAs

Christine Mayr · Uwe Zdun · Schahram Dustdar

Abstract In process-driven, service-oriented architectures (SOAs), process activities can perform service operations, data transformations, or human tasks. Unfortunately, the process activities are usually tightly coupled. Thus, when the number of activities in the process grows, focusing on particular activities of the flow such as the service operations reading or writing persistent data is a time-consuming task. In particular, in order to solve structural problems concerning persistent data access such as deadlocks in data-intensive business processes, stakeholders need to understand the underlying persistent data access details of the activities i.e. physical storage schemes, and database connections.

With our view-based model-driven approach, we provide a solution to generate flows of persistent data access activities (which we refer to as persistent data access flows). To the best of our knowledge these persistent data access flows are not used to solve structural problems in process-driven SOAs, yet. Moreover, our persistent data access flows can be flattened by diverse filer criteria e.g. by filtering all activities reading or writing from a specific database or table. Using our approach, we can enhance traceability and documentation of persistent data access in business processes. In a series of motivating scenarios from an industrial case study we present how our persistent data access flow concept can contribute to enhance productivity in service-oriented, process-driven environments. We qualitatively evaluate our concepts and prototypes, and finally, discuss the correctness and the complexity of the underlying algorithms.

C. Mayr
Distributed Systems Group,
Information Systems Institute,
Vienna University of Technology,
E-mail: christine.mayr@inode.at

U. Zdun
Research Group Software Architecture,
Faculty of Computer Science,
University of Vienna,
E-mail: uwe.zdun@univie.ac.at

S. Dustdar
Distributed Systems Group,
Information Systems Institute,
Vienna University of Technology,
E-mail: dustdar@tuwien.ac.at

1 Introduction

In process-driven, service-oriented architectures (SOAs), process activities can invoke service operations [31], transformations such as string manipulations, business logic, or human tasks to perform a certain activity. Decision nodes are used to define the possible paths through the flow. Often the process activities perform I/O operations on a persistent storage, typically an RDBMS. We refer to this special type of process activities reading or writing from a persistent storage as data access activities. Nowadays, this data access is often done by so-called data access services (DAS). DAS are variations of the ordinary service concept: They are more data-intensive and are designed to expose data as a service [44]. Like a common service consists of service operations, a DAS consists of DAS operations. The DAS can use data access objects (DAOs) that abstract and encapsulate all access to the data source and provide an interface independent of the underlying database technology [30]. The DAO manages the connection with the data source to obtain and store data. Our approach uses DAS and DAOs in our prototypical implementation.

The decision which alternative path to run in the business process often depends on persistent data. Thus, there is a tight coupling between persistent data access and business logic within a business process. This tight coupling is necessary to enable stakeholders to get a basic understanding of the overall process. However, it can hinder stakeholders to analyze, develop, and test persistent data access in a business process. In Figure 1, we use a UML [33] activity diagram to illustrate these dependencies. In UML terminology, the figure shows a main process with two activities (also called microflows in [17]). Each activity contains basic actions, the fundamental behavior units of an activity [33]. The business process consists of different types of actions, namely service operations, data access service operations, transformations, and human actions. This business process is part of our case study, which we describe in detail in Section 4.

A common problem in business process modeling is the detection of structural errors [38]. Current business process modeling systems (BPMS) [48] lack support for verification of structural problems concerning persistent data access. However, in many BPMS, such as IBM Websphere MQ Workflow, the process activities cannot request persistent data directly [37]. Therefore, these systems cannot trace persistent data access without the help of external dependencies. In other BPMS, such as Webmethods [40], the process activities are able to invoke persistent data access directly. However, they lack tool support for detecting structural persistent data access problems at modeling time. While collaborating on several service-oriented software development projects in a large enterprise, we identified a series of structural problems in business processes concerning persistent data access. During this collaboration, we identified three groups of stakeholders which are faced with these problems: the data analysts, DAS developers, and the database testers. In this article we discuss the drawbacks from the stakeholder perspectives and propose appropriate solutions.

In this article we provide the following contributions: We use a view-based model-driven approach to specify persistent data access in process-driven SOA. In particular, we specify persistent data access activities to better integrate persistent data access into processes. With these well-structured persistent data access activities, and our new view integration paths concept, presented in this article, we can generate persistent data access flows from business processes. We show how these persistent data access flows can enhance documentation and solve structural errors concerning persistent data access in processes. As a result, data analysts, DAS developers, and database testers can increase their understanding of persistent data access activities in the process flow. By exploiting our model-driven approach we can filter the persistent data access flows by diverse search criteria such as they solely con-

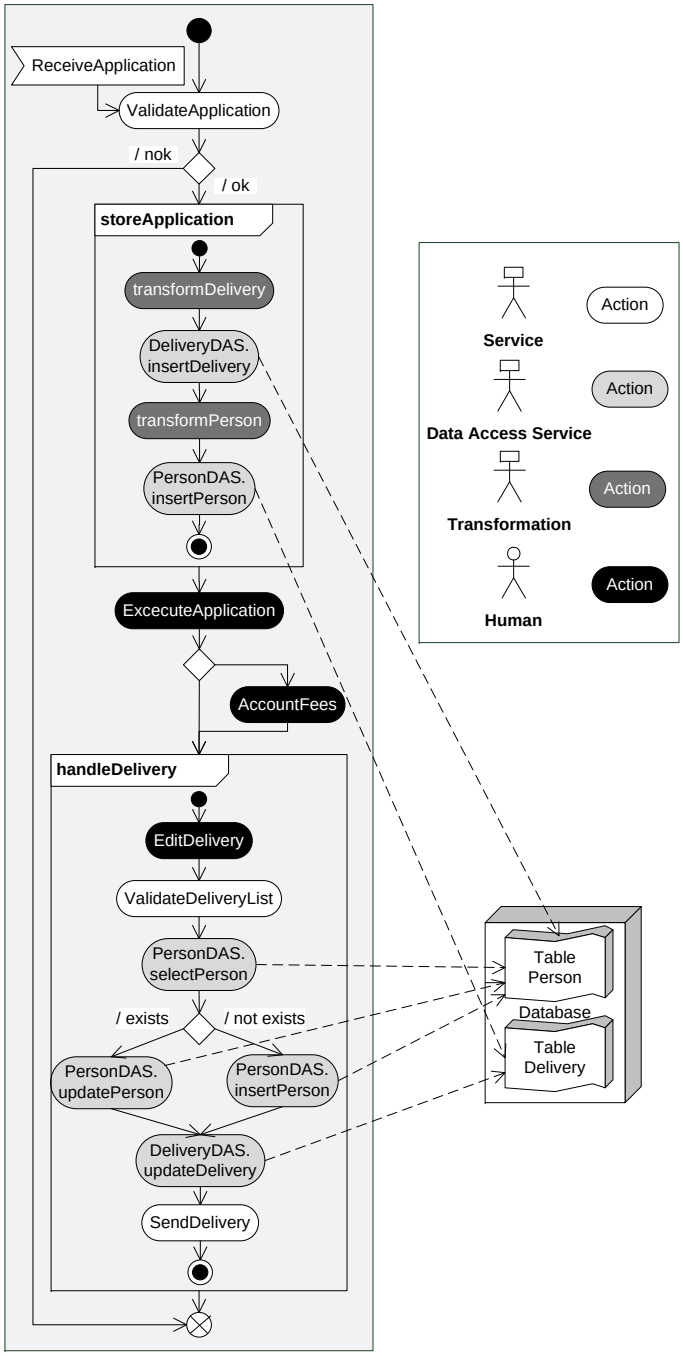


Fig. 1: Persistent Data Access in a Process-Driven SOA

tain only those activities reading or writing from specific database tables or certain ORM frameworks. Hence, development and deployment efforts can be reduced. Our model-driven solution is based on the View-based Modeling Framework (VbMF) introduced in our earlier work [43]. This framework aims at separating different concerns in a business process into different views. The main idea in our VbMF approach is to enable stakeholders to understand each view on its own, without having to look at other concerns, and thereby reduce the development complexity. The data-related extension of VbMF, the View-based Data Modeling Framework (VbDMF) [27,28], introduces a layered data model for accessing data in process-driven SOAs.

This article is organized as follows: First, in Section 2 we discuss related work. Next, Section 3 provides some background information to better understand our approach. Section 4 presents a motivation case study which we will refer to throughout this article. In Section 5 we present an overview of our approach. Next, in Section 6 we illustrate how our persistent data access flow concept can solve structural problems in business processes by presenting selected use cases. Section 7 describes the details necessary to realize our approach within a model-driven generator environment: the model-driven specification, integration, and extraction of persistent data access flows. Next, in Section 8 we show the applicability of our implementation solution and present a suitable tooling. In Section 9 we show the correctness and complexity of the underlying algorithms. We discuss the limitations of our approach in Section 10. Finally, Section 11 summarizes and concludes this article.

2 Related Work

In this section we present related work from the existing literature and related standards. We also emphasize the contribution of our work by explaining how our work compares to these related works. Finally, Table 1 summarize this comparison.

2.1 Integrating Persistent Data Access into SOAs

Many work focus on better integrating data into the overall SOA [5,49,50,46]. In the following we relate each of these works to our approach.

At this point we want to clearly differentiate our approach from other works such as BPELDT [14] which focuses on the data flows transporting data from one process activity to the next process activity. Habich et al. introduce BPEL data transitions to efficiently model 'data-grey-box web services' [14]. In contrast, in our approach, we focus on tracing the persistent data access activities themselves instead of the transitions between two process activities.

2.1.1 Modeling Data Access Services

In the literature, various related works propose using data access services for better data integration.

Carey et al. [5] examined how the AquaLogic Data Services Platform (ALDSP) supports data modeling and design. They describe the ALDSP 3.0 data service model and assert that the modeling extensions in ALDSP 3.0 provide a rich basis for modeling data services for SOA applications.

Wang et al. [46] propose a dynamic data integration model architecture based on SOA. On the basis of XML technology and web service, their architecture model enables data sharing and integration over all business systems. Thus data resource and information interoperability is realized in a cross-platform manner.

As our approach, both ALDSP [5] and the dynamic data integration model [46] use data access services to read and write data. However, they focus on separate modeling of data access services for use in external environments. In these two approaches, data integration overall business systems is established by using DAS as interface to the data. In contrast, we propose a continuous integration approach to be able to exploit the structured nature of the data access service models in business processes.

2.1.2 Modeling the Relationships between Persistent Data Access, Services and Processes

Zhang et al. [49] propose a new process data relationship model (PDRM) to specify the complex relationships among process models, data models, and persistent data access flows. In our approach, we define the activities incorporating data access as DAS activities. Likewise, Zhang et al. define these activities as data access nodes (DAN). Like our approach, they understand data access flows as persistent data access flows rather than data flows representing transient and persistent data. However, as opposed to our approach, they focus on automatic data access component generation to cluster similar data access components into larger components. In contrast, we concentrate both on the applicability and the feasibility of generating data access flows. Furthermore, unlike our model-driven solution, they solely focus on simple activities and cannot model structured process activities.

Zhang et al. [50] introduce a unique information liquidity meta-model (ILM) to separate persistent data integration logics from business services and application services. Like our approach their architecture uses a data service layer to access the data. In addition, as our approach, they use views to relate processes to new and existing services, or new and existing data definition. Whereas our approach focuses on modeling new view models and extracting new view models from existing view models, they solely concentrate on creating new models. In our approach we also focus on solving data analysis problems by creating flattened persistent data access flows from the whole process flow.

2.1.3 Business Process Modeling Systems

Our work is closely related to common commercial and open-source business process modeling systems (BPMS) [48]. Representatives of common commercial BPMS are IBM Websphere MQ Workflow [18], Webmethods [40], and TIBCO [42]. In addition, there are common representatives of open-source systems i.e. JBOSS [22] and Intalio [19].
Russel et al. [37] define a specific data interaction pattern for how BPMS access persistent data. Their main focus is to determine data patterns in business processes. On top of this data integration pattern, our conceptual approach focuses more on solving structural problems in business processes by using persistent data access flows. Unfortunately, many BPMS do not explicitly support this pattern by a direct integration of persistent data access into the process activities. In example, IBM Websphere MQ Workflow [18] and Intalio BPMS Designer [19] do not provide an explicit mechanism to invoke persistent data access from the process activities within the BPMS. In these BPMS, the persistent data is rather accessed e.g. through underlying services incorporating the persistent data access implementation.

Other BPMS such as Webmethods [40], JBOSS Messaging [21] and TIBCO [42] support integration of persistent data access into the process activities. In these BPMS, process

activities can directly request persistent data within the BPMS environment. However, as opposed to our persistent data access flow concept, these BPMS do not provide comparable support to adequately overview persistent data access in data-intensive business processes. In Webmethods [40], stakeholders can configure adapter services used to read or write data from the database, in example the InsertSQL, UpdateSQL and DeleteSQL services. As our approach, the services can configure SQL statements in a structured way. For example, statements can contain structured elements such as database connection properties, database tables, and database table columns. Furthermore, Webmethods provides filtering mechanisms to limit the adapter services by structured elements such as catalogs, schemes, and tables. JBOSS Messaging [21] supports configuration of relational database connections by the JDBC Persistent Manager. A channel mapper is used to configure SQL statements such as *Create* and *Select*. However, as opposed to our approach, JBOSS Messaging does not support structured modeling of persistent data access. In TIBCO [42], it is possible to establish the link between a process activity and structured process data models with business objects, e.g. specified in UML [23], designed with TIBCO Business Studio. TIBCO provides tooling support to read/write access from the business objects.

2.2 Solving Structural Problems in Business Processes

There are several approaches concerning solving structural problems in business processes [38,2]. Sadiq et al. [38] identify structural conflicts in process models by applying graph reduction rules. Awad et al. [2] use business process queries to detect structural problems in business processes. In contrast to these works, in our approach, besides solving structural problems in business processes, we aim at enhancing traceability and documentation of persistent data access. Moreover, we provide a model-driven solution to reduce business process complexity as we can flatten business processes by certain filter criteria.

2.2.1 Static Analyzing Techniques

There are a number of frameworks for performing static analysis to extract common data flows from the whole program. An example of these static analysis approaches is the demand-driven flow analysis as proposed by Duesterwald et al. [8]. 'The goal of demand-driven analysis is to reduce the time and space overhead of conventional exhaustive analysis by avoiding the collection of information that is not needed' [25]. As our approach, Duesterwald et al. focus on extracting sub flows from process flows on-demand. However, they aim at extracting common data flows instead of persistent data access flows. Moreover, we provide a model-driven, view-based approach to analyze and document persistent data access flows in process-driven SOAs.

In Section 4 we present how our approach can be in particular applied to testing and to deadlock detection and prevention. In the following we relate other testing and deadlock detection and prevention techniques to our solution.

2.2.2 Testing

There are a number of approaches in the literature that elaborate on test case creation and selection.

Fischer et al. [10] focus on improving test case quality for declarative programs by introducing a novel notion of data flow coverage. In their opinion, a visual representation of

the control- and/or data flow would help the users to better understand program execution. We share the opinion that a visual view increases the understandability of the data flows. However, our views are not restricted to the viewing of these data flows. Accordingly, our persistent DAS Flow View can be integrated with other views in order to form richer views, in example with the DAO View, the ORM View, and the Physical Data View.

There are several works using data flows for selecting test cases as presented in [35]. Rapps et al. apply data flow analysis techniques to examine test data selection criteria. The procedure presented associates each definition of a variable with each of its usages within a flow. The data flow criteria that they have defined can be used to traverse each path. Like our approach, each persistent data access activity in the process flow can be associated with corresponding definitions. In contrast to this formal approach, we use a visual approach for selecting our test cases.

2.2.3 Deadlock Detection and Prevention

In the following we relate our solution to various static and dynamic deadlock detection techniques.

There are many runtime approaches (such as [20] and [36]) that aim at deadlock-free sharing of resources in distributed database systems. Isloor et al. [20] distinguish between deadlock detection, deadlock prevention and deadlock avoidance techniques. Krishna et al. [36] present a graph-based deadlock prevention algorithm that reduces processing delays within the distributed environment. However, with our persistent data access flow approach we provide a visual solution in order to discover errors at the earliest stage of development – at the modeling level.

There are a number of static deadlock detection algorithms (e.g. [29]). Naik et al's [29] deadlock detection algorithm uses static analyses to approximate necessary conditions for deadlocks to occur. Their effective algorithm concentrates on detecting deadlocks between two threads and two locks.

Dedene et al. [7] present a formal approach to detect deadlocks at the conceptual level. In their work, they present a formal process algebra to verify conceptual schemes for deadlocks based on the object-oriented analysis (OOA) method M.E.R.O.DE. As our approach, Dedene et al. can check the models for deadlocks at the earliest stage in the development process.

An interesting approach is presented by Zhou et al. [51]. Like our approach, the authors use a static approach to analyze deadlocks in data flows. They in particular concentrate on analyzing deadlocks in loops. In order to determine deadlocks, they define a causality interface that abstractly represents causality of data flow actors.

In contrast to these formal deadlock analysis approaches, again, our approach is a visual solution for detecting deadlocks. Furthermore, on top of our approach, common deadlock detection and prevention techniques as described before can be performed. Furthermore our persistent data access flow concept aims at documenting the persistent data access flows within the control flow. Thus, with our approach we do not solely focus on detecting deadlocks in process flows, we rather enable a more general analysis of a series of development and testing problems.

2.2.4 Business Process Modeling Systems

In order to solve structural problems in business processes, common BPMS such as IBM Websphere MQ Workflow [18], JBOSS [22], and Intalio [19] support transaction handling.

Thus, in case of failures, the transactions can be rolled back or compensated. Whenever some actions cannot be rolled back e.g. due to external dependencies, a compensation handler can be invoked to perform an 'undo action'. In contrast, our persistent data access flow approach focuses on the underlying structural problem. Moreover, we solve the cause of the failed transactions instead of solely handle the problem. In example, if a database table is locked, due to a structural problem in the business process, our persistent data access flow approach will contribute to solve the problem more quickly. In addition, our model-driven provides up-to-date documentation of persistent data access in business processes.

3 Background

Before we go deeper into the contributions of our approach, we give some background knowledge to better understand our concepts.

3.1 Data Flow vs. Control Flow

Common graphical process modeling languages and business process management systems (BPMS) [48] can differentiate between the control flow and the data flow of a process. Examples of graphical modeling languages are the Business Process Model and Notation (BPMN) [32] and the Unified Modeling Langage (UML) [33] activity diagrams. An example of a BPMS is the IBM Websphere MQ Workflow [18] Whereas the control flow describes the sequence of activities of the process flow, the data flow describes incoming and outcoming data to and from process activities. An example of a control flow is depicted in Figure 1.

In BPMN, data is transferred in data objects that can be associated with activities. The data flow is modeled by associations from data objects to activities or vice versa. Accordingly, data objects written by one activity can be read be the subsequent activity. In IBM Websphere MQ Workflow, a data flow is modeled by connecting the activity's input and output container. Special data flow connectors define the mapping of the activity's input and output container. In UML 2.0, the data flow is specified by pin elements representing the inputs and outputs of activities. Whereas input pins provide the activities with data, output pins get the data from the activities. Figure 2 depicts a data flow in UML notation.

Lang defines that data flows between processes may represent either attributes of objects, transient data or persistent data [24]. In contrast to these data flows, in this article, we concentrate on persistent data access flows. Our persistent data access flows are control flows that solely consist of data access activities reading or writing from a persistent data storage. In contrast, whenever we refer to data flows, we outline the common data flows, representing transient and persistent data respectively, as defined by Lang [24].

3.2 Microflow and Macroflow Pattern

Our work is in particular based on the so-called Macro-Microflow pattern [17,16]. The Macro-Microflow pattern is a pattern designed for process-oriented integration in service oriented architectures. According to this Macro-Microflow pattern, a microflow represents a sub-process that runs within a macroflow activity [17,16]. Macroflows are considered to be high-level conceptual business processes whereas microflows are technical information processes [17,16,11]

Table 1: Comparison of Related Work

Representative, Short Description/ Requirement	**Carey et al. [5]**, AquaLogic Data Services Platform (ALDSP)	**Wang et al.[46]**, Dynamic Data Integration Model architecture	**Zhang et al. [49]**, Process Driven Data Access Component Generation	**Zhang et al. [50]**, Information Liquidity Metamodel (ILM)	**Duesterwald et al. [8]**, Demand-Driven Flow Analysis	**Fischer et al. [10]**, Data-flow testing of declarative programs	**Rapps et al. [35]**, Data Flow Analysis Techniques for Test Data Selection	**Isloor et al. [20], Krishna et al. [36], Naik et al. [29], Dedene et al. [7], Zhou et al. [51]**, Deadlock analysis	**Common BPMS [18,40,42,22,19]**	**Our approach**
Specifies data access services	yes, by ALDSP	yes, to enable data sharing and integration over all business systems	yes, the data access nodes (DAN) invoke the data access services	yes, by a Information Liquidity Model (ILM)-based data service metamodel	not relevant	not relevant	not relevant	not relevant	yes, e.g. Webmethods [40] supports the configuration of adapter services	yes, by VbDMF
Models data access activities	no	no	yes, by the process data relationship model (PDRM) which defines special data access nodes (DAN)	yes, by a ILM-based data service metamodel	not relevant	not relevant	not relevant	not relevant	no	yes, by VbDMF
Models sub process flows	no	no	no	yes, a process of the data service metamodel consists of 0..* processes	not relevant	not relevant	not relevant	not relevant	yes, common BPMS such as IBM MQ Workflow [18] support the micro-macro-flow pattern [17]	yes, by VbMF
Distinguishes between persistent data access flows and transient data flows	no	no	yes, by the data access nodes (DAN) of the process data relationship model (PDRM)	yes, the data service view models the persistent data access flows	no	no	no	not relevant	no	yes, by VbDMF, we can model persistent data access flows
Supports views	no	no	no	yes, supports a process definition, service definition, ILM and data view	no	no	no	no	yes, e.g. in TIBCO [42], separate design of the data models and sequence of process activities	yes
Supports view integration	no	no	no	no	no	no	yes, they associate each data access activity in the data flow with its corresponding definition	no	yes, e.g. TIBCO [42] can integrate process data models into business processes	yes, by VbDMF
Associates correspondent data definitions to data access activities	yes, by the design view	yes, by the XML View	yes, the references are modeled by the process data relationship model (PDRM)	yes, by the data service metamodel	no	no	yes, they associate each data access activity in the data flow with its corresponding definition	no	yes, e.g. in TIBCO [42], by integrating process data models to business process activities	yes, by view integration
Supports view extraction	no	no	no	no	yes, by demand-driven analysis	no	no	no	no	yes, by VbDMF
Visual solution	not relevant	not relevant	not relevant	not relevant	no	no, future work	no	no	not relevant	yes
Discovers errors at the earliest stage of development	not relevant	not relevant	not relevant	not relevant	no	no	no	no	yes, errors can be discovered at the modeling level	
Supports general analysis of a series of development and testing persistent data access problems	not relevant	not relevant	not relevant	not relevant	no	no	no	no	no	yes, by VbDMF and its ability to integrate, inherit and extract views
Documentation of persistent data access flows	no	no	possible	possible	no	no	no	no	no	yes, by VbDMF

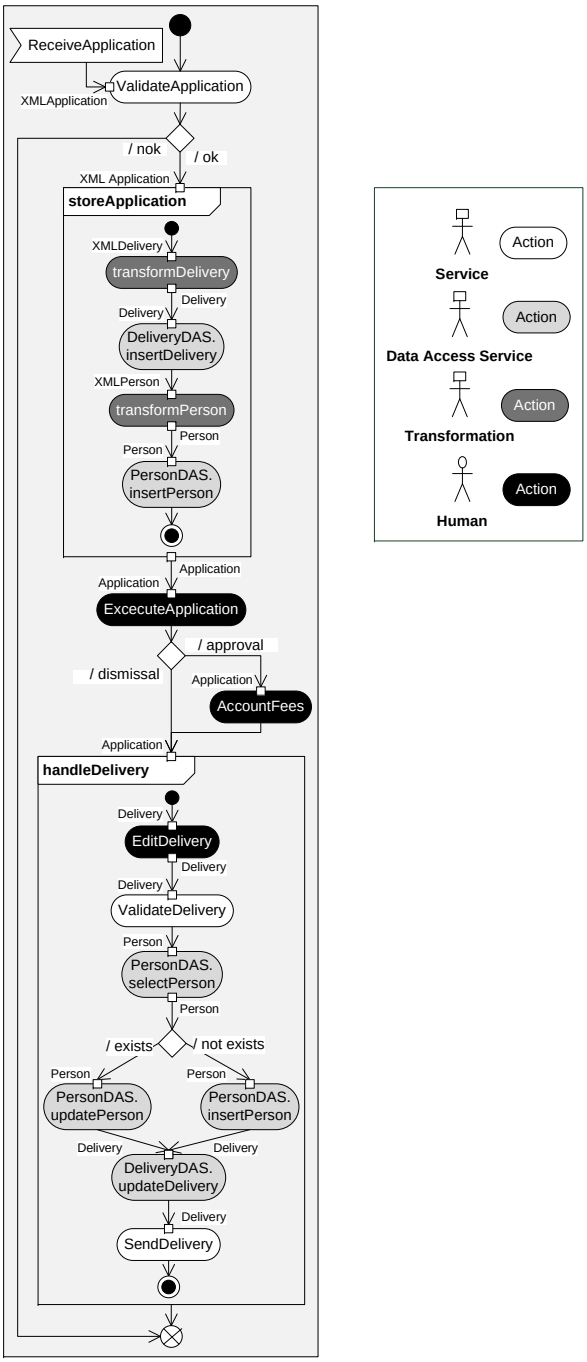


Fig. 2: Data Flow of a Business Process specified with UML pin elements

There are two types of microflows. Firstly, a short-running technical process that runs automatically and secondly, a flow of activities that can contain interrupting process activities such as human tasks and events. The first alternative, the technical microflows are not interruptible and are running in a transaction [17,16]. A flow is not interruptible if it contains no interrupting process activities such as human tasks and events. The interruptible microflows in turn can contain automatically short-running microflows. When analyzing, developing, and maintaining persistent data access, stakeholders have to focus on these microflows. When analyzing, developing, and maintaining persistent data access, stakeholders have to focus on these microflows. In Figure 1, we depict two technical microflows as technical sub processes of the whole business macroflow.

3.3 View-based Data Modeling Framework

In the following we shortly recapitulate the View-based Data Modeling Framework (VbDMF) which we apply to implement our solution. VbDMF is an extension of the basic View-based Modeling Framework (VbMF). VbMF is specified to define processes in a process-driven SOA. In contrast, VbDMF is focused on modeling persistent data access within processes.

VbMF consists of modeling elements such as (view) models, and views. A view or model instance is specified using an appropriate view model. Each model is a (semi) formalized representation of a particular business process concern. The models, in turn, are defined on top of the meta-model. We use the Eclipse Modeling Framework (EMF) meta-model to define our models. Accordingly, the VbMF core model is derived from the EMF [9] *.ecore meta-model. All views (model instances) depicted in this article are based on the XML Metadata Interchange (XMI) standard [13].

In Figure 3, the rectangles depict models of VbMF and the ellipses denote the additional models of VbDMF. In VbMF new architectural models can be *designed*, existing models can be *extended* by adding new features, views can be *integrated* in order to produce a richer view, and using *transformations* platform-specific code can be generated. As displayed by the dashed lines in Figure 3 view models of VbDMF extend basic VbMF view models namely the Information View model, the Collaboration View model, and the Flow View model. The dashed lines in Figure 3 indicate view integration, e.g., the Collaboration View integrates the Information View to produce a combined view.

In the following we shortly describe basic views of VbMF and VbDMF:

VbMF views:

- The VbMF Collaboration View model basically describes services and service operations.
- The VbMF Information View model specifies the service operations in more detail by defining data types and messages.
- The VbMF Flow View model describes the control-flow of a process.

VbDMF views:

- The VbDMF Collaboration DAO Mapping View model is an optional view model that maps DAS operations to DAO operations
- The VbDMF Information Data Object Mapping View model is an optional view model that maps DAS data types to DAO data types.

- The VbDMF DAO View model describes the DAO operations in detail.
- The VbDMF Data Object View model specifies data object types and data object member variables used to store values in object-oriented environments.
- The VbDMF ORM View model maps physical data to data object types
- The VbDMF Physical Data View model specifies the data storages such as database tables and columns accessed from the DAOs.
- The VbDMF Database Connection View model comprises a list of arbitrary, user-defined connection properties.

To summarize, whereas VbMF focuses on reducing the development complexity of business processes and services, VbDMF introduces tailored views for integrating persistent data access into the services of business processes.

4 Case Study

In this section we present our motivation case study which we will refer to throughout this article. This case study deals with a real workflow of a specific e-government application modeling the jurisdictional provisions in the context of a district court. However, the applicability of the persistent data access flows is not limited to this type of applications. The persistent data access flow concept can reasonably be applied to all applications, based on a process-driven SOA, where data is accessed from a persistent storage. In the course of this section, we also describe selected problems which we identified while collaborating in several projects for developing e-government applications. All these problems have in common that data is accessed from persistent storage. These problems reoccur in many use cases for data analysts, DAS developers, and database testers. For each problem, we illustrate how the persistent data access flows can contribute to problem solving.

First of all, let us explain the business process flow at the land registry court illustrated in Figure 1 of Section 1. As governmental processes are typically very complex [34], for reasons of simplicity, we use a flattened workflow for demonstration. We use a UML [33] activity diagram to model the process flow. Each process activity contains basic actions, the fundamental behavior units of an activity [33]. The business process consists of different types of actions, namely service operations, data access service operations, transformations,

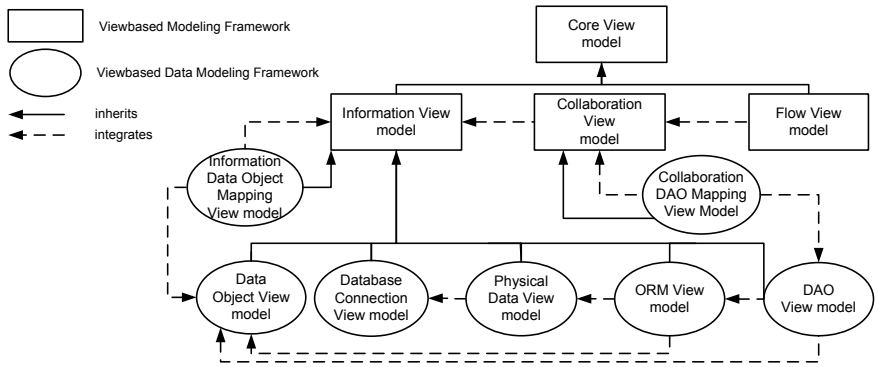


Fig. 3: VbMF and VbDMF – Overview

and human actions. The process starts when a new jurisdictional application is received. Then, the *ValidateApplication* activity invokes a service that checks the incoming jurisdictional application for correct syntax and semantic. Successfully validated applications are saved by a flow of alternate transformation activities and persistent data access activities. In case the validation fails, neither data is stored nor the delivery is sent to the applicant. In order to store data into the database by object relational mapping (ORM) mechanisms, the process data need to be transformed into data objects. The activities *transformDelivery* and *transformPerson* transform delivery and applicants process data respectively into associated data objects. After executing each of these transformation activities, the persistent data access activities *insertDelivery* and *insertPerson* respectively are invoked in order to persistently store the resulting data objects. Stored applications can be executed by the registrar within the human process activity *ExecuteApplication*. If the registrar approves the application, the service-based activity *AccountFees* will be invoked. As a dismissed application is free of charge, the service operation *AccountFees* is never invoked in case of dismissal. After accounting the fees, the registrar has to select whether the approval or dismissal shall be delivered by the system. Dependent from the registrar's decision, the approval or dismissal is delivered to the applicant. For this purpose, the process activity *ValidateDelivery* checks the recipient information for correctness and completeness before sending the delivery to the applicant. In case of successful validation, the two DAS operations *updatePerson* and *updateDelivery* are invoked in order to store the recipient information persistently. If the validation fails, the persistent data access activity *selectPerson* will return zero rows. In this case, instead of updating the person, a new person has to be inserted by invoking the persistent data access activity *insertPerson*. Finally, the service operation *SendDelivery* sends the delivery to the recipient by invoking an external service.

5 Our Approach

In this section we present the basic idea of our persistent data access flow concept. For this, we reuse the business process presented in the precedent case study Section 4.

On the left and on the right of Figure 4, the resulting persistent data access flows from the business process in the middle are shown. We define persistent data access flows as control flows containing the persistent data access activities of the whole business process flow. We differentiate simple persistent data access flows from filtered persistent data access flows.

- Simple persistent data access flows are control flows containing all and only the persistent data access activities of a business process
- Filtered persistent data access flows are control flows containing only those persistent data access activities of a business process that match certain persistent data access filter criteria

On the left of Figure 4, a simple persistent data access flow is depicted. On the right of the figure, a filtered persistent data access flow is shown. The filtered persistent data access flow in this example contains only those persistent data access activities reading or writing data from table *Person*.

In this article we use DAS with underlying DAOs as example implementation. However, our approach can be easily applied for other types of persistent data access implementations. In the following we show how our persistent data access flows depicted on the left and on the right of Figure 4 can be applied to enhance traceability and documentation of persistent

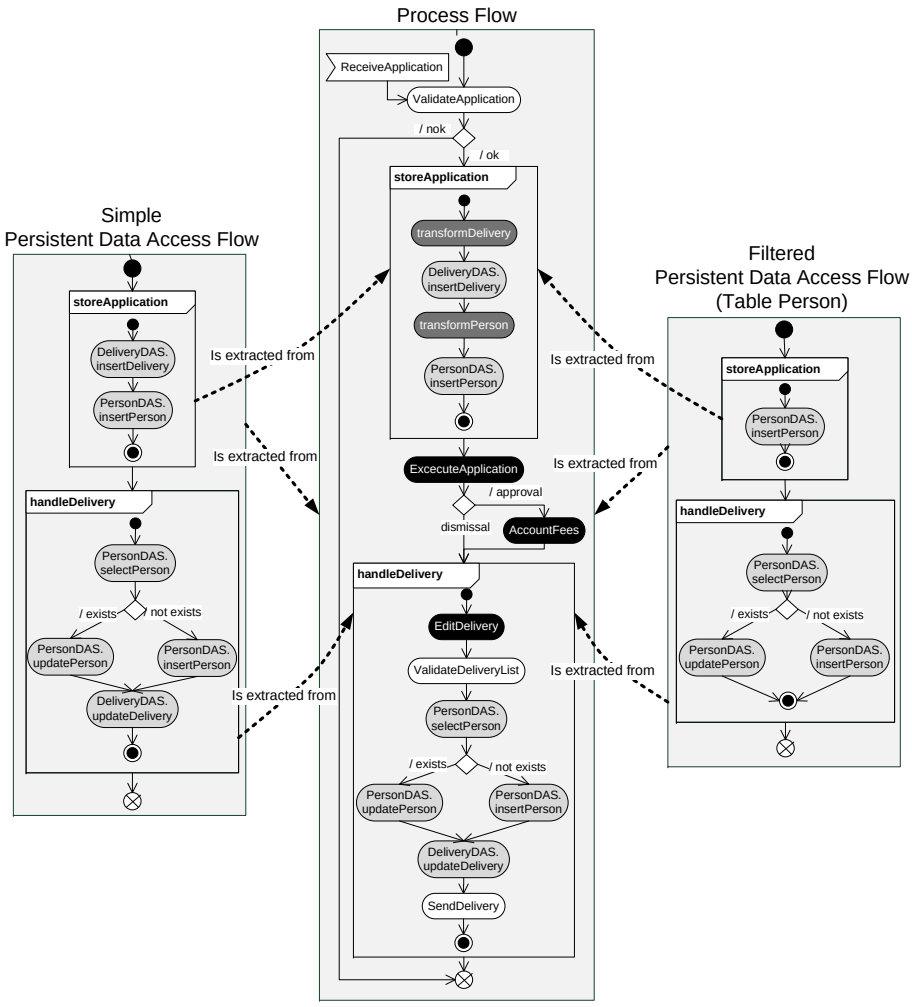


Fig. 4: Two Persistent Data Access Flows Extracted from a Business Process flow

data access in process-driven SOAs. For this purpose, in the following Section 4, we present selected problems and solutions from different stakeholders' point of view, in particular from the perspective of data analysts, DAS developers, and database testers.

6 Solving Structural Problems in Business Processes

In this section we illustrate how our persistent data access flow concept can be generalized to solve various data analysis problems. For this, we refer to the selected problems introduced in Section 1. These problems reoccur in many cases for data analysts, DAS developers, and database testers when analyzing, developing, and testing persistent data access in business

processes. For each selected problem, we describe how stakeholders can apply our persistent data access flow concept to solve it.

1. At first, we have a look at a typical data analysis problem. We show how our persistent data access flow concept can ease the manual and automated data analysis in process-driven SOAs. Our goal is not to reinvent deadlock detection, but instead show how both *manual and automatic deadlock detection* in a complex process model can be eased by applying the persistent data access flows. On top of our approach existing data analysis solutions such as deadlock detection techniques can be applied.
2. Secondly, we show how DAS developers can benefit from our view-based approach. The persistent data access flows can be applied to document the persistent data access flows in a process. Furthermore, we illustrate how to *detect design weaknesses* concerning persistent data access at the earliest possible state of the development process [7] – in the modeling phase.
3. Thirdly, we describe how our approach provides database testers with appropriate input/output data needed for *test case generation and execution*. Moreover, we explain how the persistent data access flow concept can improve the database testers' documentation. Finally, we illustrate how our persistent data access flows support testers in locating errors more quickly.

6.1 Problem & Solution: Deadlock Detection

In process-driven SOAs usually a large number of process instances run in parallel in a process-engine. These process instances often require access to competing data resources such as data from an RDBMS. Deadlocks arise when process instances hold resources required from each other. When none of these process instances will lose control over its resources, a classic deadlock situation occurs [20]. There are various deadlocks detection techniques in order to discover and resolve deadlocks. One common method to resolve deadlocks are database transaction timeouts as used by common database drivers such as the Java Database Connectivity (JDBC) driver [6]. Accordingly, after the timeout expired, process instances lose control over the held resources.

A process can perform some transformations, invoke service operations, and access the database. In order to prevent, detect, and solve deadlocks, data analysts need to focus on the persistent data access activities of a process. Moreover, stakeholders have to make sure that the DAS operations of different process flow instances always have to be processed in the same order such that no two DAS operations have to wait for competing resources.

manual deadlock detection with persistent data access flows In the following we present how our approach can contribute to detect deadlocks in business processes by using our persistent data access flow approach. Figure 5 displays the two persistent data access flows of our business process. In order to identify the persistent data access activities they are consecutively numbered.

The persistent data access flow on the left hand side simply consists of two DAS operations. The first DAS operation *DeliveryDAS.insertDelivery* (1) inserts delivery data into table *Delivery*. Afterwards the DAS operation *PersonDAS.insertPerson* (2) inserts person data into table *Person*. The persistent data access flow on the right hand side of the figure consists of a DAS operation *PersonDAS.selectPerson* (3) that selects a row from table *Person* using certain filter criteria. If the result set is empty, a new row will be inserted into table

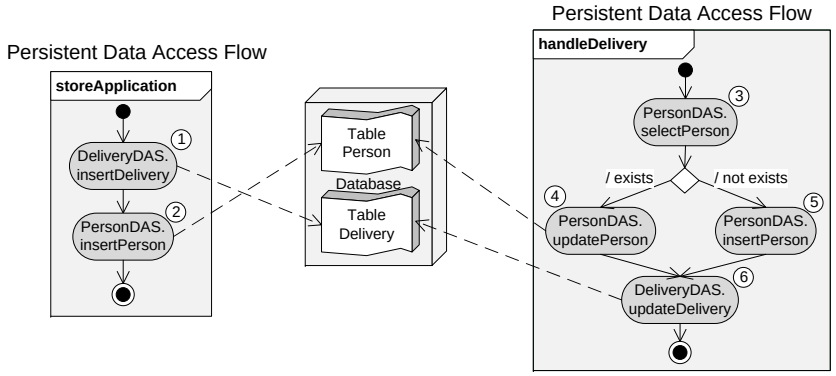


Fig. 5: Motivating Example for Manually Detecting Potential Deadlock Risks

Person by the DAS operation *PersonDAS.insertPerson* (5). Otherwise the retrieved row in table *Person* is updated by the DAS operation *PersonDAS.updatePerson* (4). Finally a row in table *Delivery* is updated by the DAS operation *DeliveryDAS.updateDelivery* (6).

All activities in a process flow instance are running in a transaction [39]. Consider two process instances *p1* and *p2* running through the main process. *P1* inserts a new row into table *Delivery* by performing the DAS operation *DeliveryDAS.insertDelivery* (1). At the same time *p2* updates a row into table *Person* by performing the DAS operation *PersonDAS.updatePerson* (4). Thus DAS operation *DeliveryDAS.insertDelivery* (1) holds table *Delivery* and DAS operation *PersonDAS.updatePerson* (4) holds table *Person*. As a result, the DAS operation *DeliveryDAS.updateDelivery* (6) cannot be executed because DAS operation *DeliveryDAS.insertDelivery* (1) holds table *Delivery*. Likewise, the DAS operation *PersonDAS.insertPerson* (2) cannot be executed because *PersonDAS.updatePerson* (4) holds table *Person*. This is the classic deadlock situation. In the figure, this deadlock situation is displayed by the intersecting arrows on the left hand side and, concomitantly, the non-intersecting arrows on the right hand side.

Without our persistent data access flow concept, analysts cannot solely focus on the persistent data access activities of the process, but must consider many other concerns at the same time. Therefore, especially if a large number of different types of activities is used in a flow model, manual deadlock detection will be an exhaustive and time-consuming task. Our approach is to provide a specific persistent data access flow that enables data analysts to focus only on the relevant information helpful for detecting deadlocks. In particular, our approach supports a visual solution to already eliminate potential deadlock risks at the modeling level. The same can be assessed for any other manual data analysis task in process-driven SOAs. Furthermore, on top of our approach, common deadlock detection techniques (such as [29,7,51]) can be performed.

Automatic deadlock detection with persistent data access flows In some cases, we want to go beyond manual data analysis in process-driven SOAs. The persistent data access flows enable us to easier implement algorithms for static deadlock detection in distributed database systems: As explained in the example above (see Figure 5), a deadlock can occur, when data resources in different persistent data access flows are accessed in a different order. Thus in order to detect possible deadlocks, we need to check the order in which database tables are

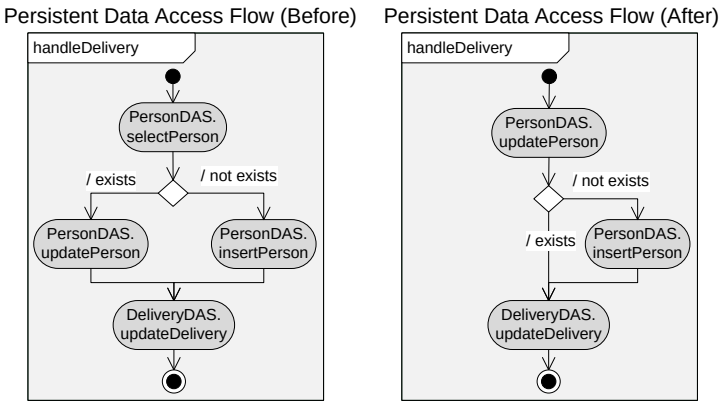


Fig. 6: Motivating Example for Detecting Inefficient Persistent Data Access Flows

accessed in each of these persistent data access flows. For this, we need to consider the paths of all persistent data access flows of the process. Accordingly, a possible deadlock algorithm compares the order in which database tables are accessed in one path *p1* with the order in which tables are accessed in another path *p2*. This pair-wise comparison needs to be done for each possible path of the persistent data access flows of a process.

6.2 Problem & Solution: Design Weakness Detection

In process-driven SOAs, at first, DAS developers have to become acquainted with the process flows including business logic activities, transformation activities, and persistent data access activities. In particular, they need a general overview of the persistent data access flows of the process e.g. they need to know which tables are accessed by a certain DAS operation. These persistent data access flows are in particular important for developers who need to review the developed database transactions in case of troubleshooting or analysis of performance leaks.

Secondly, in integrated development environments (IDE) such as Eclipse [41], it is possible to search for modules that invoke a certain DAS operation. However, in a process flow of different types of activities, to search for specific DAS operations can be a time-consuming task. Accordingly, in contrast to our approach, in common IDEs it is not possible to extract a list of DAS operations accessing a specific database table or database table column.

Thirdly, the persistent data access flow enables DAS developers to easily discover inefficient persistent data access flow. Figure 6 shows an example of such an inefficient persistent data access flow before and after redesigning it. When we look at the flow on the left hand side of the figure, we can easily recognize that eliminating the DAS operation *DAS1.select* could reduce the number of statements during process execution. The reason for this is that the update operation *DAS1.update* anyway returns the number of updated data sets. After redesign, we can see the resulting flow on the right hand side of Figure 6. There are various performance measuring tools used to discover performance leaks at runtime. However we provide a visual approach to detect inefficient source code at the earliest possible state of the

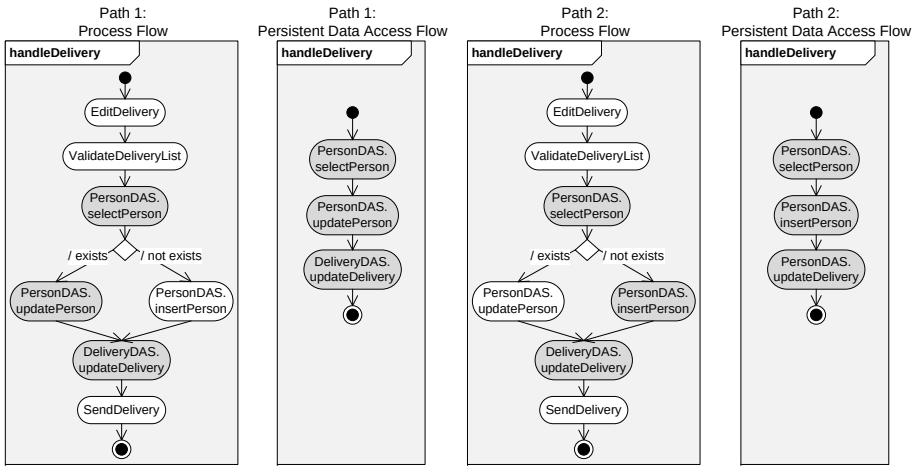


Fig. 7: Motivating Example for Testing Persistent Data Access of a Process Flow

development process [7]. Our approach is not limited to the example above. It rather can be applied to solve many other types of structural problems in business processes.

6.3 Problem & Solution: Test Case Generation

One major task during testing a process is to check whether data is correctly stored and retrieved from a central storage. For this purpose, test cases have to be created, tested, and executed, and finally, the results need to be examined [15]. In the following, we concentrate on creating test cases at two different levels:

1. Test cases for single persistent data access activities: Each persistent data access activity will have to be checked whether data is correctly stored and accessed. Each persistent data access activity can be tested independently from the whole process. In order to create, test, and execute these test cases respectively, testers require necessary input and output data for each path of the process [35] (see Figure 7). In order to provide appropriate input and output data, they need the information which tables are accessed by a specific DAS operation. Our approach enables extracting persistent data access flows by different filter criteria, such as extracting persistent data access flows containing only those activities accessing a specific column of a database table.
2. Test cases for transactions: All persistent data access activities of a process are running within a transaction. For each possible path in this transaction, a test case has to be created, tested, and executed in order to verify the correctness of each path. Figure 7 exemplifies these different paths both of the process flow and of the associated persistent data access flow. The bold arrows mark a specific path within the process flow. In order to create, test, and execute cases for transactions, testers have to overview the overall persistent data access flows of a process. For this purpose, data test developers need a documentation of these modeled persistent data access flows. However, from our experience, in industry, persistent data access flows are not documented. Furthermore, even if such kind of documentation existed, the problem of updating this documentation in a

timely manner would remain. 'The only notable exception is documentation types that are highly structured and easy to maintain, such as test cases and inline comments' [26]. As our persistent data access flows follow the MDD [45] paradigm, there is no gap between specification and development. In particular, the effort to update the specification to be synchronized with the newly implemented data access activities is not necessary. The persistent data access flow concept provides such kind of documentation implicitly and thus enables testers to gain a better understanding of the persistent data access flows of a process. As shown in Figure 7, with our persistent data access flows, database testers can easily overview the persistent data access activity paths of a process. Finally, due to the improved persistent data access documentation, we argue that using the persistent data access flow concept can decrease the participation of the different stakeholders during test case design.

Moreover, our persistent data access flow concept enables testers to locate errors more quickly when a specific test case asserting persistent data access fails [10]. Accordingly, testers will be able to verify the particular path of the flow and thus will more efficiently determine the failure reason e.g. if the failure is due to an error within the process, the test case or the provided input data. During the run, a log handler can log each persistent data access activity performed during the process. With this information, the error causing persistent data access activities can be easily retrieved by reconstructing the entire path of the persistent data access flow.

7 Solution: Model-driven Specification, Integration, Extraction

In this section we prove the technical feasibility of our approach. Our model-driven solution is based on the View-based Data Modeling Framework (see Section 3.3). Our highly structured models are used as the modeling basis for extracting our flattened persistent data access flows. In the following we present the necessary steps to be taken in order to implement our persistent data access flow concept.

- **Specification** of persistent data access activities
- **Integration** of persistent data Access activities with persistent data access implementation details
- **Extraction** of persistent data access flows from whole process flows

7.1 Specification

In Section 3.3 we already provided a general overview of our View-based Modeling Framework (VbMF) and View-based Data Modeling Framework (VbDMF). Now, we present our VbDMF Flow View model that is used to define the data access activities of a process flow. As shown in Figure 8, our VbDMF Flow View model is extended from the basic VbMF Flow View model.

The VbDMF Flow View consists of a separate persistent data access task *AtomicDASTask* extended from the basic *AtomicTask* of the VbMF Flow View. The VbMF *AtomicTask* class is a specialization of the VbMF *Task* class. The new *AtomicDASTask* allows stakeholders to structurally modeling persistent data access in business processes. On the basis of this new simple model, we can link a business process activity with persistent data access implementation details. In the following Section 7.2, we describe how stakeholders can associate

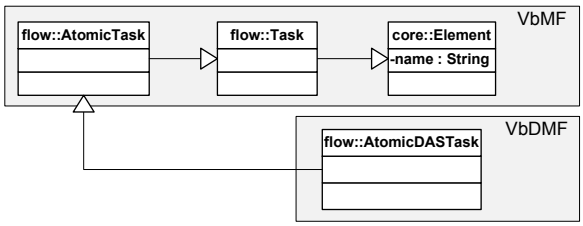


Fig. 8: VbDMF Flow View model

each *AtomicDASTask* of the VbDMF Flow View with the definition of a corresponding *DAO operation* of the VbDMF DAO View.

7.2 Integration

As explained in Section 3.3 views can be enriched by the mechanism of view integration. In this article we enhance the concept of view integration by introducing view integration paths, which we use to trace implementation details of process activities among different views.

In Figure 3, we have basically outlined the (directed) view integration dependencies between the different VbMF/ VbDMF views. By the mechanism of view integration, persistent data access activities of a business process can be integrated with their persistent data access implementation details. In example, the Collaboration DAO Mapping View integrates the Collaboration View and the DAO View. The DAO View, in turn can integrate two views, namely the ORM View and the Data Object Type View. Finally, the ORM View can integrate the Physical Data View and the Data Object View views. In order to check if a process activity reads or writes from a certain database table, the Flow View needs to be integrated with the Physical Data View. However, as depicted in Figure 3, these two views are not connected directly. Thus, we have to establish an integration path between these two views.

In order to establish such an integration path from the source view i.e. the Flow View to the target view i.e. the Physical Data View, many views need to connected. When integrating views to establish an integration path, the target view of the last view integration always becomes the source view of the next view integration. Within a view integration, we define the connection point in the source view as *start integration point* and the connection point in the target view as *end integration point*. In order to illustrate the concept of view integration paths, Figure 9 shows an integration path of four view integrations. The views are depicted in XMI notation.

– The first view integration combines the Flow View with the Collaboration DAO Mapping View. In this view integration, the entity *AtomicDASTask DASDelivery.insertDelivery* of the Flow View acts as start integration point S_1 and the entity *AtomicDASTask DASDelivery.insertDelivery* of the Collaboration DAO Mapping View acts as end integration point E_1) The Collaboration DAO Mapping View maps DAS operations to DAO operations e.g. it maps the DAS operation *DeliveryDAS.insertDelivery* of the Flow View to the DAO operation *DeliveryDAO.insert* of the DAO View. Instead of using the Collaboration DAO Mapping View, the Flow View can also be integrated with the DAO View directly by using the VbMF/VbDMF's mechanism of view integration. However,

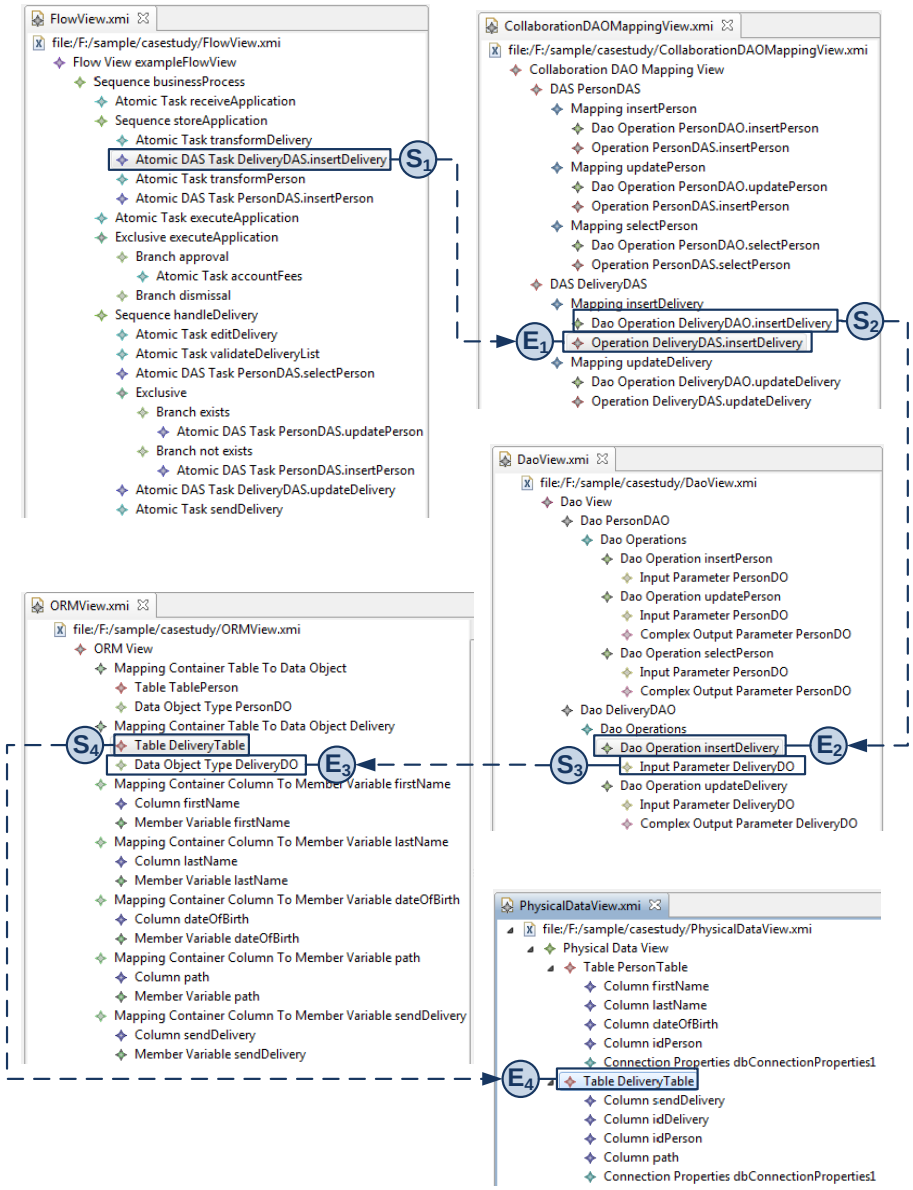


Fig. 9: VbDMF Integration Path

in this case, as we use a name-based matching algorithm for view integration, the DAO operations and the DAS operations would have to be named identically.

– The second view integration pair (S_2,E_2) combines the *AtomicDASTask DeliveryDAO.insertDelivery* entity of the Collaboration DAO Mapping View with the DAO operation *DeliveyDAO.insertDelivery* entity of the DAO View.

- The third view integration combines the DAO View with the ORM View in order to get information about the associated database tables. Each *DAO Operation* of the DAO View contains *DAO Input Parameter Types* and *DAO Output Parameter Types*. Each parameter type can be mapped to corresponding *Data Object Types* of the ORM View. In our example, the input parameter type *DeliveryDO* of the DAO View can be mapped to the correspondent entity *DeliveryDO* of the ORM View. In this view integration, the entity *DeliveryDO* of the DAO View acts as start integration point S_3 and the entity *DeliveryDO* of the ORM View acts as end integration point E_3.
- The fourth view integration pair (S_4,E_4) combines the *DeliveryTable* entity of the ORM View with the *DeliveryTable* entity of the Physical Data View. This is possible, because the ORM View maps *Tables* and *Table Columns* of the Physical Data View to *Data Object Types* and *Data Object Member Variables* of the Data Object Type View.

After illustrating the concept of view integration paths, we provide general definitions of the underlying terms.

Definition 1 *Let V_1 and V_2 be two views. If entity $S \in V_1$ matches entity $E \in V_2$ and entity $E \in V_2$ matches entity $S \in V_1$, then S is defined as the **start integration point** and E is defined as the **integration end point**. Then, V_1 is defined as the **source view** and V_2 is defined as the **target view** of this view integration.*

Definition 2 *Let V_1 be a view, and M_1 be the model of V_1, such as $V_1 = instanceOf(M_1)$. A start integration point $S_1 \in V_1$* corresponds *to an integration end point $E_1 \in V_1$ when one of the following conditions is true:*

- $S_1 \equiv E_1$
- *E_1 is a super element of S_1*
- *Let MC_1 be a mapping container entity $\in M_1$ with S_1 $instanceOf(MC_1)$ and $E_1 instanceOf(MC_1)$.*

Definition 3 *Let V_i, $i \in 1..n$ be n views. A **view integration path** is a tuple of entity pairs $P(S_i, E_i | i = 1..n-1, S_i \in V_i, E_i \in V_{i+1})$ that meets the following conditions: For each $i \in 1..n-1$ $\exists$ $S_i \in V_i$ that matches an integration end point $E_i \in V_{i+1}$, each integration end point $E_i \in V_{i+1}$ corresponds to a new start integration point $S_{i+1} \in V_{i+1}$.*

Algorithm 1: MatchFilterCriteria()

```
   Input: Entity integrationStartPoint ∈ FlowView
   Input: View searchView
   Input: Entity searchEntity
1  sourceView=FlowView;
2  if (NOT(searchView == NULL)) then
3      while (NOT sourceView.equals(searchView)) do
4          targetView = getNextView(sourceView, searchView);
5          integrationEndPoint = getIntegrationEndPoint(integrationStartPoint, targetView);
6          if (NOT(searchView.equals(targetView))) then
7              integrationStartPoint =
               RecursiveGetIntegrationStartPoint(integrationEndPoint, targetView);
8          sourceView = targetView;
9  return (RecursiveMatchEntity(integrationEndPoint, searchEntity));
```

Next, we present our algorithms used to implement the definitions above. The *MatchFilterCriteria* algorithm (see Algorithm 1) is the heart of our implementation solution. It checks if a certain process activity matches given filter criteria by implementing the concept of view integration paths. Algorithm 1 is a sub-algorithm of our recursive elimination algorithm *RecursiveClean* (Algorithm 4), which we will present in Section 7.3. Algorithm 1 consists of three basic functions:

- *getNextView(View sourceView, View targetView)* The function *getNextView* simply returns the next related view based on the *sourceView* and the *targetView*. The function returns the next view based on the view integration dependencies depicted in Figure 3. This function *NextEntity* is comparably simple and is not further illustrated.
- *getIntegrationEndPoint(Entity startEntity, View targetView)* In order to connect a source view with a target view, the start integration point of the source view need to be integrated with an end integration point of the target view. In our prototype implementation, the algorithm finds the corresponding integration end point in the target view based on name-matching [43]. As the name-matching algorithm is sufficient for our prototype implementation, we do not provide further implementations to find matching entities in the target view in this article.
- *RecursiveGetIntegrationStartPoint(Entity oldEntity, View sourceView)* Based on the end integration point of the previous view integration, this Algorithm 2 can calculate the start integration point of the next view integration. The target view of the last view integration becomes the source view of the next view integration. Thus, the old end integration point is in the same view as the new start integration point. Algorithm 2, the heart of our integration path calculation, requires the simple recursive sub-algorithm *RecursiveMatchEntity* (Algorithm 3) to check if the integration end point of the target view contains or matches given search criteria.

Three parameters are passed to Algorithm 1: A parameter *integrationStartPoint* of the Flow View, which initially is the persistent data access activity, and the filter criteria to be checked represented by the view *searchView* containing the entity *searchEntity*. As defined above, a view integration path consists of pairs of a start integration entity and end integration entity. Each start integration entity belongs to the source view and each end integration entity belongs to the target view. Accordingly, at first, a variable *sourceView* is initialized within the Flow View. As long as the current *sourceView* does not equal the *searchEntity* entity of the *searchView*, the functions *getNextView* and *getIntegrationEndPoint*, and *RecursiveGetIntegrationStartPoint* are invoked. The function *RecursiveGetIntegrationStartPoint* is invoked as long as the variable *targetView* does not equal the variable *searchView*.

In the following we describe the algorithm *RecursiveGetIntegrationStartPoint* (Algorithm 2) used to get the start integration point of the current view integration. The input parameters *currentEntity* and *targetView* are passed to Algorithm 2. According to Definition 2, there are three possibilities how to find a start integration point for the next view, the new start integration point either is the last end integration node, or the new start integration point is part of a *Matching Container*, or the new start integration point is a child of the last end integration node. According to the first possibility, the function *GetIntegrationEndPoint* checks if the current start integration point of the *currentView* matches a corresponding end integration point in the target view. If the start integration point matches a corresponding end integration point, a new start integration point is found. Otherwise, the new start integration point is either part of a *Matching Container* or it becomes a child entity of the current entity. In both cases, the *RecursiveGetIntegrationStartPoint* algorithm invokes itself recursively to check if the new start integration point matches an end integration point.

Algorithm 2: RecursiveGetIntegrationStartPoint()

```
   Input: Entity currentEntity
   Input: View targetView
1  integrationEndPoint=GetIntegrationEndPoint(currentEntity,targetView);
2  if NOT(integrationEndPoint == NULL) then
3     return integrationEndPoint;
4  else
5     if (hasChildren(currentEntity)) then
6        foreach (Entity childEntity ∈ entity.children()) do
7           return RecursiveGetIntegrationStartPoint(childEntity,targetView);
8     else
9        Entity parent = getParent(currentEntity);
10       if (parent instanceof MappingContainer) then
11          foreach (Entity childEntity ∈ parent.children()) do
12             if (NOT (childEntity.equals(currentEntity))) then
13                return childEntity;
14 return NULL;
```

Algorithm 3: RecursiveMatchEntity()

```
   Input: Entity currentEntity ∈ ViewcurrentView
   Input: Entity searchEntity ∈ ViewcurrentView
1  if (currentEntity.equals(searchEntity)) then
2     return TRUE;
3  if (hasChildren(currentEntity)) then
4     foreach Entity childEntity ∈ entity.children() do
5        RecursiveMatchEntity(childEntity);
6  return FALSE;
```

Finally, the simple recursive algorithm *RecursiveMatchEntity* (Algorithm 3) is invoked in order to check if the integration point in the target view matches the given search criteria. For this purpose, two parameters are passed to Algorithm 1. Firstly, the entity *currentEntity* is to be checked against certain filter search criteria. Secondly, the entity *searchEntity* specifies this filter criteria. The algorithm firstly checks if the entity *searchEntity* equals *currentEntity* by name-based matching. If this is true, the business process activity matches the filter criteria. If *currentEntity* has children, for each child, the algorithm invokes itself recursively.

An unsolved problem still is how to extract the persistent data access flows from the whole Flow View. In the following we present an algorithm calculating a flattened Flow View, defined as the DAS Flow View.

7.3 Extraction

In the following we present our algorithm used to extract the DAS Flow View from the whole Flow View. Due to our model-driven view-based approach we can filter data access activities by specific search criteria, such as tables, columns, DAOs, data objects etc. As a result our extracted persistent data access flows can contain only those activities accessing a specific table of a database.

Before we define the algorithm to extract persistent data access flows from the whole process flow, in the context of VbDMF, we provide the following definition:

Definition 4: *The VbDMF view incorporating the persistent data access flow is a* VbDMF DAS Flow View. *This VbDMF DAS Flow View is an extraction of the Flow View. This DAS Flow View only contains the* AtomicDASTasks *of the process flows. Each* AtomicDASTask *matches an associated* DAO Operation *of the DAO View.*

Algorithms for global data flow analysis fall into two major classes: iterative algorithms and elimination algorithms [8]. In iterative algorithms, the equations are repeatedly evaluated until the evaluation converges to a fixed point. Elimination algorithms compute the fixed point by decomposition and reduction of the flow graph to obtain subsequently smaller systems of equations. We settled for a recursive elimination algorithm and present our simple recursive elimination algorithm *RecursiveClean* (Algorithm 4) to extract the DAS Flow View from the Flow View. Algorithm 4 contains the sub-algorithm Algorithm 1 which is the heart of our recursive elimination algorithm *RecursiveClean*.

Algorithm 4: RecursiveClean()

```
   Input: Task task ∈ FlowView
   Input: View searchView
   Input: Entity searchEntity
 1 if (hasChildren(task)) then
 2     foreach Task childTask ∈ task.children() do
           /* only process non-data-related tasks                    */
 3         if (!(childTask instanceof AtomicDASTask)) then
 4             recursiveClean(childTask);
 5             if (NOThasChildren(childTask)) then
 6                 task.removeChild(childTask);
           /* only process data-related tasks                        */
 7         else if (MatchFilterCriteria(childTask, searchView, searchEntity)) then
 8             task.removeChild(childTask);
 9 else if (!(task instanceof AtomicDASTask)) then
10     task = NULL;
11 else if (MatchFilterCriteria(childTask, searchView, searchEntity)) then
12     task = NULL;
```

In the following we explain our recursive elimination algorithm *RecursiveClean* (Algorithm 4). The start *Tasks* of the Flow View are passed as mandatory input parameters to the algorithm. In addition, the optional input parameters *searchView* and *searchEntity* are passed to the algorithm in order to filter DAS operations by certain search criteria. After executing the algorithm, the persistent data access flow contains only those DAS operations matching the entity *searchEntity* of the view *searchView*. In order to filter persistent data access flows by more than one search criteria the algorithm can be performed repeatedly. If the input task *Task* has children, for all non-data-related entities, our recursive algorithm recursively steps into the different paths of the tree view. A task *Task* can have children if its type is of *Sequence*, *Parallel*, *Exclusive* or *Branch*. For each non-data-related *childTask* of the current task, the algorithm calls itself recursively. As explained before, a task is data-related if it is of type *AtomicDASTask*. When stepping through a certain path, only the non-data-related leaf-tasks are removed by recursion from the Flow View. Per default, tasks of type *AtomicDASTask* must not removed, because they are part of our resulting DAS Flow View. Likewise, tasks such as *Sequence*, *Parallel*, *Exclusive* and *Branch* containing data-related entities must not removed as well, because they are also part of the resulting DAS Flow

View. The algorithm *MatchFilterCriteria* (Algorithm 1) filters all data-related leaf-tasks of type *AtomicDASTask* provided that they do not match the search criteria. In order to check if the current leaf-tasks match the search criteria, the algorithm *MatchFilterCriteria* tries to establish a view integration path to the entity *searchEntity* of the *searchView*. Hereby the current leaf-tasks act as integration points. If a view integration path is found, Algorithm 1 returns true. The algorithm *MatchFilterCriteria* (Algorithm 1) has been described in detail before in Section 7.2.

This recursive elimination algorithm can be reused for extracting other views such as for extracting all service operations from the Flow View.

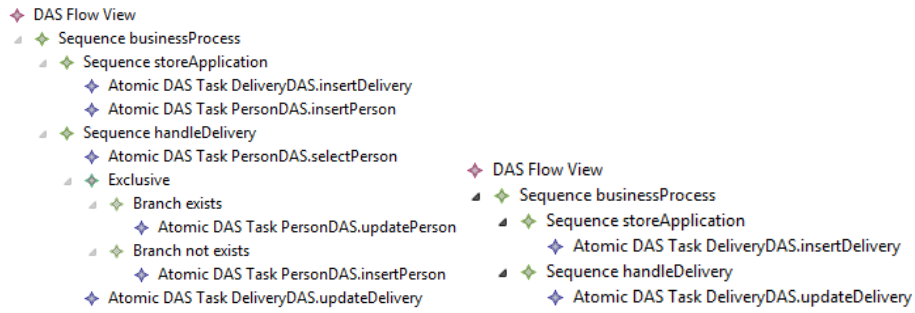
8 Applicability of the Algorithms & Tooling

In this section we show the applicability of the algorithms above and present a suitable tooling.

Firstly, we apply our algorithms to our process flow described in case study Section 4. We extract both a simple and a filtered persistent data access flow:

- Extract simple persistent data access flows: When stakeholders want to test persistent data access in process driven SOAs, they need a documentation about the persistent data access activities in the business process. Our simple persistent data access flow provides such a documentation. In the following we apply our algorithms to extract the DAS Flow View from the whole process flow. For this purpose, we invoke the recursive elimination algorithm *RecursiveClean* (Algorithm 4) with the start *Task* of the Flow View in Figure 4. The resulting Flow View of this algorithm is a DAS Flow View that only contains data access activities. We invoke the algorithm with the *NULL* value for the input parameters *searchView* and *searchEntity*. Figure 10a shows the XMI notation of the extracted DAS Flow View after invoking the algorithm.
- Extract filtered persistent data access flows: In case a deadlock occurs, data analysts want to check the business process for structural errors. For this purpose, they can extract all the data access activities that read or write from a specific database table. In our example the function *MatchFilterCriteria* of Algorithm 4 filters all data access activities that do not access a specific table *DeliveryTable*. To establish this, we set the input parameters *searchView* and *searchEntity* to the values *Physical Data View* and *Table* respectively. In addition, we set the attribute *Table.name* to *DeliveryTable*. By view integration, we can filter those data access activities not accessing the specific table *DeliveryTable*. For this purpose, the algorithm *RecursiveMatchEntity* (Algorithm 3) checks the *searchEntity* input parameter against the *DeliveryTable* entity. If the entity *DeliveryTable* matches the current *Table* entity, the concerned persistent data access activity is part of the resulting persistent data access flow. Otherwise the persistent data access activity is filtered from the Flow View. The resulting extracted DAS Flow View is shown in Figure 10b. As a result, only those data access activities accessing table *DeliveryTable* are part of the DAS Flow View.

Use Cases In the following, we give a few more use case examples fulfilled by stakeholders developing and maintaining applications in large-scale enterprises. If a certain use case occurs depends e.g. on the quality of the underlying business process and non-functional requirements e.g. the availability of external dependencies such as service providers and



(a) DAS Flow View (b) DAS Flow View (Table Delivery)

Fig. 10: Case Study: XMI notation of a simple and filtered persistent DAS Flow View

databases. These use cases mainly result from our study of analyzing data access in service-oriented environments in a large enterprise and secondly from analyzing literature in this field. They demonstrate how our persistent data access flows can be applied to specific analysis problems. Each of these use cases extracts a persistent data access flow from the whole process flow by different search criteria.

- In case a deadlock occurs, in addition to selecting all persistent data access activities accessing a specific database table, data analysts can further flatten the resulting persistent data access flow. In example, they can extract all the data access activities from the business process that read or write from a specific column of a database table.
- In case a specific database fails, stakeholders such as DAS developers need a documentation of which business process activities access a specific database. For this purpose, they need to extract all the data access activities that read or write from a specific database connection. In order to establish this, in addition to the previous four view integrations, the Physical Data View needs to be integrated with the *Database Connection View*.
- Let us consider the case that a certain service provider is shut down for any reason. Then, stakeholders such as system architects need to determine the business process activities invoking a service of the failed service provider. For this purpose, stakeholders can extract only those data access activities from the whole process flow which run on a certain URI *Service.Uri.name*. In order to establish this, the algorithm *MatchFilterCriteria* integrates the DAS Flow View with the *Collaboration View*.

Tooling In order to demonstrate applicability of our model-driven solution, we have integrated our persistent data access flow algorithms into the Eclipse-based [41] BPMS Intalio. Due to this tool integration, stakeholders can view the persistent data access flows and trace persistent data access details of a business process. In particular, we provide the following functionalities:

1. Add data access service views to the process flow. Figure 11 shows a new menu item in the process flow's context menu for adding relevant data access service views. After clicking this item, developers can select the VbDMF views, specifying the data access services of the business process, in a file chooser. Afterwards, a new directory (vb-dmf_*diagram_name*) with the selected views is created in the project folder. As a result, stakeholders can inspect the VbDMF views which are shown bottom right of Figure 12. By these views, stakeholders can view persistent data access details of persistent data

access activities such as physical storage tables, database connections, object-relational mappings, and data access object types. In Figure 12, a character is displayed top right of each view, which refers to a description below.

(a) The Collaboration View specifies the service operation definitions of the DAS operations.
(b) The DAO Collaboration Mapping View maps data access service (DAS) to underlying data access object (DAO) definitions.
(c) The DAO View models the underlying DAO operations of the DAS operations.
(d) The ORM View maps data object types of the Data Object Type View to physical database tables of the Physical Data View.
(e) The Data Object Type View specifies the data object types of the input and output parameters of the DAO View.
(f) The Physical Data View defines the tables and columns of an RDBMS and integrates the Database Connection View.

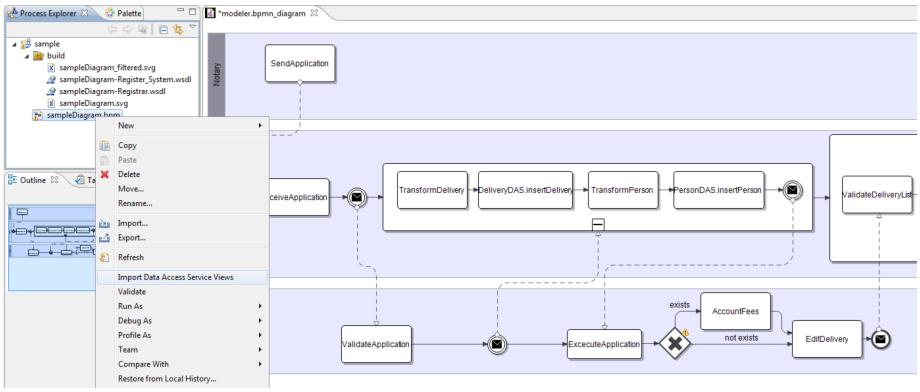


Fig. 11: Tooling: Import Data Access Service Views

2. Generate persistent data access flows Based on the selected views, developers can generate simple and filtered persistent data access flows of a process flow. In both cases, they use the filter view displayed in Figure 12. The Persistent Data Access Flow Filter View generates persistent data access flows for the selected process flow in the process explorer. In order to generate a simple persistent data access flow, stakeholders select the check box 'No Filter Criteria' and simply press the *Generate* button within the filter view. In order to produce a filtered persistent data access flow, stakeholders select filter criteria from the list. After pressing the button 'Generate', the recursive elimination algorithm *RecursiveClean* (Algorithm 4) is invoked with or without filter criteria arguments. In the example we set the filter criteria to *PhysicalDataView.table.name=DeliveryTable* to filter all persistent data access activities from the flow that do not match table *DeliveryTable*. As soon as the button 'Generate' is pressed, Algorithm 4 is invoked with the input parameter values *Physical Data View*, *Table*, and *DeliveryTable*. Top right of Figure 12 the resulting filtered persistent data access flow is shown that solely consists of the persistent data access activities reading or writing from table *DeliveryTable*.

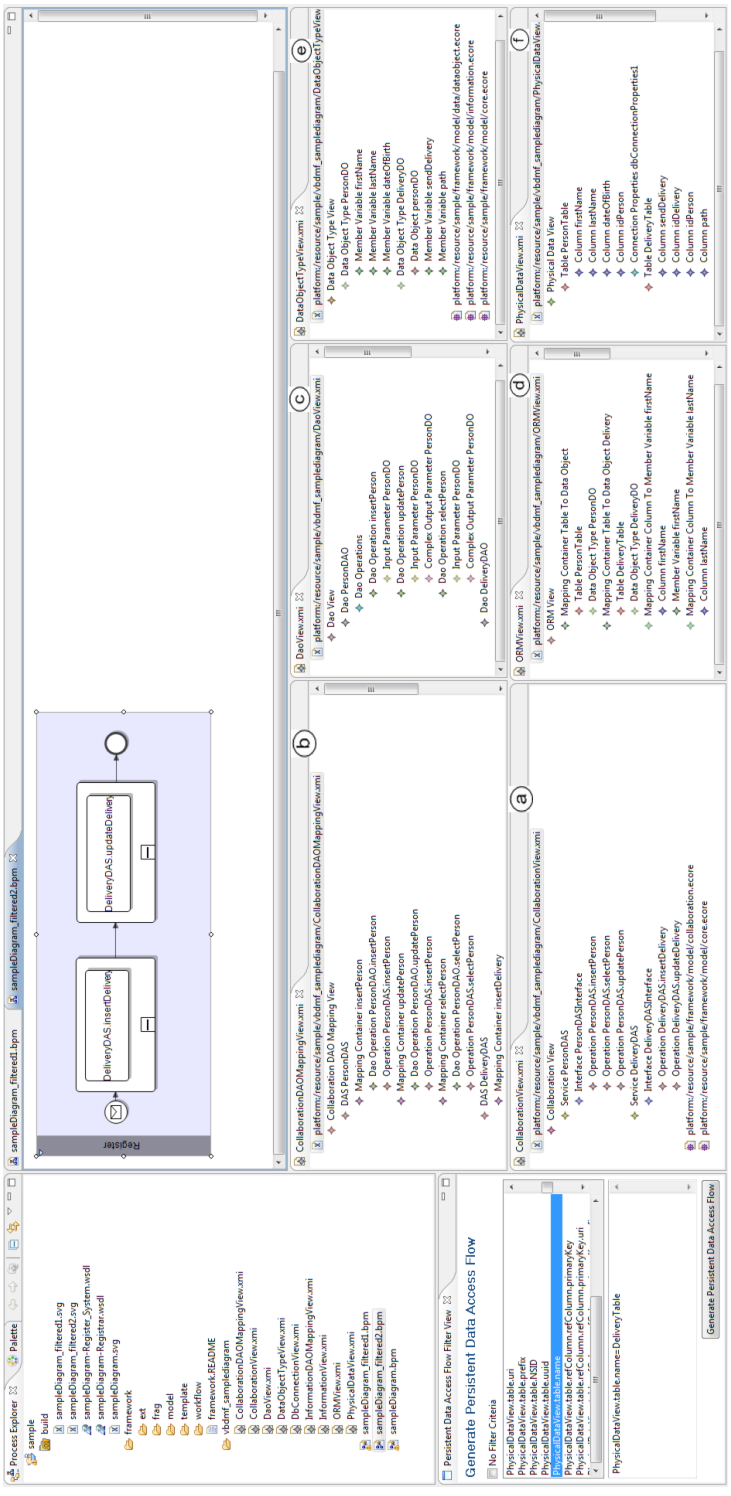


Fig. 12: Tooling: Generate Persistent Data Access Flows

Implementation details In the following we describe the implementation details of integrating the persistent data access flows into the Intalio BPMS. Figure 13 illustrates the necessary implementation steps. Each of the three implementation steps requires some input and generates output files. In the figure, the gray-labeled boxes depict the generated output files whereas the white-labeled boxes denote existing input (files).

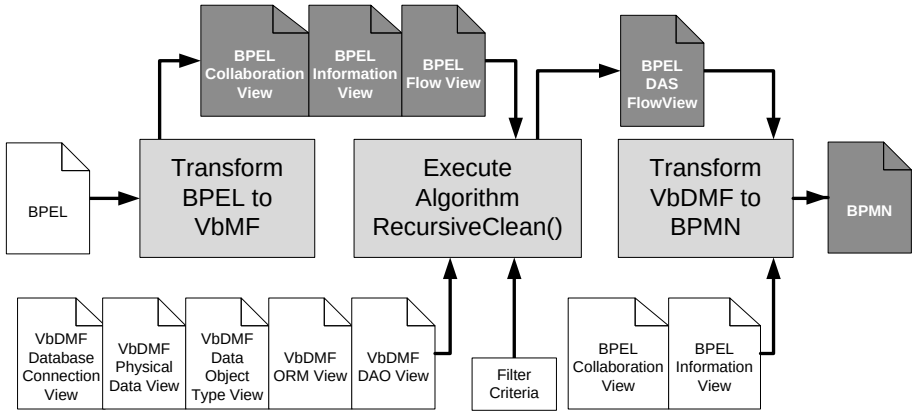


Fig. 13: Tool Integration

- **Transform BPEL to VbMF:** In order to being able to apply our algorithm to persistent data access flow, we need to transform the Intalio-generated BPEL flow into VbMF views. Whenever saving a BPMN diagram, the concerning BPEL source code is generated into the build folder of the project (shown top left of Figure 12). On top of this Intalio-generated BPEL code, we have implemented a java-based transformation that translates a BPEL file into VbMF views, namely the BPEL Flow View, BPEL Collaboration View, and BPEL Information View. After generating the views, they are also saved in folder vbdmf_*diagram_name*.
- **Execute Algorithm RecursiveClean():** The generated VbMF views BPEL Flow View, BPEL Collaboration View, and BPEL Information View as well as other data-related VbDMF views from the folder vbdmf_*diagram_name* are read by our recursive elimination algorithm *RecursiveClean*. In addition, the algorithm is fed with filter criteria specified in the Persistent Data Access Flow Filter View. The result of the algorithm is the BPEL DAS Flow View.
- **Transform VbDMF to BPMN:** Finally, we have to generate the BPMN code from the BPEL DAS Flow View. Besides the DAS Flow View, the transformation engine reads the BPEL Collaboration View and the BPEL Information View in order to transform BPEL messages and partner links to BPMN notation. In literature, there are very few approaches to map BPEL to BPMN. Weidlich et al. [47] discuss the limitations and pitfalls of such a BPEL-to-BPMN-mapping. However, in our prototype implementation we concentrate on mapping simple BPEL processes.

9 Evaluation

In this section we want to discuss both the correctness and complexity of the presented algorithms.

Correctness Here, we discuss the correctness of the algorithm *MatchFilterCriteria* (Algorithm 1) using induction. The *MatchFilterCriteria* algorithm is the heart of our recursive elimination algorithm that is used to implement our view integration path concept.

Hypothesis: *Let VT_i be the ith target view and VS_{i+1} be the (i+1)th source view of a view integration path. The algorithm is correct if the target view VT_i equals the source view VS_{i+1}* $\forall\ 2 < i < n$.

Algorithm 5 illustrates a reduced *MatchFilterCriteria* algorithm that contains the relevant lines of the while loop necessary to prove the hypothesis. In this reduced *MatchFilterCriteria* algorithm we use the following variables: VS_i denotes the ith source view of a view integration path, VT_i denotes the ith target view of a view integration path. Accordingly, $S_i \in VS_i$ denotes the start integration point of a view integration and $E_i \in VT_i$ denotes a end integration point of a view integration.

Algorithm 5: Reduced MatchFilterCriteria Algorithm

```
1 while (NOTVS_i.equals(searchView)) do
2     targetView = getNextView(VS_i, searchView);
3     E_i = getIntegrationEndPoint(S_i, targetView);
4     if (NOT(searchView.equals(VT_i))) then
5         S_{i+1} = RecursiveGetIntegrationStartPoint(E_i, VT_i);
6     VS_{i+1} = VT_i;
```

Let i be the number of while loop cycles of Algorithm 1. The number of while loop cycles equates the number of views in the view integration path. $\forall 0 < i < 2$ the hypothesis is false, because a view integration path must have at least two views in order to fulfill the hypothesis:

1. $i = 1 : S_1 \in VS_1, T_1 = NULL$
2. $i = 2 : S_1 \in VS_1, T_1 \in VT_1$

Base Case: $i = 3 : S_1 \in VS_1,\ T_1 \in VT_1,\ S_2 in VS_2,\ VS_2 = VT_1,\ T_2 \in VT_2$

Inductive Step: *Let $VS_i = VT_{i-1}$ be true* $\forall\ 2 < i < n$:
$S_1 \in VS_1,\ T_1 \in VT_1,\ ..,\ S_{n-1} \in VS_{n-1},\ T_{n-1} \in VT_{n-1},\ S_n in VS_n,\ VS_n = VT_{n-1},\ T_n \in VT_n$.

Now, we show that the hypothesis is true $\forall\ 2 < i < n+1$:

$S_1 \in VS_1,\ T_1 \in VT_1,\ ..,\ S_{n-1} \in VS_{n-1},\ T_{n-1} \in VT_{n-1},\ S_n in VS_n,\ VS_n = VT_{n-1},\ T_n \in VT_n,$
$S_(n) \in VS_n,\ T_(n) \in VT_n,\ S_{n+1} in VS_{n+1},\ VS_{n+1} = VT_n,\ T_{n+1} \in VT_{n+1}$
From this it follows that $\forall\ 2 < i < n+1$: $VS_{i+1} = VT_{(i+1)-1} = VT_i$. Hereby we have proven that our hypothesis is true.

Complexity: In the following we quantitatively measure the complexity of the presented algorithms using the Big O notation. We evaluate each of our algorithms separately before we will derive the overall performance from the parts.

Formally, the algorithm $f(n)$ is equivalent to $O(g(n))$ for all $n > 0$, if there exists a constant $c > 0$, such that $f(n) = c * g(n)$. Table 2, Table 3, Table 4, and Table 4 summarize the complexity of the presented algorithms. The complexity of each algorithm is presented in a separate table. In each table, the line number, the complexity of each line, and the maximum number of invocations are displayed. First, in Table 2 and Table 3 the complexity of the algorithms *RecursiveGetIntegrationStartPoint* and *RecursiveMatchEntity* are illustrated. As these two algorithms are invoked by algorithm *MatchFilterCriteria*, next, Table 4 shows the complexity of the algorithm *MatchFilterCriteria*. Finally, Table 5 displays the complexity of the algorithm *RecursiveClean* which invokes the algorithm *MatchFilterCriteria*. In the sub tables, we use the following literals: v to refer to the number of views in a view integration path, n to denote the number of elements within a view, and the constant k to denote the number of child elements within an integration element within a view. For example, The input parameter *DeliveryDO* of the DAOView is a child entity of the *DeliveryDAO.insert* integration point. The number of persistent data access activities within the Flow View is denoted by d.

Table 2: Complexity of Algorithm RecursiveGetIntegrationStartPoint (Algorithm 1)

Line #	Line of Algorithm	Complexity of Line	Max. # of Invocations
1	*integrationEndPoint=GetIntegrationEndPoint (currentEntity,targetView)*	O(n)	k
2	If(NOT(*integrationEndPoint == NULL*))	O(1)	k
3	RETURN (integrationEndPoint)	O(1)	k
4	Else		
5	If (*hasChildren*(*currentEntity*))	O(1)	k
6	ForEach(*Entity childEntity* ∈ *entity.children*())	O(1)	k
7	RETURN (*RecursiveGetIntegrationStartPoint (childEntity,targetView)*)	O(1)	k
8	Else		
9	*Entity parent* = *getParent*(*currentEntity*)	O(1)	k
10	If(*parent* instanceof *MappingContainer*)	O(1)	k
11	ForEach(*Entity childEntity* ∈ *parent.children*())	O(1)	k
12	If(NOT (*childEntity.equals*(*currentEntity*)))	O(1)	k
13	RETURN *childEntity*	O(1)	k
14	RETURN NULL	O(1)	k

Table 2 depicts the complexity of the recursive algorithm *RecursiveGetIntegrationStartPoint* (Algorithm 2). The algorithm *RecursiveGetIntegrationStartPoint* checks each entity of the view if if matches the current entity *currentEntity*. Thus, the function *RecursiveGetIntegrationStartPoint* is of linear complexity $O(n)$ and is invoked at most m times, whereas m corresponds to the number of child elements of entity currentEntity. However, the number of child elements m is not dependent on the number of process elements, because it is a constant factor. Therefore, the next statements are also of constant complexity. As a result, the overall performance of the algorithm *RecursiveGetIntegrationStartPoint* is linear.

Table 3: Complexity of Algorithm RecursiveMatchEntity (Algorithm 2)

Line #	Line of Algorithm	Complexity of Line	Max. # of Invocations
1	If ($currentEntity.equals(searchEntity)$)	O(1)	k
2	RETURN TRUE	O(1)	1
3	If ($hasChildren(currentEntity)$)	O(1)	k
4	Else		
5	ForEach($Entity\ childEntity \in entity.children()$)	O(1)	k
6	$RecursiveMatchEntity(childEntity)$	O(1)	k
7	RETURN *FALSE*	O(1)	1

Table 4: Complexity of Algorithm MatchFilterCriteria (Algorithm 4)

Line #	Line of Algorithm	Complexity of Line	Max. # of Invocations
1	$sourceView = FlowView$	O(1)	1
2	if ($NOT\ searchView == NULL$)	O(1)	1
3	while ($NOT\ sourceView.equals(searchView)$)	O(1)	v
4	$targetView = getNextView(sourceView, searchView)$	O(1)	v-1
5	$integrationEndPoint = getIntegrationEndPoint$ $(integrationStartPoint, targetView)$	O(n)	v-1
6	if ($NOT\ (searchView.equals(targetView))$) then	O(1)	v-1
7	$integrationStartPoint = RecursiveGetIntegrationStartPoint$ $(integrationEndPoint, targetView)$	O(1)	v-2
8	$sourceView = targetView$	O(1)	v-1
9	RETURN ($RecursiveMatchEntity$ $(integrationEndPoint, searchEntity)$)	O(1)	1

Table 5: Complexity of Algorithm RecursiveClean (Algorithm 3)

Line #	Line of Algorithm	Complexity of Line	Max. # of Invocations
1	If (($hasChildren(task)$))	O(1)	n
2	ForEach(Task $childTask \in task.children()$)	O(1)	n
3	If((!($childTask$ instanceof $AtomicDASTask$)))	O(1)	n
4	$recursiveClean(childTask)$	O(1)	n-d
5	If(($NOT\ hasChildren(childTask)$))	O(1)	n-d
6	$task.removeChild(childTask)$	O(1)	n-d
7	ElseIf($MatchFilterCriteria$) $(childTask, searchView, searchEntity)$)	O(d) O(d)	d d
8	$task.removeChild(childTask)$	O(1)	d
9	ElseIf (!($task$ instanceof $AtomicDASTask$))	O(1)	n-d
10	$task = NULL$	O(1)	n-d
11	ElseIf($MatchFilterCriteria$) $(childTask, searchView, searchEntity)$)	O(d)	d
12	$task = NULL$	O(1)	d

Table 3 summarizes the complexity of the algorithm *RecursiveMatchEntity* (Algorithm 2). As an entity has a constant number of child entities e.g. the entity *Table* has a constant number of *Columns*. Therefore, the algorithm *RecursiveMatchEntity* is of constant complexity.

Table 4 illustrates the complexity of Algorithm 4. The function *getNextView*, that is not further specified, is of constant complexity, because, according to Figure 3, it simply re-

turns the next view by the current view. In contrast to the function *getNextView*, the function *getIntegrationEndPoint* is dependent of the number of process elements within a view. The function *getIntegrationEndPoint* determines a matching element in the target view, that is the end integration point. Thus, the response time of this function grows linearly with the number of view elements. The function *RecursiveMatchEntity* checks if the current entity matches the search criteria. This function is also of constant complexity. As a result, Algorithm 4 has an linear overall performance.

Table 5 shows the complexity of the algorithm *RecursiveClean*. Each line in the algorithm is invoked linearly with the number of process activity in the business process. In particular, line 7 and 11 are invoked linearly with the number of persistent data access activities within the business process. the lines 8 and 12 are only invoked if the persistent data access activities do not match the filter criteria. Thus, the overall performance of our recursive elimination algorithm *RecursiveClean* for the number of persistent data access activities $d > 0$ is $O(d^2)$. If the number of persistent data access activities within the business process $d = 0$, the worst case response time of the recursive elimination algorithm *RecursiveClean* grows solely linear with the number of process activities $O(n)$.

In the following we summarize the resulting complexity of the algorithms.

- RecursiveGetIntegrationStartPoint: $k * O(n) + 11 * k * O(1) \equiv O(n)$
- RecursiveMatchEntity: $4 * k * O(1) + 2 * O(1) = (2 * (2k+1)) * O(1) \equiv O(1)$
- MatchFilterCriteria: $(v-1) * O(n) + v * O(1) + 2 * (v-1) * O(1) + (v-2) * O(1) + 3 * O(1)$
  $= (v-1) * O(n) + (4v-1) * O(1) \equiv O(n)$
- RecursiveClean:
  for $d = 0$: $3n * O(1) + 5 * (n-d) * O(1) + 2d * O(d) + 2d * O(1) \equiv 3n * O(1) + 5 * (n) * O(1) \equiv O(n)$

  for $d = n$: $3n * O(1) + 5 * (n-d) * O(1) + 2d * O(d) + 2d * O(1) \equiv 3d * O(1) + 2d * O(d) + 2d * O(1)$
  $= 5d * O(1) + 2d * O(d) \equiv O(d^2)$

  for $0 < d < n$: $3n * O(1) + 5 * (n-d) * O(1) + 2d * O(d) + 2d * O(1) \equiv 3 * O(n) + 5 * O(n-d) + 2 *$
  $O(d) + 2 * O(d^2) \equiv O(n) + O(d^2)$

Today, XML is a popular standard data exchange format. Thus, in literature, there is a variety of more efficient XML structural matching techniques [1,3]. By using these structural matching techniques, the worst case complexity of our recursive elimination algorithm $O(n) + O(d^2)$ for $d > 0$ can be reduced. However, the aim of this section is to quantitatively show the feasibility and applicability of our approach, which has been achieved well.

10 Discussion

Different stakeholders such as business experts, architects, and developers have different requirements to a software system. According to the pattern of separation of concerns [12], appropriate views must be provided to the different stakeholders. However, in addition to these views, read-only sub-views extracted from these rich views can facilitate tasks such as developing, and testing. Thus, besides view model extension and view integration, we introduced a further mechanism in order to generate a resulting view: The mechanism to extract views from existing views. In this connection we need to distinguish between editable and read-only views. The DAS Flow View is an example of such a read-only view. The DAS Flow View cannot be specified at modeling time, because usually connections have to be modeled in the context of the whole business flow. Hence, these extracted views are typically

read-only views. Moreover, our DAS Flow View can be generated from the Flow View on the fly. Thus the DAS Flow View does not have to be stored after adapting the corresponding Flow View. This concept is comparable to the view concept in database theory: A database view can output data stored in one more database tables. When data in one of these database tables changes, the database view can output the updated data by accessing them through the tables. The disadvantage of this on-the-fly-generation is that the generation procedure needs to be performed each time when selecting the DAS Flow View.

To the best of our knowledge, up-to-now these persistent persistent data access flows are not used to solve development, testing and analysis problems, yet. With this article, our goal is to present a visual solution for a series of persistent data access problems. Accordingly, the specified use cases in Section 6 are just examples of how our approach can be applied. Accordingly, in Section 6.1 we discover deadlocks by ensuring whether the DAS operations are properly designed. However with our approach we do not claim to discover a new approach for detecting deadlocks. The potential deadlock cause of two process-instances invoking intersecting DAS operations presented, is just one of several possible causes for a deadlock. Other mistakes such as an incorrect transaction handling or database configuration can also increase the probability of a deadlock. Instead, our persistent data access flow shall ease both manual and automatic deadlock detection in a complex process model. On top of our approach existing data analysis solutions such as deadlock detection techniques can be applied.

In the following we shortly state how our approach reduces the complexity of the process in the context of the three presented use cases. Hereby we use the definitions for process complexity specified in [4]. Four main metrics can be identified to measure the complexity of a process: activity complexity, control flow complexity, data-flow complexity, and resource complexity. The activity complexity of the process simply calculates the number of activities a process has. The control flow behavior of a process is affected by process constructs such as splits, joins, loops, and ending and starting points. The data-flow and resource complexity perspectives measure the complexity of data structures and the diversity of resources respectively. With our approach, according to the concept of separation of concerns, we could reduce the number of activities of a flow. We achieved this by extracting persistent data access flows consisting of simply data access activities. Thus we reduced the activity complexity of the process. Furthermore in the database testing use case, we resolved one complex problem into a number of simpler problems by extracting the data paths from a whole process flow. In this use case we could also decrease the control flow complexity to a minimum value. This is because a data path contains no switch constructs. By our filtering mechanism, we could also reduce the data-flow complexity and resource-complexity of business processes.

11 Conclusion and Future Work

Process flows contain different types of activities such as business logic activities, transformation activities, and persistent data access activities. When the number of activities grows, focusing on special types of activities of the process flow such as the persistent data access activities is a time-consuming task. In this work we presented a view-based, model-driven solution extracting persistent data access flows from the whole process flow. By using these persistent data access flows, different stakeholders such as data analysts, DAS developers, and database testers can focus on the persistent data access activities of the process flows and to solve structural problems in business processes. We illustrated how our tailored DAS

Flow View concept can improve data analysis, development, and testing by presenting selected use cases. Each of these use cases is an example of how persistent data access flows can increase efficiency and decrease the time to solve certain problems at the earliest state of development. Our DAS Flow View can be further tailored by different filter criteria such that the flow contains only those persistent data access activities reading or writing data from a specific database table. We have demonstrated the applicability of our approach by a suitable tooling. Furthermore, we have evaluated the feasibility by showing the correctness and complexity of the presented algorithms. Apart from focusing on the persistent data access activities, our approach can be generally applied to focus on any particular parts of the business process in a process-driven SOA.

However, further work is necessary to coping with other important requirements. As the tools are what gives value to a concept, we continue focusing on developing suitable tooling for persistent data access flows. Besides modeling data access activities, we will describe other important activities of the business process in more detail. Accordingly, we concern ourselves with activity management e.g. specifying general activity description interfaces and categorizing different types of activities, and developing new views. Furthermore, we will focus on source code re-engineering in order to being able to exploit our approach when no view model instances are available.

Acknowledgement This work was supported by the European Union FP7 projects COMPAS, grant no. 215175, and INDENICA, grant no. 257483.

References

1. Al-Khalifa, S., Jagadish, H.V., Patel, J.M., Wu, Y., Koudas, N., Srivastava, D.: Structural joins: A primitive for efficient xml query pattern matching. In: Agrawal, R., Dittrich, K.R. (eds.) ICDE. pp. 141–152. IEEE Computer Society (2002)
2. Awad, A., Puhlmann, F.: Structural detection of deadlocks in business process models. In: BIS. pp. 239–250 (2008)
3. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal xml pattern matching. In: SIGMOD Conference. pp. 310–321 (2002)
4. Cardoso, J.: Control-flow Complexity Measurement of Processes and Weyuker's Properties. In: 6th International Enformatika Conference. pp. 213–218. Transactions on Enformatika, Systems Sciences and Engineering, Vol. 8 (2005)
5. Carey, M.J., Reveliotis, P., Thatte, S., Westmann, T.: Data service modeling in the aqualogic data services platform. In: SERVICES I. pp. 78–80 (2008)
6. Database, J.S.T.: The java database connectivity (jdbc). http://java.sun.com/javase/technologies/database/ (2001)
7. Dedene, G., Snoeck, M.: Formal deadlock elimination in an object oriented conceptual schema. Data Knowl. Eng. 15(1), 1–30 (1995)
8. Duesterwald, E., Gupta, R., Soffa, M.L.: A practical framework for demand-driven interprocedural data flow analysis. ACM Trans. Program. Lang. Syst. 19(6), 992–1030 (1997)
9. Eclipse: Eclipse Modeling Framework Project. http://www.eclipse.org/modeling/emf/ (Retrieved October, 2011)
10. Fischer, S., Kuchen, H.: Data-flow testing of declarative programs. In: ICFP. pp. 201–212 (2008)
11. Georgakopoulos, D., Hornick, M.F., Sheth, A.P.: An overview of workflow management: From process modeling to workflow automation infrastructure. Distributed and Parallel Databases 3(2), 119–153 (1995)
12. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall, Englewood Cliffs (1991)
13. Group, O.M.: MOF 2.0 / XMI Mapping Specification, v2.1.1. http://www.omg.org/technology/documents/formal/xmi.htm (January 2010)
14. Habich, D., Richly, S., Preissler, S., Grasselt, M., Lehner, W., Maier, A.: Bpel-dt - data-aware extension of bpel to support data-intensive service applications. In: WEWST (2007)

15. Harrold, M.J.: Testing: a roadmap. In: ICSE '00: Proceedings of the Conference on The Future of Software Engineering. pp. 61–72. ACM, New York, NY, USA (2000)
16. Hentrich, C., Zdun, U.: Patterns for business object model integration in process-driven and service-oriented architectures. In: PLoP '06: Proceedings of the 2006 conference on Pattern languages of programs. pp. 1–14. ACM, New York, NY, USA (2006)
17. Hentrich, C., Zdun, U.: Patterns for process-oriented integration in service-oriented architectures. In: EuroPLoP. pp. 141–198 (2006)
18. IBM: Websphere mq workflow. http://www-01.ibm.com/software/integration/wmqwf/ (Retrieved January 2012)
19. Intalio: Bpm. http://www.intalio.com/bpm (Retrieved January 2012)
20. Isloor, S., Marsland, T.: The deadlock problem: An overview. Computer 13(9), 58–78 (1980)
21. JBoss Community: Jboss messaging. http://www.jboss.org/jbossmessaging (Retrieved January 2012)
22. JBoss Community: jbpm. http://www.jboss.org/jbpm (Retrieved January 2012)
23. Kurz, S., Guppenberger, M., Freitag, B.: A uml profile for modeling schema mappings. In: ER (Workshops). pp. 53–62 (2006)
24. Lang, N.: Schlaer-mellor object-oriented analysis rules. SIGSOFT Softw. Eng. Notes 18(1), 54–58 (1993)
25. Le, W., Soffa, M.L.: Refining buffer overflow detection via demand-driven path-sensitive analysis. In: PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. pp. 63–68. ACM, New York, NY, USA (2007)
26. Lethbridge, T.C., Singer, J., Forward, A.: How software engineers use documentation: The state of the practice. IEEE Softw. 20(6), 35–39 (2003)
27. Mayr, C., Zdun, U., Dustdar, S.: Model-driven integration and management of data access objects in process-driven soas. In: ServiceWave '08: Proceedings of the 1st European Conference on Towards a Service-Based Internet. pp. 62–73. Springer-Verlag, Berlin, Heidelberg (2008)
28. Mayr, C., Zdun, U., Dustdar, S.: View-based model-driven architecture for enhancing maintainability of data access services. Data Knowl. Eng. 70, 794–819 (2011)
29. Naik, M., Park, C.S., Sen, K., Gay, D.: Effective static deadlock detection. In: ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering. pp. 386–396. IEEE Computer Society, Washington, DC, USA (2009)
30. Network, S.D.: Core J2EE Pattern Catalog. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html (Copyright 1994-2008 Sun Microsystems, Inc)
31. OASIS Web Services Business Process Execution Language (WSBPEL) TC: Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (April 2007)
32. Object Management Group (OMG): Business process model and notation (bpmn) version 2.0. http://www.omg.org/spec/BPMN/2.0 (Release Date January 2011)
33. (OMG), O.M.G.: Unified modeling language. http://www.uml.org/ (Retrieved January, 2012)
34. Palkovits, S., Wimmer, M.: Processes in e-government - a holistic framework for modelling electronic public services. In: Traunmüller, R. (ed.) EGOV. Lecture Notes in Computer Science, vol. 2739, pp. 213–219. Springer (2003), http://dblp.uni-trier.de/db/conf/egov/egov2003.html#PalkovitsW03
35. Rapps, S., Weyuker, E.J.: Data flow analysis techniques for test data selection. In: ICSE. pp. 272–278 (1982)
36. Reddy, P.K., Bhalla, S.: Deadlock prevention in a distributed database system. SIGMOD Rec. 22(3), 40–46 (1993)
37. Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow data patterns: Identification, representation and tool support. In: ER. pp. 353–368 (2005)
38. Sadiq, W., Orlowska, M.E.: Applying graph reduction techniques for identifying structural conflicts in process models. In: In Proceedings of the 11th Conf on Advanced Information Systems Engineering (CAiSE'99. pp. 195–209. Springer-Verlag (1999)
39. Schmit, B.A., Dustdar, S.: Model-driven development of web service transactions. In: In Proceedings of the Second GI-Workshop XML for Business Process Management,Mar. p. 2005 (2005)
40. Software AG: Webmethods bpms. http://www.softwareag.com/at/products/wm/bpm/default.asp (Retrieved January 2012)
41. The Eclipse Foundation: Eclipse. http://www.eclipse.org/ (2012)
42. TIBCO: Bpm. http://www.tibco.com/products/bpm/ (Retrieved January 2012)
43. Tran, H., Zdun, U., Dustdar, S.: View-based and model-driven approach for reducing the development complexity in process-driven SOA. In: Abramowicz, W., Maciaszek, L.A. (eds.) Business Process and Services Computing: 1st International Conference on Business Process and Services Computing (BPSC'07), September 25-26, 2007, Leipzig, Germany. LNI, vol. 116, pp. 105–124. GI (2007)

44. Turner, M., Budgen, D., Brereton, P.: Turning software into a service. Computer 36, 38–44 (2003)
45. Völter, M., Stahl, T.: Model-Driven Software Development: Technology, Engineering, Management. Wiley (2006)
46. Wang, J., Yu, A., Zhang, X., Qu, L.: A dynamic data integration model based on soa. In: ISECS International Colloquium on Computing, Communication, Control, and Management. pp. 196 – 199. IEEE Computer Society, Washington, DC, USA (2009)
47. Weidlich, M., Decker, G., Großkopf, A., Weske, M.: Bpel to bpmn: The myth of a straight-forward mapping. In: OTM Conferences (1). pp. 265–282 (2008)
48. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer (2007)
49. Zhang, G., Fu, X., Song, S., Zhu, M., Zhang, M.: Process driven data access component generation. In: DEECS. pp. 81–89 (2006)
50. guang Zhang, X.: Model driven data service development. In: ICNSC'08. pp. 1668–1673 (2008)
51. Zhou, Y., Lee, E.A.: A causality interface for deadlock analysis in dataflow. In: EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software. pp. 44–52. ACM, New York, NY, USA (2006)