Working Papers

ERCIS - European Research Center for Information Systems

Working Paper No. 38

Data-Flow Analysis of BPMN-Based Process-Driven Applications: Detecting Anomalies across Model and Code

Konrad Schneid, Sascha Di Bernardo, Herbert Kuchen, Sebastian Thöne

ISSN 1614-7448

cite as: Konrad Schneid, Sascha Di Bernardo, Herbert Kuchen, Sebastian Thöne: Data-Flow Analysis of BPMN-Based Process-Driven Applications: Detecting Anomalies across Model and Code. In: Working Papers, European Research Center for Information Systems No. 38. Eds.: Becker, J. et al. Münster 2021.

Contents

List of Figures

List of Tables

Working Paper Sketch

Type

Research Report

Title

Data-Flow Analysis of BPMN-Based Process-Driven Applications: Detecting Anomalies across Model and Code[1].

Authors

Konrad Schneid, Sascha Di Bernardo, Herbert Kuchen, Sebastian Thöne

Abstract

Process-Driven Applications (PDA) combine Business Process Management and less-code approaches. They are typically based on executable process models, human tasks, and adapter code to external software services. Process data is shared across these artifacts, managed by a process engine. Therefore process engineers and programmers must ensure the correct data flow within these components, but also in between. However, previous approaches have only considered the data-flow analysis of those components separately. This paper provides a concept for detecting data-flow anomalies in BPMN-based Process-Driven Applications across all artifacts. The main idea is to create a single Data-Flow Analysis (DFA) graph based on the process model's abstract syntax. Call graphs representing the internal flows of the referenced source code are transformed and merged into the DFA graph. Then, the resulting graph is extended by labels indicating data-object operations occurring at its nodes. Eventually, a combined forward and backward analysis is performed to uncover data-flow anomalies as indicators of potential errors. The analysis concept was implemented as a prototype designed for the Camunda BPM Framework, proving its practicality in several case studies.

Keywords

Process-Driven Application, Data-Flow Anomalies, Control-Flow Graph Analysis, BPMN

[1] This paper is an extended version of our work published in the SAC 2021 conference proceedings [17].

1 Introduction

The ISO standard BPMN (Business Process Model and Notation) maintained by the Object Management Group (OMG) is the de-facto standard for modeling business processes [10, 13]. By being easy to understand without extensive prior knowledge, BPMN provides a common language for business experts and software developers, resulting in improved business-IT alignment. BPMN models can be enriched with technical attributes, allowing them to be executed by process engines such as Camunda BPM or Activiti [10, 1]. The resulting systems called Process-Driven Applications play an important role, especially in enterprises practicing Business Process Management (BPM). This kind of model-driven development combines the strengths of BPM and less-code approaches, leading to faster development processes [21].

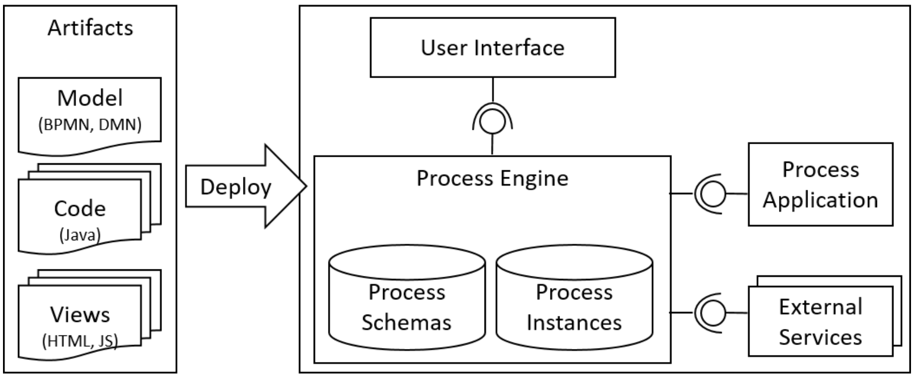


Figure 1: Artifacts and Runtime Components of a PDA.

Besides the mentioned process engine and the machine-interpretable process models, source code and form views are essential components of a PDA, as illustrated in Figure 1. In this context, source code is typically linked in BPMN service tasks to invoke external implementations of automated steps. Service tasks are also frequently used to integrate third-party systems such as ERP solutions this way. Forms, e.g., in HTML, can be referenced in the process model for user tasks. In addition to the BPMN model, (DMN-) decision tables can be applied to decouple business rules from the overall process model. The process engine's task is to coordinate the deployed process according to the model schema. Assigned user tasks can be accessed and completed via a kind of task inbox provided by the workflow engine. If the process arrives at a service task, the linked source code is called and executed. A process application can serve as a wrapper and takes care of the application startup and shutdown.

Process models in BPMN are defining both the control flow and the data flow. Basically, a BPMN model consists of the following element types: Events are depicted by small circles, while tasks are visualized by rounded rectangles. Diamond shapes illustrate gateways for branching and merging the flow. Data objects are represented by document icons associated with tasks as their input or output, respectively [3, 13].

A simplified example of a BPMN process model for treating incoming invoices is shown in Figure 2. Once the invoice is received, it is scanned and uploaded as part of a human task. In a subsequent service task, the PDF document is analyzed using OCR, transferring its metadata to an invoice data object. The invoice is checked for correctness based on the scanned invoice data and the corresponding order information. If the invoice is approved, an automated bank transfer is initiated. Otherwise, a correct invoice is requested from the supplier and submitted to the clerk for rechecking. This simple process model is used as a running example in this paper and already contains three data flow anomalies:

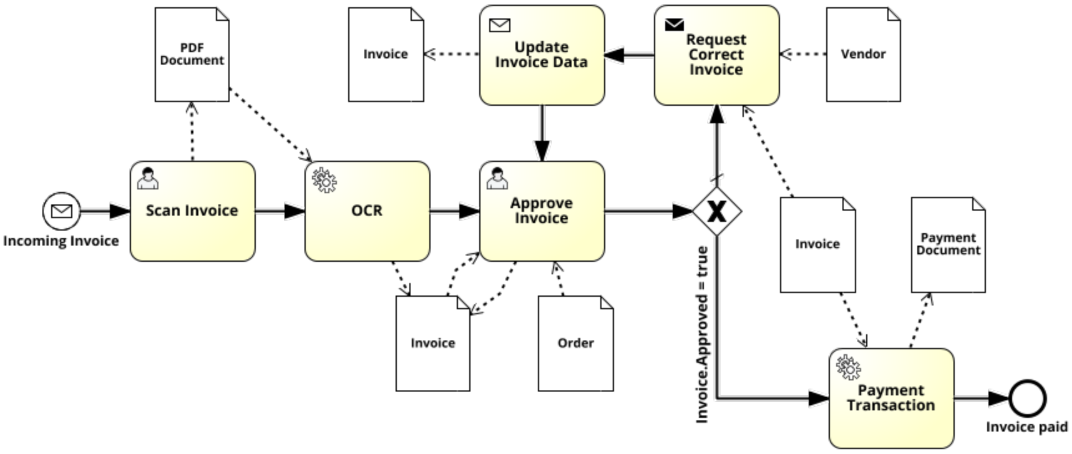


Figure 2: BPMN Example of Treating Incoming Invoices with Data-Flow Anomalies.

1. The user task *approve invoice* attempts to read the data object *order* to compare this information with the invoice data. However, the data object *order* has not been initialized before and consequently does not contain any information. Note: The BPMN specification provides the element type *data input* for such cases. To correct this mistake, the modeler could declare the data object *order* as this element type. In that case, the task is not started until the linked data input is available, which would avoid the error-constellation mentioned before.

2. A similar scenario is presented in the task *request correct order*. In this activity, an attempt is also made to read a previously uninitialized data object (here: *vendor).*

3. During the service task *payment transaction*, the data object *payment document* is created. However, this information is not used in the entire process. It remains doubtful whether this behavior is intended. Data resulting from process steps can also be specified as *data output* according to the BPMN metamodel. Nevertheless, it is not clear if and how the resulting data object will be processed further.

These error types should be identified as early as possible in the development process, ideally through static analysis techniques. Especially when practicing continuous software engineering, such static checks can be integrated into continuous integration/delivery pipelines. This way, the data-flow analysis runs every time changes on the PDA artifacts (e.g., process model, source code, etc.) are pushed into the common repository. We have identified the following three types of anomalies as good indicators for potential faults:

Undefined-Read Anomaly: A data object is read without a previous assignment of some initial value, as in case (1) and (2) of our running example.

Never-Used Anomaly: A data object is written, but the definition is never read before the data object is updated or the process ends. The latter case corresponds to case (3) of our example.

Undefined-Undefined Anomaly: The BPMN specification does not cover delete accesses to data objects. However, process engines such as Camunda offer such options [5]. This gives the possibility to attempt to delete a previously deleted or never before initialized data object.

Another anomaly we identified but did not yet cover in this paper is the treatment of race condition anomalies [9]. These anomalies occur due to common accesses to data objects in concurrent sections, such as in defined parallel threads or through parallel gateways in BPMN models. Due to race conditions, different results may be obtained depending on the process's execution sequence. The support of detecting race conditions is work in progress and will be presented in a future paper.

An additional challenge arises from the practice that process engineers often do not explicitly model the data objects and the process model's data flow. This is also driven by the fact that popular process engines such as Camunda BPM do not support data object elements in the process model; the framework is just ignoring them. Camunda provides so-called *process variables* for handling process data instead [5]. These process variables are hidden behind the technical XML representation and do not provide any visualization. Figure 3 shows the running example without data objects, as it would typically be designed. Additionally, data operations are hidden in other artifacts such as source code or views, making the intended analysis even more difficult.

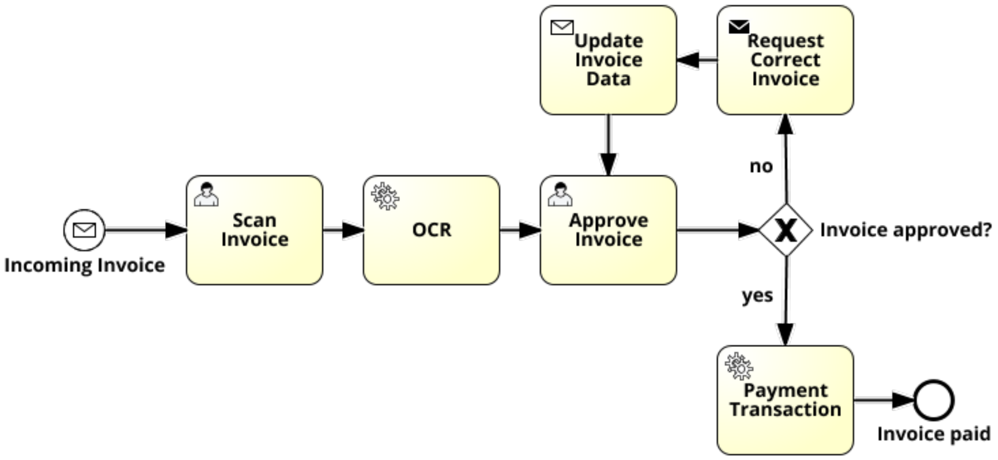


Figure 3: The Same Process Model with Hidden Data Objects.

To address this issue, we present a design concept to gather the scattered data operations of all involved artifacts of a PDA by an in-depth analysis. We demonstrate how data operations can be uncovered at the process model level, in linked user forms, or even in source code. We suppose that process engine-specific APIs can be used to identify data object processing in source code.

However, it is challenging to determine the program routines linked in the process model as well as the order of their execution. Furthermore, as described before, considering artifacts separately is not sufficient. For the desired data-flow analysis, an integrated representation of all possible control flows with corresponding data operations is indispensable. For this purpose, we present an algorithm and demonstrate its functionality on a prototype designed for the Camunda BPM.

For the study presented in this paper, we pursue the established Design Science (DSR) paradigm [14]. This method implies developing a prototypical artifact to solve an identified problem. We identified the lack of an integrated view of the data-flow analysis for PDAs in our project's context. In order to address this issue, we have developed a design concept and a prototype for the Camunda BPM Engine. In the next step, the DSR cycle envisages evaluating the artifact created to discover possible optimization potential. Based on this, the created artifact can be improved in further iterations. In our case, we have conducted numerous case studies and interviewed BPM experts, during which we discovered minor optimization potentials.

The rest of the paper is structured as follows: The proposed anomaly detection is introduced at a glance in Section 2. The details of the DFA graph construction are described in Section 3, while the computing of data flows and the identification of anomalies are explained in Section 4. Next, the developed prototype and the evaluation are presented in Section 5. The discussion of related work is reflected in Section 6. Finally, Section 7 concludes this paper and gives a short outlook.

2 Overview of the Anomaly Detection Solution

The proposed anomaly detection can be divided into two main challenges: The first challenge is to construct a single data-flow view across all artifacts of a Process-Driven Application, while the second hurdle is the identification of data-flow anomalies. We have developed a five-step process to tackle these challenges, as illustrated in Figure 4 and discussed in detail in the following.

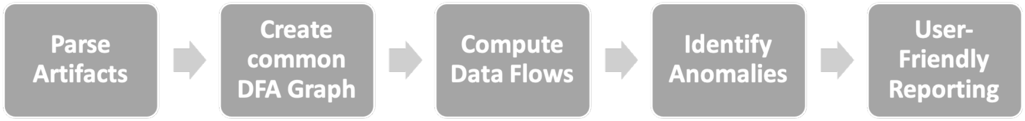


Figure 4: The Anomaly Detection Process at a Glance.

The data flow relevant information is reflected in process models, source code, but also in other artifacts such as user forms. We introduce a construction algorithm for the common data-flow view, called DFA graph, representing all possible data flow paths. Each node of the directed graph expresses a single step of a possible run. The abstract syntax of a BPMN model is almost a kind of control-flow graph. However, input- and output data of user task forms, conditional expressions, and the like need to be identified. Based on the parsing information, the individual nodes are extended by labels that reveal any data operations executed during that step, clustered into *defined* (write-access), *used* (read-access), and *killed* (removal). The integration of the data-flow information, obtained from source code linked in the process model, into the DFA graph, is more complicated. Therefore we start generating a call graph using the Java framework *Soot* [19]. We transform the result into the DFA graph structure enhanced with the same previously introduced labels in the next step. Finally, we integrate this subgraph into the single DFA graph. How the DFA graph is constructed and data-flow information are identified is discussed in detail in Section 3.

A variety of traditional static analysis methods such as *Available Expressions*, *Very Busy Expressions*, or *Liveness Analysis* can be performed on the integrated DFA graph [2]. For our intended anomaly detection, the calculation of the data flow between the individual DFA graph nodes is necessary. Therefore, we use a combination of a forward and backward analysis to set up pre- and post-conditions for each node. The forward analysis, a variant of a reaching definition analysis, allows to check the following situations:

- has a data object (certainly) been defined before it is used
- has a data object (certainly) not been undefined (i.e. killed before or not defined) when it is killed

The backward analysis allows to determine whether a definition will (possibly) be used. Based on these analyses, the DFA graph can be scanned for data-flow anomalies. The backward and forward analysis and the detection of anomalies are discussed in detail in Section 4.

3 Building the DFA Graph

As a starting point for our intended analysis, we construct a directed control-flow graph. The DFA graph represents all possible data-operation sequences, whereby its nodes express every possible execution step. The nodes are labeled, indicating which data object is *defined*, *used*, or *killed* at this position.

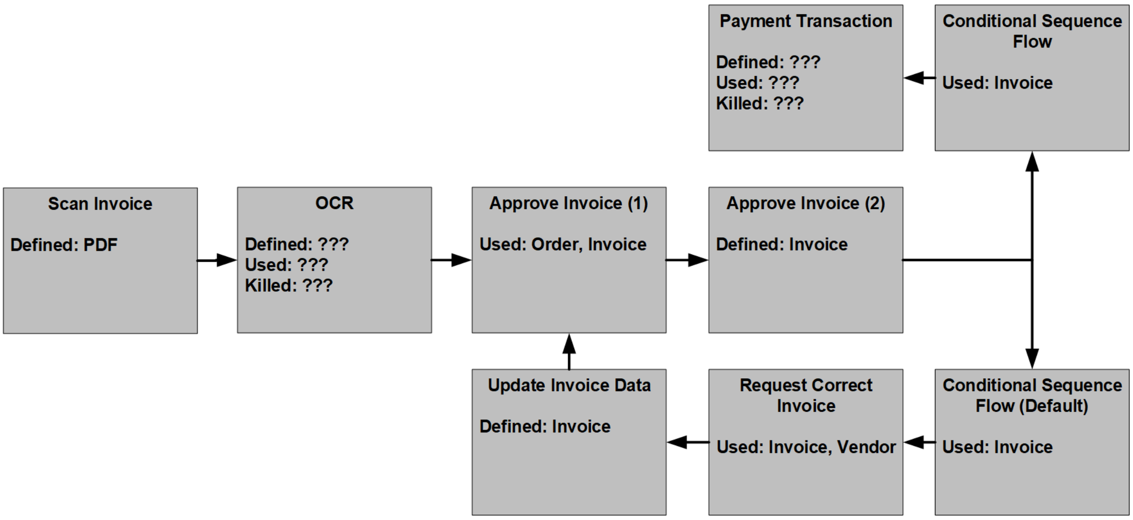


Figure 5: Early (Incomplete) State of the DFA Graph.

We start the construction of the DFA graph using the process model in its abstract syntax, i.e. as an instance of the BPMN metamodel. In the next step, we reduce the model by elements which are irrelevant for the control flow, such as lanes or text annotations, and adjust all edges according to the flow semantics. If data operations occur in a node, it is supplemented with appropriate labels. Nodes without data operations can be removed in order to reduce the complexity. In this case, it is necessary to connect the predecessor(s) of the node to be removed with the corresponding successor(s). This procedure is demonstrated using the example from Section 1 in Figure 5. Depending on the node type, a substitution of the node is necessary. The following paragraphs explain the treatment of process model node types and source code in detail.

3.1 Substitution and Labeling of BPMN Elements

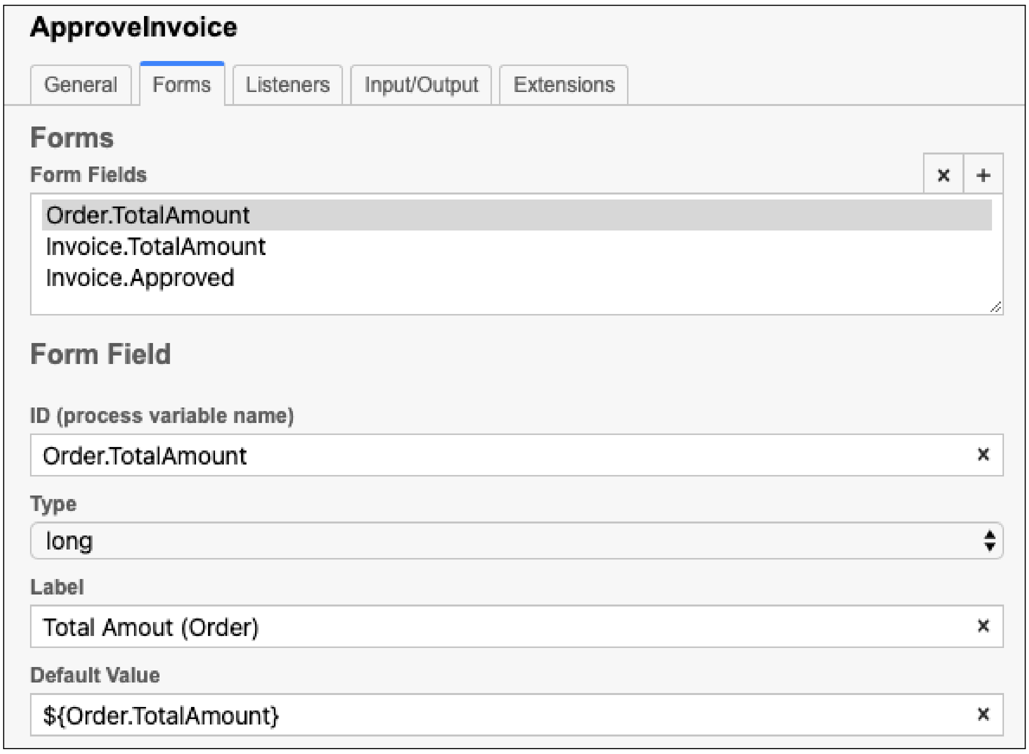
The BPMN standard defines numerous different control-flow elements, which must be treated differently during the DFA construction. In this subsection, we described the treatment in detail below for *sequence flow*, *user task*, *service task*, *script task*, and *call activity* nodes.

Sequence Flow Nodes: Sequence flows normally do not contain any data operations and can consequently be removed for the analysis as described above. However, this statement does not apply to conditional sequence flows. This sequence flow type usually depends on some condition that evaluates process data and is therefore relevant for the analysis we are pursuing. These flows typically occur after splitting XOR-gateways. As in the process model illustrated in Figure 2, such an evaluation of conditions takes place to decide on the subsequent flow direction. Listing 1 demonstrates this for our running example by a conditional expression with the data object *invoice*. Once a data object has been evaluated in such a condition, we add a *used* label for data object (here: *invoice*) to the corresponding node.

```
<sequenceFlow>
  <conditionExpression xsi:type="tFormalExpression">
    ${invoice.approved = true}
  </conditionExpression>
</sequenceFlow>
```

Listing 1: Conditional Sequence Flow Definition.

User Task Nodes: User tasks represent workflow steps which involve human interaction and are managed and executed by the BPM engine. Metainformation for rendering forms can be defined in the process model for this purpose, whereby the exact specification is left open in the BPMN standard. Platform-specific extensions are used, e.g., to directly embed the declaration of required forms into the user task definition. Figure 6 illustrates the configuration of form data for the user task *approve invoice* using the Camunda Modeler, while Listing 2 demonstrates the underlying XML specification. In the XML syntax, the attribute *id* directly reveals the data object bound to a form field. Alternatively, Camunda allows to embed references to HTML files that describe the required forms. In that case, we have to include these files in our analysis and scan them for relevant data bindings.



ApproveInvoice

Forms

Form Fields

Order.TotalAmount
Invoice.TotalAmount
Invoice.Approved

Form Field

ID (process variable name): Order.TotalAmount

Type: long

Label: Total Amout (Order)

Default Value: ${Order.TotalAmount}

Figure 6: Definition of Form Fields on Process Model Level using the Camunda Modeler.

If new data is requested during the user task, the corresponding data object will be labelled as *defined* at the current node. However, sometimes existing information is presented in addition to the request for new data. We have such a situation in Listing 2, as it includes read access to the data objects *order* (see read-only flag) and read and write access to the data object *invoice*. In such a case, the DFA graph's user-task node has to be substituted by two consecutive nodes. The first node is labeled with *used* for all data shown during the user task, representing the read access. The second node is labeled with *defined*, indicating the write access to the corresponding data. This procedure is necessary to represent the correct order of accesses. Otherwise, it cannot be clearly determined whether read access occurs first and then write access or the other way around.

```
<bpmn:userTask id="approveInvoice">
      <bpmn:extensionElements>
        <camunda:formData>
          <camunda:formField id=" Order.TotalAmount" label="Total␣Amout␣(Order)" type="
              ↪ long" defaultValue="${Order.TotalAmount}">
            <camunda:validation>
              <camunda:constraint name="readonly" />
            </camunda:validation>
          </camunda:formField>
          <camunda:formField id="Invoice.TotalAmount" label="Total␣Amount␣(Invoice)"
              ↪ type="long" defaultValue="${Invoice.TotalAmount}">
            <camunda:validation>
              <camunda:constraint name="readonly" />
            </camunda:validation>
          </camunda:formField>
          <camunda:formField id="Invoice.Approved" type="boolean" />
        </camunda:formData>
      </bpmn:extensionElements>
</bpmn:userTask>
```

Listing 2: Embedded Form Declaration of the User Task *Approve Invoice*.

Service Task Nodes: Service tasks are usually used for automated steps in a workflow by invoking external implementations. A referenced implementation, e.g., a Java method, can access process data in various statements. Therefore, it is not sufficient to represent a service task by a single node, but we have to substitute the service task node with a subgraph representing the data flow within the external implementation. How such a subgraph is created, is explained in detail in Subsection 3.2. The substitution of the service task node is performed by redirecting its adjacent edges: the start node of the subgraph becomes new target of every ingoing edge, and the end node of the subgraph becomes new source of every outgoing edge. This way, the service task node remains unconnected and can be removed.

In Figure 7, the procedure is demonstrated for a service task using the example from Section 1. In the upper part of Figure 7, the service task is referring to a control-flow graph with four nodes of a fictitious Java method. In the second part of the Figure, the service task node has been completely replaced by the control-flow graph resulting from the external implementation. For better readability, the labels have been hidden in this figure.

Script Task Nodes: Script Tasks can contain implementations such as Groovy or JavaScript scripts and are executed by the process engine. These scripts can be embedded in the model or in referenced script files. This node type is treated in the same way as the previously introduced service task, but typically requires support for a different scripting language.

Call Activity Nodes: A call activity acts as a wrapper and calls a referenced process definition, which has to be embedded into the DFA graph. The procedure works as follows: First, the call activity node must be duplicated for the correct resolution of input and output mappings to the external process definition. Next, the graph of the referenced process definition is inserted between the nodes just duplicated. Remember: An external process definition can have several start and end events. In this case, it is necessary to embed further split and / or join nodes.

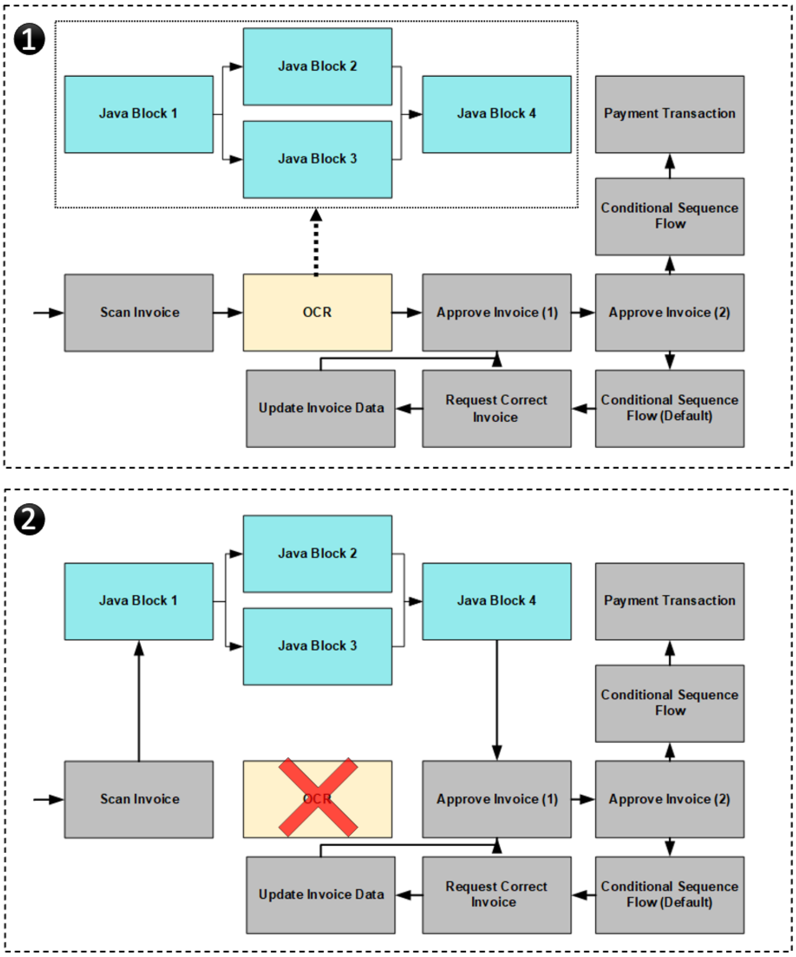


Figure 7: Substitution of a Service-Task Node by the Detailed Subgraph.

3.2 Subgraphs for Referenced Source Code

In referenced source code, process data is typically accessed via an API provided by the process engine. In the case of Camunda BPM, a software developer must implement a Java Delegate interface to access data objects (Camunda synonym: process variable) in Java Delegates [6]. The library provided by Camunda allows read, write, and delete accesses via the API. For this purpose, the following methods are mainly used:

- getVariable(variableName)
- setVariable(variableName, value)
- removeVariable(variableName)

A simplified example implementation of the *ORC* service task, known from the running example, is presented in Listing 3 - using a Java Delegate implementation.

Several tools exist for analyzing such Java code implementations, for instance, SpotBugs, Checkstyle, or Klocwork [20, 7, 15]. The Java optimization framework Soot [23] proved to be particularly suitable for our purpose. The framework, developed by McGill University, analyzes external Java source code and converts it into an intermediate code representation. Furthermore, the tool is used to generate call graphs. This way, inter-procedural calls are resolved, as they contain all outgoing edges and their targets. The intermediate three-address code representation Jimple [23] is used during the analysis. While Java bytecode offers more than 200 possible statement types, Jimple reduces them down to 19. The transfer to the intermediate code representation already implies optimizations such as removing unused code and linearizing expressions [23].

```java
public class Orc_Delegate implements JavaDelegate {

        public void execute(DelegateExecution execution) throws Exception {

                // Read Process Variable
                PdfDocument pdf = (PdfDocument) execution.getVariable("pdf");

                // Initialization InvoiceData
                InvoiceData invoice = new InvoiceData( );

                // Set invoice data (from embedded data or OCR)
                if (pdf.isDataEmbedded()) {
                    invoice.setMetadata( pdf.getEmbeddedData() );
                }
                else {
                    invoice.setMetadata( pdf.getOcrData() );
                }

                // Write Process Variable
                execution.setVariable("invoice", invoice);
        }
    }
```

Listing 3: Referenced Implementation with Read and Write Access to Process Data.

Through Soot, call graphs are generated for source code references detected in the process model. Such a call graph is illustrated in Figure 8 using the example of Listing 3.

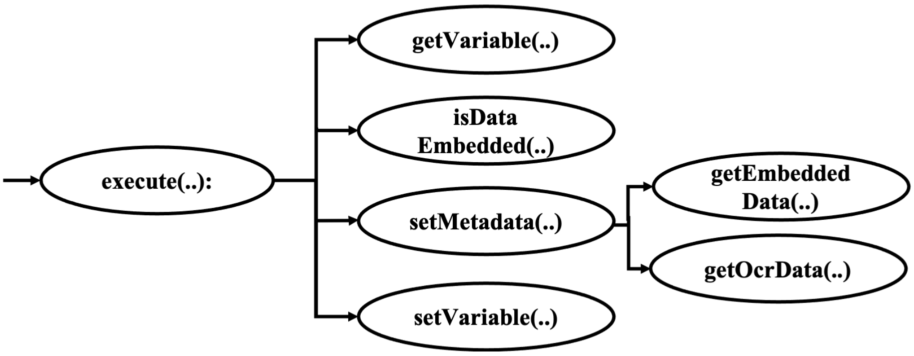


Figure 8: Call Graph Representation of Listing 3.

A multi-stage procedure was established based on the generated call graphs and the Jimple representation to identify data accesses in source code, as illustrated in Figure 9 and considered in detail in the following paragraphs.

The procedure starts by the *Graph Iterator* iterating over all *basic blocks* of the generated call graph. Each basic block, representing a maximal sequence of consecutive instructions, is transferred to a node. These nodes include information about the predecessors and successors, which results in a control-flow representation of the source code [23]. The graph originated from the source code must be integrated into the common DFA graph, as mentioned before. For the desired data-flow analysis, it is furthermore necessary to resolve all instructions within a node. The resolution is handled by the *Block Iterator*, iterating over the lowest level, the atomic statements.

During the iteration, the analysis distinguishes between different statement types such as *Invoke Statements* and *Assignment Statements* to discover data-object access. The final step for finding data-object operations is concealed behind the *Parse Expression* module. Here, the called method's name is compared to predefined methods for accessing data objects through the process engine's API. If there is a match, the data object with the corresponding access type label is attached to the node. After the procedure, the created graph, including the data-flow information, is attached to the referring DFA graph node. As explained in Subsection 3.1, the created graph is used as a substitution of the referencing DFA node.

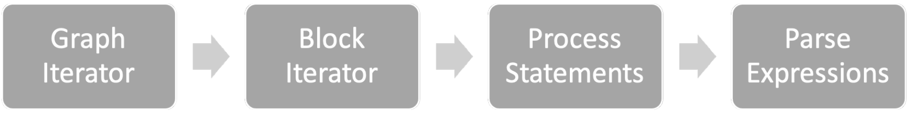


Figure 9: Processing of Referenced Source Code using Soot.

Referenced implementations, such as the introduced Java Delegates, can invoke numerous other classes. These do not have to be self-developed, but can also result from embedded Java libraries such as *java.**, *sun.**. *jdk.**, or *javax.**. Since no direct access to process data is to be expected in these cases, such classes should be excluded from the analysis. Therefore, we propose maintaining a whitelist and blacklist for the upcoming analysis, limiting the area to be analyzed. These lists should be customized to project-specific conditions. In that manner, problem explosions can be avoided, and the analysis time can be kept low. Account should be taken of the fact that code loaded via reflection at runtime cannot be considered in the analysis.

4 Computing Data Flows and Identify Anomalies

Based on the constructed DFA graph resulting from the process model, referenced source code, and other artifacts such as user forms, the intended data-flow analysis can be performed. As mentioned at the beginning, we intend a combined forward and backward analysis [2]. For this purpose, we set up data-flow sets computed by a fixpoint computation.

The nodes of the constructed DFA graph were already labeled with *defined* for write access, *killed* for delete access, and *used* for read access on data objects. Through these labels, the data operations within a node were considered. What is still missing is the determination of the data flow between the individual nodes. For each node, the incoming and outgoing data flow must be calculated [2].

To represent all incoming and outgoing data objects of a single node, we extend the previously presented node labels by three more data sets named IN, OUT, and $USED$. The data set IN subsumes all data objects which are certainly available to the current node. It can be defined as intersection of all output sets of its predecessors, as given by Equation 1. OUT, on the other hand, contains all data objects that are passed on by the current node to all its downstream successors. Differences between IN and OUT emerge from data operations (definitions, deletions) taking place in the current node. For this reason, the formal definition of OUT is based on set IN and local activities of the current node, as given by Equation 2.

As OUT depends on IN, and IN depends on the predecessors of the current node, the content of the two sets can be computed iteratively using a forward fixpoint computation. As soon as a fixed point is reached and the sets remain stable, the calculation terminates. Note that this will happen after a finite number of iterations, since both the graph and the set of data objects are finite, and the computed sets are monotonically increasing in each round.

Definition: Let n be an arbitrary node of the graph and pred(n) the set of n's direct predecessors and succ(n) the set of n's direct successors in the graph, then the additional label sets of n can be defined using the dot notation from OO programming as follows:

$$n.IN = \bigcap_{p \in pred(n)} p.OUT \tag{1}$$

$$n.OUT = (n.IN \cup n.defined) \setminus n.killed \tag{2}$$

$$n.USED = n.used \cup (\bigcup_{s \in succ(n)} s.USED) \setminus (n.defined \cup n.killed) \tag{3}$$

The third set, called $USED$, collects all data objects whose current value, i.e., the value they had right before reaching the current node, is used on at least one of the downstream paths. This information is required to detect never-used anomalies later on. The formal definition is given by Equation 3. This time, we have to construct the union of all successors, because it is sufficient if the data object is used on at least one path. Note that new definitions or deletions of a data object will cause later uses to refer to these definitions or deletions rather than to the previous value. Thus, they have to be subtracted from the set of data objects, which are used later. Similar to IN and OUT above, one can perform a fixpoint computation, this time backwards, until $USED$ remains stable for all nodes.

The algorithm can handle any form of cycles in the graph and, as mentioned, is guaranteed to terminate. The data-flow calculation applied to our running incoming invoice example results in a labeled DFA graph as in Figure 10. After the calculation is completed, the data-flow anomalies introduced in Section 1 can be identified using the Equations 4, 5 and 6.

$$n.UndefinedRead \quad :\Leftrightarrow \quad n.used \setminus n.IN \neq \emptyset \tag{4}$$

$$n.NeverUsed \quad :\Leftrightarrow \quad n.defined \setminus \bigcup_{s \in succ(n)} s.USED \neq \emptyset \tag{5}$$

$$n.UndefinedUndefined \quad :\Leftrightarrow \quad n.killed \setminus n.IN \neq \emptyset \tag{6}$$

The anomaly definitions are checked for each node of the DFA graph. If an anomaly condition applies to a data object, the anomaly type, location, and affected data object are reported. The intended analysis is sound and complete w.r.t. the identification of data-flow anomalies. However, it may happen that anomalies are reported, which do not have any negative impact. Such a scenario results from semantic dependencies. Let us take the example from Listing 3 and modify it in line 6 as follows:

```
if (1 != 1) {
  PdfDocument pdf = (PdfDocument) execution.getVariable("pdf");
}
```

In addition, we assume that `pdf` is undefined beforehand (which is not the case in our running example). Here, an undefined-read anomaly would be reported, although the call to `getVariable()`

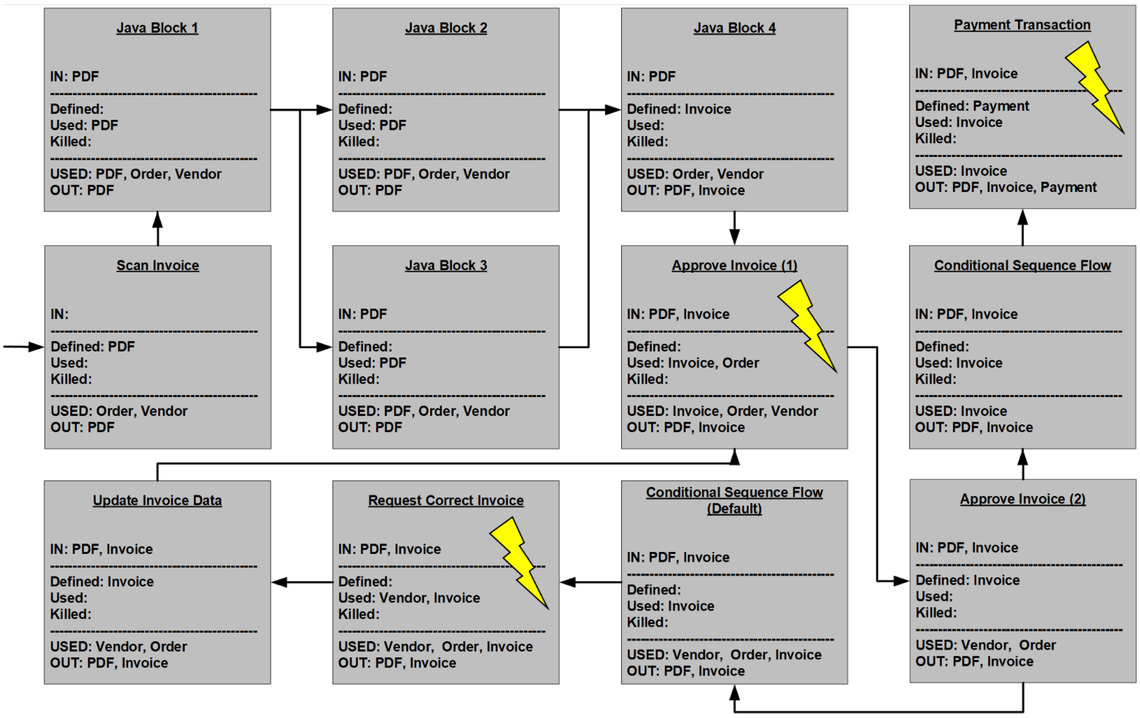


Figure 10: Labeled DFA Graph of the Running Example Produced by our Algorithm.

will never be reached. Nevertheless, we think that such cases should also be reported. Code smells like that should be improved. Otherwise, small changes might lead to the reported anomaly and corresponding errors.

In the example from Section 1, the undefined-read condition matches for the data object *order* in *approve invoice* and for the data object *vendor* in *request correct invoice*. The condition never-used matches for *payment document* in *payment transaction* (cf. Figure 10). Since the nodes of the DFA graph still have references to the original process model, a tool can visualize anomalies directly in the process model.

5 Prototype and Evaluation

We integrated the data-flow analysis into a Camunda-based prototype called viadee Process Application Validator (vPAV)[2]. The tool also includes other static checks in order to discover inconsistencies in Process-Driven Applications such as broken implementation references in the model or the violation of conventions. The developer can integrate vPAV into projects or deployment pipelines via Maven. The checks are executed by JUnit tests. The implemented data-flow analysis considers any Java code, which can be referenced in service tasks, but also Camunda-specific callback methods (so-called execution listeners). The concept and the tool were evaluated by case studies in order to test the feasibility and performance. Furthermore, semi-structured interviews with two BPM experts were used to evaluate the prototype and its concept. Due to the small number of interviews, we can only extract qualitative insights.

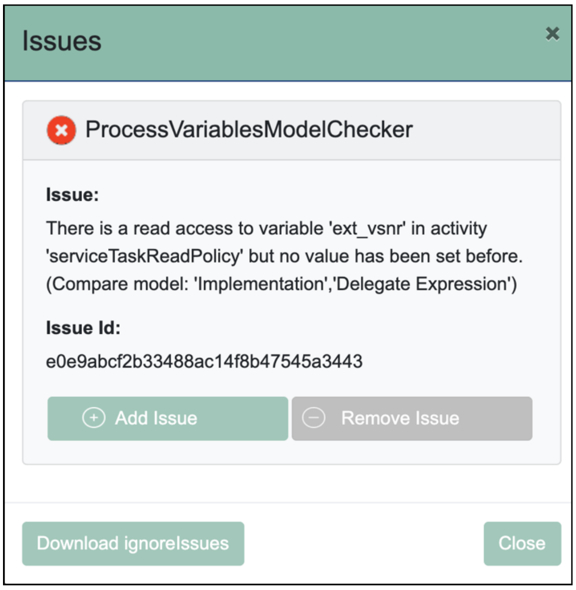
For the expert interviews, an interview protocol has been developed, which consists of three parts.

[2] Available at https://github.com/viadee/vPAV/tree/sac2021.

In the first part, the experts have been asked about the current practice in developing PDAs and about occurring data-flow errors and eliminating them. In the second part, the interviewees have embedded the prototype into their current implementation project and run the data-flow analysis. Finally, the BPM experts have been asked about their experience with the use of the prototype and the data-flow analysis, focusing on feasibility, performance, and usability. The interview template and the transcribed interviews are available in a public repository[3].

Current Practice: Both BPM experts testified during the interviews that the data flow is not documented in their implementation projects using the BPMN standard elements or in an alternative way. As suspected, access to process data takes place both in the process model and in the referenced source code. In the first interview, the most common data-flow anomalies (code smells) mentioned were never used process variable definitions. This type of anomaly occurs mainly in the early development stage of a PDA. The second BPM expert mentioned that undefined-read anomalies frequently occur because of typos in the definition or the read. Data-flow errors were detected so far in projects via unit tests or integration tests.

Feasibility: We used various case studies in order to evaluate the operational feasibility of our analysis. These include very simplified process examples to check the functionality of anomaly detection, but also a realistic business process of an insurance company [24]. The latter process demonstrated in Figure 12 consists of 30 BPMN elements and the model refers to 15 service implementations in Java, including also third-party frameworks such as JUnit and Spring [26]. The data-flow analysis discovered a total of 31 data operations in the PDA. All previously (deliberately) injected data-flow anomalies, such as the undefined-read anomaly reported in Figure 11, were successfully detected.



Issues

ProcessVariablesModelChecker

Issue:

There is a read access to variable 'ext_vsnr' in activity 'serviceTaskReadPolicy' but no value has been set before. (Compare model: 'Implementation','Delegate Expression')

Issue Id:

e0e9abcf2b33488ac14f8b47545a3443

Figure 11: Information Box for a Detected Undefined-Read Anomaly.

After the analysis, an HTML report with the process model is provided. The report visualizes for each BPMN element all detected data operations and anomalies. Moreover, the report offers a list of existing data objects, including the corresponding model elements and access type. Such a summary is demonstrated for the insurance example in Figure 12. As mentioned before, the interviewed BPM experts have also used the tool in their real implementation projects. However, since their projects were already far developed, the data-flow analysis did not detect any anomaly, since all anomalies had been detected and removed by hand before. Neverthelsee, one of the

[3] Available at https://fh-muenster.sciebo.de/s/lKTcP2NEym4rMhH.

BPM experts considered it helpful to see at which point in the process model which data objects are processed (Interview 2). Embedding the tool in implementation projects was found to be particularly useful, especially in the early stage of the development process. Due to the early prototypical state of our tool, a few data accesses could not be resolved correctly, since the tool can not yet handle complex inheritance structures and might errorneously report anomalies in such situations.

Performance: We measured the performance of the data-flow analysis in seven different case studies with VisualVM [25]. This tool allows extracting runtime, memory consumption, and CPU usage of a local application session. We repeated the measurement ten times for each case study to compensate for any possible runtime variations. An overview of the runtime considering the number of BPMN elements, implemented classes, and lines of code is given in Table 1. The full measurement results and references to the publicly available PDAs are provided in the mentioned repository. Note that much more classes are loaded during the analysis besides the listed implemented classes. This is mainly caused by third-party frameworks and libraries. For instance, the tool loaded over 4000 classes during the insurance case study analysis (process no. 7). By maintaining the black- or whitelist filter, the number of classes considered and thus the runtime still can be kept low. However, the runtime was in all test runs under five seconds, which seems acceptable for the practical use in implementation projects.

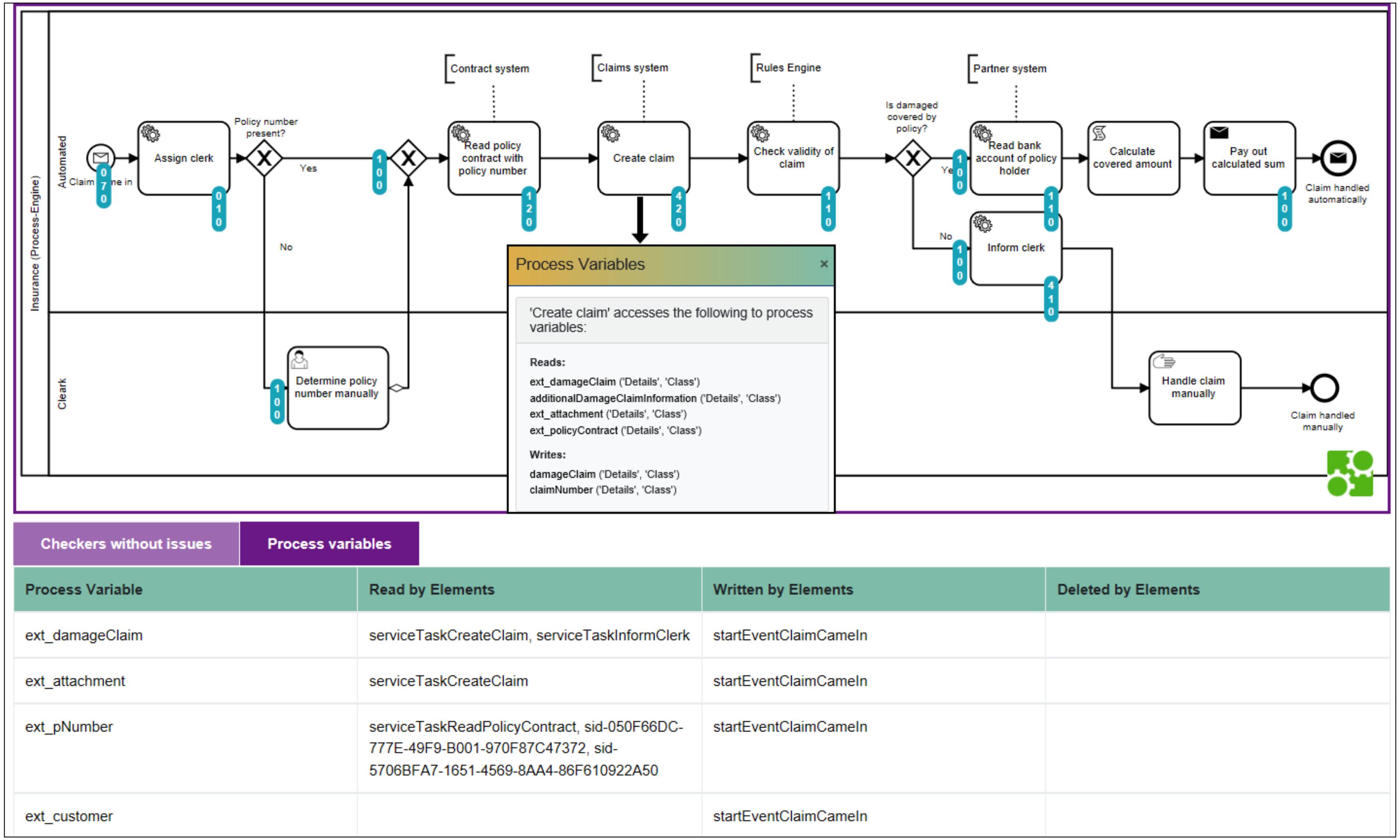
Process No.	BPMN-Elements	Classes implemented	Lines of Code	Runtime (in s)
1	9	3	64	1.602
2	7	2	18	1.756
3	5	2	36	2.173
4	7	3	80	2.296
5	41	27	743	2.372
6	9	13	213	2.467
7	162	53	1906	3.267

Table 1: Measured Processing Times (in s) of the Tool in different Case Studies.

The performance was not measured during the ongoing implementation projects. However, one BPM expert did not even notice that the analysis was active (Interview 1). The fast execution increases the motivation to use the tool during the implementation phase.

Usability: The BPM experts were able to gain an impression of the prototype's usability during their PDA development. The usability evaluation can be subdivided into integrating the tool into implementation projects and the analysis report. One BPM expert experienced integrating the prototype into his implementation project as easy thanks to the extensive documentation provided (Interview 2), while the other expert assesses the integration as intuitive (Interview 1). The experts particularly highlighted the visualization of data-flow information on the process model level. The report is ”very well presented” (Interview 2) and contains all necessary information. For a process engineer, the report is easy to understand and in a suitable format, while the same understanding cannot be assumed for business experts without prior technical knowledge.

Overall, the developed prototype and the underlying concept for data-flow analysis could convince during the evaluation. It is an essential contribution to the static analysis of Process-Driven Applications, although there is still potential for an improvement, e.g. with respect to the treatment of the mentioned complex inheritance structures.



Process Variables ×

'Create claim' accesses the following to process variables:

Reads:

ext_damageClaim ('Details', 'Class')
additionalDamageClaimInformation ('Details', 'Class')
ext_attachment ('Details', 'Class')
ext_policyContract ('Details', 'Class')

Writes:

damageClaim ('Details', 'Class')
claimNumber ('Details', 'Class')

Process Variable	Read by Elements	Written by Elements	Deleted by Elements
ext_damageClaim	serviceTaskCreateClaim, serviceTaskInformClerk	startEventClaimCameIn	
ext_attachment	serviceTaskCreateClaim	startEventClaimCameIn	
ext_pNumber	serviceTaskReadPolicyContract, sid-050F66DC-777E-49F9-B001-970F87C47372, sid-5706BFA7-1651-4569-8AA4-86F610922A50	startEventClaimCameIn	
ext_customer		startEventClaimCameIn	

Figure 12: Composite Screenshot of the Data-Flow Analysis Report without Anomalies

6 Related Work

Current research is so far restricted separate data-flow analysis of of program code and process model. The analysis of process models for the XML-based notation *Business Process Execution Language* (BPEL) has been considered, e.g., by JI ET AL.[11]. They use a forward data-flow analysis based on a modified Reaching Definition algorithm. For this approach, the authors developed an eXtended Control Flow Graph (XCFG), on which the analysis is applied. YANG ET AL. have also concentrated on a forward analysis of process models in BPEL notation [28]. Starting from a so-called Activity Object Tree, a control-flow graph is created to identify data-flow anomalies. Another approach focusing on the analysis of BPEL processes is provided by MOSER ET AL. [12]. Their idea is to generate a Concurrent Single Static Assignment Form (CSSA) of the process model, which is used to analyze the data flow and data dependencies.

The intention of SARNO AND SINAGA is to use ontologies based on the event logs and the process model to achieve additional information for anomaly detection [16]. Learning multi-level class association rules is used to discover anomalies in processes. CORTES-CORNAX ET AL. perform an automated analysis on a Process Intermediate Format, resulting from the workflow-oriented domain-specific language Mangrove [8].

Other researchers have focused, as our work, on the analysis of process models in BPMN notation. STACKELBERG ET AL. perform analysis after transferring the process model to Petri Nets [27]. The approach of AWAD ET AL. follows the same idea, but additionally repairs a subset of data-flow errors automatically [4]. TRČKA ET AL. define nine anti-patterns expressed in the temporal logic CTL* for BPMN models, which can be detected using a model checker [22]. However, none of the mentioned approaches considers the inclusion of referenced source code and other artifacts.

In a previous paper, we have already presented a first approach for static analysis of all artifacts of a PDA [18]. The concept includes both the process model and referenced source code in the data-flow analysis. However, the approach has two significant weaknesses: First, the regular expression-based search for data-access operations just considers the directly referenced program routines. This way, no methods invoked from that entry point are taken into account. The second, more fatal weakness is the lack of consideration for the order of data operations. The missing sequence easily leads to wrong results. This critical research gap is closed in this paper by the presented concept. The desired data-flow analysis is achieved based on an integrated representation of the complete control and data flow.

7 Conclusion

Data-flow analysis approaches have so far been restricted to considering either process models or source code. However, for the popular enterprise application architectures of Process-Driven Applications, this separate consideration is not sufficient. Only an integrated view of all artifacts allows a significant data-flow analysis. In this paper, we have presented an approach based on an integrated DFA graph resulting from the process model and referenced artifacts such as source code or user forms. This DFA graph contains labels indicating data operations within a node and the data flow between these.

This approach's conception was based on the following steps: First, potential locations were identified where process data could be treated inside and outside the process model. As a result, operations on data objects within the process model can be determined and marked; further steps were necessary for operations outside the model. Source code referenced from the process model was analyzed using the Java framework Soot. The source code was first transferred into a call-graph and in another step into a control-flow graph, providing information about access to data objects. This control-flow graph was integrated in a final step into a common DFA graph.

Data sets were defined using equations based on a combined forward and backward analysis for the analysis of the data flow between the DFA graph nodes. This extension provides the foundation to identify the anomalies we have identified: undefined-read, never-used, and undefined-undefined.

While the data-flow analysis concept is platform-independent, the technical implementation is specifically designed for a process engine. In our case, a prototypical implementation was created for the popular process engine Camunda BPM. The prototype detects data-flow anomalies in PDAs and reports them user-friendly directly in the process model. Apart from anomaly detection, the tool is also useful to report the trace of data operations, which can be required, e.g., by regulations such as the European General Data Protection Regulation.

The data-flow analysis and the developed tool were convincing during the evaluation. On the one hand, the functionality was demonstrated by the correct detection of data operations on the source-code level and the process-model level in several case studies. On the other hand, BPM experts revealed positive feedback regarding its usefulness and usability in semi-structured interviews. Only the partially incorrect resolution of data accesses in the source code in hierarchical constructions should be improved in further work.

This paper presented an approach for an integrated data-flow analysis of a PDA, which allows detecting anomalies early in the development process.

References

[1] Activiti. Open source business automation — activiti. `https://www.activiti.org`, 2020.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] Thomas Allweyer. *BPMN 2.0: introduction to the standard for business process modeling*. BoD–Books on Demand, 2016.

[4] Ahmed Awad, Gero Decker, and Niels Lohmann. Diagnosing and repairing data anomalies in process models. In Stefanie Rinderle-Ma, Shazia Sadiq, and Frank Leymann, editors, *Business Process Management Workshops*, pages 5–16, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[5] Camunda. Camunda best practices - handling data in processes. `https://camunda.com/best-practices/handling-data-in-processes/`, 2016.

[6] Camunda. Delegation code — docs.camunda.org. `https://docs.camunda.org/manual/7.8/user-guide/process-engine/delegation-code/`, 2018.

[7] Checkstyle. Checkstyle 8.36.1. `https://checkstyle.sourceforge.io/`, 2020.

[8] Mario Cortes-Cornax, Ajay Krishna, Adrian Mos, and Gwen Salaün. Automated analysis of industrial workflow-based models. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, page 120–127, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Philip Cox, Simon Gauvin, and Andrew Rau-Chaplin. Adding parallelism to visual data flow programs. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, page 135–144, New York, NY, USA, 2005. Association for Computing Machinery.

[10] Jakob Freund and Bernd Rücker. *Real-Life BPMN: With introductions to CMMN and DMN*. CreateSpace Independent Publishing Platform, 2006.

[11] S. Ji, B. Li, and P. Zhang. Xcfg based data flow analysis of business processes. In *2019 5th International Conference on Information Management (ICIM)*, pages 71–76. IEEE, 2019.

[12] S. Moser, A. Martens, K. Gorlach, W. Amme, and A. Godlinski. Advanced verification of distributed ws-bpel business processes incorporating cssa-based data flow analysis. In *IEEE International Conference on Services Computing (SCC 2007)*, pages 98–105. IEEE, 2007.

[13] OMG. About the business process model and notation specification version 2.0.2. `https://www.omg.org/spec/BPMN/`, 2014.

[14] Ken Peffers, Marcus Rothenberger, Tuure Tuunanen, and Reza Vaezi. Design science research evaluation. In Ken Peffers, Marcus Rothenberger, and Bill Kuechler, editors, *Design Science Research in Information Systems. Advances in Theory and Practice*, pages 398–410, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[15] Perforce. Static code analysis for c, c , c#, and java. `https://www.perforce.com/products/klocwork`, 2020.

[16] R. Sarno and F. P. Sinaga. Business process anomaly detection using ontology-based process modelling and multi-level class association rule learning. In *2015 International Conference on Computer, Control, Informatics and its Applications (IC3INA)*, pages 12–17. IEEE, 2015.

[17] Konrad Schneid, Herbert Kuchen, Sebastian Thöne, and Sascha Di Bernardo. Uncovering data-flow anomalies in bpmn-based process-driven applications. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, page 1504–1512, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Konrad Schneid, Claus A. Usener, Sebastian Thöne, Herbert Kuchen, and Christian Tophinke. Static analysis of bpmn-based process-driven applications. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 66–74, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Soot. Soot - a java optimization framework. `https://github.com/soot-oss/soot/`, 2020.

[20] SpotBugs. Spotbugs. `https://spotbugs.github.io`, 2020.

[21] Volker Stiehl. *Process-driven Applications with BPMN*. Springer, 2014.

[22] Nikola Trčka, Wil M. P. van der Aalst, and Natalia Sidorova. Data-flow anti-patterns: Discovering data-flow errors in workflows. In Pascal van Eck, Jaap Gordijn, and Roel Wieringa, editors, *Advanced Information Systems Engineering*, pages 425–439, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[23] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[24] viadee Unternehmensberatung AG. vpav example process. `https://github.com/viadee/vpav_case_study/tree/develop/`, 2021.

[25] VisualVM. Visualvm: Home. `https://visualvm.github.io/`, 2021.

[26] VMware. Spring — home. `https://spring.io/`, 2021.

[27] Silvia Von Stackelberg, Susanne Putze, Jutta Mülle, and Klemens Böhm. Detecting data-flow errors in bpmn 2.0. *Open Journal of Information Systems (OJIS)*, 1(2):1–19, 2014.

[28] Xuehong Yang, Junfei Huang, and Yunzhan Gong. Static data flow analysis and anomalies detection for bpel. In *2009 International Conference on Test and Measurement*, volume 2, pages 18–21. IEEE, 2009.

Working Papers, ERCIS

Nr. 1 Becker, J.; Backhaus, K.; Grob, H. L.; Hoeren, T.; Klein, S.; Kuchen, H.; Müller-Funk, U.; Thonemann, U. W.; Vossen, G.; European Research Center for Information Systems (ERCIS). Gründungsveranstaltung Münster, 12. Oktober 2004.

Nr. 2 Teubner, R. A.: The IT21 Checkup for IT Fitness: Experiences and Empirical Evidence from 4 Years of Evaluation Practice. 2005.

Nr. 3 Teubner, R. A.; Mocker, M.: Strategic Information Planning – Insights from an Action Research Project in the Financial Services Industry. 2005.

Nr. 4 Gottfried Vossen, Stephan Hagemann: From Version 1.0 to Version 2.0: A Brief History Of the Web. 2007.

Nr. 5 Hagemann, S.; Letz, C.; Vossen, G.: Web Service Discovery – Reality Check 2.0. 2007.

Nr. 6 Teubner, R.; Mocker, M.: A Literature Overview on Strategic Information Management. 2007.

Nr. 7 Ciechanowicz, P.; Poldner, M.; Kuchen, H.: The Münster Skeleton Library Muesli – A Comprehensive Overview. 2009.

Nr. 8 Hagemann, S.; Vossen, G.: Web-Wide Application Customization: The Case of Mashups. 2010.

Nr. 9 Majchrzak, T.; Jakubiec, A.; Lablans, M.; Ükert, F.: Evaluating Mobile Ambient Assisted Living Devices and Web 2.0 Technology for a Better Social Integration. 2010.

Nr. 10 Majchrzak, T.; Kuchen, H: Muggl: The Muenster Generator of Glass-box Test Cases. 2011.

Nr. 11 Becker, J.; Beverungen, D.; Delfmann, P.; Räckers, M.: Network e-Volution. 2011.

Nr. 12 Teubner, A.; Pellengahr, A.; Mocker, M.: The IT Strategy Divide: Professional Practice and Academic Debate. 2012.

Nr. 13 Niehaves, B.; Köffer, S.; Ortbach, K.; Katschewitz, S.: Towards an IT consumerization theory: A theory and practice review. 2012

Nr. 14 Stahl, F., Schomm, F., Vossen, G.: Marketplaces for Data: An initial Survey. 2012.

Nr. 15 Becker, J.; Matzner, M. (Eds.).: Promoting Business Process Management Excellence in Russia. 2012.

Nr. 16 Teubner, R.; Pellengahr, A.: State of and Perspectives for IS Strategy Research. 2013.

Nr. 18 Stahl, F.; Schomm, F.; Vossen, G.: The Data Marketplace Survey Revisited. 2014.

Nr. 19 Dillon, S.; Vossen, G.: SaaS Cloud Computing in Small and Medium Enterprises: A Comparison between Germany and New Zealand. 2015.

Nr. 20 Stahl, F.; Godde, A.; Hagedorn, B.; Köpcke, B.; Rehberger, M.; Vossen, G.: Implementing the WiPo Architecture. 2014.

Nr. 21 Pflanzl, N.; Bergener, K.; Stein, A.; Vossen, G.: Information Systems Freshmen Teaching: Case Experience from Day One (Pre-Version of the publication in the International Journal of Information and Operations Management Education (IJIOME)). 2014.

Nr. 22 Teubner, A.; Diederich, S.: Managerial Challenges in IT Programmes: Evidence from Multiple Case Study Research. 2015.

Nr. 23 Vomfell, L.; Stahl, F.; Schomm, F.; Vossen, G.: A Classification Framework for Data Marketplaces. 2015.

Nr. 24 Stahl, F.; Schomm, F.; Vomfell, L.; Vossen, G.: Marketplaces for Digital Data: Quo Vadis?. 2015.

Nr. 25 Caballero, R.; von Hof, V.; Montenegro, M.; Kuchen, H.: A Program Transformation for Converting Java Assertions into Controlflow Statements. 2016.

Nr. 26 Foegen, K.; von Hof, V.; Kuchen, H.: Attributed Grammars for Detecting Spring Configuration Errors. 2015.

Nr. 27 Lehmann, D.; Fekete, D.; Vossen, G.: Technology Selection for Big Data and Analytical Applications. 2016.

Nr. 28 Trautmann, H.; Vossen, G.; Homann, L.; Carnein, M.; Kraume, K.: Challenges of Data Management and Analytics in Omni-Channel CRM. 2017.

Nr. 29 Rieger, C.: A Data Model Inference Algorithm for Schemaless Process Modeling. 2016.

Nr. 30 Bünder, H: A Model-Driven Approach for Graphical User Interface Modernization Reusing Legacy Services. 2019.

Nr. 31 Stockhinger, J.; Teubner, R.: How Digitalization Drives the IT/IS Strategy Agenda. 2020.

Nr. 32 Dageförde, J. C.; Kuchen, H.: Free Objects in Constraint-logic Object-oriented Programming. 2020.

Nr. 33 Plattfaut, R.; Coners, A.; Becker, J.; Vollenberg, C.; Koch, J.; Godefroid, M.; Halbach-Türscherl, D.: Patient Portals in German Hospitals – Status Quo and Quo Vadis. 2020.

Nr. 34 Teubner, R.; Stockhinger, J.: IT/IS Strategy Research and Digitalization: An Extensive Literature Review. 2020.

Nr. 35 Distel, B.; Engelke, K.; Querfurth, S.: Trusting me, Trusting you – Trusting Technology? A Multidisciplinary Analysis to Uncover the Status Quo of Research on Trust in Technology. 2021.

Nr. 36 Becker, J.; Distel, B.; Grundmann, M.; Hupperich, T.; Kersting, N.; Löschel, A.; Parreira do Amaral, M.; Scholta, H.: Challenges and Potentials of Digitalisation for Small and Mid-sized Towns: Proposition of a Transdisciplinary Research Agenda. 2021.

Nr. 37 Lechtenberg, S.; Hellingrath, B.: Applications of Artificial Intelligence in Supply Chain Management: Identification of main Research Fields and greatest Industry Interests. 2021.