# Ensuring the Canonicity of Process Models

Henrik Leopold[a], Fabian Pittke[b], Jan Mendling[b]

[a] *Vrije Universiteit Amsterdam*
*De Boelelaan 1081, 1081 HV Amsterdam,*
*The Netherlands*
[b] *WU Vienna, Austria*
*Welthandelsplatz 1, 1020 Wien, Austria*

**Abstract**

Process models play an important role for specifying requirements of business-related software. However, the usefulness of process models is highly dependent on their quality. Recognizing this, researches have proposed various techniques for the automated quality assurance of process models. A considerable shortcoming of these techniques is the assumption that each activity label consistently refers to a single stream of action. If, however, activities textually describe control flow related aspects such as decisions or conditions, the analysis results of these tools are distorted. Due to the ambiguity that is associated with this misuse of natural language, also humans struggle with drawing valid conclusions from such inconsistently specified activities. In this paper, we therefore introduce the notion of canonicity to prevent the mixing of natural language and modeling language. We identify and formalize non-canonical patterns, which we then use to define automated techniques for detecting and refactoring activities that do not comply with it. We evaluated these techniques by the help of four process model collections from industry, which confirmed the applicability and accuracy of these techniques.

*Keywords:* conceptual modeling, business process, canonical representation, non-canonicity patterns, pattern refactoring

## 1. Introduction

Process models are an important means to specify requirements in business-related software development projects [1]. Nevertheless, practitioners often struggle with the definition of fully correct and meaningful models [2]. The reasons for this are manifold. For instance, many modelers in practice have limited modeling experience [3], modeling projects often involve an overwhelming number of models [3], and the work of modelers involved in one project is difficult to coordinate [4]. The implications of incorrect and inconsistent models are severe. In the worst case, they entail wrong design decisions and a considerable increase of the overall development costs [5, 6].

To ensure process model correctness and consistency, researchers proposed several automated analysis techniques. Such techniques can, for instance, check whether a process model contains deadlocks [7], is compliant with expected behavior [8], and meets predefined naming conventions [9]. The shortcoming of these techniques is, however, that they already make assumptions about the way modelers have used natural language to label the process model activities. As a result, these techniques are hardly of any help if the logic of modeling and routing elements is textually described in activity labels. As an example, consider the activity "*Consult expert and prepare report*" from one of the models we encountered in practice. Apparently, this activity label consists of two separate activities, i.e., "*consult expert*" and "*prepare report*", which are

*Email addresses:* h.leopold@vu.nl (Henrik Leopold), fabian.pittke@wu.ac.at (Fabian Pittke), jan.mendling@wu.ac.at (Jan Mendling)

linked using the conjunction "*and*". The problem is that the execution semantics between these linked activity parts is not clearly defined. The word "*and*" might either refer to a parallel or a sequential execution. The specification of the activity as in this example mixes natural language and control structure in a way that is inherently ambiguous. This makes it impossible to draw valid conclusions from formal analysis results and, thus, difficult to develop process-related systems that are in line with the specification.

In this paper, we address this problem by introducing the notion of *canonicity* to prevent the mixing of natural language and modeling language. Based on this notion, we automatically check for problems caused by the violation of canonicity and point to reworks for resolving them. More specifically, we provide the following contributions. First, we introduce the notion of canonicity for process models and provide an operationalization of the concept. Second, we formalize a number of non-canonical patterns we discovered in models from practice. Third, we develop algorithms to recognize whether a given label suffers from these patterns and to refactor the detected cases into canonical model fragments. In order to demonstrate the applicability of the proposed techniques, we conduct extensive experiments with four real-world process model collections.

The rest of the paper is structured as follows. Section 2 illustrates the problems of non-canonical process model activity labels and reviews how prior research approaches have addressed this issue. Section 3 explains how we operationalize the notion of canonicity and our strategies to recognize and refactor instances that do not comply with it. Section 4 evaluates our techniques with process model collections from practice. In Section 5, we discuss implications and limitations of our work, before Section 6 concludes the paper.

## 2. Background

This section introduces the background of our research. First, Section 2.1 illustrates the problem of mixing modeling language and natural language and reflects upon the implications of non-canonical activities for system analysis and design. Section 2.2 discusses in how far prior research from the field of process model analysis has addressed the issue of non-canonical activities.

### *2.1. Problem Illustration*

In order to elaborate on the problem of non-canonical process model activities, we use the exemplary BPMN process model depicted in Figure 1. It shows a process model of a company receiving new goods that are meant to be stocked in the warehouse. The process starts by screening the documents of the delivery. Then, the delivery is identified within the company before the inspection of the delivery begins. Afterwards, it needs to be determined if the delivery is complete or not. Depending on the result, the missing items are either requested from the supplier or the inventory is updated and the delivery documents are archived. Finally, the delivered goods are moved to the warehouse.

Figure 1 shows several process model activities that are non-canonical. For example, the activity "*Screen delivery documents if necessary*" instructs the model reader to quickly check the delivery documents. Implicitly, it further instructs the reader to make a decision about the necessity. This means it would be more consistent to convey this decision through an exclusive choice gateway. The next example concerns the activity "*Delivery identification before inspection*". This activity describes a sequence of two activities with a particular order. Indeed, we first need to identify the delivery within the company and then inspect it accurately. This means it would be more consistent to create two separate activities in a sequence. Although both cases represent an inconsistent mix of natural and modeling language, it has to be noted that the intention of the modeler is clear. Nevertheless, there are also less understandable cases. For example, the activity "*Update inventory and archive delivery documents*" is ambiguous. Although we know that the activity involves an update of the inventory and an archiving of the delivery documents, we are unclear about the sequence of these activities. The intention of the conjunction "*and*" is not obvious and may be interpreted as a parallel as well as a sequential execution of the activities .

These examples illustrate that non-canonical activities can significantly reduce the usefulness of process models. First, non-canonical activities can obfuscate the information that a human reader can derive from a process model. Thus, models with such flaws do no provide a precise and clear specification of system

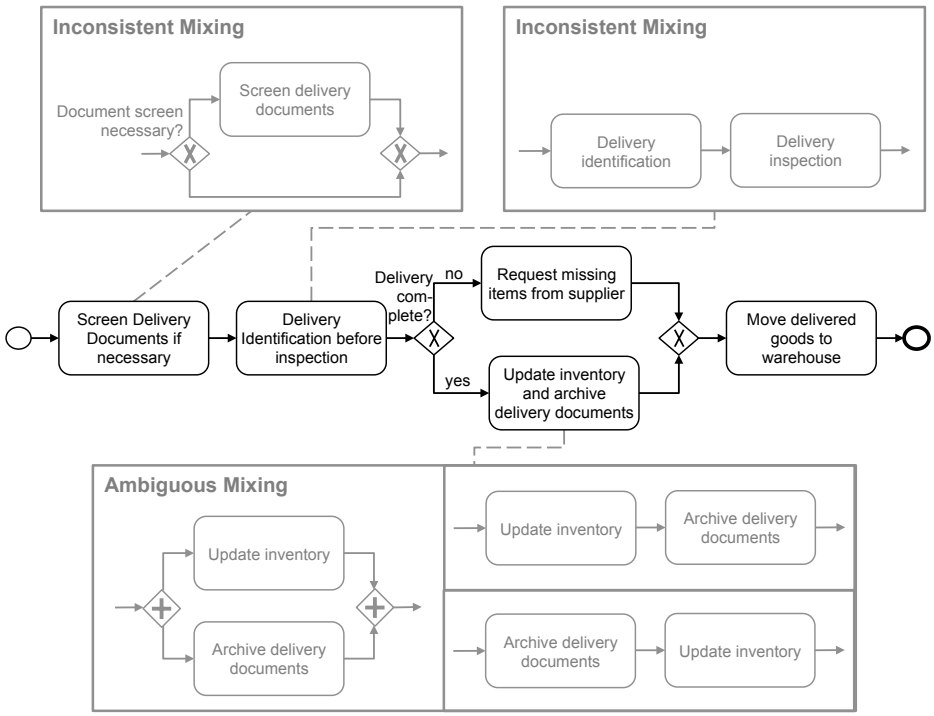



Figure 1: Process Model with Mixing Natural Language and Modeling Language

requirements. Second, non-canonical activities also affect the applicability of automated approaches. In the context of systems analysis and design, this particularly concerns techniques that aim to transform process models into executable process-aware systems [10, 11]. These approaches assume that each process model activity refers to a single system function that needs to be implemented. If system functions are incorporated in non-canonical activities as described above, the generated executable code will be incomplete.

In the next section, we investigate to what extent prior research has addressed the problem of non-canonical activities.

### *2.2. Prior Research*

Canonicity promotes the specification of process model elements in such a way that they do not refer to several streams of actions. In this way, canonicity defines a correctness criterion for process models that, if fulfilled, provides the basis for their analysis. Since there is a plethora of research approaches that verify particular properties of process models, we discuss prior research on model verification and assess their focus with regard to canonicity. Table 1 gives an overview of existing verification approaches. We can distinguish between those classes of verification techniques which relate to the structure of the model and those that consider text labels.

The verification of the *formal structure* of the model covers research that focus on the syntax and on the semantics of a process model. *Syntactical properties* define which elements of a process model are allowed to be connected upon which circumstances. Examples include the workflow net property [12], structural soundness of Petri nets [13] or syntactical correctness of EPCs [14]. If syntactical properties are fulfilled, *semantic properties* can be verified. Most prominent is the soundness property [12] and its derivatives for Petri nets [15, 17, 16, 18]. These analyses for Petri nets can be used for other languages by transformation from BPMN [30], BPEL [31], UML Activity Diagrams [32], EPCs [33, 34], and YAWL [35]. Comparable formalisms such as behavioural profiles [8] or meta graphs [19] are utilized as well. Efficient tools such as Woflan [36] or LoLa [37, 7] support the verification of various properties that tie to the formal structure of the model.

Table 1: Quality Assurance Approaches for Process Models

| Approach | Authors | Focus on Canonicity |
|---|---|---|
| **Quality Assurance of Model Structure** | | |
| Workflow net property | Van der Aalst [12] | None |
| Structural soundness | Weske [13] | None |
| Syntactical correctness | Mendling & Nüttgens [14] | None |
| Soundness | Van der Aalst [12] | None |
| Relaxed soundness | Dehnert et al. [15] | None |
| Lazy soundness | Puhlmann & Weske [16] | None |
| Soundness with data | Sidorova et al.[17] | None |
| Soundness derivates | Van der Aalst et al. [18] | None |
| Property verification of workflows with metagraphs | Basu & Blanning [19] | None |
| Consistency Measurement with behavioral profiles | Weidlich et al. [8] | None |
| **Quality Assurance of Labels** | | |
| *Syntactical Properties* | | |
| Correction of naming guideline violations | Leopold et al. [9] | Partly |
| Enforcement of naming guidelines with ontologies | Becker et al. [20, 21] | Partly |
| Assessment of naming conventions with grammar parsers | Delfmann et al. [22] | Partly |
| *Semantic Properties* | | |
| Linguistic consistency checking | Van der Vos et al. [23] | None |
| Detection and Resolution of ambiguous terms | Pittke et al. [24, 25] | None |
| Detection of non-uniformly specified terms | Koschmider & Blanchard [26] | None |
| Detection of semantic errors | Gruhn & Laue [27] | None |
| Detection of semantic errors | Weber et al. [28] | None |
| Linking process labels with ontology concepts | Di Francescomarino & Tonella [29] | None |

The verification of the *text labels* of a model has been studied in separate works. *Syntactical properties* of labels define whether a certain grammatical structure is used. As an example, consider the activity label "*Plan data transfer*", which may refer to the "*planning*" of a "*data transfer*" or the "*transfer*" of "*plan data*". Ambiguity can result from an unclear grammar of the label, such that it is widely recommended to use the verb-object style [38, 39, 40], which would suggest the action "*plan*" in this case. Different works support automatic checking of label syntax. Becker at al. propose ontology support to enforce specific naming guidelines right from the start of modeling [21, 20]. Delfman et al. apply standard grammar parsers to check naming conventions of model elements [22]. Leopold et al. propose an approach to detect linguistic guideline violations of modeling elements, such as activities, gateways, and events and to transform them into the desired style of labeling [9].

*Semantic properties* of labels relate to their meaning and potential issues of interpretation. A frequent problem is the usage of vaguely and ambiguously defined text labels. As an example, consider the label "*Assess application*" where the word "*application*" might either refer to a "*job application*" or a "*software program*". Such issues are discussed in the following works. The approach of Van der Vos et al. checks the semantic consistency of process models with linguistic relations [23]. Pittke et al. propose a technique that detects ambiguous terminology in process model elements with the help of WordNet [24] and BabelNet relations [25]. The approach of Koschmider et al. detects inconsistently specified terms with regard to the level of detail [26]. Gruhn and Laue investigate whether the natural language in activities violates the logic that is imposed by control flow splits. For example, an application cannot be rejected or accepted in the same process instance [27]. Weber et al. formalize an approach that checks whether the pre-conditions and post-conditions of activities are in line with the control flow [28]. Finally, the approach by Di Francescomarino and Tonella supports the annotation of process element labels to ontology concepts in order to avoid unclear or overlapping terms [29].

Our review of prior research illustrates that the concept of canonicity in process models has been only addressed to a limited extend. Most of the approaches focus on errors that either relate to the model definition or the textual naming of modeling elements, generally assuming that the respective elements are canonical and do not refer to several streams of action. Though not explicitly discussed, they require the concept of canonicity in order to provide reliable results. However, we also identified approaches that partly address the canonicity concept. For instance, the approach from Leopold et al. defines permissible naming guidelines and detects violations [9]. Typically, non-canonical labels are marked as violations. In a similar way, the approaches from Becker et al. enforce a particular naming style for model elements and thus indirectly facilitate model elements to comply with the canonicity concept [21, 20]. However, these approaches do not explicitly discuss the canonicity problem and only provide building blocks to tackle it. They do not provide a comprehensive solution.

In order to solve these problems, it is essential to clearly define the concept of canonicity in process models. Canonicity requires that each process model activity clearly refers to one distinct stream of action and that an activity does not incorporate additional information with regard to other perspectives of the process, such as the control flow, the data, or the resources perspective. Therefore, we develop a formal definition and operationalization of canonicity in the next section. Based on this definition we then specify respective identification and resolution mechanisms.

## 3. A Technique for Ensuring Canonicity in Process Models

In this section, we present our technique for recognizing and refactoring non-canonical process model activities. In Section 3.1, we operationalize the concept of canonicity and provide a formal definition. In Section 3.2 we introduce the formalism for recognizing non-canonical process model activities. In Section 3.3, we then introduce our technique to refactor them.

### *3.1. Operationalization of Canonicity*

In this paper, we use an adapted version of the definition of a process model introduced in [9]. On the one hand, this definition explicitly distinguishes between element types, such as gateways, events, and activities. On the other hand, it also refers to the textual content of process models and the assignment of text labels to these element types. As far as text labels are concerned, current research proposes that text labels should specify at least two components: an action and a business object to which the action is applied [39, 38]. In some cases, text labels may also specify additional information such as the receiver of a message or the purpose. As an example, consider the activity "*Move delivered goods to warehouse*". It contains the action "*to move*" and the business object "*delivered goods*". Moreover, it specifies that the goods are moved to the warehouse. Recognizing that actions represent verbs, business objects represent nouns, and additions are a composition of a preposition and a noun, we define a process model as follows.

**Definition 3.1.** (**Process Model**). A process model $P = (A, E, G, F, L, W_V, W_N, \lambda, \alpha, \beta, \gamma)$ consists of six finite sets $A$, $E$, $G$, $L$, $W_V$, $W_N$, a binary relation $F \subseteq (A \cup E \cup G) \times (A \cup E \cup G)$, a partial function $\alpha : L \nrightarrow W_V$, a partial function $\beta : L \nrightarrow W_N$, a partial function $\gamma : L \nrightarrow W_N$, and a partial function $\lambda : (A \cup E \cup G) \nrightarrow L$, such that

- $A$ is a finite non-empty set of activities.
- $E$ is a finite non-empty set of events.
- $G$ is a finite set of gateways.
- $W_V$ is a finite set of verbs.
- $W_N$ is a finite set of nouns.
- $F$ is a finite set of sequence flows. Each sequence flow $f \in F$ represents a directed edge between element types.

- $L$ is a finite set of text labels.
- The partial function $\alpha$ assigns an action $w_V \in W_V$ to a label $l \in L$.
- The partial function $\beta$ assigns a business object $w_N \in W_N$ to a label $l \in L$.
- The partial function $\gamma$ assigns an additional fragment $w_N \in W_N$ to a label $l \in L$.
- The partial function $\lambda$ defines the assignment of a label $l \in L$ to an activity $a \in A$, an event $e \in E$, or a gateway $g \in G$.

It has to be noted that this definition allows the assignment of several actions or business objects to an element type. As shown in Fig. 1, the activity "*Update inventory and archive delivery documents*" comprises two actions ("*to update*" and "*to archive*"), two business objects ("*inventory*" and "*delivery documents*"). Since the activity instructs to perform two different actions on two different business objects, it actually describes two individual streams of action, i.e., (1) the update of the inventory and (2) the archiving of the delivery documents. What is more, the activity includes logical information ("*and*") that adds a parallel or sequential character to it. According to the discussion from Section 2, activity labels should be free of such information. Therefore, we require an additional condition (*canonicity*) that prevents a mix of several components and logical constructs.

**Definition 3.2.** (**Canonicity**). A process model is *canonical* iff each activity label consists of exactly one action, one business object, and no more than one addition. Formally:

$$\forall l \in L : |\alpha(l)| = 1 \wedge |\beta(l)| = 1 \wedge |\gamma(l)| \leq 1$$

Applying this definition to the process model in Figure 1 reveals that the model does not comply with this condition. As already pointed out, the activity label "*Update inventory and archive delivery documents*" refers to two business objects and two actions. Similarly, the activity label "*Delivery identification before inspection*" refers to two actions and one business object. Building on this notion of canonicity, we develop recognition and refactoring mechanisms in the subsequent sections.

### *3.2. Automatic Recognition of Non-Canonical Activities*

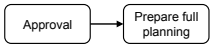
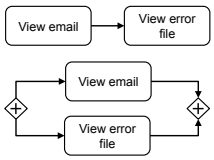
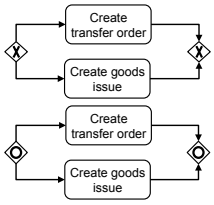
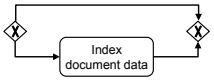
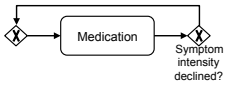
The automatic recognition of non-canonical process model activities requires a proper understanding of how the lack of canonicity manifests itself in process model activities. In prior work, we manually analyzed more than 12,000 activities from three large process model collections from industry and collected indicators for a lack of canonicity, such as keywords and certain grammatical structures [41]. As a result, we described a number of non-canonical patterns that are based on the indicators we found. To verify the completeness of the non-canonical patterns, we cross-checked the manual analysis of that paper and analyzed a process model collection from a telecommunication company. The outcome was a set of nine non-canonical patterns that covered all non-canonical instances of more than 25,000 process model activities. In the following, we discuss the identified patterns and represent their structure by the help of syntactical concepts in order to leverage the forthcoming automatic refactoring of these patterns. We distinguish between patterns that concern the control flow (Section 3.2.1), the resources and data perspective (Section 3.2.2), and implicit model elements (Section 3.2.3).

#### *3.2.1. Non-Canonical Control Flow Patterns*

In total, we found five non-canonical patterns that relate to the control flow of process models. Table 2 provides an overview of these patterns, their structure, and their interpretation.

Activities suffering from the *sequence* pattern incorporate sequential text information. This information imposes additional conditions on the task by stating that the task has to be executed in preparation for or as a consequence of another task. The pattern directly relates to the basic sequential flow of activities [42, 11, 13]. Typically, this pattern links two distinct activities by using temporal prepositions, such as "*before*" or "*after*", to express the order of activities. The canonicity of these labels is violated due to the

Table 2: Overview and Interpretation of Control Flow Patterns

| Pattern | Pattern Structure | Example | Interpretation |
|---|---|---|---|
| Sequence | $A_1$ $O_1$ "*before*" $A_2$ $O_2$<br>$A_1$ $O_1$ "*after*" $A_2$ $O_2$ | "*Prepare full planning after approval*"<br>"*Note requirement to collect excess after logdement*" |  |
| Parallel | $A_1$ $O_1$ "*and*" $A_2$ $O_2$<br>$A_1$ $O_1$ "*+*" $A_2$ $O_2$<br>$A_1$ $O_1$ "*&*" $A_2$ $O_2$<br>$A_1$ $O_1$ "*/*" $A_2$ $O_2$ | "*View email and error file*"<br>"*Start quick scan + clarify low level requirements*"<br>"*Assign carrier & prepare paperwork*"<br>"*Generate contract options / proposals*" |  |
| Decision | $A_1$ $O_1$ "*or*" $A_2$ $O_2$<br>$A_1$ $O_1$ "*/*" $A_2$ $O_2$ | "*Create transfer order or goods issue*"<br>"*Account login / register*" |  |
| Skip | *A O* "*as*" ("*required*"\| "*necessary*")<br>*A O* "*if*" ("*required*"\| "*necessary*") | "*Update claim exposure estimate as required*"<br>"*Index document data if necessary*" |  |
| Iteration | "*repeat*" *A O* "*until*" *C*<br>*A O* "*per*" *O*<br>*A O* "*for each*" *O* | "*Repeat medication until symptom intensity declines*"<br>"*Check SLA per client*"<br>"*Notify updated invoice for each order*" |  |

Legend: *A*: Action, *O*: Business Object, *C*: Condition or Status of a Business Object

inclusion of several actions, business objects, or additional information fragments. Thus, we describe the set of labels that fall into the *sequence* pattern as follows:

$$P_{Sequence} \quad = \quad \{l \in L \mid m(l, \texttt{*(before|after)*}) \wedge \ (|\alpha(l)| \neq 1 \vee |\beta(l)| \neq 1 \vee |\gamma(l)| > 1)\}$$

where $m(l,$`regex`$)$ denotes a logical predicate that evaluates to true if the label matches the regular expression `regex`.

Labels suffering from the *parallel activities* pattern combine several actions, business objects or combinations of these in a single activity element. Hence, a single activity element instructs people to perform multiple streams of action. Typically, this pattern includes the conjunction "*and*" or special characters such as "*+*", "&", or "/" to indicate several distinct activities. However, the interpretation of this pattern is not obvious. The meaning of the conjunction elements is not clearly defined and open for several interpretations. Consequently, the label may refer to a sequence of activities as well as to a parallel execution of multiple activities, which is another basic control flow pattern [42, 11, 13]. The canonicity condition is violated since these labels include several actions and business objects. Accordingly, we formalize the set of labels that violate the *parallel activities* pattern as follows:

$$P_{Parallel} \quad = \quad \{l \in L \mid m(l, \texttt{*(and|+|\&|/)*}) \wedge \ (|\alpha(l)| \neq 1 \vee |\beta(l)| \neq 1 \vee |\gamma(l)| > 1)\}$$

where $m(l,$`regex`$)$ denotes a logical predicate that evaluates to true if the label matches the regular expression `regex`.

The *decision* pattern implies a control flow split and may lead to several exclusive or inclusive streams of action. Similarly to the previous pattern, this pattern is using multiple actions, business objects, or combinations of these and thus violates canonicity. Typically, this pattern occurs when two alternatives are linked with the conjunction "*or*". Alternatively, the special character "/" may also point to this pattern. The interpretation of the *decision* pattern is also ambiguous. On the one hand, it might refer to an exclusive decision as presented in [42, 11, 13]. On the other side, the interpretation as an *inclusive or* gateway, which

would combine the possibilities of choosing among multiple alternatives [42, 43], is also valid. We formalize the set of labels that violate the *decision* pattern as follows:

$$P_{Decision} \quad = \quad \{l \in L \mid m(l, \texttt{*(or|/)*}) \wedge \ (|\alpha(l)| \neq 1 \vee |\beta(l)| \neq 1 \vee |\gamma(l)| > 1)\}$$

where $m(l,\texttt{regex})$ denotes a logical predicate that evaluates to true if the label matches the regular expression `regex`.

In general, the *skip* pattern implies a decision about an activity that needs only be conducted under specific conditions. If the conditions are not met, the activity is skipped and the process continues without executing this activity. Typically, this pattern combines prepositions, such as "*if*" or "*as*", with the past participle of the verb "*to require*" or adjectives that express optionality. The interpretation of the *skip* pattern corresponds to a consistent solution that is similar to the switch pattern for conditional routing [11] in which the respective activity is executed or not. In many cases, activity labels of this pattern correctly specify a single action, a single business object, and an optional addition. However, since they use additional words that do not belong to any of these components, they still violate canonicity. The optional character of this pattern has significant impact on the control flow and encodes relevant information in the activity label. We formalize the set of labels that violate the *skip* pattern as follows:

$$P_{Skip} \quad = \quad \{l \in L \mid m(l, \texttt{(*(if|as)(necessary|required))})\}$$

where $m(l,\texttt{regex})$ denotes a logical predicate that evaluates to true if the label matches the regular expression `regex`.

The *iteration* pattern is arranged in such a way that the natural language fragment asks for an iteration or a loop construct. In most of the cases, the iteration is expressed by the language pattern "*repeat ... until ...*" or a statement such as "*per object*". In these cases, the label also contains the iteration condition. The interpretation of the *iteration* pattern resembles the arbitrary cycle pattern [42, 13] or, more specifically, the while and the repeat pattern [11]. Similar to the *skip* pattern, the canonicity condition is violated because of the use of words that cannot be associated with any of the components action, business object, or addition. We formalize the set of labels that match the *iteration* pattern as follows:

$$P_{Iteration} \quad = \quad \{l \in L \mid m(l, \texttt{(repeat * until *|* per *)})\}$$

where $m(l,\texttt{regex})$ denotes a logical predicate that evaluates to true if the label matches the regular expression `regex`.

*3.2.2. Non-Canonical Resource and Data Patterns*

Table 3 provides an overview of patterns that hide resource and data information in text labels. Furthermore, we provide a core structure and interpretation of the patterns.

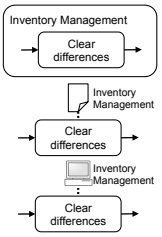
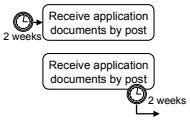
The *extra specification* pattern refers to activities that ambiguously incorporate additional information fragments into the activity label. The activity label violates the canonicity condition because of having more than one addition. The additional information themselves may include a specification of business objects, the refinement of entire activities into subprocesses, or the specification of process resources. The most prominent examples make use of brackets or dashes to indicate additional specifications. The interpretation of such labels is unclear and strongly dependent on the context. It may refer to multiple activities in form of a subprocess that are specified elsewhere [44]. Moreover, it may refer to resources that are used by the activity, such as data or systems [45, 46]. Accordingly, we formalize labels that fall into the *extra specification* pattern as follows:

$$P_{Extra} \quad = \quad \{l \in L \mid m(l, \texttt{(*(*)|*[*]|*:*)}) \wedge \ (|\alpha(l)| \neq 1 \vee |\beta(l)| \neq 1 \vee |\gamma(l)| > 1)\}$$

where $m(l,\texttt{regex})$ denotes a logical predicate that evaluates to true if the label matches the regular expression `regex`.

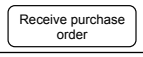
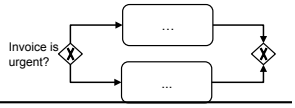
The *time exception* pattern is similar to the latter one. It, however, incorporates temporal information. This may include temporal prepositions that clarify the duration (e.g. in minutes, hours, or days) of an

Table 3: Overview and Interpretation of Resource and Data Patterns

| Pattern | Pattern Structure | Example | Interpretation |
|---|---|---|---|
| Extra Specification | *A O (O)*<br>*A O [O]*<br>*O : A O*<br>*O - A O* | "*Clear differences (Inventory Management)*"<br>"*Updating [Investment Projects]*"<br>"*Sales quotes: Budgeting*"<br>"*Rental Unit - Assign Application*" |  |
| Time Exception | *A O (CD ("min"\|"h"\|"day"\|"week"))* | "*Receive application documents by post (2 weeks)*" |  |

Legend: *A*: Action, *O*: Business Object, *CD*: Cardinal Number

Table 4: Overview and Interpretation of Implicit Object Patterns

| Pattern | Pattern Structure | Example | Interpretation |
|---|---|---|---|
| Implicit Action | *O C*<br>*A O "?"* | "*Purchase order received*"<br>"*Provide repairer with the job list?*" |  |
| Implicit Decision | "*determine*" if *O C*<br>"*check*" if *O C* | "*Determine if personal message is required*"<br>"*Check if invoice is urgent*" |  |

Legend: *A*: Action, *O*: Business Object, *C*: Condition or Status of a Business Object

activity or other time-related constraints and exceptions. Typically, the time information is provided in brackets and, in many cases, unclear. The temporal information may represent waiting time, i.e., time that must pass before the process continues normally [47], or the temporal information could be interpreted in the sense of an attached intermediate event. The latter implies that the execution of the activity is canceled as soon as the time limit is reached triggering a new stream of actions that is not specified [48]. Altogether, labels belonging to the *time exception* pattern are described as follows:

$$P_{Time} \quad = \quad \{l \in L | m(l, \texttt{*((0-9)(min|h|day|week))}) \ \wedge \ (|\alpha(l)| \neq 1 \vee |\beta(l)| \neq 1 \vee \ |\gamma(l)| > 1)\}$$

where $m(l,\texttt{regex})$ denotes a logical predicate that evaluates to true if the label matches the regular expression `regex`.

### *3.2.3. Implicit Element Error Patterns*

Finally, we discuss patterns that implicitly refer to decisions and actions. Table 4 shows the core pattern structure and the consistent interpretations of these patterns.

Process model activities suffering from the *implicit action* pattern erroneously combine labeling style and modeling construct, i.e. activity, event, or gateway. As an example, consider an activity that is described as an event or vice versa. As shown in the examples of the table, the activity "*Purchase order received*" might refer to the respective event or the activity "*Receive purchase order*". Moreover, they also violate the canonicity criterion because they specify a particular state or condition of an object rather than an action that needs to be applied on the object. As far as the interpretation is concerned, we assume that the modeling construct takes precedence over the text label. Following this assumption, the text label would imply an activity in the process. Thus, we interpret these cases as a regular activity with a correct style of labeling [39]. Accordingly, we define the set of all Implicit Action labels as follows:

$$P_{IAction} \quad = \quad \{l \in L \mid m(l, \texttt{*?}) \vee (|\alpha(l)| \neq 1 \vee |\beta(l)| \neq 1 \vee \ |\gamma(l)| > 1)\}$$

**Algorithm 1:** Pattern Detection from Process Model Activity Label

```
1: detectAntiPatterns(Label l)
2: if l.matches('(before|after)') ∧ isCanonical(l) = false then
3:     l.violatesSequencePattern(true)
4: if l.matches('(and|+|&|/)') ∧ isCanonical(l) = false then
5:     l.violatesMultiplePattern(true)
6: if l.matches('(or|/)') ∧ isCanonical(l) = false then
7:     l.violatesDecisionPattern(true)
8: if l.matches('(as|if) (required|necessary)') then
9:     l.violatesSkipPattern(true)
10: if l.matches('(repeat * until|per)') then
11:     l.violatesIterationPattern(true)
12: if l.matches('((*)|[*]|:|-)') ∧ isCanonical(l) = false then
13:     l.violatesExtraInformationPattern(true)
14: if l.matches('(CD (min|h|day|week))') ∧ isCanonical(l) = false then
15:     l.violatesTimePattern(true)
16: if l.matches('?') ∨ isCanonical(l) = false then
17:     l.violatesImplicitActionPattern(true)
18: if l.matches('(determine|check) if') ∧ isCanonical(l) = false then
19:     l.violatesImplicitDecisionPattern(true)
20: return l
```

where $m(l,\texttt{regex})$ denotes a logical predicate that evaluates to true if the label matches the regular expression `regex`.

The *implicit decision* pattern entails a decision in the process flow. It includes a specific condition that has to be checked in order to proceed. Many instances of this pattern contain a verb asking for the verification or investigation of the conditions and the conditional word *if*. Regular language patterns include *determine if*, *validate if*, *check if*, and *confirm if*. However, the respective activities only contain an action, but further need a business object. Therefore, these activities are not canonical. The interpretation is equal to the basic construct of an exclusive choice [42, 42]. All labels that fall into the *implicit decision* pattern are formalized as follows:

$$P_{IDecision} = \{l \in L \mid m(l, \texttt{(check|determine) if *}) \wedge (|\alpha(l)| \neq 1 \vee |\beta(l)| \neq 1 \vee |\gamma(l)| > 1)\}$$

where $m(l,\texttt{regex})$ denotes a logical predicate that evaluates to true if the label matches the regular expression `regex`.

*3.2.4. Automatic Detection of Non-canonical Patterns*

From a technical perspective, the pattern detection approach analyzes whether the input label matches a particular non-canonical pattern and marks the recognized pattern in case of a positive match. For this purpose, the approach uses the formalization presented in the sections 3.2.1, 3.2.2 and 3.2.3. The detection itself works in two steps. The first step evaluates if the activity label contains specific keywords that point towards a certain pattern. In a second step, the approach checks whether the activity label violates the criterion of canonicity by extracting actions, business objects as well as additions and counting the appearances of the respective component. In case of positive evaluations, the label is marked with the respective pattern. Algorithm 1 summarizes the pattern detection for each pattern.

*3.3. Refactoring of Non-canonical Activities*

The refactoring of non-canonical activities consists of two main steps. In the first step, we extract the relevant information from the activity label. In the second step, we reorganize the extracted information in such a way that canonicity is restored. The following subsections elaborate on the details of both steps.

**Algorithm 2:** Component Extraction from Process Model Activity Label

```
1: parseNonCanonicalLabel(Label l)
2: LabelComponents lc = new LabelComponents()
3: List CanonicalLabels ← ∅
4: List conditions ← ∅
5: List extraInfo ← ∅
6: if l.matchesSequencePattern() ∨ l.matchesMultiplePattern() ∨ l.matchesDecisionPattern() then
7:    CanonicalLabels ∪ l.splitByKeyword()
8: if l.matchesSkipPattern() then
9:    CanonicalLabels ∪ l.removeKeyword()
10: if l.matchesIterationPattern() then
11:    CanonicalLabels ∪ l.splitByKeyword()[1]
12:    conditions ∪ l.splitByKeyword()[2]
13: if l.matchesExtraPattern()∨ l.matchesTimePattern() then
14:    CanonicalLabels ∪ l.removeExtraInfos()
15:    extraInfo ∪ l.extractInfoByKeyword()
16: if l.matchesImplicitActionPattern() then
17:    l ← l.removeKeyword()
18:    α(l) ← transformStatusToAction(l)
19: if l.matchesImplicitDecisionPattern() then
20:    conditions ∪ l.removeKeyword()
21: lc ← AL ∪ condition ∪ extraInfo
22: return lc
```

*3.3.1. Information Extraction*

The first step concerns the extraction of relevant information from the deficient model activity. Depending on the respective pattern, we need to remove specific keywords and extract relevant text fragments, such as conditions and resources. Moreover, we also identify actions and business objects by using available label annotation techniques [49]. Reconsider the activity label "*Investigate risk or credit history of customer*" from Figure 1. Based on the detection technique, we recognize the *Decision* non-canonicity pattern, because the activity label uses the keyword "*or*" and consists of two different business objects that indicate two different streams of action. Accordingly, the label parsing technique removes the keyword "*or*" and extracts the action "*to investigate*", the business objects "*risk*" and "*credit history*" as well as one additional fragment "*of customer*". As the second business object lacks an action, the component also adds the action "*to investigate*" to restore the canonicity of the second activity.

This procedure is formalized by Algorithm 2. The algorithm takes a label $l$ as an input, which was marked with the respective pattern violations. In the beginning, we initialize a label component data structure $lc$ to store the extracted components (Step 2) as well as three lists for the canonical activity labels, the conditions, and the extra information that may be specified in a non-canonical label (Steps 3-5). In the following steps, we extract the relevant parts depending on the identified pattern. If the *sequence* pattern, the *multiple activity* pattern, or the *decision* pattern have been recognized, we split the label using the respective keyword and add the parts to the canonical label list (Steps 6–7). If the label matches the *skip* pattern, the parsing algorithm solely removes the text fragment that indicates the optionality of the label and adds the canonical activity to the list (Steps 8–9). In case of the *iteration* pattern, the algorithm splits the label at the keyword position, resulting in two parts. According to the pattern, the first part specifies the activity and is stored in the activity list. The second part, which relates to the iteration condition, is stored in the condition list (Steps 10–12). Step 13 processes labels that violate the *extra specification* and the *time exception* pattern are processed. On the one hand, the algorithm removes the additional information and stores the canonical activity in the activity list (Step 14). On the other hand, it also stores the extra information in a separate list (Step 15). If the label matches the implicit *decision* pattern, the algorithm begins with removing specific keywords (Step 17), before it assigns the missing action the label by transforming the state or condition of a business object into an action (Step 18). Finally, the algorithm parses implicit decision labels by removing

the keywords and adding the remaining part to the set of conditions (Steps 19–20). The algorithm terminates by adding the activity, the condition, and the extra information list to the set to the label component (Step 21) and returning the label component (Step 22).

*3.3.2. Canonicity Restoration*

The second step concerns the actual restoration of the canonicity of deficient model activities. To this end, we have to remove the deficient activity, insert a blank model fragment based on the possible interpretations of the patterns, and instantiate them with proper element names. Moreover, we take the label component data structure of the label parsing as input and use these components to assign labels to model fragments. In the following, we describe the detailed transformation steps for each pattern separately.

In case of the *sequence* pattern, the deficient activity label needs to be replaced by two distinct activities. In order to enforce a strong order relation between these two activities, we further need to specify a directed sequence flow between the two added activities. Finally, the activities are assigned with the respective label from the canonical activity list from the label component. Each of these steps is described in the following definition.

**Definition 3.3.** (**Sequence Pattern Refactoring**) Let $a^1 \in A$ be an activity that matches the *sequence* non-canonicity pattern, $(x, a^1), (a^1, y) \in F$ the incoming and outgoing sequence flows of $a^1$, and $lc$ the parsed label component. The activity $a^1$ is refactored with the following steps:

- $A \setminus \{a^1\} \cup \{a_1, a_2\}$
- $F \setminus \{(x, a^1),(a^1, y)\} \cup \{(x, a_1),\ (a_1, a_2),(a_2, y)\}$
- $\lambda(a_1) = lc.AL[1]$, $\lambda(a_2) = lc.AL[2]$

The refactoring of the *parallel activities* and the *decision* pattern are very similar. In both cases, the refactoring involves the removal of the deficient activity and the insertion of a splitting gateway, a joining gateway, and a number of blank activities depending on the number of canonical labels that are stored in the canonical activity list. Afterwards, control flow edges are added to implement a parallel or interleaving order of activities. Finally, each blank activity is assigned a label from the list of canonical activities. Note that parallel gateways are used in case of the *parallel Activities* pattern and inclusive or exclusive decision gateways in case of the *decision* pattern. The following definitions describe the refactoring steps:

**Definition 3.4.** (**Parallel Activities Pattern Refactoring**) Let $a^2 \in A$ be an activity that matches the *parallel activities* non-canonicity pattern, $(x, a^2), (a^2, y) \in F$ the incoming and outgoing sequence flows of $a^2$, and $lc$ the parsed label component. The activity $a^2$ is refactored with either one of the following alternatives:

1. Refactoring with parallel activities
   - $A \setminus \{a^2\} \cup \{a_1, ..., a_n\}$
   - $G \cup \{g^S_{and}, g^J_{and}\}$
   - $F \setminus \{(x, a^2), (a^2, y)\} \cup \{(x, g^S_{and}),\ (g^S_{and}, a_1),...,$ $(g^S_{and}, a_n),\ (a_1, g^J_{and}),...,\ (a_n, g^J_{and}),\ (g^J_{and}, y)\}$
   - $\lambda(a_1) = lc.AL[1]$, ..., $\lambda(a_n) = lc.AL[n]$
2. Refactoring with a sequence of activities
   - $A \setminus \{a^2\} \cup \{a_1, a_2\}$
   - $F \setminus \{(x, a^2),(a^2, y)\} \cup \{(x, a_1),\ (a_1, a_2),(a_2, y)\}$
   - $\lambda(a_1) = lc.AL[1]$, $\lambda(a_2) = lc.AL[2]$

**Definition 3.5.** (**Decision Pattern Refactoring**) Let $a^3 \in A$ be an activity that matches the *decision* non-canonicity pattern, $(x, a^3), (a^3, y) \in F$ the incoming and outgoing sequence flows of $a^3$, and $lc$ the parsed label component. The activity $a^3$ is refactored with either one of the following alternatives:

1. Refactoring with exclusive decision
   - $A \setminus \{a^3\} \cup \{a_1, ..., a_n\}$
   - $G \cup \{g^S_{xor}, g^J_{xor}\}$
   - $F \setminus \{(x, a^3), (a^3, y)\} \cup \{(x, g^S_{xor}), (g^S_{xor}, a_1), ..., (g^S_{xor}, a_n), (a_1, g^J_{xor}), ..., (a_n, g^J_{xor}), (g^J_{xor}, y)\}$
   - $\lambda(a_1) = lc.AL[1]$, ..., $\lambda(a_n) = lc.AL[n]$
2. Refactoring with inclusive decision
   - $A \setminus \{a^3\} \cup \{a_1, ..., a_n\}$
   - $G \cup \{g^S_{or}, g^J_{or}\}$
   - $F \setminus \{(x, a^3), (a^3, y)\} \cup \{(x, g^S_{or}), (g^S_{or}, a_1), ..., (g^S_{or}, a_n), (a_1, g^J_{or}), ..., (a_n, g^J_{or}), (g^J_{or}, y)\}$
   - $\lambda(a_1) = lc.AL[1]$, ..., $\lambda(a_n) = lc.AL[n]$

The refactoring of the *skip* pattern mainly involves the same steps as the *decision* pattern. Accordingly, the defective activity is removed from the process model and a model fragment is added that specifies an exclusive decision. In contrast to the decision pattern, one path of this decision is empty in order to skip the activity if necessary. The following definition summarizes the necessary refactoring steps:

**Definition 3.6.** (**Skip Pattern Refactoring**) Let $a^4 \in A$ be an activity that matches the *skip* non-canonicity pattern, $(x, a^4), (a^4, y) \in F$ the incoming and outgoing sequence flows of $a^4$, and $lc$ the parsed label component. The activity $a^4$ is refactored with the following steps:

- $A \setminus \{a^4\} \cup \{a'\}$
- $G \cup \{g^S_{xor}, g^J_{xor}\}$
- $F \setminus \{(x, a^4), (a^4, y)\} \cup \{(x, g^S_{xor}), (g^S_{xor}, a'), (a', g^J_{xor}), (g^S_{xor}, g^J_{xor}), (g^J_{xor}, y)\}$
- $\lambda(a') = lc.AL[1]$

The refactoring of the *iteration* patterns needs to replace the deficient activity with an arbitrary loop construct. Consequently, we insert two gateways together with a blank activity to the process model. In contrast to the previously discussed patterns, we need to alter the positions of the exclusive join and the exclusive split gateway to implement the loop. Afterwards, we assign the activity label and the iteration condition to the respective constructs.

**Definition 3.7.** (**Iteration Pattern Refactoring**) Let $a^5 \in A$ be an activity that matches the *iteration* non-canonicity pattern, $(x, a^5), (a^5, y) \in F$ the incoming and outgoing sequence flows of $a^5$, and $lc$ the parsed label component. The activity $a^5$ is refactored with the following steps:

- $A \setminus \{a^5\} \cup \{a'\}$
- $G \cup \{g^S_{xor}, g^J_{xor}\}$
- $F \setminus \{(x, a^5), (a^5, y)\} \cup \{(x, g^J_{xor}), (g^J_{xor}, a'), (a', g^S_{xor}), (g^S_{xor}, g^J_{xor}), (g^S_{xor}, y)\}$
- $\lambda(a') = lc.AL[1]$
- $\lambda(g^S_{xor}) = lc.conditions[1]$

The refactoring of the *extra specification* pattern is not straightforward. As the additional information may refer to different aspects of a process model, such as resources, roles, or even sub-process activities, we cannot propose a comprehensive refactoring of this pattern. In general, we replace the original label with an canonical one and set the control flow edges accordingly. For the extra information fragment, we require the decision of a business analyst to correctly specify this information with available modeling syntax.

**Definition 3.8.** (**Extra Specification Pattern Refactoring**) Let $a^6 \in A$ be an activity that matches the *extra specification* non-canonicity pattern, $(x, a^6), (a^6, y) \in F$ the incoming and outgoing sequence flows of $a^6$, and $lc$ the parsed label component. The activity $a^6$ is refactored with the following steps:

- $A \setminus \{a^6\} \cup \{a', a^S\}$
- $F \setminus \{(x, a^6), (a^6, y)\} \cup \{(x, a'), (a', y)\}$
- $\lambda(a') = lc.AL[1]$

The refactoring of the *time exception* pattern requires the insertion of a new activity and a new event. The activity replaces the deficient one and is labeled as an canonical activity. The new event precedes the activity and is labeled with the respective time information of the original label. These steps are formally described by the following definition:

**Definition 3.9.** (**Time Exception Pattern Refactoring**) Let $a^7 \in A$ be an activity that matches the *time exception* non-canonicity pattern, $(x, a^7), (a^7, y) \in F$ the incoming and outgoing sequence flows of $a^7$, and $lc$ the parsed label component. The activity $a^7$ is refactored with the following steps:

- $A \setminus \{a^7\} \cup \{a'\}$
- $E \cup \{e'\}$
- $F \setminus \{(x, a^7), (a^7, y)\} \cup \{(x, e'), (e', a'), (a', y)\}$
- $\lambda(a') = lc.AL[1]$
- $\lambda(e') = lc.extraInfo[1]$

The refactoring of an *implicit action* pattern is the most straightforward one since it only requires the transformation of the condition or the status into an imperative action. Thus, we only need to reassign the activity with the correctly parsed label without inserting a new activity.

**Definition 3.10.** (**Implicit Action Pattern Refactoring**) Let $a^8 \in A$ be an activity that matches the *implicit action* non-canonicity pattern and $lc$ the parsed label component. The activity $a^8$ is refactored with the following steps:

- $\lambda(a^8) = lc.AL[1]$

The refactoring of the *implicit decision* pattern is similar to the *decision* pattern. We remove the deficient activity and replace it with an exclusive choice model fragment. However, as the activities falling into that pattern do not specify subsequent activities, we cannot fill the blank activities with meaningful labels. Therefore, we only assign a gateway label to the split gateway at the beginning of the fragment.

**Definition 3.11.** (**Implicit Decision Pattern Refactoring**) Let $a^9 \in A$ be an activity that matches the *implicit action* non-canonicity pattern, $(x, a^9), (a^9, y) \in F$ the incoming and outgoing sequence flows of $a^9$, and $lc$ the parsed label component. The activity $a^9$ is refactored with the following steps:

- $A \setminus \{a^9\} \cup \{a_1, ..., a_n\}$
- $G \cup \{g^S_{xor}, g^J_{xor}\}$
- $F \setminus \{(x, a^9), (a^9, y)\} \cup \{(x, g^S_{xor}), (g^S_{xor}, a_1), ...,$
  $(g^S_{xor}, a_n), (a_1, g^J_{xor}), ..., (a_n, g^J_{xor}), (g^J_{xor}, y)\}$
- $\lambda(g^S_{xor}) = lc.conditions[1]$

## 4. Evaluation

In this section, we present the results of an evaluation with four large process model collections. The goal of the evaluation was to demonstrate the applicability of the presented techniques in terms of accuracy. Section 4.1 first discusses the evaluation setup. Section 4.2 then introduces the test data of our evaluation. Sections 4.3 and 4.4 finally present the experimental results of the detection and the refactoring.

### *4.1. Evaluation Setup*

The overall goal of the evaluation is to assess the performance of the presented techniques. To this end, we created a human benchmark against which we could compare the algorithmic results. The creation of the human benchmark involved two researchers, who examined the activities of the process models from the test colllection and classified them according to the presented patterns. The researchers identified non-canonical labels independently from each other and resolved contradicting cases afterwards. In a subsequent step, the researchers refactored the identified instances by using the possible interpretations of the non-canonical activity.

Using the human benchmark, we were able to compare the algorithmic and the manually examined results and to classify each activity as either true-positive, true-negative, false-positive, or false-negative. Based on these sets, we respectively calculated the metrics precision and recall [50]. In the context of the detection of non-canonical activities, the precision is the number of correctly recognized pattern instances divided by the total number of pattern instances retrieved by the algorithms. The recall is the number of correctly recognized patterns instances divided by the total number of patterns instances. As it is important that both precision and recall yield sufficiently high values, we also take the f-measure (harmonic mean of precision and recall) into account.

For running the experiments, we implemented the algorithms in the context of a Java prototype. We deployed the techniques on a MacBook Pro with a 2.4 GHz Intel Core Duo processor and 4 GB RAM, running on Mac OS X 10.7.5 and Java Virtual Machine 1.7. The next section introduces the data sets we used for the evaluation experiments.

### *4.2. Data Sources*

To demonstrate that the proposed techniques are applicable to a wide range of real-world process models, we selected process models from different input sources and with varying characteristics. As shown in Table 5, the selected process model collections are heterogeneous with regard to the characteristics such as size, standardization, the expected degree of modeling quality, and the modeling domain:

- **SAP Reference Model**: The SAP Reference Model (SRM) contains 604 Event-Driven Process Chains organized in 29 different functional branches [51]. Examples are procurement, sales, and financial accounting. The model collection includes 2,432 activity labels. Since the SAP Reference Model was designed as an industry-independent recommendation, we expect a high degree of standardization and a high model quality which should result in smaller numbers of non-canonical pattern occurrences.

- **Insurance Model Collection**: The Insurance Model Collection (IMC) contains 349 EPCs dealing with the claims handling activities of a large insurance company. In total, the models include 1840 activities and hence are slightly smaller than the models from the SRM. Compared to SRM, we expect a bigger number of pattern occurrences due to the low level of competence of casual modelers [3, 52] and higher error rates in industry model collections [53].

- **AI Collection**: The models from the AI collection stem from academic training and cover diverse domains (see http://bpmai.org). From the available models, we filtered those with proper English labels. The resulting subset includes 1,091 process models with 8,339 activity labels. Since the collection targets no specific industry and has been mainly created by students, the number of pattern occurrences is expected to be the highest among all considered collections.

Table 5: Demographics of the Applied Process Model Collections

| **Characteristic** | **SRM** | **IMC** | **AI** | **TelCo** |
|---|---|---|---|---|
| No. of Models | 604 | 349 | 1,091 | 803 |
| No. of Activity Labels | 2,432 | 1,840 | 8,339 | 12,088 |
| Modeling Language | EPC | EPC | BPMN | EPC |
| Domain | Independent | Insurance | Academic Training | Communication |
| Standardization | High | Medium | Low | Medium |

**Legend**: SRM: SAP Reference Model Collection, IMC: Insurance Model Collection, AI: Academic Initiative Collection, TelCo: Telecommunication Model Collection

Table 6: Pattern Detection Results

| | | **P1** | **P2** | **P3** | **P4** | **P5** | **P6** | **P7** | **P8** | **P9** |
|---|---|---|---|---|---|---|---|---|---|---|
| **SRM** | Precision | - | .94 | .98 | - | - | 1 | - | .2 | - |
| | Recall | - | .97 | .94 | - | - | .99 | - | 1 | - |
| | F-measure | - | .95 | .96 | - | - | .99 | - | .33 | - |
| **IMC** | Precision | 1 | .99 | .96 | 1 | - | .94 | - | .65 | .98 |
| | Recall | 1 | 1 | 1 | 1 | - | .89 | - | .75 | .98 |
| | F-measure | 1 | .99 | .98 | 1 | - | .91 | - | .70 | .98 |
| **AI** | Precision | 1 | .96 | .93 | 1 | 1 | .99 | 1 | .92 | .98 |
| | Recall | .80 | 1 | 1 | 1 | .71 | .98 | .42 | .72 | .86 |
| | F-measure | .89 | .98 | .96 | 1 | .83 | .98 | .59 | .81 | .92 |
| **TelCo** | Precision | 1 | .98 | .99 | 1 | .89 | .97 | - | .74 | 1 |
| | Recall | 1 | .95 | 1 | .89 | .94 | .56 | - | .73 | .98 |
| | F-measure | 1 | .96 | .99 | .94 | .92 | .71 | - | .73 | .99 |

- **TelCo Collection**: The TelCo collection contains the processes from an international telecommunication company. It comprises 803 process models with 12,088 activities in total. Thus, the TelCo collection is the biggest model collection which we employ for our experiments. Similar to the IMC collection, the TelCo collection was also created by casual and semi-professional modelers which leads to the assumption that the techniques will recognize pattern occurrences between the SAP and the AI collection.

### *4.3. Detection Results*

Table 6 shows precision, recall, and f-measure of the pattern detection.

In general, the numbers show that our pattern detection techniques works satisfactory. In the majority of the cases, the f-measure is higher than 80%, indicating a reliable recognition of relevant deficient model activities. However, we also observe notable deviations among the different process model collections and among the patterns.

With respect to the deviations among the *process model collections*, we observe that modeling experience and standardization are reflected by the results. In the SRM, the algorithm did not find any instances of the *sequence* pattern (P1), the *skip* pattern (P4), and the *implicit decision* pattern (P9). This observation confirms our initial assumption that the SRM, as a reference model collection for several industries, is characterized by a consistent and correct modeling style. The TelCo and the AI collection, by contrast, contain a considerable number of these patterns. While we expected a notable number of pattern occurrences in the AI collection, we surprisingly also found many of these patterns in the TelCo collection. In general, the results demonstrate the validity and generalizability of the patterns to other process model collections.

Within the *patterns*, the performance of the algorithm is stable and does not deviate in most of the cases. However, we observe a significant deviation of the performance for the *implicit action* pattern (P8) in the

Table 7: Pattern Refactoring Results

| | | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **SRM** | $L_c$ | - | 102 | 47 | - | - | 168 | - | 2 | - |
| | $L_e$ | - | 30 | 19 | - | - | 22 | - | 3 | - |
| | $AR$ | - | .77 | .71 | - | - | .88 | - | .40 | - |
| **IMC** | $L_c$ | 5 | 297 | 101 | 44 | - | 60 | - | 17 | 156 |
| | $L_e$ | 0 | 94 | 52 | 4 | - | 5 | - | 14 | 3 |
| | $AR$ | 1 | .76 | .66 | .92 | - | .92 | - | .55 | .98 |
| **AI** | $L_c$ | 7 | 327 | 58 | 1 | 23 | 119 | 31 | 215 | 42 |
| | $L_e$ | 3 | 110 | 30 | 0 | 9 | 13 | 0 | 107 | 3 |
| | $AR$ | .7 | .75 | .66 | 1 | .72 | .90 | 1 | .67 | .93 |
| **TelCo** | $L_c$ | 5 | 1002 | 283 | 2 | 15 | 60 | - | 120 | 64 |
| | $L_e$ | 0 | 293 | 117 | 2 | 2 | 31 | - | 95 | 1 |
| | $AR$ | 1 | .77 | .71 | .5 | .88 | .66 | - | .56 | .98 |

SRM collection and the *extra information* pattern (P6) in the TelCo collection. In the former case, the precision only amounts to 0.2, which reveals a large number of false positives. A deeper investigation showed that these labels contain business objects that could be misinterpreted as a state. As examples, consider the labels "*Parked Document Posting*" or "*Earned value calculation*". The detection algorithm suffers from the ambiguity of the first word. Instead of recognizing it as being part of the real business object ("*parked document*"), the algorithm interprets the word as a state of another business object ("*document posting*"), which does not resemble the original intention ("*post a parked document*"). In the latter case, the technique suffers from a low recall of 0.56. Again, we further investigated the reasons and found that many of the activities in the TelCo collection miss a signal phrase to indicate this pattern. For example, the label "*PLC 231 Create detailed configuration system*" incorporates the resource statement "*PLC 231*". However, this statement is not separated by a dash or a colon. Therefore, it does not result in a violation of the respective pattern. Despite these deviations, we consider the algorithm to work accurately and of being suitable to successfully detect patterns in real-world process models.

### *4.4. Refactoring Results*

In order to evaluate the pattern refactoring technique, we focus on the results of the information extraction technique, because the created model fragments highly depend on the label components that have been derived. Thus, we determine whether or not the defective label has been parsed properly and whether it would label the respective modeling fragment correctly. As a baseline, we consider all activities that violate a particular pattern as the set of all labels that should be corrected. Within this set, a label is either appropriately or erroneously corrected. Therefore, we define the metric *refactoring precision* as the share of appropriately corrected activities compared to all activities. Let $L_c$ be the set of all labels that have been parsed correctly and $L_e$ the set of labels that have been parsed with errors. Then, the refactoring precision (RP) is given as follows:

$$RP = \frac{|L_c|}{|L_c|+|L_e|} \tag{1}$$

Table 7 summarizes the results of appropriately and erroneously refactored labels as well as the corresponding refactoring precision. The results show that the correction technique is working satisfactory (RP score of 0.8 on average) and it is fairly stable among the different patterns. However, we also observe notable differences depending on the pattern type. If we consider the *parallel activity* (P2) and the *decision* pattern (P3), the RP score is stable around 0.7 to 0.75, yet rather small in comparison to other patterns. An analysis of erroneous corrections revealed the *referential ambiguity* as the main reason. For instance, consider the activity label "*Create and submit quote and photos using fax or Email*". In this case, the relation between actions and business objects is more complicated because the two actions ("*create*" and "*submit*") might

either refer to one or two business objects ("*quote*" and "*photos*"). Ultimately, this might result in three or four distinct activities. Another example involves the correction of the *implicit action* pattern (P8) with an RP score of only around 0.50. A deeper investigation revealed that a corrected label has not been created because the business object was not recognized (e.g. see the label "*rcs software update embedded*") or the state could not be recognized and transformed into an appropriate action (e.g. see the label "*created detailed implementation plan*"). These effects are worsened if a label violates several patterns at the same time, as for instance in the label "*implemented offer change or removal*", which makes the correction of these patterns challenging. Despite these deviations, we regard the performance of the correction technique as satisfactory given the various possibilities of natural language to combine several patterns in one activity label.

## 5. Implications

This section discusses the implications of our research. Section 5.1 and 5.2 identify implications of our work for research and for practice. Section 5.3 reflects upon threats to validity.

### *5.1. Implications for Research*

As a major implication for research, we introduced the notion of canonicity for process models. Canonicity enables a unique specification of model activities since each model activity refers to only one stream of action. This concept has notable impacts on process and workflow model analysis techniques since it uncovers hidden aspects of process models. For example, canonicity explicates hidden control flow or data objects within process models, which once refactored become accessible to analysis techniques of various kinds [19, 54, 55]. Thus, it contributes to the reliable applicability of process models in different scenarios, such as the design of workflow systems requiring a flawless specification of the underlying process [56].

Furthermore, we structure a yet unformalized dimension of process model quality. In a broader setting, model quality has been discussed in terms of syntactical, semantic, and pragmatic aspects [57, 58]. However, these discussions are not aware of quality aspects with regard to the textual content or a combination of textual content and formal content. The results of this paper complement the concept of process model quality by addressing the combination of formal and textual content and proposing a distinction between them. Thus, ambiguous and misleading process descriptions are identified and refactored in order to best resemble the underlying process.

In addition, the identified patterns foster the creation of a new process model conceptualization, in which each activity only refers to one stream of action which is applied on one object. An operationalization of this concept is highly beneficial for several process model analysis techniques and for applications that rely on syntactically and semantically sound process models, such as the generation of process descriptions [59], automatic execution [60], or process model merging [61]. So far, these techniques assume that each activity contains a single piece of information that is not subject to additional conditions. The new concept overcomes this restriction and provides means that permit efficient and accurate analysis of process models.

### *5.2. Implications for Practice*

The findings of this paper also have two important implications for practice. First, the explication of the patterns can be used to extend the capabilities of process modeling tools. In such a context, tools can be used to automatically look for patterns and to provide possible solutions to resolve these patterns during process modeling. Thus, modeling errors could be prevented right from the beginning, which would save cost and time-intensive rework in later stages [5, 6, 62].

Second, the patterns can also be used with regard to organization-specific modeling conventions. In general, modeling conventions aim to maintain a consistent modeling practice within an organization in terms of used modeling elements and naming of these elements [63, 64, 39, 20]. In this setting, the patterns may extend existing conventions with workflow constructs to express complex scenarios. Thus, the patterns provide counterexamples and frequent errors that should be avoided when creating process models.

### 5.3. Threats to Validity

The results of this paper have to be discussed in the light of different threats to validity. In particular, we discuss those threats that impact the generalizability of the techniques (external validity), the completeness of the non-canonicity patterns, and the validity of the techniques themselves (construct validity).

Regarding the external validity, we identify the *representativeness* of our test data and the language adaption as a threat to validity. The four process model collections we used can hardly be seen as representative in a statistical sense. Therefore, we cannot completely rule out that our technique would yield different results for other model collections. However, we tried to minimize this risk by selecting process model collections that vary along different dimensions such as degree of standardization, domain, and size. Hence, we are confident that the successful application of our techniques is not limited to a particular set of models and that it will also uncover and rework a large share of non-canonical process model activities in other collections.

A second aspect to note is that the presented techniques have been applied to process models with *English* labels only. This raises the question of adaptability to different target languages. In general, the recognition technique can be easily adapted to different target languages by adjusting key phrases of the respective patterns. Moreover, we may use localized parsers for process models [9] or natural language text [65, 66, 67]. For the correction technique, we require further adaptions with regard to parsing and the refactoring of activities. The parsing of activities is facilitated by using insights on labeling styles of the target language [68]. In contrast to that, the refactoring technique does not require further adaptions since it builds only the model fragment and uses the labels from the parsing component. Hence, we do not consider the the adaption to different languages as a threat to external validity because the adaptions are associated with manageable efforts.

The *completeness of non-canonicity patterns* might be affected because we only used four process model collections to derive the patterns. Thus, the inclusion of other process model collections might expand the set of the non-canonicity patterns. As mentioned before, we tried to minimize this risk by selecting process model collections that vary along different characteristics. Moreover, we also investigated an independent repository, i.e. the TelCo collection, for additional patterns. Since no new non-canonicity pattern emerged during its investigation, we are confident to have reached a point of saturation with regard to the completeness of the patterns.

The *validity of the techniques* (construct validity) might be affected if activities vary or combine the specified patterns. Consequently, deficient activity labels are not recognized or cannot be refactored by our techniques, which raises the question if the designed techniques fulfill their intended purpose. To that end, we used the evaluation metrics precision and recall. By definition, these metrics distinguish between the instances that are relevant and retrieved and those that are either irrelevant or not retrieved. Consequently, the greater the value of these metrics, the more accurate the respective techniques work. Since our experiments showed considerably high values of precision and recall, we are confident that the techniques do what they are supposed to and rework only the problematic activities in process models.

## 6. Conclusion

In this paper, we introduced the notion of canonicity in order to prevent the mix natural of natural language and modeling language within one process model activity. To this end, we formalized the notion of canonicity and reoccurring patterns that violate canonicity. Based on these formalizations, we designed techniques for the automatic recognition and refactoring of these patterns. As shown in the evaluation experiments, the proposed techniques are capable of recognizing and correcting the discussed patterns that can be found in several process model collections from practice. Thus, we can provide a reliable baseline for further analysis of process models by separating modeling language and natural language from each other.

In future research, we first aim to conduct several user studies on the cognitive effects of process models. Since process models tend to become more complex after correcting deficient activities, model understanding and problem solving capabilities might be affected to some extend. Thus, we want to conduct a user experiment and assess the process model understanding capabilities with and without the application of

our approach. Second, we want to extend our techniques to improve the processing of abbreviations and compound words. Our study has revealed that specific abbreviations and terms incorporate additional resources, which are erroneously processed by the techniques. Thus, addressing these syntactical phenomena should improve the results of the presented techniques.


[1] E. Cardoso, J. Almeida, G. Guizzardi, Requirements engineering based on business process models: A case study, in: Enterprise Distributed Object Computing Conference Workshops, 2009. EDOCW 2009. 13th, 2009, pp. 320–327.

[2] H. Leopold, J. Mendling, O. Gunther, What we can learn from quality issues of bpmn models from industry, IEEE Software.

[3] M. Rosemann, Potential Pitfalls of Process Modeling: Part A, Business Process Management Journal 12 (2) (2006) 249–254.

[4] R. M. Dijkman, M. La Rosa, H. A. Reijers, Managing large collections of business process models-current techniques and challenges, Computers in Industry 63 (2) (2012) 91–97.

[5] B. W. Boehm, Understanding and controlling software costs, Journal of Parametrics 8 (1) (1988) 32–68.

[6] A. Abran, P. Bourque, R. Dupuis, J. W. Moore, L. L. Tripp, Guide to the Software Engineering Body of Knowledge - SWEBOK, 2004th Edition, IEEE Press, 2004.

[7] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, K. Wolf, Analysis on Demand: Instantaneous Soundness Checking of Industrial Business Process Models, Data & Knowledge Engineering 70 (5) (2011) 448–466.

[8] M. Weidlich, J. Mendling, M. Weske, Efficient consistency measurement based on behavioral profiles of process models, IEEE Trans. Software Eng. 37 (3) (2011) 410–429.

[9] H. Leopold, R.-H. Eid-Sabbagh, J. Mendling, L. G. Azevedo, F. A. Baião, Detection of naming convention violations in process models for different languages, Decision Support Systems 56 (2013) 310–325.

[10] C. Ouyang, M. Dumas, S. Breutel, A. ter Hofstede, Translating standard process models to bpel, in: Advanced Information Systems Engineering, Springer, 2006, pp. 417–432.

[11] C. Ouyang, M. Dumas, A. H. M. ter Hofstede, W. M. P. van der Aalst, Pattern-based translation of bpmn process models to bpel web services, International Journal of Web Services Research (IJWSR) 5 (1) (2008) 42–62.

[12] W. M. P. van der Aalst, Verification of workflow nets, Application and Theory of Petri Nets (1997) 407–426.

[13] M. Weske, Business Process Management: Concepts, Languages, Architectures, 2nd Edition, Springer, 2012.

[14] J. Mendling, M. Nüttgens, EPC syntax validation with XML schema languages, in: M. Nüttgens, F. J. Rump (Eds.), EPK 2003 - Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, Proceedings des GI-Workshops und Arbeitskreistreffens (Bamberg, Oktober 2003), GI-Arbeitskreis Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, 2003, pp. 19–30.

[15] J. Dehnert, P. Rittgen, Relaxed soundness of business processes, in: Advanced Information Systems Engineering, Springer, 2001, pp. 157–170.

[16] F. Puhlmann, Soundness verification of business processes specified in the pi-calculus, in: R. Meersman, Z. Tari (Eds.), On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, Vol. 4803 of Lecture Notes in Computer Science, Springer, 2007, pp. 6–23. doi:10.1007/978-3-540-76848-7_3.

[17] N. Sidorova, C. Stahl, N. Trcka, Soundness Verification for Conceptual Workflow Nets with Data: Early Detection of Errors with the Most Precision Possible, Information Systems 36 (7) (2011) 1026–1043.

[18] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, M. T. Wynn, Soundness of workflow nets: classification, decidability, and analysis, Formal Asp. Comput. 23 (3) (2011) 333–363. doi:10.1007/s00165-010-0161-4.
URL http://dx.doi.org/10.1007/s00165-010-0161-4

[19] A. Basu, R. W. Blanning, A formal approach to workflow analysis, Information Systems Research 11 (1) (2000) 17–36.

[20] J. Becker, P. Delfmann, S. Herwig, L. Lis, A. Stein, Formalizing Linguistic Conventions for Conceptual Models, in: Conceptual Modeling - ER 2009, LNCS, Springer Berlin Heidelberg, 2009, pp. 70–83.

[21] J. Becker, P. Delfmann, S. Herwig, L. Lis, A. Stein, Towards Increased Comparability of Conceptual Models - Enforcing Naming Conventions through Domain Thesauri and Linguistic Grammars, in: ECIS 2009, 2009, pp. 1–1.

[22] P. Delfmann, S. Herwig, L. Lis, A. Stein, Supporting distributed conceptual modelling through naming conventions-a tool-based linguistic approach., Enterprise Modelling and Information Systems Architectures 4 (2) (2009) 3–19.

[23] B. van der Vos, J. A. Gulla, R. van de Riet, Verification of conceptual models based on linguistic knowledge, Data & Knowledge Engineering 21 (2) (1997) 147 – 163.

[24] F. Pittke, H. Leopold, J. Mendling, Spotting terminology deficiencies in process model repositories, in: S. Nurcan, H. A. Proper, P. Soffer, J. Krogstie, R. Schmidt, T. A. Halpin, I. Bider (Eds.), Enterprise, Business-Process and Information Systems Modeling (BPMDS 2013), Vol. 147 of Lecture Notes in Business Information Processing, Springer, 2013, pp. 292–307.

[25] F. Pittke, H. Leopold, J. Mendling, Automatic detection and resolution of lexical ambiguity in process models, Software Engineering, IEEE Transactions on PP (99) (2015) 1–1. doi:10.1109/TSE.2015.2396895.

[26] A. Koschmider, E. Blanchard, User assistance for business process model decomposition, in: Proceedings of the 1st IEEE International Conference on Research Challenges in Information Science, 2007, pp. 445–454.

[27] V. Gruhn, R. Laue, Detecting common errors in event-driven process chains by label analysis., Enterprise Modelling and Inf. Sys. Architectures 6 (1) (2011) 3–15.

[28] I. Weber, J. Hoffmann, J. Mendling, Beyond Soundness: on the Verification of Semantic Business Process Models, Distributed and Parallel Databases 27 (3) (2010) 271–343.

[29] C. Di Francescomarino, P. Tonella, Supporting ontology-based semantic annotation of business processes with automated suggestions, in: Enterprise, Business-Process and Information Systems Modeling, Springer, 2009, pp. 211–223.

[30] R. M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in bpmn, Information and

Software Technology 50 (12) (2008) 1281–1294.
[31] N. Lohmann, P. Massuthe, C. Stahl, D. Weinberg, Analyzing interacting WS-BPEL processes using flexible model generation, Data Knowl. Eng. 64 (1) (2008) 38–54. doi:10.1016/j.datak.2007.06.006.
URL http://dx.doi.org/10.1016/j.datak.2007.06.006
[32] R. Eshuis, R. Wieringa, Tool support for verifying UML activity diagrams, IEEE Trans. Software Eng. 30 (7) (2004) 437–447. doi:10.1109/TSE.2004.33.
URL http://dx.doi.org/10.1109/TSE.2004.33
[33] W. M. P. van der Aalst, Formalization and verification of event-driven process chains, Information and Software Technology 41 (10) (1999) 639 – 650.
[34] J. Mendling, B. F. van Dongen, W. M. P. van der Aalst, Getting rid of or-joins and multiple start events in business process models, Enterprise IS 2 (4) (2008) 403–419. doi:10.1080/17517570802245433.
URL http://dx.doi.org/10.1080/17517570802245433
[35] M. T. Wynn, H. M. W. E. Verbeek, W. M. P. van der Aalst, A. H. M. ter Hofstede, D. Edmond, Reduction rules for YAWL workflows with cancellation regions and or-joins, Information & Software Technology 51 (6) (2009) 1010–1020. doi:10.1016/j.infsof.2008.12.002.
URL http://dx.doi.org/10.1016/j.infsof.2008.12.002
[36] H. Verbeek, T. Basten, W. M. P. van der Aalst, Diagnosing workow processes using woflan, The Computer Journal 44 (4) (2001) 246–279.
[37] K. Wolf, Generating petri net state spaces, in: Petri Nets and Other Models of Concurrency–ICATPN 2007, Springer, 2007, pp. 29–42.
[38] J. Mendling, H. A. Reijers, J. Recker, Activity Labeling in Process Modeling: Empirical Insights and Recommendations, Information Systems 35 (4) (2010) 467–482.
[39] J. Mendling, H. A. Reijers, W. M. P. van der Aalst, Seven Process Modeling Guidelines (7PMG), Information and Software Technology 52 (2) (2010) 127–136.
[40] B. Silver, BPMN Method and Style, with BPMN Implementer's Guide, 2nd Edition, Cody-Cassidy Press, 2011.
[41] F. Pittke, H. Leopold, J. Mendling, When language meets language: Anti patterns resulting from mixing natural and modeling language, in: F. Fournier, J. Mendling (Eds.), BPM Workshops and Doctoral Consortium 2014, LNBIP, Springer, 2014, pp. 1–12.
[42] W. M. P. van der Aalst, A. H. M. ter Hofstede, Workflow patterns: On the expressive power of (petri-net-based) workflow languages, in: Proceedings of the 4th Workshop on the Practical Use of Coloured Petri Nets and CPN Tools, 2002, pp. 1–20.
[43] D. R. Christiansen, M. Carbone, T. Hildebrandt, Formal semantics and implementation of bpmn 2.0 inclusive gateways, in: Web Services and Formal Methods, Springer, 2011, pp. 146–160.
[44] R. M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in bpmn, Information and Software Technology 50 (12) (2008) 1281–1294.
[45] N. Russell, A. H. Ter Hofstede, D. Edmond, W. M. van der Aalst, Workflow data patterns: Identification, representation and tool support, in: Conceptual Modeling–ER 2005, Springer, 2005, pp. 353–368.
[46] N. Russell, W. M. van der Aalst, A. H. ter Hofstede, D. Edmond, Workflow resource patterns: Identification, representation and tool support, in: Advanced Information Systems Engineering, Springer, 2005, pp. 216–232.
[47] A. Lanz, B. Weber, M. Reichert, Workflow time patterns for process-aware information systems, in: Enterprise, Business-Process and Information Systems Modeling, Springer, 2010, pp. 94–107.
[48] N. Russell, W. van der Aalst, A. ter Hofstede, Workflow exception patterns, in: Advanced Information Systems Engineering, Springer, 2006, pp. 288–302.
[49] H. Leopold, S. Smirnov, J. Mendling, On the refactoring of activity labels in business process models, Information Systems 37 (5) (2012) 443–459.
[50] R. A. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, ACM Press / Addison-Wesley, 1999.
[51] G. Keller, T. Teufel, SAP(R) R/3 Process Oriented Implementation: Iterative Process Prototyping, Addison-Wesley, 1998.
[52] P. Trkman, The critical success factors of business process management, International Journal of Information Management 30 (2) (2010) 125–134.
[53] J. Mendling, Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness, Vol. 6 of LNBIP, Springer, 2008.
[54] W. M. P. van der Aalst, A. Kumar, Xml–based schema definition for support of interorganizational workflow, Information Systems Research 14 (1) (2003) 23–46.
[55] S. X. Sun, J. L. Zhao, J. F. Nunamaker, O. R. L. Sheng, Formulating the data-flow perspective for business process management, Information Systems Research 17 (4) (2006) 374–391.
[56] A. Basu, A. Kumar, Research commentary: Workflow management issues in e-business, Information Systems Research 13 (1) (2002) 1–14.
[57] J. Krogstie, O. I. Lindland, G. Sindre, Defining quality aspects for conceptual models, in: Proceedings of the international working conference on Information system concepts: Towards a consolidation of views, Chapman & Hall, Ltd., 1995, pp. 216–231.
[58] J. Krogstie, G. Sindre, H. Jorgensen, Process models representing knowledge for action: a revised quality framework, European Journal of Information Systems 15 (1) (2006) 91–102.
[59] H. Leopold, J. Mendling, A. Polyvyanyy, Supporting process model validation through natural language generation, Software Engineering, IEEE Transactions on 40 (8) (2014) 818–840. doi:10.1109/TSE.2014.2327044.
[60] J. Fabra, V. De Castro, P. Álvarez, E. Marcos, Automatic execution of business process models: Exploiting the benefits of

model-driven engineering approaches, Journal of Systems and Software 85 (3) (2012) 607–625.
[61] H. Leopold, M. Niepert, M. Weidlich, J. Mendling, R. M. Dijkman, H. Stuckenschmidt, Probabilistic optimization of semantic process model matching, in: BPM'12, 2012, pp. 319–334.
[62] V. Ambriola, V. Gervasi, On the systematic analysis of natural language requirements with circe, Automated Software Engineering 13 (1) (2006) 107–167.
[63] R. Schuette, T. Rotthowe, The Guidelines of Modeling - An Approach to Enhance the Quality in Information Models, in: Proceedings of the 17th International Conference on Conceptual Modeling, Springer-Verlag, London, UK, 1998, pp. 240–254.
[64] J. Becker, M. Rosemann, C. von Uthmann, Guidelines of business process modeling, in: Business Process Management, Springer, 2000, pp. 30–49.
[65] D. Klein, C. D. Manning, Fast Exact Inference with a Factored Model for Natural Language Parsing, in: NIPS 2003, Vol. 15, MIT Press, 2003.
[66] D. Klein, C. D. Manning, Accurate Unlexicalized Parsing, 41st Meeting of the Association for Computational Linguistics (2003) 423–430.
[67] R. Socher, J. Bauer, C. D. Manning, A. Y. Ng, Parsing with compositional vector grammars, in: In Proceedings of the ACL conference, Citeseer, 2013.
[68] H. Leopold, Natural Language in Business Process Models - Theoretical Foundations, Techniques, and Applications, Vol. 168 of Lecture Notes in Business Information Processing, Springer, 2013.