Specification and Detection of Business Process Antipatterns

Francis Palma[1,2], Naouel Moha[1], and Yann-Gaël Guéhéneuc[2]

[1] Département d'informatique, Université du Québec à Montréal, Canada
moha.naouel@uqam.ca
[2] Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
{francis.palma,yann-gael.gueheneuc}@polymtl.ca

Summary. Structured business processes (SBPs) are now in enterprises the prominent solution to software development problems through orchestrating Web services. By their very nature, SBPs evolve through adding new or modifying existing functionalities. Those changes may deteriorate the process design and introduce *process antipatterns*—poor but recurring solutions that may degrade processes design quality and hinder their maintenance and evolution. However, to date, few solutions exist to detect such antipatterns to facilitate the maintenance and evolution and improve the quality of process design. We propose SODA-BP (Service Oriented Detection for Antipatterns in Business Processes), supported by a framework for specifying and detecting *process antipatterns*. To validate SODA-BP, we specify eight antipatterns and perform their detection on a set of randomly selected 35 SBPs form a corpus of more than 150 collected processes from an open-source search engine. Some of the SBPs were modified by adding, removing, or modifying process elements to introduce noise in them. Results shows that SODA-BP has an average detection precision of more than 75% and recall of 100%.

Key words: Business process, Antipatterns, Specification, Detection

1 Introduction

BPMN (Business Process Model and Notation) [1] is broadly used by business analysts for modeling workflows using a graphical notation. BPEL4WS (Business Process Execution Language for Web Services) [2] provides an executable form for graphical process models and is now the *de-facto* standard to implement structured business processes (SBPs) on top of Web services technology.

Like any other software artefacts, SBPs may evolve, *i.e.*, changes may take place (1) by modifying the existing tasks and–or adding new tasks or elements (2) by modifying the flow in the processes. This evolution of SBPs may deteriorate their designs over time and introduce poor but recurring solutions to process design problems—*process antipatterns*. Process antipatterns describe common design problems in SBPs that may hinder their maintenance and evolution and result in poor quality of design (QoD) [3].

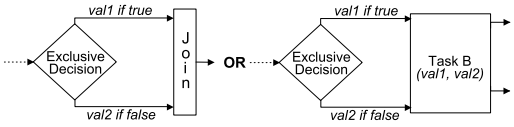

Fig. 1: Deadlock Through Decision-Join Pair

The *Deadlock Through Decision-Join Pair* [3] as shown in Figure 1 using the IBM WebSphere Business Modeling notation[1] is a common process antipattern where a *decision* node may appear before a *join* gateway. This structure leads to a deadlock: the *decision* always triggers a single output while the *join* waits for inputs on all of its branches. Alternatively, as a variant of this antipattern, if a *task* produces two alternative outputs, *i.e.*, behaves like an *exclusive decision*, and the following *task* requires both the outputs as its input, then there is also a *deadlock*. The presence of such antipatterns in SBPs degrades the QoD and may hinder their maintenance and evolution. Therefore, for SBPs, the automatic detection of such antipatterns is an important activity by assessing their design (1) to ease their maintenance and evolution and (2) to improve their QoD.

In the literature, a number of process antipatterns in graphical process models [3–6] have been defined and several approaches have been proposed to analyse and detect those antipatterns [7–13]. To date, however, the detection of antipatterns in structured processes still did not receive much attention. The approaches dedicated to graphical process models from the literature cannot be exploited for SBPs due to several conceptual differences [**?**]. The transition from graph-oriented BPMN to block-structured BPEL4WS is not isomorphic, prone to semantic ambiguities, differ in representing some *constructs*, and are implemented with two different classes of language.

We introduced an approach in our previous work [**?**] for detecting process antipatterns in SBPs. We presented seven antipatterns using `if-then` *inference rules* and performed the detection of two antipatterns on three example processes. In this paper, we provide a complementary approach called SODA-BP (Service Oriented Detection for Antipatterns in Business Processes) supported by an underlying framework, SOFA (Service Oriented Framework for Antipatterns) to specify and detect process antipatterns. We also define a specification language after a thorough domain analysis of process antipatterns from the literature [3–6]. SODA-BP relies on this specification language to specify process antipatterns in terms of metrics, process elements, and–or constructs.

To validate our approach, first, we specify process antipatterns using our defined domain specific language. Then, we implement their detection algorithms following the specified rules, and, finally, we apply those algorithms on several SBPs, which, in turn, return identified antipatterns. Our detection results show the effectiveness of SODA-BP: it can detect eight process antipatterns with an average precision of more than 75% and with a recall of 100% in 35 processes.

[1] http://www-03.ibm.com/software/products/en/modeler-advanced

Thus, compared to our previous work [**?**], the main contributions of this paper are: (1) the definition of a rule-based domain specific language to specify process antipatterns, (2) the definition of ten new business process-specific metrics, (3) the specification of eight process antipatterns from the literature [3–6] using the defined language, (4) the extension of the SOFA framework by adding ten process-specific metrics from its early version [**?**] to allow the detection of process antipatterns, and, finally, (5) the validation of SODA-BP for eight process antipatterns on a set of 35 BPs randomly-selected form a corpus of more than 150 collected processes from an open-source search engine.

The rest of the paper is organised as follows. Section 2 discuses the motivation. Section 3 briefly describes the contributions from the literature on the specification and detection of process antipatterns. Section 4 presents our approach SODA-BP, while Section 5 presents its validation along with detailed discussions. Finally, Section 6 concludes the paper and sketches future work.

2 Motivation

A typical business-driven development starts with modeling process tasks and later aims at their technical implementations. Thus, models are the central artefact in any development project and are used to map business requirements and information technology (IT).

Modeling phase is the primary step where business analysts visually model business processes, *i.e.*, using BPMN notations [1]. At this step, the models describe the workflow of the processes but do not contain all the information to execute them. In the next step, the execution logic or code to orchestrate predefined services is derived from those design models, *e.g.*, using BPEL4WS [2], which is the standard executable language for interactions among Web services. Antipatterns in a business process can be introduced in the two steps discussed before: (1) due to the Business-IT gap, *i.e.*, when desired user requirements may not properly be achieved by the services during service development step; and (2) due to the IT-IT gap, *i.e.*, during the process implementation step where some technical limitations may hinder the appropriate translation of business models into executable and logically composed services chain in the form of tasks.

From Graph-based Business Models to Executable Processes: Business analysts mostly rely on BPMN-models to transform user requirements into workflow models trying to reflect users' business goals as well as to guide in-house developers for the technical parts. The BPMN specification [1] provides an informal and partial mapping of such graph-based models to executable concrete processes. Therefore, developers must take caution while writing the structured SBPs because there exist some significant conceptual differences between graphical process models and executable processes. Here, among many, we list some of conceptual differences that may lead to the introduction of antipatterns: (1) The transition from workflow models to executable processes is not isomorphic and prone to semantic ambiguities, which may cause the loss of design considerations; (2) Workflow models and executable processes originate from different sources

(*i.e.*, users and business analysts *vs.* technical analysts) and are employed at different stages of the BP management life cycle; (3) Workflow models and executable processes differ in representing some significant *constructs* and *control flows*. For example, block-structured executable processes and graph-oriented models differ in representing *loops*, *splits*, *joins*, *conditions*, and *goto*; and (4) The transformation between artefacts, *i.e.*, from workflow models to executable processes, is mostly performed manually by the architects and developers, which creates higher risk of introducing design anomalies.

Therefore, process antipatterns exist and will likely be introduced and must be detected as early as possible.

3 Related Work

The literature has already a rich catalog of antipatterns defined by the process modeling community [3–6], with some analysis and detection approaches [7–13], most of which deal with graphical models.

For example, Onoda *et al.* [4] described a set of five deadlock antipatterns. Later, Maruta *et al.* [9] proposed detection algorithms for those antipatterns. With a focus on the quality of process modeling, Persson *et al.* [5] and Stirna *et al.* [6] discussed 13 antipatterns with their possible causes and impacts and presented them as common mistakes that the modelers should avoid. Koehler and Vanhatalo [3] also reported 14 antipatterns in IBM WebSphere process models, while Laue and Awad [12] proposed the first visualisation approach of process antipatterns after detecting them in graphical models and argued that visualising the antipatterns can ease their understanding and correction.

Gruhn and Laue [10] employed a heuristic-based approach to detect modeling antipatterns. By translating different elements of graphical models into Prolog facts and rules, the authors detected modeling errors that may hinder the soundness and correctness of the models (*e.g.*, deadlocks). Instead of using Prolog, Trčka *et al.* [11] described eight process antipatterns using temporal logic. Finally, relying on Petri nets, Awad *et al.* [8] performed the detection and correction of data-flow anomalies in graphical process models. Ouyang *et al.* [14], using the Petri nets, mainly focused on the reachability and message-consuming activity analysis.

Based on these previous works, the gaps in the literature can be summarised as follows: (1) the approaches to detect antipatterns were studied mostly for graphical models, while the structured business processes (SBPs) were not considered, *i.e.*, a concrete approach for specifying and detecting process antipatterns in SBPs is lacking and (2) diverse runtime quality aspects, *e.g.*, *availability* or *response time* of involved Web services were not considered, those can be computed by concretely executing the SBPs. We plan to perform such dynamic analysis of SBPs as one of our future works.

Considering the conceptual differences between graph-oriented BPMN and block-structured BPEL4WS representations discussed in [**?**], the detection approaches discussed above dedicated to graphical process models are not applicable to the structured processes.

4 The SODA-BP Approach

We consequently developed the SODA-BP approach (Service Oriented Detection for Antipatterns in Business Processes) dedicated to structured business processes (SBPs). SODA-BP involves three steps:
Step 1: Specification of Process Antipatterns – In this step, we identify relevant properties of process antipatterns that we use to define a domain-specific language (DSL). We use this DSL to specify process antipatterns based on rules.
Step 2: Generation of Detection Algorithms – This step involves the generation of detection algorithms from the specifications in the former step. In this paper, we performed this step by implementing concretely the algorithms in conformance with the rules specified in Step 1. We plan to automate this step in the future.
Step 3: Detection of Process Antipatterns – In the last step, we apply the implemented detection algorithms from Step 2 on SBPs to detect and report process antipatterns.

The next sections present the first two steps in details. The last step is discussed in Section 5, where we report the validation of SODA-BP.

4.1 Specification of Process Antipatterns

To specify process antipatterns, we carried out a thorough domain analysis of antipatterns for SBPs by investigating their definitions and descriptions from the literature, namely [3–6, 11, 15]. After the domain analysis, we identified all the quantifiable properties related to each antipattern, which include all static properties related to process design, *e.g.*, presence of *fork*, *merge*, *gateways*, *inputs*, and *outputs*, etc. In general, we can easily identify those static properties from the abstract processes. Those properties play a key role and are sufficient in defining a DSL, which allows engineers to specify antipatterns in the form of rules, using their own experience and expertise.

The DSL provides the engineers with a high-level domain-related abstractions to express various properties of process antipatterns. Indeed, a DSL gives more flexibility than implementing the *ad-hoc* detection algorithms by focusing on *what* to detect and not *how* [16]. Other rule-based declarative languages exist, like the Object Constraint Language (OCL) [17] that describes rules to apply on Unified Modeling Language (UML) models However, these languages do not suit our purpose because we specify process antipatterns with a higher level of abstraction with discrete *domain* expressiveness.

We define the syntax of our DSL using a Backus-Naur Form (BNF) grammar. We apply a rule-based technique for specifying process antipatterns, *i.e.*, each rule card combines a set of rules. Figure 2 presents the grammar of our DSL. A *rule_card* denoted with `RULE_CARD` includes a name and a rule body (see Figure 2, line 1). The content of the rule card is identified as *content_rule* (line 3), and may enclose a *metric*, a *process_fragment*, or even a reference to another `RULE_CARD` (lines 3 to 4). A *process_ fragment* is the smallest part of a process model with the arrangement of at least two process elements that modelers place while modeling the processes [18]. A *process_fragment* (see lines 5 to 7) can be a binary

```
1  rule_card          ::= RULE_CARD rule_cardName {(rule)+};
2  rule               ::= RULE ruleName {content_rule};
3  content_rule       ::= metric | process_fragment
4                        | RULE_CARD rule_cardName

5  process_fragment   ::= binary_rel | binary_rel relType element
6                        | element relType binary_rel | ruleType operator binary_rel
7                        | element relType binary_rel relType element
8  ruleType           ::= ruleName | rule_cardName

9  metric             ::= id_metric comparator num_value
10 id_metric          ::= NICF | NIDF | NII | NIO | NOF | NOM | NUI | NUO | NIU | NIP
11 comparator         ::= < | ≤ | = | ≠ | ≥ | >

12 binary_rel         ::= element relType element | element operator element
13                        | ruleType operator ruleType | element operator ruleType
14                        | ruleType operator element

15 operator           ::= AND | OR | NOT
16 relType            ::= S:PRECEDE | W:PRECEDE | BACKCONN
17 element            ::= PROCESS | FORK | MERGE | JOIN | TASK | S_NODE | X_DECISION | I_DECISION

18 rule_cardName, ruleName ∈ string
19 num_value ∈ double
```

Fig. 2: BNF grammar of the DSL of SODA-BP (we show only BP-specific metrics).

relation (*binary_rel*), *i.e.*, a simple relation between two elements (line 12) or can describe more complex relations by combining different *binary_rel* with other elements through a relation type, (*i.e.*, *relType*). A *binary_rel* may also connect two rules or elements using an *operator* (line 13). The *operator* set includes common logical operators like `AND`, `OR`, `NOT` (line 15). The *metric* can contain an *id_metric* compared with a numeric value (line 9). The *comparator* includes the common mathematical comparators (line 11). The DSL has three typical relation types (*relType*), *e.g.*, a strong or immediate precedence (`S:PRECEDE`), a weak precedence (`W:PRECEDE`), and a back connection (`BACKCONN`) (line 16). In `W:PRECEDE`, an *element* may not appear right after another *element*. In contrast, for a `S:PRECEDE`, the precedence is with the immediate *element*. The *element* set includes most common modeling elements from a task (`TASK`) to different types of decision gateways, such as `X_DECISION` or `I_DECISION` (line 17). Other elements are `FORK`, `MERGE`, `JOIN`, the stop node (`S_NODE`), etc.

Our metric suite (line 10) currently includes 23 static and dynamic metrics. The ten newly defined process-specific static metrics are: number of identical control-flows (`NICF`), number of identical data-flows (`NIDF`), number of identical inputs (`NII`), number of identical outputs (`NIO`), number of forks (`NOF`), number of merges (`NOM`), number of unused inputs (`NUI`), number of unused outputs (`NUO`), number of inputs undeclared (`NIU`), and number of inputs unproduced (`NIP`). New metrics can be added in the DSL to specify new antipatterns.

Figure 3 shows the rule card for *Deadlock Through Decision-Join Pair* [3] antipattern, introduced in Section 1. When an *exclusive decision* appears before a *join* gateway (or a *task*), as shown in the rule card: `X_DECISION S:PRECEDE (JOIN OR TASK)`, *i.e.*, an *exclusive decision* immediately precedes a *join* (or a *task*). Then, this structure in the process may lead to a deadlock because the *exclusive decision* always triggers a single output, whereas the immediate *join* or

```
RULE_CARD Deadlock {
  RULE Deadlock {X_DECISION S:PRECEDE (JOIN OR
  TASK)};
};
```

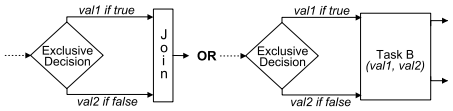

Fig. 3: The *Deadlock Through Decision-Join Pair* process antipattern.

task in the process requires input on all of its branches. We specify seven other process antipatterns as shown in Figure 5.

4.2 Generation of Detection Algorithms

The second step involves the implementation of the detection algorithms from the rule cards specified for each process antipattern. For each process antipattern, we implement all the related metrics following its specification and implement the detection algorithm in Java, which can directly be applied on any BP. However, in the future, we want to automate this algorithm generation process following a similar technique presented in [**?**].

4.3 Detection of Process Antipatterns

To ease the detection step, we preprocess the SBPs and generate process structure trees (PSTs). Such parsing of processes also helps their comprehension and allows finding reusable sub-processes and applying rules on process trees [19]. For our purpose, we automatically transform the SBPs into more abstract and simplified PSTs after eliminating inessential process elements and attributes.

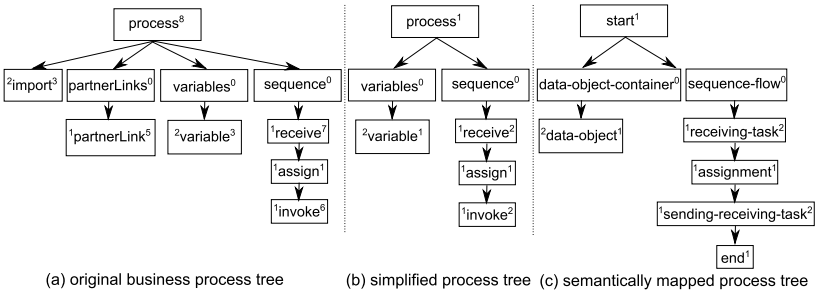

Fig. 4: An example process structure tee (PST) of a business process (each node is preceded by a integer representing the number of instances of that node and succeeded by another representing number of attributes of a node).

SBPs are complex entities and their sizes cause their structural complexity to grow further. However, every implementation details are not required for our analysis since we are interested in the static analysis of SBPs. In fact, we do not lose any essential process information specific to our rules-based approach,

i.e., all the tasks, input/output data, control-flow information, and so on (see Figure 4(b)). Subsequently, we automatically generate a final process tree that is semantically equivalent to the previous tree and are mappable to our language (see Figure 4(c)). Engineers can use the latter version of the process tree for further analysis, *e.g.*, for implementing rules to apply on business processes.

Underlying Framework: The SOFA (Service Oriented Framework for Antipatterns) framework was originally presented elsewhere [**?**] supported detecting antipatterns in service-based systems. We further extend the framework to support the detection of process antipatterns. SOFA itself is developed using the SCA standards [20] and is composed of several components with distinct functional supports. The SOFA framework includes several components: (1) the *Rule Specification* component for specifying rules, (2) the *Algorithm Generation* component for automatically generating detection algorithms based on specified rules, and (3) the *Detection* component for applying generated detection algorithms on the SBPs.

We added a new `Process Handler` component to SOFA to allow the detection of process antipatterns. The different functionalities performed by the `Process Handler` component are: (1) it parses a given process and filter unnecessary information for generating a PST, (2) it then maps the abstract process model to our rule-based language, and (3) uses the *Detection* component to apply the detection algorithms on the process trees, which reports the detected process antipatterns.

We extended the SOFA framework from its early version by adding ten new business process-specific metrics as described in Section 4.1. Combining those new metrics and different *process elements*, we specify eight business process-specific antipatterns. We list them in Table 1 and show their specifications in Figure 5. The addition of an antipattern requires the implementation of each of its metric if it is not already available following its specification. A metric can be reused for other antipatterns if they share that metric in their specifications.

5 Validation

Through our experiment, we aim to show (1) the generality and extensibility of our DSL and SOFA framework and (2) the accuracy and performance of the detection algorithms in terms of precision and recall.

Assumptions: We define three assumptions to evaluate in our experiment:
A1. Generality: *The DSL allows the specification of different process antipatterns.* This assumption supports the applicability of the SODA-BP, which relies on metric-based rule cards for specifying process antipatterns.
A2. Accuracy: *Antipattern detection algorithms have at least a precision of 75%, and a recall of 100%.* Assuming that the antipatterns have a negative impact on the design, we target a recall of 100% for antipatterns, which ensures that we do not miss any existing antipatterns. The *precision* concerns the detection

accuracy of our specified rules and the corresponding detection algorithms. We also measure the *specificity* of our specified rules.

A3. Extensibility: *Our DSL and SOFA framework are extensible by adding new metrics and process antipatterns.* Through this assumption, we show that new antipatterns can be specified by adding new or combining existing metrics, *process elements*, and different operators and comparators, and later, those antipatterns can be detected using the extended SOFA framework.

Table 1: Description of the eight process antipatterns.

Cyclic Deadlock through Join-Fork and Join-Decision Pair: A ***backward connection*** exists from an ***exclusive decision*** (or a ***fork***) to a ***join***. The ***join*** waits for inputs on all of its branches. However, one of its ***incoming branches*** can only receive the input after the ***join*** has been executed in the first cycle, because its input is initiated from an ***exclusive decision***, later in the process. This ***cyclic dependency*** between the ***join*** and the ***decision*** (or ***fork***), where the ***join*** must be executed before the ***decision*** (or ***fork***) may cause a ***cyclic deadlock*** [3].
Cyclic Lack of Synchronisation through Merge-Fork Pair: Occurs in the cyclic structures when ***backward connections*** appear in ***branches*** that are ***executed*** in ***parallel***, which are ***not synchronised*** by a ***join*** before the backward connection is added to the process. In such case, each of the backward connections results from the same ***fork*** and ends in a ***merge*** located earlier in the process. This antipattern may result in an infinite iterations of the process [3].
Dangling Inputs and Outputs: The ***inputs*** and ***outputs*** of an ***activity*** or ***gateway*** remain ***unconnected*** or ***unused***. Dangling outputs are ***produced*** by a ***task*** or ***sub-process***, but ***never used*** anywhere in the process. In contrast, dangling inputs might cause ***deadlocks*** if the data input of– a ***gateway*** or an ***activity*** is never provided, which is required by the process [3].
Deadlock Through Decision-Join Pair: The ***decision*** node appears ***before*** a ***join gateway***. This structure leads to a ***deadlock***: the ***decision*** triggers a single output, while the ***join*** waits for inputs on all of its branches, but only one input is supplied. Alternatively, if a task produces two alternative outputs, *i.e.*, behaves like an ***exclusive decision***, and the following task requires both the outputs as its input, then there is also a ***deadlock*** [3].
Lack of Synchronisation through Fork-Merge Pair: The ***fork-merge*** pair appears. The ***fork*** triggers ***output*** on all of its **outgoing branches**, while the ***merge*** always wait for input on only one of its incoming connections. Later in the process, another final ***merge*** may cause ***synchronisation problem*** because the latter ***merge*** requires all the inputs, which are ***not available*** yet [3].
Missing Data: Certain ***data elements*** are ***required*** but were ***not created*** or have been ***deleted***. This may cause ***deadlock*** for a certain activity, or even for the whole process depending on the execution context and the design of the process [11].
Multiple Connections between Activities: The ***redundant control-flow*** and–or ***data-flow*** connections exist between tasks. This antipattern has two variants: (i) ***multiple control-flows*** between ***tasks*** that increase the process structural complexity, and (ii) ***multiple data-flow connections*** of the ***same type*** from a task [3].
Passing Shared Data along Several Branches: Shared inputs or ***outputs*** are ***duplicated*** along several ***branches***. Typically, the duplication of outputs or inputs of a task hinders the reusability of a process fragment. The best practice is to use a ***fork*** for ***distributing*** single ***input*** among the ***branches*** and a ***join*** for ***aggregating*** unique outputs into a single flow [3].

Subjects: We specify eight process antipatterns by applying our SODA-BP approach. Table 1 describes those antipatterns collected from the literature [3–6, 11, 15]. We selected those antipatterns because they were described in the literature as being most common and frequent. In Table 1, we mark the relevant properties related to the specification for each antipattern in bold-italics. We show the specifications and graphical representations of those antipatterns in Figure 5 using the IBM WebSphere notation[1] to ease their understanding.

Objects: Real structured business processes (SBPs) are often not freely available for validation purposes. We used the `ohloh.net` portal as our source of SBPs

because it provides a collection of publicly-available processes. That portal facilitates searching business processes along with their underlying Web services. We performed the experiment on a set of 35 SBPs randomly-selected form a corpus of more than 150 collected processes from `ohloh.net`. The complete set of analysed SBPs is available online at `sofa.uqam.ca/soda-bp/`, where we also detail their modifications.

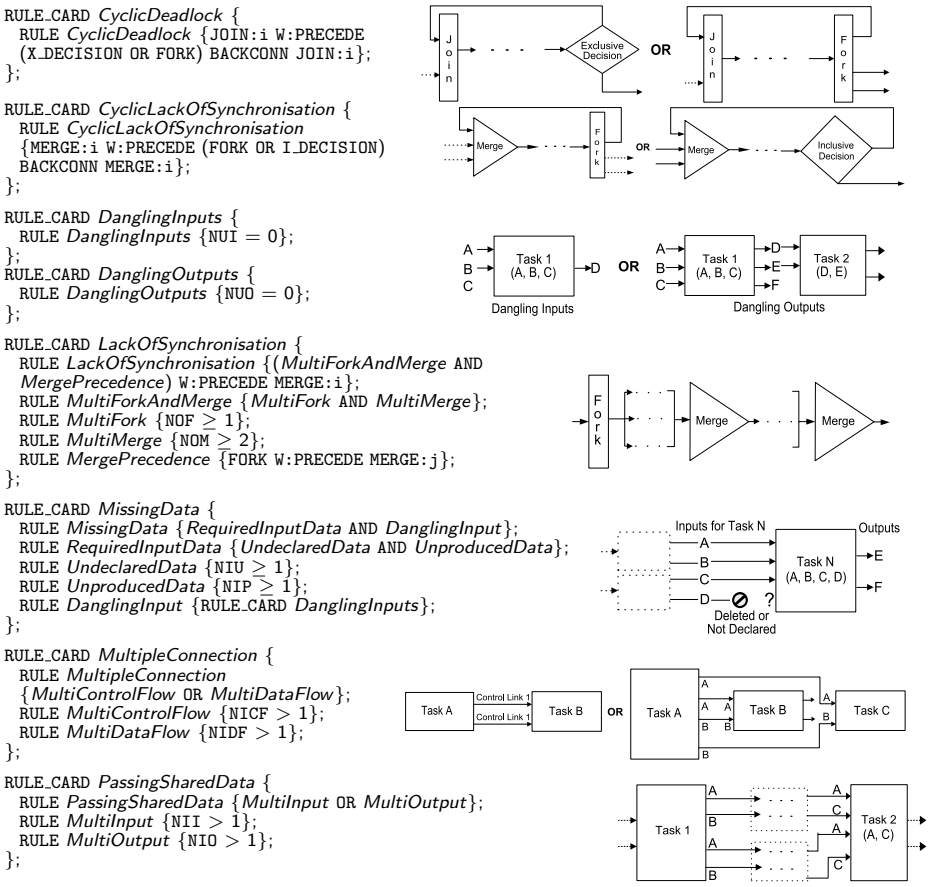

Fig. 5: Rule cards for different process antipatterns.

Process: We specified the rule cards for eight process antipatterns, implemented their detection algorithms, and applied those algorithms on the 35 SBPs to detect any existing antipatterns in two steps. First, we introduced some noise within the selected SBPs by adding, removing, or changing process elements to thoroughly validate the detection. The introduction of noise was performed by two Master students who were not the part of our experiment by adding or removing variables and parallel or sequence of tasks, and so on. However, they

made sure such introduction of noise did not affect the stability of the original processes. Then, we performed detection on the set of 35 SBPs.

We performed the validation of the detection results by analysing the process elements manually (1) to validate that those process elements are true positives and (2) to identify false negatives, *i.e.*, occurrences of antipatterns missed in the SBPs. We used the measures of precision and recall to show our detection accuracy. Precision concerns the ratio between the true detected antipatterns and all detected antipatterns. Recall is the ratio between the true detected antipatterns and all existing true antipatterns. Two students performed a thorough and independent analysis after we provided them with the SBPs and a short description of each process antipattern. The manual validation of the processes was a laborious task that demanded 30 minutes to an hour per process depending on the size of the process, for each antipattern.

5.1 Results

Table 2 shows the detailed detection results of the eight process antipatterns. We found six antipatterns, namely *Lack of Synchronisation*, *Passing Shared Data*, and *Dangling Inputs/Outputs*, and so on, in 16 processes. In Table 2, we report the antipatterns in the first column, followed by the list of processes having these antipatterns in the second. In the third column, we present the metric values and the different process elements involved in the detected antipatterns in column 4. Finally, the last two columns report the precision (P) and recall (R). Detailed detection results are also available online at `sofa.uqam.ca/soda-bp/`.

5.2 Discussions of the Results

Figure 6(a) graphically shows the detection of the *Lack of Synchronisation* antipattern in the original version of the auction process. The auction process involves two *forks* (`NOF`=2) and two *merges* (`NOM`=2), with the first fork receiving values from the users through the `provide` task simultaneously and storing them into two different variables, *i.e.*, `buyerData` and `sellerData`. After the two process threads merge, there is another parallel invocation of `answer` task again with two different parameters, *i.e.*, `sellerAnswerData` and `buyerAnswerData`. A synchronisation problem occurs in the first place where the process may not receive values from the `provide` task at the same time and the *merge* may proceed even with a single response. A good practice to avoid this synchronisation problem is to use a *join* gateway instead, considering the operational difference between the *merge* and *join* gateway.

We detected *Multiple Connections* in the noisy version of loan_approval process (see Figure 6(b) and Table 2) because it contains more than one control-flows (`NICF`>1). In Figure 6(b), the same "receive-to-assess" control-flow link was defined with the duplicated transition conditions for the `request` receiving task, hence, `NICF`=2 for "receive-to-assess". Manual validation confirmed this antipattern detection and thus we had a precision and recall of 100%.

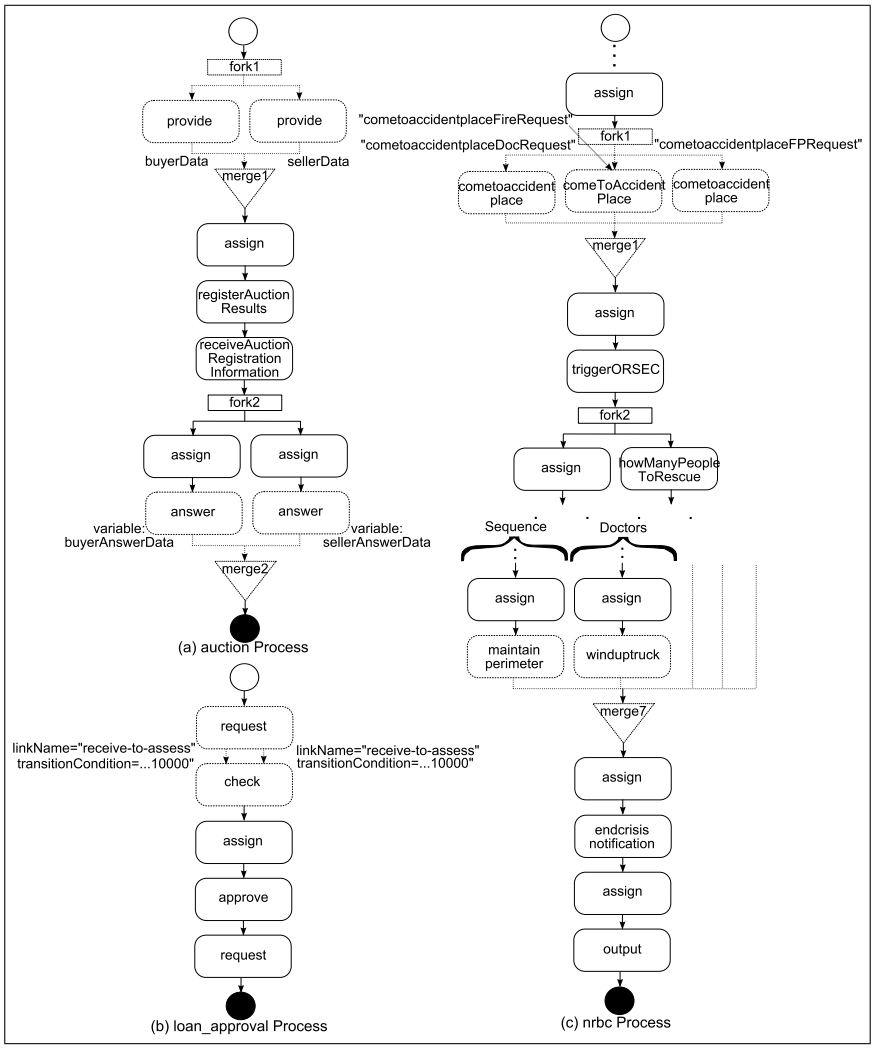

Fig. 6: The detection of *Lack of Synchronisation* in auction process and nrbc process; and the detection of *Multiple Connections* in loan_approval process.

The occurrence of *Lack of Synchronisation* was also detected in the original versions of the BuyBook, LoanFlowPlus, and nrbc process. The nrbc process (see Figure 6(c)) has four *forks* (`NOF`=4) and seven *merges* (`NOM`=7) within its full process scope. A synchronisation problem may occur in the beginning of the process flow with the invocation of `comeToAccidentPlace` task simultaneously with three input variables `cometoaccidentplaceDocRequest`, `cometoaccidentplaceFireRequest`, and `cometoaccidentplaceFPRequest`. After those three threads *merge*, later in the process, another concurrent execution of two

Table 2: Details on the eight antipattern detection results for the 35 processes.

Antipatterns	Processes	Metrics	Elements Involved	P	R
Cyclic Deadlock	*none detected*	*N/A*	*N/A*	–	–
Cyclic Lack Of Synchronisation	*none detected*	*N/A*	*N/A*	–	–
Dangling Inputs	Loan	NUI=0	"creditRatingInput123"	[3/3] 100%	[3/3] 100%
	purchaseOrder	NUI=0	"shippingRequest1"		
	SalesforceFlow	NUI=0	"output1"		
Dangling Outputs	AbsenceRequest	NUO=0	"createTaskResponse1"	[4/4] 100%	[4/4] 100%
	FlightBooking	NUO=0	"Output2"		
	Loan	NUO=0	"creditRatingOutput123"		
	SalesforceFlow	NUO=0	"Output1"		
Deadlock	*none detected*	*N/A*	*N/A*	–	–
Lack of Synchronisation	auction	NOF=2,NOM=2	*MergePrecedence=true*	[8/8] 100%	[8/8] 100%
	BuyBook	NOF=1,NOM=3	*MergePrecedence=true*		
	LoanFlowPlus	NOF=1,NOM=2	*MergePrecedence=true*		
	nrbc	NOF=4,NOM=7	*MergePrecedence=true*		
	Travel	NOF=1,NOM=2	*MergePrecedence=true*		
	AbsenceRequest	NOF=1,NOM=3	*MergePrecedence=true*		
	GovernanceBPEL	NOF=1,NOM=3	*MergePrecedence=true*		
	VacationRequest	NOF=1,NOM=2	*MergePrecedence=true*		
Missing Data	ClaimsApproval	NIU=1,NIP=1,NUI=0	"dummyVar"	[5/5] 100%	[5/5] 100%
	Correlation	NIU=1,NIP=1,NUI=0	"CorrelationProcess-OperationIn"		
	Loan	NIU=1,NIP=1,NUI=0	"creditRatingInput123"		
	purchaseOrder	NIU=1,NIP=1,NUI=0	"shippingRequest1"		
	SalesforceFlow	NIU=1,NIP=1,NUI=0	"output1"		
Multiple Connections	loan_approval	NICF=2	"receive-to-assess"	[2/2] 100%	[2/2] 100%
	purchaseOrder	NICF=2	"ship-to-invoice" "ship-to-scheduling"		
Passing Shared Data	BuyBook	NII=2	"BookRequest"	[3/5] 60%	[3/3] 100%
	LoanFlowPlus	NII=2	"loanApplication"		
	Travel	NII=2	"FlightDetails"		
	DILoanService	NII=2	"output"		
	GovernanceBPEL	NII=2	"applicationDeployerPL1"		
Average				93.3%	100%

control-flows, *i.e.*, `Sequence` and `Doctors` follow a final *merge*. The first *merge*, *i.e.*, merge1, may trigger even without the three invocations of the `comeToAccidentPlace` task having finished. If that happens, the latter concurrent execution of `Sequence` and `Doctors` will not wait and proceed with the consequent execution, which may lead to a lack of synchronisation at the final *merge*, *i.e.*, merge7. The manual analysis confirmed the detection and we thus have a precision and recall of 100% for the *Lack of Synchronisation* antipattern.

The *Dangling Inputs*, *Dangling Outputs*, and *Missing Data* antipatterns were detected in five modified processes. We also detected the *Lack of Synchronisation* antipattern in three other modified SBPs, namely, AbsenceRequest, GovernanceBPEL, and VacationRequest. However, we did not detect any occurrences of three antipatterns within the selected SBPs, namely *Cyclic Deadlock*, *Cyclic Lack of Synchronisation*, and *Deadlock*.

5.3 Discussions of the Assumptions

Following our detection results, we assess all the three assumptions.

A1. Generality: We specified eight process antipatterns (see Table 1) using the rule cards (see Figure 5) from the literature [3–6, 11, 15]. We defined simpler process antipatterns with few rules, such as *Dangling Inputs and Outputs* and *Passing Shared Data* but also more complex antipatterns with rules combining metrics and *process elements*, such as *Lack of Synchronisation*. Also, we specified rule cards in combination with other rule cards, such as the specification of *Missing Data* antipattern, which includes the *Dangling Inputs* antipattern. Similarly, we can specify other antipatterns defined in the literature. Thus, we can support our first assumption regarding the generality of our DSL.
A2. Accuracy: As shown in Table 2, we obtained an average recall of 100% and an average precision of 93.3% with the set of 35 SBPs. We also have an average specificity, *i.e.*, the proportion of true negatives identified correctly, of 99.02%. Therefore, with an average precision of 93.3%, recall of 100%, and specificity of 99.02%, we positively support our second assumption on the detection accuracy.
A3. Extensibility: We claim that our DSL and SOFA framework are extensible for new antipatterns. In this paper, with ten new process-specific metrics, we specified and detected eight process-specific antipatterns using our framework. In our previous work [**?**], we specified ten SCA-specific antipatterns using the rule-based language and detected them in SCA systems using SOFA. The designed language is flexible enough for integrating new metrics in the DSL. SOFA also supports the addition of new antipatterns through the implementation of new metrics. With this extensibility feature of our DSL and the SOFA framework we support our third assumption.

5.4 Comparison with the State of the Art Approaches

We compare here SODA-BP with 11 state-of-the-art approaches shown in Table 3. Most of the approaches do not support automatic detection of antipatterns, but the ones that do, focus only on BMPN models and use Petri nets. For example, Maruta *et al.* [9] has an accuracy of 100% with five antipatterns, but the reported results are only for three process models. Gruhn *et al.* [10] and Laue *et al.* [12] have accuracy close to SODA-BP with only six antipatterns after analysing more than 100 models. In contrast, SODA-BP analyses and performs detection for eight antipatterns on 35 processes with the accuracy of more than 90%. Overall, considering the trade-off between the number of BP antipatterns specified and detected and the number of processes analysed, SODA-BP has a good detection accuracy of more than 90%.

Table 3: Comparison of SODA-BP with the state-of-the-art approaches.

Contributions	Focus Groups	Perform Automatic Detection?	Number of Patterns / Antipatterns	Accuracy
Awad *et al.* [8]	Data-flow anomalies	×	5	-
Gruhn *et al.* [10]	BPMN soundness properties	✓	6	98.31%
Koehler *et al.* [3]	BPMN modeling errors	×	18	-
Laue *et al.* [12]	BPMN modeling errors	✓	6	95.41%
Lei *et al.* [13]	Data access exceptions in BPEL	×	2	-
Maruta *et al.* [9]	Deadlock patterns in BPs	✓	5	100%
Onoda *et al.* [4]	Deadlock patterns in BPs	×	5	-
Ouyang *et al.* [14]	Reachability analysis	×	3	-
Persson *et al.* [5]	Enterprise modeling practices	×	13	-
Stirna *et al.* [6]	Enterprise modeling practices	×	9	-
Trčka *et al.* [11]	Data-flow errors in models	×	9	-
SODA-BP	Process antipatterns in BPEL4WS	✓	8	93.3%

6 Conclusion and Future Work

Structure business processes (SBPs) are now the prominent means to build enterprise solutions by orchestrating Web services. The presence of process antipatterns in SBPs may hinder maintenance and degrade their quality of design. Therefore, the detection of antipatterns is important to maintain and improve the process quality.

In this paper, we presented the SODA-BP approach to specify and detect process antipatterns. We specified eight process antipatterns from the literature to support their detection. Then, we applied SODA-BP on a set of 35 SBPs randomly-selected form a corpus of more than 150 collected processes from an open-source search engine, some of which were modified by adding, removing, or modifying process elements to introduce noise within the models. Results show that SODA-BP detects process antipatterns with an average precision of more than 75% and the recall of 100%.

In this paper, we analysed the SBPs statically and currently we are performing dynamic analyses by executing them to collect their runtime properties. We plan also to replicate our evaluation of SODA-BP on real SBPs from our industrial partners. Finally, we intend to focus on the automatic correction of process antipatterns in the future.

Acknowledgment This work is supported by the NSERC grant, a Canada Research Chair, and a FRQNT grant.

References

1. OMG: (Object Management Group): Business Process Modeling Notation (BPMN) version 1.2. Technical report, www.bpmn.org (January 2009)
2. Alves, A., et al.: Web Services Business Process Execution Language Version 2.0. Technical report (2007)
3. Koehler, J., Vanhatalo, J.: Process Anti-Patterns: How to Avoid the Common Traps of Business Process Modeling. IBM WebSphere Developer Technical Journal (February 2007)

4. Onoda, S., Ikkai, Y., Kobayashi, T., Komoda, N.: Definition of Deadlock Patterns for Business Processes Workflow Models, IEEE Computer Society (1999)
5. Persson, A., Stirna, J.: How to Transfer a Knowledge Management Approach to an Organization - A Set of Patterns and Anti-patterns. PAKM '06, Springer-Verlag (2006) 243–252
6. Stirna, J., Persson, A.: Anti-patterns as a Means of Focusing on Critical Quality Aspects in Enterprise Modeling. In: Enterprise, Business-Process and Information Systems Modeling. Volume 29. Springer Berlin Heidelberg (2009) 407–418
7. Dijkman, R., Dumas, M., nuelos, L.G.B., Käärik, R.: Aligning Business Process Models. In: IEEE International Enterprise Distributed Object Computing Conference. (September 2009) 45–53
8. Awad, A., Decker, G., Lohmann, N.: Diagnosing and Repairing Data Anomalies in Process Models. In: BPM 2009 Workshops. Volume 43., Berlin, Heidelberg, Springer-Verlag (2010) 5–16
9. Maruta, T., Onoda, S., Ikkai, Y., Kobayashi, T., Komoda, N.: A Deadlock Detection Algorithm for Business Processes Workflow Models. In: IEEE International Conference on Systems, Man, and Cybernetics, Vol 1. (October 1998)
10. Gruhn, V., Laue, R.: A Heuristic Method for Detecting Problems in Business Process Models. BPM **16** (September 2010) 806–821
11. Trčka, N., van der Aalst, W.M., Sidorova, N.: Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows. CAMISE '09, Berlin, Heidelberg, Springer-Verlag (2009) 425–439
12. Laue, R., Awad, A.: Visualization of Business Process Modeling Anti Patterns. In: Proceedings of the 1st International Workshop on Visual Formalisms for Patterns. Volume 25. (2010)
13. Lei, K., Zhang, P.P., Lang, B.: Data Access Exception Detecting of WS-BPEL Process Based on Workflow Nets. In: International Conference on Computational Intelligence and Software Engineering (CiSE), 2010. (2010) 1–6
14. Ouyang, C., Verbeek, E., van der Aalst, W.M., Breutel, S., Dumas, M., ter Hofstede, A.H.: WofBPEL: A Tool for Automated Analysis of BPEL Processes. In Benatallah, B., Casati, F., Traverso, P., eds.: International Conference on Service-Oriented Computing. Volume 3826 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2005) 484–489
15. Van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Distributed and Parallel Databases **14**(1) (July 2003) 5–51
16. Consel, C., Marlet, R.: Architecturing Software Using A Methodology for Language Development. Lecture Notes in Computer Science **1490** (September 1998) 170–194
17. Group, O.M.: Object Constraint Language (OCL) (February 2014)
18. Ma, Z., Leymann, F.: A Lifecycle Model for Using Process Fragment in Business Process Modeling. In: Proceedings of Business Process Modeling, Development, and Support. (2008)
19. Vanhatalo, J., Vlzer, H., Koehler, J.: The Refined Process Structure Tree. Data & Knowledge Engineering **68**(9) (2009) 793–818 Sixth International Conference on Business Process Management (BPM 2008) Five selected and extended papers.
20. OASIS: SCA Service Component Architecture - Assembly Model Specification. Open SOA, www.osoa.org. (March 2007) Version 1.00.