Instantaneous, Comprehensible, and Fixable Soundness Checking of Realistic BPMN Models

Tim Kräuter[1], Patrick Stünkel[1], Adrian Rutle[1], Harald König[2,1], and Yngve Lamo[1]

[1] Western Norway University of Applied Sciences, Bergen, Norway
tkra@hvl.no, patrick.stuenkel@hvl.no, aru@hvl.no, yla@hvl.no
[2] University of Applied Sciences, FHDW, Hanover, Germany
harald.koenig@fhdw.de

Abstract. Many business process models have control-flow errors, such as deadlocks, which can hinder proper execution. In this paper, we introduce our new soundness-checking tool that can instantaneously identify errors in BPMN models, make them comprehensible for modelers, and even suggest corrections to resolve them automatically. We demonstrate that our tool's soundness checking is instantaneous, i.e., it takes less than 500ms, by benchmarking our tool against synthetic BPMN models with increasing size and state space complexity, as well as realistic models provided in the literature. Moreover, the tool directly displays possible soundness violations in the model and provides an interactive counterexample visualization of each violation. Additionally, it provides fixes to resolve the violations found, which are not currently available in other tools. The tool is open-source, modular, extensible, and integrated into a popular BPMN modeling tool.

Keywords: BPM · Verification · Soundness · Safeness · Control-Flow

1 Introduction

Business Process Modeling Notation (BPMN) is becoming increasingly popular for automating processes and orchestrating people and systems. However, many process models suffer from control-flow errors, such as deadlocks and lack of synchronization [7]. These errors hinder the correct execution of BPMN models and may be detected late in the development process, resulting in elevated costs.

In this paper, we describe our new tool, which checks BPMN process models for soundness and safeness [6], which entails finding control-flow errors already during modeling. Figure 1 shows an overview of our tool, which we provide online [12]. The tool front-end is based on the popular *bpmn.io* ecosystem, while the soundness checker is implemented in Rust for fast performance, memory efficiency, and memory safety.

The tool can check models after each change since soundness checking is *instantaneous* according to [7], i.e., it takes 500ms or less. Furthermore, we ensure the results are *comprehensible* by highlighting possible violations directly in the
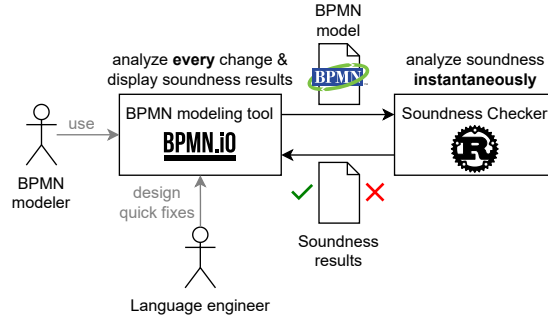
Fig. 1. Overview of the tool

model and displaying an interactive counterexample visualization. Finally, the tool suggests *fixes* for the most common soundness violations and can be extended to suggest more fixes independent of the soundness checker. However, the tool is still a *prototype* and cannot provide fixes for all possible violations due to the wide variety of control-flow errors.

In the remainder of the paper, we describe how instantaneous, comprehensible, and fixable soundness checking is achieved in section 2. Then, we explain our tool implementation in section 3 before presenting related work (section 4). Finally, we discuss possible limitations in section 5 and conclude in section 6.

2   Soundness Checking of BPMN Models

Soundness was introduced for workflow nets in [13] and stems from the field of Petri Nets. We will use the formal definition directly given for BPMN by [6]. Soundness is composed of the three following sub-properties [6]: *(i) Option to complete*: any running process instance must eventually complete, *(ii) Proper Completion*: at the moment of completion, each token of the process instance must be in a different end event and *(iii) No dead activities*: any activity can be executed in at least one process instance. Option to complete is vital since it guarantees the absence of deadlocks, while proper completion is less important, but it helps enforce BPMN best practices, such as not reusing end events. Furthermore, It is crucial not to have dead activities since, similar to dead code, they should be removed or they indicate a deeper issue that must be investigated.

In addition, we check *Safeness* to find possible missing synchronizations. A BPMN model is *safe* if, during its execution, no more than one token occurs simultaneously along the same sequence flow [6]. Safeness helps to find *lack of synchronization* [7], for example, if outgoing sequence flows of a parallel gateway are merged using an exclusive gateway. This would lead to all subsequent activities being executed twice, which is often not desired.

In the remainder of this section, we describe how the tool achieves *instantaneous*, *comprehensible*, and *fixable* soundness checking of realistic BPMN models.

2.1   Instantaneous Soundness Checking

Instantaneous soundness checking is defined as taking 500 ms or less in [7]. In this section, we demonstrate that our soundness checker is instantaneous by validating it from three viewpoints. First, we investigate how our tools react to rapidly *growing model size*. We use a benchmark based on synthetically generated BPMN models. Second, we study how our tool deals with a *growing number of parallel branches* of varying size. Third, we run soundness checking for realistic BPMN process models available in the literature and public datasets.

For all our benchmarks, we use the hyperfine benchmarking tool [11] (version 1.18.0), which calculates the average runtime when executing each soundness check ten or more times. The benchmarks were run on Ubuntu 22.04.4 with an AMD Ryzen 7700X processor (4.5GHz) and 32 GB of RAM. All used BPMN models, our tools to generate them, and benchmarking scripts are given in the artifacts of this paper [12].

Growing Model Size We use an extended version of the data set of models provided in [10], which consists of 300 synthetically generated BPMN models, and increase it to 500 models. Every BPMN model contains a start event, a fixed amount of *blocks*, and an end event. The three unique blocks are shown in Figure 2. To generate the models, we start with the first block and add one more block at a time, always following the same order until we reach 500. Figure 2 shows the third BPMN model in the data set, which contains each block once [10].
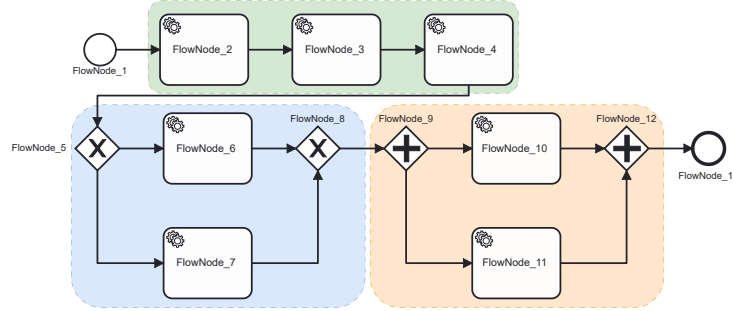

Fig. 2. BPMN model with three blocks [10]

Figure 3 shows the average runtime of our tool when checking soundness and safeness for the BPMN models with increasing model size. Our tool explores the entire state space while simultaneously verifying all properties.

Our tool spends from 1 ms up to 157 ms for the BPMN models compared to 0.7 s up to 14 s in [10] for only 300 blocks on a similar machine. Thus, our tool can be classified as *instantaneous* according to [7] even if the model size increases to over 4000 BPMN elements (500 blocks, 2168 states). Models of this size are not human-readable anymore and are usually divided into smaller models according
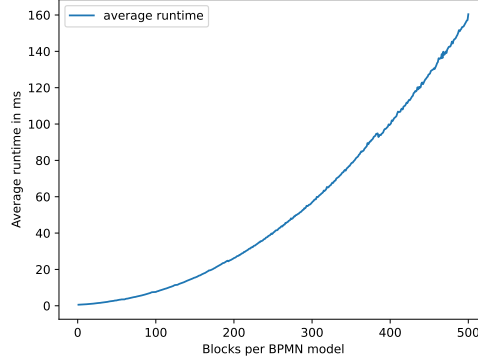
**Fig. 3.** Soundness checking runtime for models of increasing size

to best practices [7]. Thus, models found in practice are likely to be smaller than those found in our benchmark. However, they might be more complex, leading to a larger state space, as discussed in the next section.

Growing Number of Parallel Branches An increased model size leads to a bigger state space that must be analyzed. In the previous section, a linear increase in model size led to a similar growth in the state space. However, models with a growing number of parallel branches lead to an exponential increase in the state space, i.e., a state space explosion [3]. In this section, we benchmark our tool against a synthetic data set of models that lead to a state space explosion. This represents a *worst case* scenario for soundness checking.

We generated a data set of models [12] with a growing number of parallel branches with increasing length, similar to [4]. Figure 4 depicts how a model with n parallel branches with length m is generated.

Table 1 shows the average runtime of our tool when checking soundness and safeness for BPMN models with a varying number of parallel branches and branch lengths. Our tool explores the entire state space while simultaneously verifying all properties.

The models' state space grows exponentially, leading to the same growth in runtime. In our case, the number of states is given by $(m+1)^n + 3$ due to our pragmatic encoding (see section 3), clearly showing state space explosion. Our analysis is not instantaneous anymore when approaching 17 parallel branches of length 1 (see Table 1). However, analysis is still instantaneous for more reasonable models with five parallel branches of length 5 or 3 branches of length 10-20. [4] report 2-3s of runtime for most soundness properties and 30 s for no dead activities for the model with five parallel branches of length 1, which took 1 ms in our tool. The difference is due to faster hardware and implementation differences, such as our pragmatic encoding leading to smaller state spaces, see section 3.
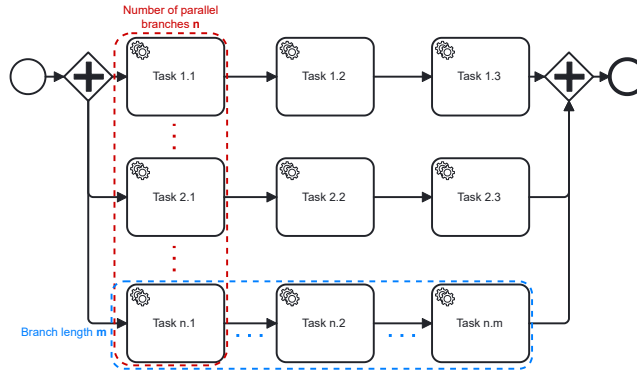
Fig. 4. Models with a growing number of parallel branches and branch length

Table 1. Benchmark results of the parallel branches models

Branches	Branch Length	Runtime	States
5	1	1 ms	35
10	1	4 ms	1.027
15	1	164 ms	32.771
16	1	368 ms	65.539
17	1	806 ms	131.075
20	1	8.859 ms	1.048.579
5	3	3 ms	1.027
5	5	19 ms	7.779
3	10	4 ms	1.334
3	20	28 ms	9.264

We believe, as also reported in [7], that models with a high degree of parallelism are uncommon. For example, only 3% of the industrial models in [7] have more than 1000 states. If complex models are more common than anticipated, one can implement *partial order reduction* [3], which has shown great results for the Petri Net model checker LoLA in [7]. Due to partial order reduction, they reduced the analysis time and checked previously intractable models. One can implement partial order reduction similarly in our tool since the semantics of BPMN and Petri Nets have many similarities. However, translating BPMN to Petri nets is not always possible.

Realistic Models We apply our tool to eight realistic models, where three models (e001, e002, e020) are taken from [8], and the remaining five models

are part of the Camunda BPMN for research repository[3]. We had to slightly
adapt the Camunda models such that they can be executed standalone, which
is described in our artifacts [12].

Table 2 shows the average runtime of soundness and safeness checking using
our tool and the number of states for each model. The results show that our tool
can check soundness and safeness *instantaneously* for the given realistic models.

Table 2. Benchmark results of the realistic BPMN models

Model name	Runtime	States	Violated Properties
e001 [8]	1 ms	31	Option To Complete, No Dead Activities
e002 [8]	1 ms	31	Option To Complete, No Dead Activities
e020 [8]	5 ms	2.112	Option To Complete
credit-scoring-async[3]	1 ms	50	Option To Complete
credit-scoring-sync[3]	1 ms	99	Option To Complete
dispatch-of-goods[3]	1 ms	103	Safeness, Proper Completion
recourse[3]	1 ms	71	Option To Complete
self-service-restaurant[3]	1 ms	74	Option To Complete

Each model violates one or more soundness properties. *Option To Complete*
is violated since the given models might not terminate if their execution or-
der leads to losing messages. *Safeness* and *Proper Completion* are violated in
dispatch-of-goods due to a parallelization which is not synchronized later. How to
automatically fix violated properties is discussed in subsection 2.3. For example,
the violations in dispatch-of-goods are automatically resolvable.

Furthermore, our tool takes 1-7ms for e001, e002, and e020 while [8] and [9]
report 3.66-10.26s and 1-1.75s. The benchmarks in [9] were run on a machine
with similar hardware, while the machine used in [8] was slightly less powerful.

2.2   Comprehensible Soundness Checking

The first step to fixing a soundness violation is understanding the problem. Thus,
a tool must present soundness violations clearly and provide the necessary details
to the modeler. We aim to make soundness checking comprehensible by providing
textual feedback and utilizing the BPMN model's graphical structure.

We highlight the problematic elements in the BPMN model that cause sound-
ness violations. Figure 5 depicts how we highlight problematic elements using
red overlays. In addition, there is a summary panel in the top-right corner.

Using the overlays, one can immediately see which elements cause soundness
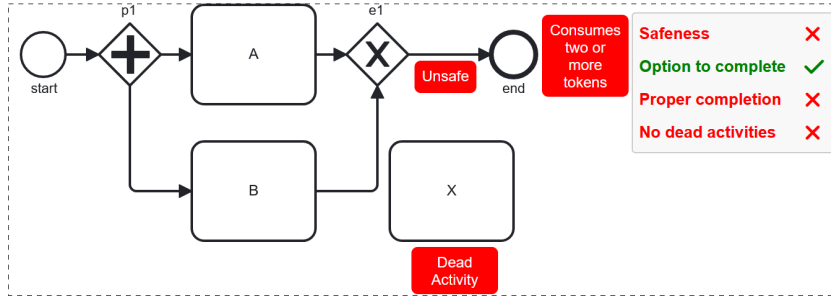violations, which helps to pinpoint the root problems in the BPMN models. For

---

[3] https://github.com/camunda/bpmn-for-research

Fig. 5. *Soundness violations* in the analysis tool

Safeness, we highlight sequence flows that can become *unsafe*, i.e., two tokens can be located at the sequence flow. For *Proper Completion*, we identify the end events that can consume more than one token, and for *No Dead Activities*, we highlight the dead activities. In contrast, we cannot highlight elements for *Option To Complete* since it means the process execution must not always terminate, which cannot easily be attributed to single BPMN elements.

However, just by looking at the BPMN model, it can still be hard to understand soundness violations due to the interplay of the BPMN element's execution semantics. In the BPMN specification, execution semantics are described using the concept of moving *tokens*. Tokens are used universally in the industry and research to depict process execution information [1,4,5,8,9,10].

We use tokens in our tool to *interactively* visualize the counterexamples, i.e., violation witnesses of our soundness properties. Our soundness checker provides counterexamples for all properties except *No Dead Activities*. Then, we visualize these counterexamples directly in the BPMN editor by showing how tokens move from the process start to a state that violates the given soundness property. Figure 6 shows a screenshot visualizing the Safeness counterexample for the same BPMN model as shown in Figure 5.
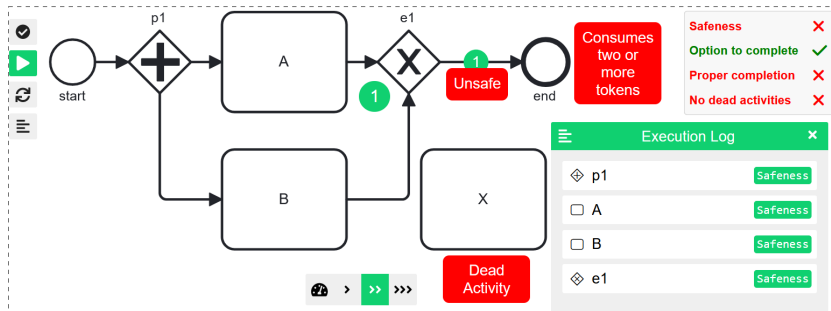

Fig. 6. Interactive *counterexample visualization* in the analysis tool

In Figure 6, the visualization has been *paused* just before the *unsafe* state was reached. One token is already located at the sequence flow, which is marked as unsafe, while a second token is currently waiting at the exclusive gateway e1. The visualization can be resumed or restarted using the play and restart button on the left side. When resumed, the gateway e1 will execute, resulting in two tokens at the subsequent sequence flow, i.e., an unsafe execution state. In addition, one can control the visualization speed using the bottom buttons next to the speedometer.

Furthermore, our tool shows an *execution log*, which states the history of executing BPMN elements. In Figure 6 the parallel gateway p1, the activities A and B, as well as the exclusive gateway e1 have each been executed once before the pause. This is useful since an unsuspected execution order is often the culprit for property violations.

In summary, we aim to make soundness checking *comprehensible* even for users unaccustomed to the BPMN execution semantics. First, we directly high-light problematic elements in the model. Second, we provide an *interactive* vi-sualization of counterexamples using tokens with the ability to pause the visu-alization, control the visualization speed, and show an execution log.

2.3   Fixable Soundness Checking

If possible, we provide an automatic fix similar to *quick fixes* in Integrated Devel-opment Environments (IDEs) for detected violations. Modelers can then select these quick fixes to restore soundness. Figure 7 shows a screenshot of our tool, where quick fixes are depicted as green overlays containing a light bulb icon. This icon is typically used to indicate quick fixes in IDEs. Our tool simultaneously shows soundness violations (Figure 5) and quick fixes (Figure 7).
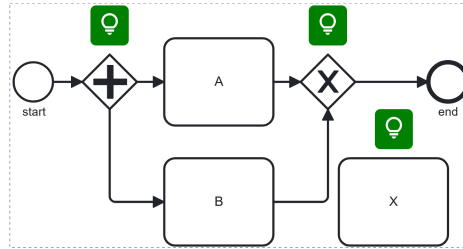
Fig. 7. *Quick fixes* in the analysis tool

The user can apply a quick fix by clicking on a green overlay and instantly see the changes in soundness and safeness. If unhappy with the result, a user can undo all changes since each quick fix is entirely revertible due to the command pattern. A user might not like a quick fix if it not only fixes a specific property but also has unintended side effects. For example, it might invalidate a different soundness property.

In the following sections, we describe the implemented resolution strategies for the different soundness properties. However, we do not expect these strategies to cover all possible resolutions and fit all BPMN models. Thus, our tool is extensible, so one can provide custom resolution strategies. This is possible due to its *modular* design compatible with standard mechanisms of the *bpmn.io* ecosystem, which *decouples* the soundness analysis from reporting violations, visualizing counterexamples, and providing quick fixes.

Safeness The soundness checker will provide a counterexample and identify the unsafe sequence flows. We use this information together with the structure of the BPMN model to find resolutions for *Safeness* violations. Possible reasons for *Safeness* violations which we address are:

(**1**) An exclusive gateway with multiple incoming sequence flows might be executed more than once, leading to multiple tokens at its outgoing sequence flows.
(**2**) Similarly, implicitly encoded exclusive gateways exist, for example, if an activity has multiple incoming sequence flows. Implicitly encoding exclusive gateways is allowed but violates best practices [2]. Similar to (**1**), this leads to unsafe sequence flows if the activity is executed more than once.

Resolutions for (1): A straightforward solution is to change the exclusive gateway to a parallel gateway. Since the gateway was activated multiple times, it indicates that it perhaps should have been a parallel gateway or that there was an unintended parallelization before. Thus, we can analyze the BPMN model and try to find the parallelization that causes the *Safeness* violation. If we find a parallel gateway, another solution is to change this parallel gateway to an exclusive one. This leads to two possible solutions with the overarching goal of matching gateways.

The parallelization can also be implicitly encoded, for example, using an activity with multiple outgoing sequence flows. This does not comply with best practices [2] but is allowed. In this case, we can add an explicit exclusive gateway to eliminate the implicit parallelization and achieve matching gateways.

Resolutions for (2): Similarly to (**1**), the goal of each quick fix is to obtain matching gateways. This is not always possible since BPMN models must not be well-structured. Thus, one solution is to find the parallelization that causes the *Safeness* violation and change it to an exclusive gateway, as described in (**1**). Quick fix (**a**) in Figure 8 shows this solution, where a parallel gateway is changed to an exclusive one. We color the changes and additions in green to highlight the effect of quick fixes.

Another solution is to remove the implicit exclusive gateway and add a parallel gateway instead, see quick fix (**b**) in Figure 8. The quick fix moves elements automatically to make space to insert the parallel gateway and then reconnects and adds sequence flows accordingly. Even though these are multiple individual operations, we ensured that an undo operation would revert the entire quick fix. All the implemented quick fixes can be reverted using one undo operation.
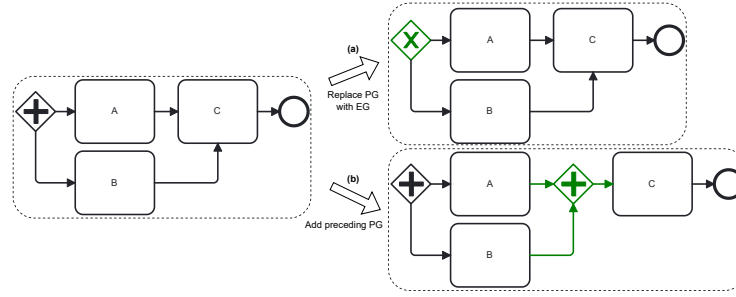
Fig. 8. Example quick fixes for *Safeness*

Our demo application in the artifacts [12] contains examples of the Safeness quick fixes discussed here and all upcoming soundness properties.

Proper Completion For *Proper Completion*, our soundness checker will provide a counterexample and identify the problematic end events that consume more than one token. Possible reasons for *Proper Completion* violations which we address are:

(**1**) End event with multiple incoming sequence flows can be executed twice or more. This could be due to a parallelization that is never synchronized.
(**2**) If there is only one incoming sequence flow, then this flow must be unsafe, i.e., hold more than one token in a possible execution.

Resolution for (1): If multiple incoming sequence flows are the cause, we can add additional end events to match the number of sequence flows. Figure 9 shows an example of applying this quick fix. We ensure an undo operation would revert the quick fix using the command pattern.
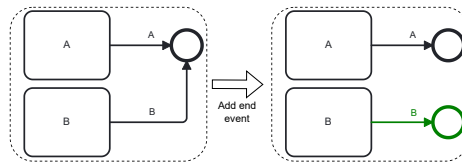

Fig. 9. Example quick fix for *Proper Completion*

Resolution for (2): If a problematic end event only has one incoming sequence flow, it must be unsafe. Thus, other *Safeness* quick fixes can apply, which will also resolve *Proper Completion*.

Option to Complete Violations of *Option to Complete* can have multiple reasons.

(**1**) A parallel gateway that synchronizes multiple incoming sequence flows but never executes leads to a violation.

(**2**) An event that is never triggered but relied upon leads to a violation.

To know the reason for a given violation, we analyze the counterexample provided by the soundness checker. The counterexample provides a trace that leads to a state in which the process cannot complete. By analyzing the last state in this trace, i.e., the state in which execution cannot continue, we can determine which element is the cause. Thus, we can provide quick fixes for the possible reasons.

Resolutions for (1): A straightforward way to fix sequence flow not continuing past a parallel gateway is to change it to an exclusive gateway. Similar to the Safeness quick fixes, we obtain matching gateways. Exclusive gateways do not synchronize, and thus, execution can continue. Figure 10 (**a**) shows an example of this quick fix.
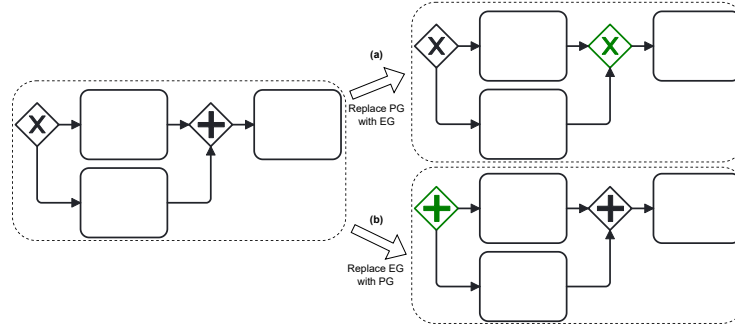

Fig. 10. Example quick fixes for *Option To Complete*

Another way to fix violations is to find the split in the sequence flow, for example, the exclusive gateway in Figure 10, and make this split a parallelization, see quick fix (**b**) in Figure 10. We present both possible resolutions to the user, who can choose the appropriate one. In the example in Figure 10, it is possible to spot the mismatching gateways. However, this might not be straightforward in bigger BPMN models with more flow nodes and sequence flows.

Resolutions for (2): We have only implemented quick fixes for untriggered message events since we do not support other intermediate events. If an untriggered message event has no incoming message flows, we can add a message flow from the closest message throw event or send task to the event. Currently, we only use spatial proximity to find the right source of the new message flow, but other factors, such as name similarity or reachability, can be considered. Generalizing this idea to different event types requires adding the missing event trigger. Analyzing other elements in the model or involving the modeler can help find a potential trigger.

No Dead Activities A dead activity might have multiple reasons:

(**1**) An activity can be disconnected, i.e., it has no incoming sequence flow.
(**2**) An activity can be a receive task with no incoming message flows.
(**3**) An activity can also be part of the BPMN model that is not reachable during execution because, for example, a parallel gateway or event preceding the activity cannot be executed.

Resolutions for (1)/(2): If the activity has no incoming sequence flow/message flow, we can propose adding the missing flow. A sequence flow is added from the nearest flow node, which is not disconnected or dead, while a message flow is added from the nearest message throw event or send task.

Resolutions for (3): When the dead activity is connected, this means it is unreachable during execution, and often the process itself cannot terminate, i.e., violates *Option to Complete*. Thus, other quick fixes can potentially be applied.

3   Implementation

In this section, we briefly describe our implementation of the BPMN soundness-checking tool. The tool's main goal is to be performant, comprehensible, and well-integrated into modeling tools without introducing unnecessary friction.

3.1   Tool overview

Our tool is open-source and available as an artifact [12]. The tool architecture is shown in Figure 1, while screenshots of our tool with different features enabled are given in Figure 5, Figure 6, and Figure 7. The front-end of the tool is built in web technologies using the *bpmn.io* ecosystem, especially the bpmn-js-token-simulation [1], while the soundness and safeness checking is implemented in the Rust programming language.

We developed soundness checking in Rust due to its memory efficiency, absence of garbage collection and lightweight hardware abstractions, resulting in an execution performance comparable to C or C++, while retaining memory safety guarantees. A fast programming language and direct implementation of the BPMN semantics play the key role in achieving *instantaneous* soundness checking. We would like to emphasize that the current implementation does feature specific optimization techniques such as partial order reduction, therefore, we assume that performance may be amplified even further for certain types of models or violations, see, e.g., the parallel branches example.

The tool can be integrated into existing BPMN modeling tools since its model capabilities are invoked through a *web-service* interface or as *command-line application*. Furthermore, the front-end is *modular* such that one can either fully integrate it, pick only specific features, or extend it, for example, to add custom quick fixes.

3.2   Soundness checking in Rust

We implemented a standard breadth-first state space exploration [3]. While generating the state space, we can check safety properties, such as *Safeness* and *Option To Complete*, *on-the-fly* for each found state [3]. For example, Listing 1 shows how to find unsafe sequence flows in a state to check *Safeness* by inspecting the number of tokens in each state.

Listing 1. Find the IDs of *unsafe* sequence flows in a given state

```rust
pub fn find_unsafe_sf_ids(&self) -> Vec<&String> {
    self.snapshots.iter()
        .flat_map(|snapshot| snapshot.tokens.iter())
        .filter(|&(_, token_amount)| *token_amount >= 2)
        .map(|(sf_id, _)| sf_id)
        .collect() }
```

Self refers to a struct holding a set of snapshots, i.e., process instances with a map of tokens, that links sequence flow IDs to the number of tokens they are holding.

Similarly, for *Option To Complete*, we examine whether a state is stuck. This means that some process instance is still ongoing, indicated by the presence of tokens, and the state has no outgoing transitions. In contrast, we check *No Dead Activities* by remembering all executed activities during state space generation and comparing them to all activities in the model. Furthermore, *Proper Completion* is checked by remembering executed end events in each state.

3.3   Pragmatic BPMN Semantics Encoding

In addition to selecting an efficient programming language for developing the soundness checker, we also opted for a pragmatic, straightforward encoding of BPMN semantics. Our encoding contains sufficient details to check soundness and safeness while omitting unnecessary intermediate states, which leads to smaller state spaces. For example, Figure 11 shows that we do not encode the start and end of a task but rather the execution as a whole. As discussed earlier, this simple abstraction avoids one additional state and pays off, especially in models with many parallel branches, which can significantly reduce the overall state space. For example, our tool finds only 2.112 states for the BPMN model e020 (see subsection 2.1), compared to the 3.558/3.060 in [8]/[9].

Similar minimal encodings can be applied to other BPMN elements, such as message events, to keep the state space small. Nevertheless, one must always be sure that such optimizations do not compromise the checked properties, similar to when applying partial-order reduction techniques [3].

Our pragmatic encoding has potential downsides if one wants to check custom properties in the future, such as Activity A and B not executing simultaneously. Since we are only interested in soundness and safeness, we are consciously trading a smaller state space for the inability to check such custom properties.
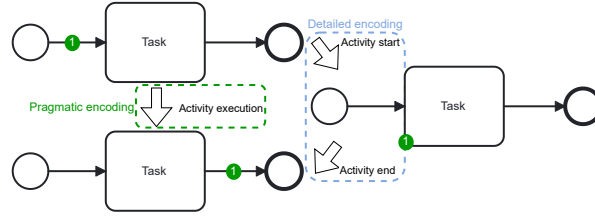
Fig. 11. Pragmatic task execution encoding (left) and detailed encoding (right)

4   Related Work

BPMN specification coverage: Most related work focuses on the depth of BPMN formalization while providing soundness and safeness checking. Thus, these approaches show how different BPMN elements can be formalized and compare themselves with each other regarding supported elements [4,8,9,10]. The supported BPMN elements come close to the capabilities of popular process orchestration platforms. Our tool supports the same depth of BPMN elements as [4], i.e., the most used gateways, tasks, and events. A detailed comparison is available in [12], and we plan to support more elements in the future. We focus on tool performance and capabilities concerning soundness comprehension, soundness resolution, and seamless integration into BPMN modeling tools.

    Tool performance: Comparing tool performance without standard benchmarks and a reproducible environment is challenging. However, other publications indicate that other tools take several seconds up to half a minute to check single soundness properties [4,8,10]. In contrast, our approach instantaneously checks *all* soundness properties and safeness of the same models. The difference in performance lies probably in our pragmatic BPMN encoding optimized for soundness checking and its direct implementation in a performant programming language rather than in transforming BPMN into general model-checking tools.

    Tool capabilities: Another way to compare the different BPMN formalizations and soundness-checking tools is to investigate their capabilities. Most tools formalize large parts of the BPMN specification and allow soundness and safeness checking [4,8,9,10]. Some tools investigate additional aspects such as the introduction of *data* and *time* during verification [5,8]. Furthermore, other tools allow specifying and checking *custom temporal logic properties* [4,9] and even provide graphical interfaces to easy the specification [10] Moreover, some tools provide interactive BPMN simulation using token-flow animation [1,5], while others *visualize counterexamples* for soundness violations using tokens [8].

    Our tool focuses on *instantaneous* soundness and safeness checking and does not support custom properties or data and time. We do not provide BPMN simulation since other tools already offer this. However, we use token-flow animation to visualize counterexamples for soundness violations interactively. This improves comprehension compared to previous static, less interactive visualizations. In addition, to the best of our knowledge, our tool is the only one that

provides *quick fixes*, i.e., automatic resolutions if soundness properties are violated. To sum up, our tool incorporates and enhances several ideas from the state of the art while adding novel concepts, such as quick fixes. We advocate for a pragmatic approach, prioritizing performance and understanding above all else to ensure seamless integration into BPMN modeling tools.

5   Limitations & Threats to Validity

Our tool is a *prototype* that focuses on soundness and safeness and does not check custom properties. Furthermore, the suggested quick fixes cannot repair all possible violations since the structure of BPMN models allows nearly arbitrary combinations of elements.

Assessing tool performance becomes difficult when there are no standardized benchmarks for comparison. In this publication, a direct comparison of our tool's performance with related work is not feasible. This is due to variations in benchmark conditions, such as hardware, operating systems, and employed methodologies. We claim that our tool can deal with most BPMN models instantaneously, as previously discussed. It is advisable to consider the results reported by other researchers on the same models only as a general reference. To solve this problem, we advocate defining a standardized benchmarking process to compare tool performance directly and transparently.

The main threat to the validity of our instantaneous soundness-checking claim is that we could not test our tool with a large set of models directly from the industry. To mitigate this threat, we validated our claim against models sourced from existing literature, public repositories, and two artificial datasets. These datasets, in terms of model size and state space complexity, closely resemble or even surpass industrial models.

6   Conclusion & Future work

In this paper, we describe a novel tool that provides instantaneous, comprehensible, and fixable BPMN soundness checking and is integrated into a popular BPMN modeling tool. We benchmarked our tool against synthetic and realistic BPMN models to demonstrate instantaneous soundness checking. Our artifacts contain the synthetic data sets of BPMN models and transparently describe our methodology [12]. Our methodology and data sets can be used to better benchmark the performance of soundness-checking tools in the future.

Three main challenges for providing soundness-checking capabilities to non-expert users are identified in [7]. First, a soundness checker must be able to check all or most user-created models, i.e., it must support the most used BPMN elements. This is not a problem for most tools, including ours, which has similar capabilities to [4], and we plan to increase our tool's BPMN coverage as needed. Second, soundness checking must be *instantaneous* since long runtimes are unacceptable and often interpreted as tool errors [7]. Third, the biggest challenge for soundness checking is *consumability*, i.e., reporting the found violations in a

comprehensible user interface. Our new tool addresses all these challenges, focusing on instantaneous and comprehensible soundness checking and even providing quick fixes for common soundness violations. The tool is a BPMN-specific soundness checker written in Rust paired with an intuitive user interface based on the popular *bpmn.io* ecosystem, which allows extending the tool, for example, to provide custom quick fix suggestions.

In future work, we aim to improve our tool by providing more quick fixes, considering advanced BPMN elements such as different events, and ranking quick fixes based on their usefulness. For example, the impact of quick fixes on soundness properties can be part of the ranking since the resulting model can be checked instantaneously in the background. Other metrics, such as least change and least surprise from the model repair field, can be used, or one can include previous user behavior. Finally, we aspire to test our tool in a real-world scenario to gather feedback and measure its impact on productivity.

References

1. Camunda Services GmbH: Bpmn-js Token Simulation. https://github.com/bpmn-io/bpmn-js-token-simulation (Mar 2024)
2. Camunda Services GmbH: Bpmnlint. https://github.com/bpmn-io/bpmnlint (Mar 2024)
3. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer International Publishing, Cham (2018). `https://doi.org/10.1007/978-3-319-10575-8`
4. Corradini, F., Fornari, F., Polini, A., Re, B., Tiezzi, F., Vandin, A.: A formal approach for the analysis of BPMN collaboration models. Journal of Systems and Software **180**, 111007 (Oct 2021). `https://doi.org/10.1016/j.jss.2021.111007`
5. Corradini, F., Muzi, C., Re, B., Rossi, L., Tiezzi, F.: Formalising and animating multiple instances in BPMN collaborations. Information Systems **103**, 101459 (Jan 2022). `https://doi.org/10.1016/j.is.2019.101459`
6. Corradini, F., Muzi, C., Re, B., Tiezzi, F.: A Classification of BPMN Collaborations based on Safeness and Soundness Notions. Electronic Proceedings in Theoretical Computer Science **276**, 37–52 (Aug 2018). `https://doi.org/10.4204/EPTCS.276.5`
7. Fahland, D., Favre, C., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Analysis on demand: Instantaneous soundness checking of industrial business process models. Data & Knowledge Engineering **70**(5), 448–466 (May 2011). `https://doi.org/10.1016/j.datak.2011.01.004`
8. Houhou, S., Baarir, S., Poizat, P., Quéinnec, P., Kahloul, L.: A First-Order Logic verification framework for communication-parametric and time-aware BPMN collaborations. Information Systems **104**, 101765 (Feb 2022). `https://doi.org/10.1016/j.is.2021.101765`
9. Kräuter, T., Rutle, A., König, H., Lamo, Y.: Formalization and Analysis of BPMN Using Graph Transformation Systems. In: Fernández, M., Poskitt, C.M. (eds.) Graph Transformation, vol. 13961, pp. 204–222. Springer Nature Switzerland, Cham (2023). `https://doi.org/10.1007/978-3-031-36709-0_11`
10. Kräuter, T., Rutle, A., König, H., Lamo, Y.: A higher-order transformation approach to the formalization and analysis of BPMN using graph transformation systems (2023). `https://doi.org/10.48550/ARXIV.2311.05243`

11. Peter, D.: Hyperfine. https://github.com/sharkdp/hyperfine/ (Oct 2023)
12. Tim Kräuter: BPM-2024: Artifacts. https://github.com/timKraeuter/BPM-2024-Extended (Jul 2024)
13. Van Der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. Journal of Circuits, Systems and Computers **08**(01), 21–66 (Feb 1998). `https://doi.org/10.1142/S0218126698000043`