A Library for Detecting Inconsistencies in Declarative Process Models

Sabine Nagel[1,*], Patrick Delfmann[1]

[1]University of Koblenz, Universitätsstr. 1, 56070, Koblenz, Germany

Abstract
We present a comprehensive library for detecting and classifying inconsistencies in declarative process models (DPMs) based on inconsistency structures (i.e., recurring patterns with shared characteristics that describe how combinations of constraints become inconsistent). Here, we not only focus on inconsistencies in the classic-logical sense but also on potential inconsistencies that only occur during run-time. Our approach allows for (1) detecting inconsistency cores based on the previously defined structures, (2) identifying design-time and run-time inconsistencies by extending these cores, and (3) classifying and measuring inconsistencies. We demonstrate the applicability of our implementation with real-life datasets.

Keywords
Declare, Declarative Process Model, Declarative Process Specification, Inconsistency Detection

1. Introduction

Declarative process models (DPMs), for example, using the modeling language Declare, offer a flexible alternative to imperative models by implicitly defining process behavior through constraints, rather than defining exact execution paths [1]. Unfortunately, even a single logical contradiction among constraints can make an entire model unsatisfiable, which is referred to as *inconsistent*. Therefore, being able to detect such inconsistencies is crucial. However, existing approaches are typically limited to checking the satisfiability of the entire model, without isolating the specific problematic combinations of constraints. More critically, current approaches [2, 3] are unable to fully identify run-time inconsistencies (i.e., inconsistencies that are triggered by activity occurrences in a trace) as they only cover specific templates or patterns. To address these gaps, we build on previous work [4] and present a library that detects and classifies inconsistencies based on an extensive collection of *inconsistency structures*, which are recurring patterns that explain how combinations of constraints can result in inconsistencies. Our library enables (1) the detection of *inconsistency cores*, i.e., minimal sets of activities and constraints responsible for an inconsistency, (2) the identification of both design-time and run-time inconsistencies through core extension, and (3) the classification and measurement of cores and inconsistencies. Thereby, we not only aim to detect inconsistencies, but also to support inconsistency comprehension, as the underlying inconsistency structures explicitly describe the causes and components of a minimally inconsistent constraint set. This is especially important when humans are involved in the resolution of inconsistencies, as shown in related works on inconsistency comprehension [5, 6].

2. Preliminaries and Related Work

Definition 1 (Declarative Process Model [1]). A declarative process model (DPM) is a tuple $\mathbf{DPM} = (\mathbf{T}, \mathbf{A}, \mathbf{C})$, where $\mathbf{T}$ is a finite, non-empty set of constraint templates, $\mathbf{A}$ is a finite, non-empty set of activities, and $\mathbf{C}$ is a finite set of constraints that instantiate the templates in $\mathbf{T}$.

Proceedings of the Best BPM Dissertation Award, Doctoral Consortium, and Demonstrations & Resources Forum co-located with 23rd International Conference on Business Process Management (BPM 2025), Seville, Spain, August 31st to September 5th, 2025.
*Corresponding author.
snagel@uni-koblenz.de (S. Nagel); delfmann@uni-koblenz.de (P. Delfmann)
0000-0003-4838-8246 (S. Nagel); 0000-0003-4441-0311 (P. Delfmann)

Generally, we distinguish between *existence* (e.g., Init(a), AtLeastOne(a)) and *relation* constraints (e.g., Response(a,b), NotChainPrecedence(a,b)). While existence constraints are automatically activated upon the start of a trace, relation constraints are either activated by their first parameter (forward constraints; e.g., AlternateResponse(a,b)), their second parameter (backward constraints; e.g., Precedence(a,b)) or both parameters (coupling constraints; e.g., Succession(a,b)). When a DPM contains contradictory constraints, it immediately becomes unsatisfiable, which is referred to as inconsistent.

Definition 2 (Inconsistency [7])**.** An inconsistency is a tuple $I = (Act, C)$, where $Act \subseteq \mathbf{A}$ is the set of activities that minimally activate a set of constraints $C \subseteq \mathbf{C}$ such that $C \cup \bigcup_{act \in Act} \textsc{AtLeastOne}(act) \models \bot$. I is a *minimal inconsistency* if there is no $Act' \subset Act$ or $C' \subset C$ such that (Act', C') is an inconsistency.

To differentiate between design-time and run-time inconsistencies, we consider a minimal inconsistency $I = (Act, C)$ a *classic inconsistency* if $C \models \bot$ and $Act = \emptyset$ and a *potential inconsistency* if $C \models \top$. Furthermore, if $|Act| = 1$ we say that a potential inconsistency is an *actual issue* and if $|Act| > 1$ it is considered a *potential issue* [4, 7]. As multiple inconsistencies can share a common underlying problem, we extend the definition by [4, 7] and define an *inconsistency core*. Table 1 contains a variety of exemplary cores representing different inconsistency types. For instance, the example for POS1 is a classic inconsistency, the one for POS2 an actual issue, and the one for CARD2 a potential issue.

Definition 3 (Inconsistency Core [7])**.** An *inconsistency core* is a tuple $IC = (Act_C, C)$, where $Act_C \subseteq \mathbf{A}$ is a non-empty set of activities that minimally activate C and thus serve as core activations, and $C \subseteq \mathbf{C}$ is a set of constraints. Additionally, IC must be a minimal classic or potential inconsistency, such that no proper subset (Act', C') with $Act' \subseteq \mathbf{A}$ or $C' \subset C$ forms a minimal classic or potential inconsistency.

Related works on inconsistency detection focus mainly on LTL, e.g., [8, 9, 10]. For DPMs, most approaches determine unsatisfiability of an entire model by translating constraints into deterministic finite automata (DFAs) and constructing a product automaton to assess satisfiability [11]. While detecting individual, minimal inconsistencies in DPMs using DFAs is theoretically sound, it suffers from severe scalability issues due to the exponential growth in the number of automata product computations required. Moreover, DFA-based techniques are inherently limited to classic inconsistencies and cannot detect potential inconsistencies that depend on specific activity occurrences. As a result, Corea et al. [2] have implemented a first approach to detect actual issues in DPMs but only consider limited scenarios (e.g., certain path-based inconsistencies). More recently, Corea and Thimm [3] proposed an approach for identifying potential issues, but only based on a small subset of templates.

3. Detection Approach

(1) Preparation. Our library was implemented in Java and accepts Declare models in various formats (TXT, JSON, and CSV), which enables seamless integration with the outputs of existing process mining algorithms. To ensure broad applicability, we support a wide range of Declare templates and also handle naming variations (e.g., AtLeast1 and Participation are treated as AtLeastOne) to enable compatibility with different tools and datasets. As an optional preprocessing step, we check for redundant constraints based on a subsumption hierarchy [1]. Redundant constraints are temporarily removed from the model but are retained for later use, e.g., to assign responsibility (culpability) to constraints or to help guide the selection of appropriate weakening strategies during inconsistency resolution. Although more advanced forms of redundancy exist (e.g., those derived from combinations of constraints), we intentionally delay such checks until after inconsistency resolution, as this might distort inconsistency measurement and significantly change the model. Furthermore, detecting and removing redundant constraints becomes much more feasible after consistency has been restored, as DFAs can be utilized again (which always have an empty language while inconsistencies are present). To improve computational efficiency, we pre-compute and store filtered subsets of constraints to be reused throughout the detection processes. Many inconsistency core detection algorithms rely on graph-based reasoning, which is why we construct multiple graphs using JGraphT[1] and utilize both built-in and

[1]https://jgrapht.org

Table 1
Inconsistency Structures and Exemplary Cores

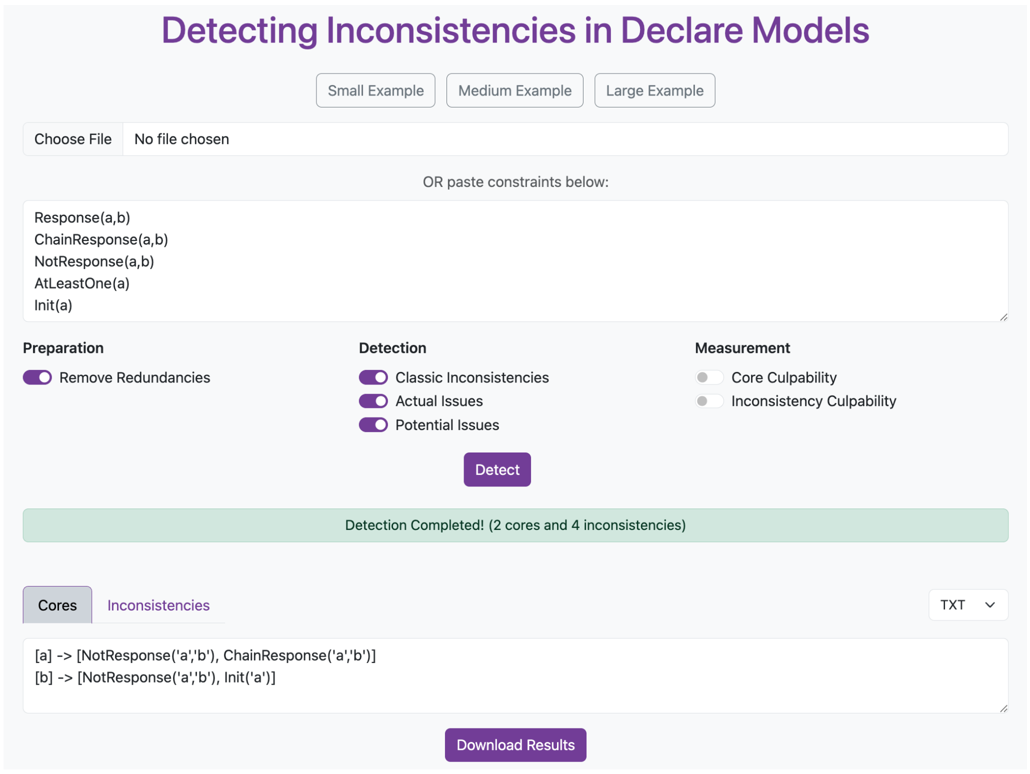
ID	Structure	Exemplary Core
POS1	Multiple Start/End Events	$\{\emptyset\}, \{\textsc{Init}(a), \textsc{Init}(b)\}$
POS2	Multiple Direct Predecessors/Successors	$\{a\}, \{\textsc{ChainResponse}(a, b), \textsc{ChainResponse}(a, c)\}$
CARD1	Explicit Contradictory Cardinality	$\{\emptyset\}, \{\textsc{AtLeastTwo}(a), \textsc{AtMostOne}(a)\}$
CARD2	Implicit Contradictory Cardinality	$\{a, c\}, \{\textsc{ChainResponse}(a, b), \textsc{ChainResponse}(c, b)\}$
COEX	Contradictory Co-Existence	$\{a, b\}, \{\textsc{NotCoExistence}(a, b)\}$
ORD1	Path-Implied Contradictory Order	$\{a\}, \{\textsc{Response}(a, b), \textsc{Precedence}(c, a), \textsc{NotResponse}(c, b)\}$
ORD2	Position-Implied Contradictory Order	$\{b\}, \{\textsc{Init}(a), \textsc{NotResponse}(a, b)\}$
ORD3	Contradictory Chain	$\{a\}, \{\textsc{ChainResponse}(a, b), \textsc{NotChainResponse}(a, b)\}$
BOUND1	Lack of Space	$\{\emptyset\}, \{\textsc{Init}(a), \textsc{Precedence}(b, a)\}$
BOUND2	Loop	$\{a\}, \{\textsc{Response}(a, b), \textsc{Response}(b, a)\}$

custom graph search algorithms. Custom implementations were necessary to incorporate advanced validity checks during the search process, which allowed us to improve efficiency by discarding invalid subtrees early.

(2) Core Detection. We implemented a set of specialized algorithms that are designed to detect cores corresponding to a specific inconsistency structure. These structures are grouped into four categories: position, cardinality, relation, and boundary inconsistencies. An overview of the implemented structures, along with an exemplary core for each, is provided in Table 1. Each algorithm is designed to comprehensively capture cores related to its respective structure. Due to space limitations, we refer to our Git repository for additional examples and extended explanations. Depending on the characteristics of a structure, we apply different algorithmic strategies. For some structures (e.g., POS1, CARD1) the core can be derived directly from a specific combination of constraints. In contrast, other structures (e.g., ORD1, BOUND2) require graph-based search to identify valid cores.

(3) Inconsistency Detection. Once cores have been identified, we proceed to detect complete inconsistencies by analyzing how these cores can be activated within the model. The detection process is configurable, so users can apply filters to focus on specific types of inconsistencies (e.g., only classic or only potential ones) or restrict the detection to certain structures. First, we detect all valid activation paths for all core activations by identifying activating existence constraints in the model (i.e., existence constraints that imply a minimum cardinality of one) and searching for valid paths in a previously defined activation graph. An activation path is considered valid if it does not result in an additional inconsistency when attached to the core, making the resulting constraint set no longer *minimally* inconsistent. For cores with more than one activation, we additionally must ensure compatibility of all path combinations before generating a respective inconsistency, as incompatible paths (e.g., paths that have an additional activity overlap) also make the resulting inconsistency no longer minimal. Each of these inconsistencies includes the core, its activation paths, and the (optional) existence activations required to trigger the inconsistency. This comprehensive detection process enables us to uncover both design-time (classic) and run-time (potential) inconsistencies across the entire model.

(4) Inconsistency Measurement. Our library also allows basic inconsistency and culpability measurement. This includes measuring the degree of inconsistency of the entire model by, for example, counting the number of cores and/or inconsistencies (optionally categorized by type or structure), as well as measuring culpability of individual constraints, i.e., the number of cores and/or inconsistencies in which the constraint is involved.



Detecting Inconsistencies in Declare Models
Small Example
Medium Example
Large Example
Choose File
No file chosen
OR paste constraints below:

Response(a,b)
ChainResponse(a,b)
NotResponse(a,b)
AtLeastOne(a)
Init(a)

Preparation
- [✓ Remove Redundancies]

Detection
- [✓ Classic Inconsistencies]
- [✓ Actual Issues]
- [✓ Potential Issues]

Measurement
- Core Culpability
- Inconsistency Culpability

Detection Completed! (2 cores and 4 inconsistencies)

```
[a] -> [NotResponse('a','b'), ChainResponse('a','b')]
[b] -> [NotResponse('a','b'), Init('a')]
```

Figure 1: Core & Inconsistency Detection User Interface

4. Usage and Maturity

To enable easy access to our inconsistency detection library, a basic version can be used directly in the browser[2] (cf. Figure 1). Additionally, the code can be obtained by cloning our Git repository[3] and executed locally. The latter also allows a direct integration into existing modeling or mining tools. Additionally, a screencast showcasing our application can be found here[4]. To demonstrate the feasibility of our approach, we evaluated our library using DECLARE models mined from real-life datasets[5]. For the considered datasets, we computed all cores and the corresponding inconsistencies. We ran our experiments on MacOS with an M2 chip and 16GB RAM. In Table 2 we provide an overview of the number of detected cores and inconsistencies, along with the run-times for the core and the inconsistency detection separately.

Table 2
Core and Inconsistency Detection Results

Model		Core Detection		Inconsistency Detection			
Event Log	Constraints	Cores	Time (ms)	Classic	Actual	Potential	Time (ms)
Sepsis	207	15	8	0	7736	62569	694
BPI 2017	305	10	38	0	51021	565795	4087
BPI 2019	52	5	9	0	5	0	31
BPI 2020	357	3554	2445	47	4025	247103	331517

[2]https://uni-ko.de/detection
[3]https://uni-ko.de/detection-git
[4]https://uni-ko.de/detection-screencast
[5]https://data.4tu.nl/search?search=bpi and https://data.4tu.nl/datasets/33632f3c-5c48-40cf-8d8f-2db57f5a6ce7/1

5. Conclusion

In this work, we present a comprehensive library for detecting and classifying inconsistencies in Declare models. Our approach identifies both classic (design-time) and potential (run-time) inconsistencies by detecting minimal inconsistency cores and exploring their activation paths. The current implementation is based on a predefined set of Declare templates. While this ensures compatibility and simplifies implementation, it restricts the analysis to models built using these templates. However, additional templates can simply be added by specifying their activation semantics and linking them to relevant inconsistency structures. As a next step, we aim to develop a dashboard that offers visualizations of inconsistencies and improved support for model exploration and diagnosis. Moreover, our detection algorithms will be integrated directly into declarative process modeling and rule mining tools. This will allow inconsistencies to be identified and resolved during model design or as part of the mining process, which, in turn, would improve model correctness and reduce the likelihood of run-time issues.

Acknowledgments

This paper was funded by the Deutsche Forschungsgemeinschaft (grant number DE 1983/9-3).

Declaration on Generative AI

During the preparation of this work, the authors used GPT-4o to check grammar and spelling.

References

[1] C. Di Ciccio, M. Montali, Declarative Process Specifications: Reasoning, Discovery, Monitoring, in: Process Mining Handbook, volume 448, Springer Int. Publishing, 2022, pp. 108–152.

[2] C. Corea, M. Deisen, P. Delfmann, Resolving Inconsistencies in Declarative Process Models based on Culpability Measurement, in: In Proceedings der 14. Internationalen Tagung der Wirtschaftsinformatik, Siegen, Germany, 2019.

[3] C. Corea, M. Thimm, Towards Handling Potential Issues in Business Rule Bases, Journal of Applied Logics 11 (2024) 565–592.

[4] S. Nagel, P. Delfmann, Identification, Abstraction and Classification of Inconsistency Structures in Declarative Process Models, in: ECIS 2023 Research Papers, 2023.

[5] S. Nagel, P. Delfmann, Exploring Cognitive Effects of Inconsistency Characteristics on Understanding Inconsistencies in Declarative Process Models, in: Proceedings of the 57th Hawaii International Conference on System Sciences (HICSS), 2024.

[6] S. Nagel, P. Delfmann, Investigating Inconsistency Understanding to Support Interactive Inconsistency Resolution in Declarative Process Models, in: ECIS 2022 Research-in-Progress Papers, 2022.

[7] S. Nagel, P. Delfmann, Interactive Resolution of Inconsistencies in Declarative Process Models, in: Proceedings der 19. Internationalen Tagung der Wirtschaftsinformatik (WI 2024), 2024.

[8] A. Ielo, G. Mazzotta, R. Peñaloza, F. Ricca, Enumerating Minimal Unsatisfiable Cores of LTLf formulas, arXiv preprint arXiv:2409.09485, 2024.

[9] M. Roveri, C. Di Ciccio, C. Di Francescomarino, C. Ghidini, Computing Unsatisfiable Cores for LTLf Specifications, Journal of Artificial Intelligence Research 80 (2024) 517–558.

[10] V. Schuppan, Extracting unsatisfiable cores for ltl via temporal resolution, Acta Informatica 53 (2016) 247–299.

[11] C. Di Ciccio, F. M. Maggi, M. Montali, J. Mendling, Resolving Inconsistencies and Redundancies in Declarative Process Models, Information Systems 64 (2017) 425–446.