Towards Declarative Process Execution: Functionalities and Architecture

Johannes Loebbecke[1,*], Dominik Voigt[1], Juergen Mangler[1] and Stefanie Rinderle-Ma[1]

[1]*Technical University of Munich, TUM School of Computation, Information, and Technology, Garching, Germany*

Abstract

Flexible, trusted, and compliant service orchestration, where executions adhere to constraints imposed by legal or business requirements, has become paramount for organizations. Using Business Process Management Systems (BPMS) to orchestrate services that implement human and computerized tasks can increase trust, enable efficient communication, and lead to flexible and transparent service orchestration. Existing BPMS generally use the imperative paradigm, which is suited for orchestrating strict processes with few deviations, while declarative models have been proposed for orchestrating processes with many and deeply diverging paths. However, no existing process execution engine supports the declarative paradigm while preserving the advantages of BPMS. We propose a list of functional requirements of a declarative engine that can be achieved through loosely coupled integration with existing BPMS as a path towards declarative process execution.

Keywords

Process Orchestration, Declarative Process Modeling, Declarative Process Execution, Process Compliance

1. Introduction

Compliant business process execution has become a central task for organizations regardless of their domain [1]. In particular, frequent changes in regulations, market conditions, and customer demands [2] require trusted and flexible service orchestration to achieve compliance while remaining competitive. Organizations commonly orchestrate their value-generating services through business process management systems (BPMS). BPMS enable trusted and traceable service orchestration through process design and modeling tools integrated with simulation and execution engines, as well as various visualization and optimization tools. Existing BPMS (e.g, Camunda, Tim Solutions, Aristaflow)[1] mainly support the imperative paradigm where the service execution order is given by an imperative process model which explicitly represents all paths that the process can take. While imperative process models are well-suited for modeling and enacting strict processes with few deviations (e.g., a manufacturing process), they are unsuited for processes with many variations (e.g., treatment guidelines in a hospital) [3, 4]. In comparison, following a declarative paradigm, the constraints to which the process execution must adhere, implicitly define the possible execution paths. This ensures that the rules governing process execution are immediately visible from the model, and highly dynamic processes can be concisely described. Furthermore, if declarative rules can be extracted, e.g., from a regulatory document, compliant execution is ensured, without requiring a mapping between model elements/event log and rules. However, while various declarative modeling techniques have been proposed [5, 6, 7] and researchers argue for declarative frames for AI augmented BPMS [8, 9], there is a lack of research on declarative process execution [10], with no available BPMS supporting execution of both imperative and declarative models. Here, we differentiate between process simulation and process execution, where process simulation typically involves using an automata to generate potential execution traces for a given model. In contrast, process execution involves executing services by using a process execution engine to call endpoints associated with activities according to the semantics defined in the process model.

Cooperative Information Systems - Early Research Achievement and Demos, 2025

*Corresponding author. This author contributed more than 50 percent.

johannes.loebbecke@tum.de (J. Loebbecke); dominik.voigt@tum.de (D. Voigt); juergen.mangler@tum.de (J. Mangler); stefanie.rinderle-ma@tum.de (S. Rinderle-Ma)

[1]https://camunda.com, https://tim-solutions.de/, https://artistaflow.com, last access: 2025-07-10

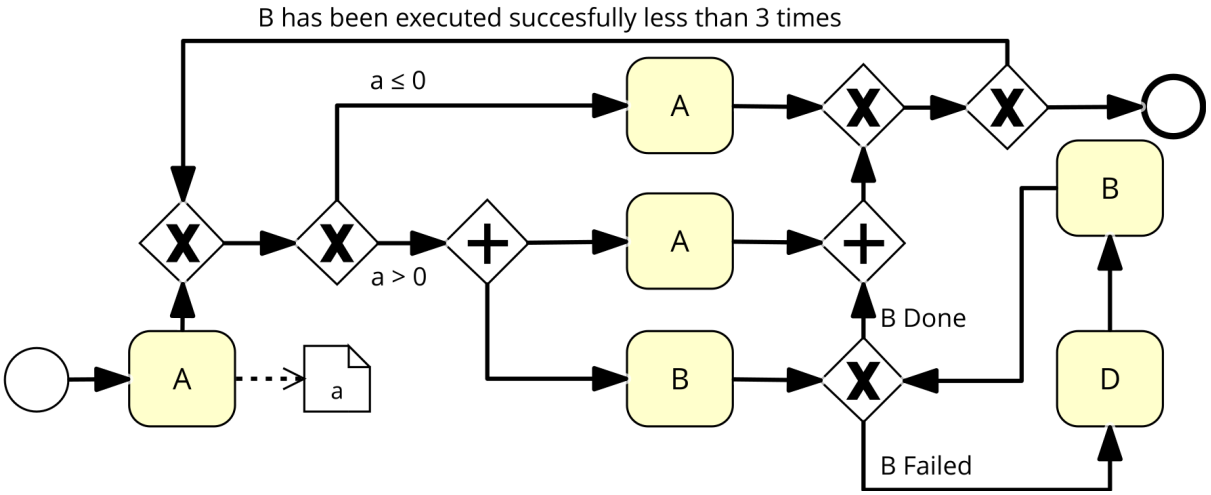


Figure 1: Imperative Version of the Running Example Process

Consider the process described by the following rules as a running example:
R1 The process starts with Activity **A**.
R2 Every time **D** is executed, it is directly followed by **B**.
R3 If **B** has been successfully executed at least 3 times, the process ends.
R4 If an execution of **B** fails, it is followed by an execution of **D**.
R5 If dataobject $a \leq 0$ and neither **A** nor **B** is running, then **B** is executed.
R6 Otherwise, if $a > 0$ and neither **A** nor **B** are running, **A** and **B** are executed in any order.

The imperative version of this process is illustrated in Fig. 1. While R1 and R2 are suited for representation in an imperative process, since they define an initial activity and a strict control flow relationship, R3 → R6 are more suited to declarative processes, as they lead to overly complex models for representing simple rules. In particular, in an imperative model, rules like R4, which act as repairing rules that reset the process state through an execution of **D** after a failed execution of **B**, require inserting a subprocess or loop after every occurrence of **B**, of which there may be multiple in a larger process. In contrast, the declarative model requires only a single rule. However, it is challenging to model these rules at execution-level using existing declarative notations such as DCR [5] and Declare [6] as they lack the required expressivity (cf. Sect. 2). To identify required artifacts for BPMS-integrated, declarative process execution, we propose 8 functional requirements of a declarative execution engine in Sect. 2 and describe how they can be achieved through a BPMS-integrated architecture in Sect. 3. In Sect. 4, we discuss related work, focusing on the default deny paradigm compared with the default allow paradigm and the recently proposed AI-augmented Business Process Management Systems (ABPMSs) [8] concept.

2. Declarative Process Execution Functionalities (DPEF)

Based on related work [5, 6, 10, 11], principles of declarative specification languages, and features of existing BPMS, we propose a set of declarative process execution functionalities DPEF 1→8 where DPEF 1→4 are architectural, DPEF 5→7 notational, and DPEF 8 logging functionalities.
DPEF 1: Declarative modeling notation agnostic: While several declarative modeling notations have been proposed, there is no agreement on which notation is preferred and all prominent notations are on a higher abstraction level than what is required for execution (cf. DPEF5/6). However, abstract notations can function as templates for execution-level notations to enable better understanding and communication, by abstracting from the underlying logic [5]. Accordingly, a declarative execution engine should be notation agnostic, enabling execution of abstract declarative processes, by mapping higher level notations to the execution-level notation.

DPEF 2: Hybrid processes and Hot swapping: Hybrid processes are generally understood as models that combine elements from both imperative and declarative notations. Declarative process execution should enable such processes, as purely declarative models can be challenging to understand [12] and many processes consist of both strict and flexible parts, such as R1 and R2 compared to R3→ R6 of the running example. The most common hybrid process type is the hierarchical hybrid process, where sub-processes can be both declarative and imperative [11]. We further argue that, in addition to hierarchical hybrid process execution, it is paramount to enable a form of "Hot swapping", where for each instance the execution engine can be swapped online, as long as a fitting model in the corresponding paradigm is available. In this line, integrating a declarative execution engine into a existing BPMS also enables instantiating imperative models through declarative execution and simulation. Adding activities to an imperative model, whenever the declarative execution engine calls said activity, results in a sequential model that describes a single execution path while considering parallelism, without any exclusive choices. A process modeler can online review the generated imperative model to identify incorrect specifications or store it as a desired execution path for future use.

DPEF 3: Extended Lifecycle Management: A major advantage of BPMS we explored in [13, 14] is the ability to start, stop, and evolve running instances. Consider an instance that runs into an error state, e.g., due to incorrect specifications or unforeseen changes in the process environment. In this case, the ability to stop the instance, adapt the model and continue the execution without losing the state of the instance is paramount. This feature is also essential for declarative processes which are generally regarded as more challenging to correctly specify [15, 16], so process modelers need to be able to stop the execution and adapt the declarative specifications when undesired execution paths emerge.

DPEF 4: Multiple conflict resolution protocols: In rule-based systems, conflict resolution refers to the problem of deciding on which rules to fire, in case of multiple matching rules in a single state [17]. In the context of declarative process execution, this maps to deciding on which activities to execute in case of multiple matched activities after rule matching is completed. Consider a set of rules R, which define in a state s that activities A, B, and C can be executed next, however one rule enforces that A and B can not be executed in parallel. Accordingly, either A and C or B and C can be executed next. In this case multiple conflict resolution techniques can be applied, such as executing the set which was matched first or executing the set which is preferred according to some metric such as resource utilization or execution costs. Declarative process execution should enable various potential conflict resolution techniques that can be dynamically modified.

DPEF 5: Execution level expressivity: While existing declarative notations have been extended to model various process perspectives [18, 19], declarative execution requires several additional artifacts that have not been considered in a unified notation such as sub-processes, endpoint addresses, activity lifecycles, and corresponding data processing points. To illustrate the required execution level expressivity, we use the activity lifecycle we utilize for the Cloud Process Execution Engine (CPEE) in [13], which is depicted in Fig. 2. While the exact terminology varies between BPMS[2] a complete declarative execution engine needs to consider the *calling*, *receiving*, *manipulating*, and *done* lifecycle states, as well as the *prepare*, *rescue*, *update*, and *finalize* data accesses, which are arbitrary code pieces. For a more extensive discussion of the required expressivity we refer to [13].

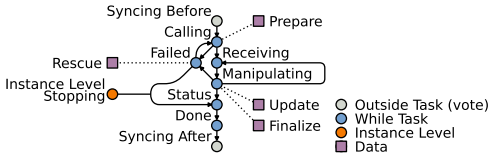


Figure 2: Activity Lifecycle from [13]

DPEF 6: Explicit final state: A central question for declarative process execution is when a process should reach its final state. DCR [5] assumes that the final state is reached when no more activities can execute, which leads to overly complicated processes that can end earlier than intended when considering execution-level expressivity. For example, if required resources are temporarily unavailable but will become available soon, the process would terminate prematurely unless potential delays are explicitly defined in the rules. Declare [6] instead defines specific activities as *last occurring*, which

[2] cf. https://docs.camunda.io/docs/components/zeebe/technical-concepts/process-lifecycles/, last access 2025-08-22

hinders expressivity, since data values, lifecycle states, and timestamps can not lead to a final state. Consider, for example, the end state that should be reached through R3. In Declare, R3 cannot be modeled since standard Declare lacks activity lifecycles, and even when adding this notion, it is unsuitable since both A and B could both be the *last occuring* activity due to the previous parallel relationship. Meanwhile, in DCR, R3 can be modeled, but requires creating individual nodes for each activity lifecycle and waiting nodes, such that correct executions of B exclude all other nodes and incomplete executions do not end the process prematurely. Defining instead an explicit final state, which can follow from an arbitrary condition, ensures that a modeler can concisely model R3.

DPEF 7: Order Independence: A core concept of declarative modeling, as defined for declarative specification languages such as Alloy[3], is order independence, i.e., the order of declaration does not change the model's behavior. For declarative process execution, this implies that the order in which rules are added to the rule set (i.e., the process) should not enforce a specific execution order. We do not extend this notion to performance metrics, defining that the rule order can impact performance metrics, such as execution time, without violating order independence. Here, we identify a conflict between potential conflict resolution protocols and strict order independence. Consider a simple conflict resolution protocol where the first matching rule is preferred over other matching rules. This protocol trivially violates order independence but is easy to predict, enables simpler modeling, and likely improves performance, as not all rules need to be processed in every state.

DPEF 8: XES logging: Over the last two decades, a vibrant academic and industrial community has developed tools and techniques for process mining, where event logs are used to discover process models and verify various properties [20]. To utilize these advancements, a declarative execution engine must write detailed execution logs that are compatible with the XES[4] standard, ideally with additional information highlighting declarative specifics, such as the applied conflict resolution technique.

3. An Architecture for BPMS-integrated Declarative Execution

We discuss a BPMS-integrated declarative process execution architecture as illustrated in Figure 3, where all new components are highlighted in purple, while existing modules are highlighted in blue.

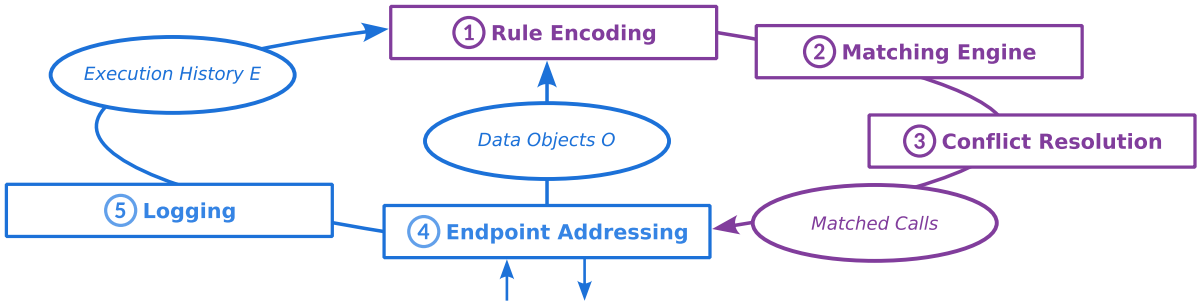


Figure 3: BPMS-integrated Declarative Process Execution. Blue modules exist, while purple modules are new

For this solution, the rules that declaratively define the process need to be encoded in a highly expressive, directly integrable, execution-level notation (1), which is processed using some pattern matching engine (2) to create a set of matched activity calls that can be executed according to the rule set in the next state. Several engines are viable, but we suggest development based on the Rete algorithm [17] due to its reliability and performance, which have been verified in both academia and industry. At this point, conflict resolution (3) is used to identify which matched calls should be executed. We propose five conflict resolution strategies, with different advantages and disadvantages regarding manual effort, understandability, performance, preserving order independence and logical mapping: (1) Everything parallel, (2) Order dependent priority (e.g., first/last matching rule), (3) Manual priority,

[3] https://alloytools.org/, last access 2025-08-22
[4] https://www.xes-standard.org/, last access 2025-08-22

(4) Rule *difficulty* dependent priority (e.g., more specific rules have priority), (5) Metric-based priority (rule which performs better according to some metric such as resource utilization has higher priority). Additionally, existing rule-based systems[5] utilize *Refraction* (each time a rule is matched, it is removed from the next set of rules) and *Recency* (Rules that did not match in previous states are preferred). While the matching engine and conflict resolution protocol are different modules in Fig. 3, resolution protocols are often integrated into engines to improve performance [17]. Conclusively, the artifacts required for integrated declarative execution are: a **highly expressive, directly integrable, execution-level notation** for modeling and encoding declarative rules which can be represented using templates in higher level notations a **performant, traceable and controllable matching engine** to identify which services should be called in every state and **implemented and rigorously evaluated conflict resolution techniques** all integrated into a existing BPMS with low coupling to enable the DPEF. Designing the notation to be directly integrable, i.e., containing the execution history and data objects created by the logging (5) and endpoint addressing modules (4), aims at enabling DPEF2 and DPEF3. The notation needs to be highly expressive to support execution-level expressivity (DPEF5) and explicit final states (DPEF6). Due to its higher expressivity, templates from higher abstraction languages can be mapped to the execution-level notation to enable DPEF1. The matching engine should be traceable and controllable to further ensure DPEF2 and DPEF3, while multiple multiple conflict resolution protocols are analyzed (DPEF4) with the goal of preserving order independence (DPEF7). Finally, since the logging functionality is reused from the original BPMS, DPEF8 is achieved.

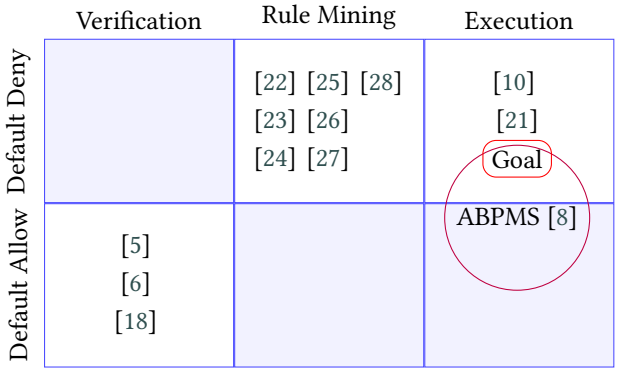
4. Related Work

We analyze whether related contributions follow the *default deny* compared to the *default allow paradigm*, which can be seen in Fig. 4 and provide an overview of prominent declarative notations while highlighting approaches focusing on execution.

Default Allow vs Default Deny: Utilizing terminology from network security [6], a major question for declarative execution is whether to follow the default deny paradigm, where any behavior which is not explicitly defined in the rules is forbidden, compared to the default allow paradigm, where any behavior which is not forbidden by the rules is allowed. For this analysis, we do not differentiate between monitoring, verification, and conformance checking, as monitoring can be understood as the runtime version of conformance checking [21], while declarative conformance checking is exactly verification of an event log against a set of rules.

While [16] argues for a default allow approach due to its higher flexibility, example rules generally follow the default deny paradigm [4, 10, 21] and default deny is regarded as easier to manage and more secure in network security. We argue that default allow is more suited for verification, since default deny verification implies that any observed behaviour that is not defined in the rules is a violation. Meanwhile, default deny is more suited for execution, since it is unrealistic to consider all potential behaviour that should not be allowed when considering execution level expressivity. An interesting question is whether to see AI augmented Business Process Management Systems(ABPMSs) [8] as default deny or default allow systems. While the proposed idea of a frame that inhibits the AI execution behaviour follows the declarative, default

Figure 4: Related declarative process contributions



[5] https://www.ibm.com/docs/en/odmoc?topic=mode-reteplus-algorithm, last access 2025-08-22
[6] https://community.jisc.ac.uk/library/advisory-services/modes-firewall-operation, last access 2025-08-22

allow paradigm, agent instructions follow the imperative, default deny paradigm. Accordingly, ABPMS can be seen as three layered systems (cf. Fig. 5), where the highest layer (1) imposes rules in a declarative, default allow manner, the middle layer contains imperative, default deny procedures (2), while the lowest layer contains the default allow AI Agents (3). This observation highlights the importance of DPEF 1/2 and seems to align with [9], who use a hybrid process representation containing Declare Rules and Petri Net models to model ABPMS.

Related Work: Most related contributions focus on Declare [6] and/or DCR graphs [5]. We present an overview, while focusing on declarative process execution. For their semantics, see [4, 5, 6].

Declare: Declare, initially proposed by Pesic [6] and motivated in [16], is a pattern-based notation with a formal mapping in LTL, for describing the rules governing a business process. Through a graphical notation for each pattern, it abstracts from the logic notation to enable modeling constraints without a background in formal languages. The expressivity of Declare has been extended in several contributions [4], most notably to support timed and data constraints using MP-Declare [18]. While several Declare rule mining approaches [27, 28] exist, there are no actively maintained tools for modeling and executing Declare Models [4, 10]. In general, Declare is suited for representing process behaviour mined from logs, but unsuited for enabling BPMS-integrated declarative execution due to its lack of expressivity and its default allow paradigm leading to highly complex processes [6].

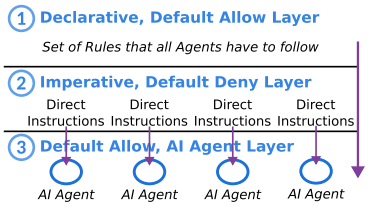


Figure 5: ABPMS Paradigm

DCR graphs: DCR graphs [5] express execution semantics through nodes and edges, where each node represents events, while edges describe the relations between events. Each event has a marking which indicates whether the event is currently *relevant* (i.e., can be executed) and whether it is currently *required* (has to be executed before the process can stop). Since unincluded events cannot be executed [4], DCR graphs follow the default deny paradigm. DCR has also been extended to support various time and data conditions [19], and several approaches discover DCR graphs from event logs [25, 26]. Furthermore, DCR is supported through a proprietary process engine for the modeling and simulation of DCR graphs [12] maintained by dcrsolutions[7]. However, the engine is not notation-agnostic, lacks extended lifecycle management features, and execution-level details are closed source.

Declarative Process Execution: Awad et al. [10] present a default deny based execution engine for DCR graphs and BPMN models based on Complex Event Processing (CEP) called BEST. CEP expressions, with a mapping to models/graphs, are used to process events arriving through event streams. BEST supports declarative execution of hybrid, imperative, and declarative models (DPEF1, DPEF2) as well as typical change operations through filter conditions (DPEF3). Furthermore, the notation supports explicit end states (DPEF6) and the custom log format can be trivially transformed into XES format (DPEF8). They do not consider conflict resolution (DPEF4), the notation has to be extended to support execution level expressivity (DPEF5), and order independence is not discussed (DPEF7). In general, BEST seems suited for orchestrating micro-services communicating via a centralized event bus/broker, such as an MQTT-broker, while we aim for more general service orchestration, enabling hot swapping and consider different conflict resolution protocols. Alman et al. [21] present formal execution and monitoring semantics for hybrid processes based on data-aware Petri nets. They further analyze their proposed semantics in [11] regarding 16 requirements for modeling using a multi-modal paradigm, which considers hybrid processes, collaborative processes, and multi-instance constrained processes. While suitable for modeling, simulation, and monitoring, the formal semantics are challenging to apply to execution due to implicit assumptions such as perfectly available, consistent, and partition-tolerant data (cf. CAP Theorem) as well as simultaneous execution across distributed nodes.

[7] https://dcrsolutions.net/, last access 2025-08-22

5. Conclusion

Summary: Declarative process notations enable modeling and mining processes with many and deeply diverging paths, but there is a lack of research on declarative process execution. We present a pathway towards BPMS-integrated declarative execution via a set of proposed functionalities and illustrate the resulting architecture. We further discuss whether existing work, ABPMS [8] and our proposed approach follow the default deny compared to the default allow paradigm.
Limitations and Future Work: We present a research direction that requires the development of several artifacts which have to be rigorously evaluated. First an execution-level declarative notation has to be designed and evaluated regarding its usability and expressivity. Second, a traceable and controllable matching engine which can be directly integrated into existing BPMS needs to be developed and evaluated regarding performance and reliability. Finally, we only provide a short textual description of different conflict resolution techniques, which will be formalized, implemented and analyzed in the future. Particularly their effects on understandability, technology acceptance level, performance, rule inconsistencies [29] and the deontic modalities of obligations, permissions and prohibitions [30] need to be analyzed. We are currently actively designing and evaluating a DMN-based, execution-level notation with example processes and simulated execution available on GitHub [8].

Acknowledgments

This study was (partly) funded as part of the TRPro project by the German Research Association (Deutsche Forschungsgemeinschaft) as "Sachbeihilfe" under the project number 514769482 and

Declaration on Generative AI

During the preparation of this work, the author(s) used Grammarly to grammar and spelling check. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. Hashmi, G. Governatori, H. Lam, M. T. Wynn, Are we done with business process compliance: state of the art and challenges ahead, Knowl. Inf. Syst. 57 (2018) 79–133. doi:10.1007/S10115-017-1142-1.
[2] S. Sadiq, G. Governatori, K. Namiri, Modeling control objectives for business process compliance, in: Business Process Management, 2007, pp. 149–164.
[3] A. Abbad-Andaloussi, A. Burattin, T. Slaats, E. Kindler, B. Weber, Complexity in declarative process models: Metrics and multi-modal assessment of cognitive load, Expert Systems with Applications 233 (2023) 120924.
[4] T. Slaats, Declarative and hybrid process discovery: Recent advances and open challenges, Journal on Data Semantics 9 (2020) 3–20.
[5] T. T. Hildebrandt, R. R. Mukkamala, Declarative event-based workflow as distributed dynamic condition response graphs, arXiv preprint arXiv:1110.4161 (2011).
[6] M. Pesic, H. Schonenberg, W. M. P. van der Aalst, DECLARE: full support for loosely-structured processes, in: Enterprise Distributed Object Computing Conference, 2007, pp. 287–300.
[7] S. Goedertier, J. Vanthienen, Designing compliant business processes with obligations and permissions, in: BPM Workshops, 2006, pp. 5–14.
[8] M. Dumas, F. Fournier, L. Limonad, A. Marrella, M. Montali, J.-R. Rehse, R. Accorsi, D. Calvanese, G. De Giacomo, D. Fahland, et al., Ai-augmented business process management systems: a research manifesto, ACM Transactions on Management Information Systems 14 (2023) 1–19.

[8] https://github.com/JohannesLbck/PeDMN_Prototype, last access 2025-08-22

[9] G. Acitelli, A. Alman, F. M. Maggi, A. Marrella, Achieving framed autonomy in ai-augmented business process management systems through automated planning, Information Systems (2025) 102573.
[10] A. Awad, F. Awaysheh, H. A. López, Best: A unified business process enactment via streams and tables for service computing, arXiv preprint arXiv:2501.14848 (2025).
[11] A. Alman, F. M. Maggi, S. Rinderle-Ma, A. Rivkin, K. Winter, Towards a multi-model paradigm for business process management, in: International Conference on Advanced Information Systems Engineering, Springer, 2024, pp. 178–194.
[12] S. Debois, T. T. Hildebrandt, M. Marquard, T. Slaats, The dcr graphs process portal., in: BPM (Demos), 2016, pp. 7–11.
[13] J. Mangler, S. Rinderle-Ma, Cloud process execution engine: architecture and interfaces, arXiv preprint arXiv:2208.12214 (2022).
[14] S. Rinderle-Ma, M. Reichert, Advanced migration strategies for adaptive process management systems, in: Conference on Commerce and Enterprise Computing, IEEE, 2010, pp. 56–63.
[15] S. Zugal, J. Pinggera, B. Weber, The impact of testcases on the maintainability of declarative process models, in: International Workshop on Business Process Modeling, Development and Support, Springer, 2011, pp. 163–177.
[16] M. Pesic, Constraint-based workflow management systems: shifting control to users (2008).
[17] C. L. Forgy, Rete: A fast algorithm for the many pattern/many object pattern match problem, in: Readings in artificial intelligence and databases, Elsevier, 1989, pp. 547–559.
[18] A. Burattin, F. M. Maggi, A. Sperduti, Conformance checking based on multi-perspective declarative process models, Expert Syst. Appl. 65 (2016) 194–211.
[19] T. T. Hildebrandt, H. Normann, M. Marquard, S. Debois, T. Slaats, Decision modelling in timed dynamic condition response graphs with data, in: International Conference on Business Process Management, Springer, 2021, pp. 362–374.
[20] W. Van Der Aalst, A. Adriansyah, A. K. A. De Medeiros, F. Arcieri, T. Baier, T. Blickle, J. C. Bose, P. Van Den Brand, R. Brandtjen, J. Buijs, et al., Process mining manifesto, in: International conference on business process management, Springer, 2011, pp. 169–194.
[21] A. Alman, F. M. Maggi, M. Montali, F. Patrizi, A. Rivkin, A framework for modeling, executing, and monitoring hybrid multi-process specifications with bounded global–local memory, Information Systems 119 (2023) 102271.
[22] F. M. Maggi, A. J. Mooij, W. M. van der Aalst, User-guided discovery of declarative process models, in: Symposium on computational intelligence and data mining (CIDM), IEEE, 2011, pp. 192–199.
[23] F. M. Maggi, C. Di Ciccio, C. Di Francescomarino, T. Kala, Parallel algorithms for the automated discovery of declarative process models, Information Systems 74 (2018) 136–152.
[24] A. Alman, C. Di Ciccio, D. Haas, F. M. Maggi, A. Nolte, Rule mining with rum, in: 2020 2nd International Conference on Process Mining (ICPM), IEEE, 2020, pp. 121–128.
[25] S. Debois, T. T. Hildebrandt, P. H. Laursen, K. R. Ulrik, Declarative process mining for dcr graphs, in: Proceedings of the Symposium on Applied Computing, 2017, pp. 759–764.
[26] V. Nekrasaite, A. T. Parli, C. O. Back, T. Slaats, Discovering responsibilities with dynamic condition response graphs, in: International Conference on Advanced Information Systems Engineering, Springer, 2019, pp. 595–610.
[27] F. M. Maggi, C. Di Ciccio, C. Di Francescomarino, T. Kala, Parallel algorithms for the automated discovery of declarative process models, Information Systems 74 (2018) 136–152.
[28] C. Di Ciccio, F. M. Maggi, M. Montali, J. Mendling, On the relevance of a business constraint to an event log, Information Systems 78 (2018) 144–161.
[29] S. Nagel, P. Delfmann, Identification, abstraction and classification of inconsistency structures in declarative process models, in: European Conference on Information Systems , ECIS, 2023. URL: https://aisel.aisnet.org/ecis2023_rp/428.
[30] M. Hashmi, G. Governatori, M. T. Wynn, Normative requirements for regulatory compliance: An abstract formal framework, Inf. Syst. Frontiers 18 (2016) 429–455. doi:10.1007/S10796-015-9558-1.