Responsible Composition and Optimization of Integration Processes under Correctness Preserving Guarantees

Daniel Ritter[a], Fredrik Nordvall Forsberg[b], Stefanie Rinderle-Ma[c]

[a]*SAP, Walldorf (Baden), Germany*
[b]*Department of Computer and Information Sciences, University of Strathclyde, Glasogw, Scotland*
[c]*School of Computation, Information and Technology, Technische Universität München, Garching, Germany*

Abstract

Enterprise Application Integration deals with the problem of connecting heterogeneous applications, and is the centerpiece of current on-premise, cloud and device integration scenarios. For integration scenarios, structurally correct composition of patterns into processes and improvements of integration processes are crucial. In order to achieve this, we formalize compositions of integration patterns based on their characteristics, and describe optimization strategies that help to reduce the model complexity, and improve the process execution efficiency using design time techniques. Using the formalism of timed DB-nets — a refinement of Petri nets — we model integration logic features such as control- and data flow, transactional data storage, compensation and exception handling, and time aspects that are present in reoccurring solutions as separate integration patterns. We then propose a realization of optimization strategies using graph rewriting, and prove that the optimizations we consider preserve both structural and functional correctness. We evaluate the improvements on a real-world catalog of pattern compositions, containing over 900 integration processes, and illustrate the correctness properties in case studies based on two of these processes.

Keywords: Enterprise application integration, enterprise integration patterns, optimization strategies, pattern compositions, petri nets, responsible programming, trustworthy application integration

1. Introduction

In a highly digitized and connected world, in which enterprises get more and more intertwined with each other, the integration of applications scattered across on-premises, cloud and devices is crucial for enabling innovation, improved productivity, and more accessible information [1]. This is facilitated by process technology based on integration building blocks called integration patterns [2, 3, 4, 5]. The composition of these patterns into integration processes can result in complex models that are often vendor-specific, informal and ad-hoc [5]; optimizing such integration processes is often desirable, but hard. In most cases complex process control flows are further complicated by data flow, transactional data storage, compensation, exception handling, and time aspects [6].

In previous work [7], we found that already simple integration processes show improvement potential, e.g., when considering data dependencies that allow for (sub-)process parallelization. In order to consider such improvements, it is crucial to also consider data flow in the model, but approaches for verification and formal analysis of "realistic data-aware" integration processes are currently missing, as recent surveys on event data [8, 9], workflow management [10], and in particular application integration [5] report. Such approaches are needed in order to formally prove the structural and functional correctness of compositions of patterns and their optimizations, which in turn is needed to enable a responsible development of integration scenarios where integration processes behave as intended.

To enable such approaches for the process modeller, we propose a *responsible composition and optimization (ReCO) process* for patterns, that covers the following objectives: (i) inherently correct structural process representation, (ii) means for representing and proving functional process execution correctness, (iii) semantic integration pattern aspects of control and data flow, transactional data storage, compensation, exception handling, and time, (iv) automatic identification and application of optimizations, and (v) correctness-preserving process changes. We argue that existing approaches do not fully support responsible integration pattern composition and optimization with correctness preserving guarantees (cf. related work in Sec. 9).

Figure 1 visualises the challenges that need to be overcome to enable such a responsible composition and optimization process achieving objectives (i)–(v). Integration developers and experts provide semantically meaningful pattern realizations in expressive specialised languages such as timed db-nets [6] (**1**). However, these languages are cumbersome to use for process modelers, and makes an automatic identification of improvements difficult. Process modelers on the other hand like higher level languages and notations to specify processes as a composition of integration patterns [3, 4, 5, 11] (**2**). While improvements are conveniently defined on the higher level mod-

Email addresses: daniel.ritter@sap.com (Daniel Ritter), fredrik.nordvall-forsberg@strath.ac.uk (Fredrik Nordvall Forsberg), stefanie.rinderle-ma@tum.de (Stefanie Rinderle-Ma)

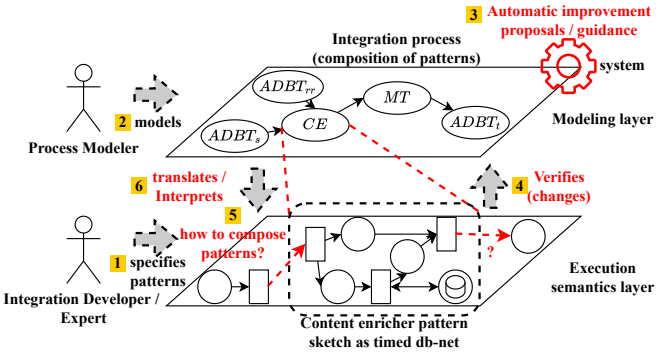


Figure 1: End-to-end perspective from integration process modeling to verifiable execution semantics and automatic, correctness-preserving improvements (current gaps or missing aspects in red color).

eling layer for automatic changes to processes [7] (3) in manual modeling and automatic improvement cases, it is currently not possible to verify that the improved process has the same behaviour as the original process with respect to the execution semantics of the composed integration patterns (4). This is because the composition of pattern definitions is currently not formally specified (5), and the modeling and execution semantics layers are not connected (6).

This work aims to fill these gaps, based on the following research questions that guide the design and development of a responsible composition and optimization process:

Q1 How can the user be supported and guided during pattern composition and process modeling?

Q2 When are pattern compositions correct?

Q3 How to responsibly determine and apply optimizations?

Question Q1 is related to objective (i), Q2 to objectives (ii)–(iii), and Q3 to (iv)–(v). In the conference version of this paper [7], we provided the foundations for Q1 (and partially Q3). Pattern compositions were represented as typed pattern graphs, based on pattern characteristics and contracts, which inherently guarantee structurally correct compositions, and thus guide and support the user. We could not use existing languages / notations like BPMN or EIP icon notation [2], since they either structurally or semantically do not provide the required notions of data flow, persistence (e.g., [3, 4, 11, 12]) and boundaries for checking structural correctness (e.g., [5]). Furthermore, the graph-based representation of integration patterns allows for the realization of optimizations as graph rewriting rules. Our evaluation showed that effective improvements could be identified and applied to real-world integration processes, while structural correctness was preserved. However, functional correctness was not considered, meaning that process changes might not be *responsible* (cf. objectives (ii)–(iii), (v)).

In this work, we extend our user-facing and structural correctness guaranteeing graph-based representation with an execution semantics using timed DB-nets [6]. To support the same notion of correctness based on pattern contracts as in [7], we define a new notion of *open* timed db-nets that are capable of representing the data exchange between patterns. We then show how they can be composed, and specify their execution semantics. By interpreting pattern compositions in graph representation as compositions of open timed db-nets, and by proving that the translation results in structurally correct and semantically well-behaved nets, we can answer Question Q2. All in all, this makes automatic optimization of integration processes feasible, but now also taking functional correctness into account, thus answering question Q3 fully. Hence this enables the study of ReCO for the first time.

Methodology. We follow the principles of design science research methodology by Peffers et al. [13] to answer the research questions above: *"Activity 1: Problem identification and motivation"* is based on a literature review and the assessment of vendor driven solutions [5], as well as quantitative analysis of integration pattern characteristics of EAI building blocks (existing catalogs of 166 integration patterns) and process improvements [7], resulting in requirements to a suitable formalism. We then address *"Activity 2: Define the objectives for a solution"* by formulating objectives (i)–(v). For *"Activity 3: Design and development"*, we create several artifacts/contributions to answer questions Q1–Q3 and realize objectives (i)–(v):

(a) a specification of an extensible structural correctness enforcing representation that allows for efficient application of improvements,

(b) an extension of the definition of the formalism of execution semantics by inter-pattern data exchange analysis capabilities based on open timed db-nets (cf. 4 , 5),

(c) an interpretation procedure of graph representation as open timed db-nets (cf. 6), and

(d) optimization realizations on the graph representation leveraging the interpretation to prove their correctness (cf. 3).

Outline. We introduce the ReCO process in Sec. 2, together with an integration process modeling example. In Sec. 3, we analyze recurring integration pattern characteristics, which are relevant for developing our formalisms, and collect optimization strategies for integration processes. We also identify eight requirements for formalizing integration pattern compositions. In Sec. 4, we describe integration pattern graphs, and use them to specify pattern compositions with inherent structural correctness. We also give an abstract cost model which can be used to determine if an optimization is an improvement or not. Section 5 extends the timed DB-net formalism [6] to open nets to introduce compositional aspects, and uses this extended formalism to capture the dynamics of integration patterns — that is, how data flows through the system. We also briefly describe an implementation of simulation of timed DB-nets as an extension of CPN Tools. In Sec. 6, we combine the two formalisms by showing how integration pattern graphs can be interpreted as open timed DB-nets. Next in Sec. 7 we realize optimization strategies as rewrite rules for integration pattern graphs, and show that these optimisations preserve the functional correctness of patterns when interpreted as timed DB-nets. In Sec. 8, we evaluate the

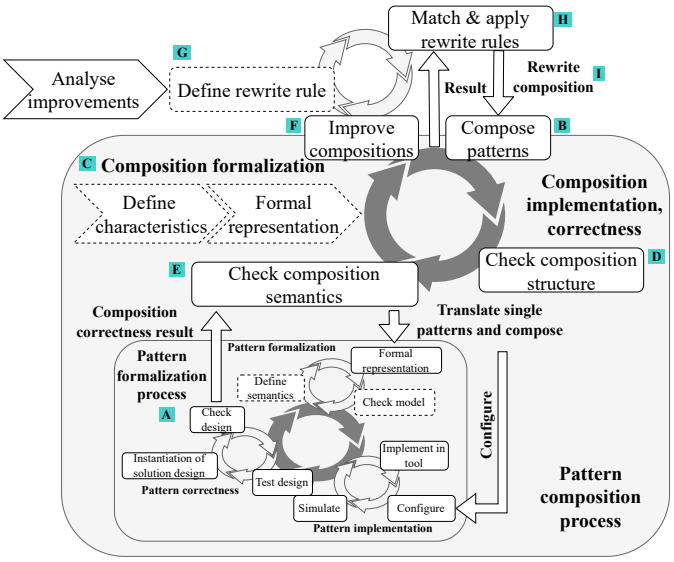


Figure 2: Responsible pattern composition and optimization process (white colored boxes denote the main steps, grey colored box shows the *pattern formalization process* from [6], to which we bridge through translation of single patterns and compositions)

optimizations on a large body of integration processes, and apply the ReCO process to two integration processes. We discuss related work in Sec. 9 and conclude in Sec. 10.

History of this paper. This paper extends our previous conference paper [6] with several new contributions. Firstly, this paper is structured along the ReCO process, whereas the conference version was not. Secondly, the dynamic semantics in the form of open timed DB-nets and their implementation as an extension of CPN Tools (Sec. 5), and the interpretation of integration pattern graphs into it (Sec. 6) are new, together with the proof that the optimizations proposed preserve the functional behaviour of the corresponding timed DB-nets (Sec. 7.7). The case studies of the ReCO process in Sec. 8.2 are also new in this version.

2. Responsible Composition and Optimization Process for Patterns

To reason about responsible pattern composition and optimization, one first needs to consider formalizations of *patterns*, *compositions*, and *composition improvements*. In this section, we introduce a pattern composition and optimization process that covers all of these topics, and we describe the process modeling in more detail by an integration process example. Finally, we specify the problem of responsible pattern composition and optimization.

2.1. Responsible Composition and Optimization Process

Our approach to a responsible pattern composition and optimization process is described in Fig. 2. We briefly discuss the main concepts involved.

Patterns The *pattern formalization process* (cf. **A** in Fig. 2) is specified in [6]. For ReCO, patterns are provided as modeling building blocks by the integration platform. Integration experts and platform developers start developing patterns by defining their semantics in formal representation that allows to formally analyze the pattern models. In addition, the realization of a pattern can be configured and simulated as well as tested.

Composition In ReCO, a process is modeled and configured by a user/modeler in a process modeling tool by composing (cf. **B** *compose patterns*) existing, single, semantically correct, integration patterns, from which the model is then translated into a formal representation by the integration platform (cf. **C**). With a thorough understanding of pattern characteristics within a process, composition can be formalized with inherent structural correctness (cf. **D** *check composition structure*). Semantic correctness of the whole composition of patterns can be checked through a notion of composition added to the pattern formalization (cf. **E** *check composition semantics*).

Improvements The formal foundation of structurally and semantically correct processes allows for proposed improvements that preserve the processes' correctness (cf. **F** *improve compositions*). For ReCO, the improvements could be defined by integration experts, but also users in a formalism provided by the integration platform that allows for process rewriting suitable to the underlying formalisms (cf. **G** *define rewrite rule*). The improvements are automatically matched (cf. **H** *match & apply rewrite rules*), and if applicable, it leads to the application of an improvement (cf. **I** *rewrite composition*).

To summarise, our vision is that integration experts and platform developers specify patterns as building blocks of an integration platform (cf. **A**), where the aspect of correct composition of multiple patterns into processes in terms of structure and semantics is based on a formalism used by the platform (cf. **B**–**E**). Correctness-preserving process improvements should be developed and proved to be correct by integration experts, and automatically applied by the platform (cf. **F**–**I**).

2.2. Potential Process Optimization by Example

We now introduce the composition stage by example of an integration process that could be improved using ReCO.

Many organizations have started to connect their on-premise applications such as Customer Relationship Management (CRM) systems with cloud applications such as SAP Cloud for Customer (COD) using integration processes similar to the one shown in Fig. 3. A *CRM Material* is sent from the CRM system via EDI (more precisely the SAP IDOC transport protocol) to an integration process running on SAP Cloud Platform Integration (CPI) [14]. The integration process enriches the message header (MSG.HDR) with additional information based on a document number for reliable messaging (i.e., AppID), which allows redelivery of the message in an exactly-once service quality [4]. The IDOC structure is then mapped to the COD service description and sent to the COD receiver.

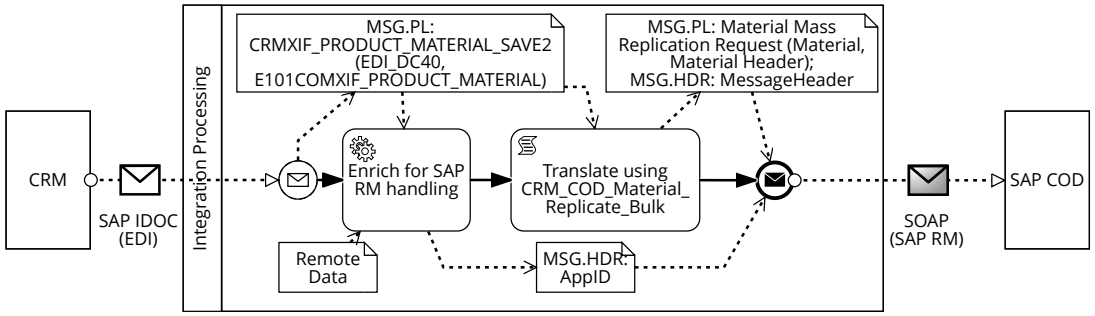


Figure 3: Replicate material from SAP Business Suite (a *hybrid integration* scenario in BPMN)

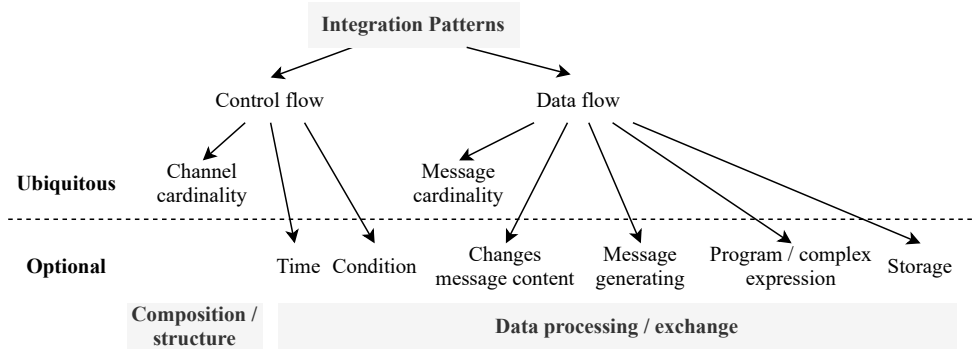


Figure 4: Categorizing integration pattern characteristics according to control and data flow

Already in this simple integration process, an obvious improvement can be applied: the data-independent Content Enricher and Message Translator patterns [2] could be executed in parallel. Importantly, such a change does not alter the behaviour of the integration process.

In this paper, we seek to find a mechanism to combine the inherently, structurally correct pattern composition formalism from [7] with the work on timed db-nets [6] that allow for semantically correct definitions of integration patterns, and to prove that improvements are correctness-preserving. The interaction between the user/modeler and the integration system requires a ReCO process that addresses the objectives (i)–(v).

3. Background and Requirements

In this section, we give a brief background on application integration patterns and their optimizations, by analyzing recurring pattern characteristics as well as collecting existing optimizations as strategies. We also derive and discuss requirements for a suitable formalism for pattern compositions in the context of the optimization strategies.

3.1. Integration Pattern Characteristics

Enterprise integration patterns (EIPs) [2] with recent additions [4, 5] form a suitable and important abstraction when implementing application integration scenarios. Besides their original differentiation in functional categories such as *message channels*, *message routers*, *message transformations*, and *messaging endpoints* among others, there are more subtle means of classifying patterns by *pattern characteristics* that consider the control and data flow within and between integration patterns, and thus help greatly when formalizing pattern compositions.

We analyzed all patterns from the literature [2, 4, 5] regarding their control and data flow characteristics. Our findings are summarised in Fig. 4. The characteristics of *channel* and *message cardinality* (CC and MC, respectively) are ubiquitous and can be found in every pattern. We also identified a number of optional and non-exclusive characteristics: if the pattern *changes message contents* (CHG), if it is *message generating* (MG), if it has *conditions* (CND), and if it has *programs / complex expressions* (PRG). Additionally, *time* and *storage* (also found in [6]) are important requirements, especially for our later considerations on runtime semantics, but on the level of compositions, they are not very relevant due to their pattern local nature, and are subsumed under control and data flow.

Together these characteristics summarize all relevant control and data flow properties of the considered integration patterns. In this work, composition and structure becomes relevant for checking structural correctness properties, while data processing and exchange characteristics are required mostly for a composition's semantic correctness. Both notions of correctness are especially relevant for modeling as well as any improvements to the composition (e.g., in the form of optimizations).

3.1.1. Ubiquitous Characteristics

Every pattern has both a channel and a message cardinality, covering control and data flow aspects that are relevant for composing patterns.

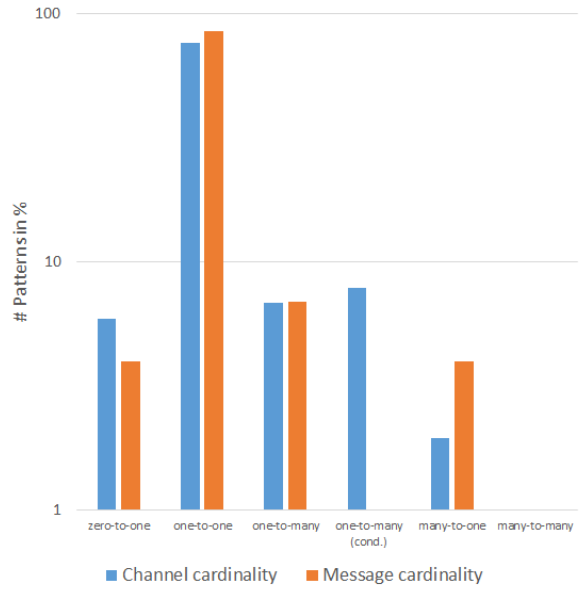


Figure 5: Channel and message cardinalities ("zero-to-one" includes "one-to-zero")

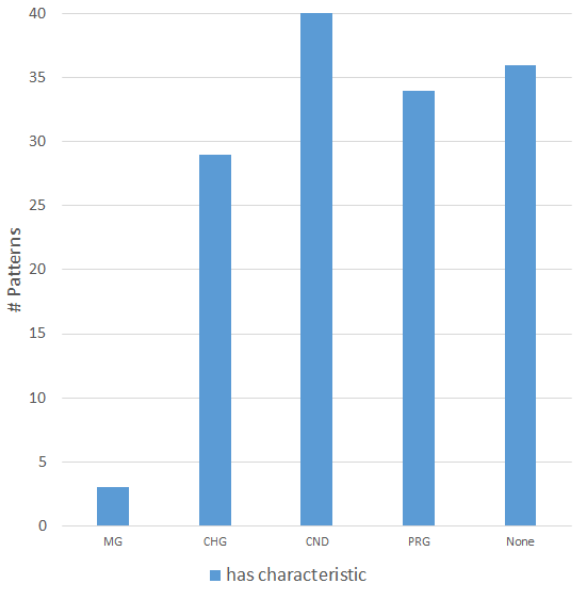


Figure 6: Further characteristics and configurations

Channel Cardinality. The channel cardinality specifies the number of incoming and outgoing message channels of an integration pattern. It is especially important for the structural correctness of a pattern composition. The relative number of patterns of each channel cardinality can be found in Fig. 5. A *zero-to-one* or *one-to-zero* cardinality is exclusively found in *start-* and *end* components of a composition, like message endpoints (e.g., commutative endpoint [5]) or integration adapters (e.g., event-driven consumer [2]). Most of the transformation patterns and some of the routing patterns are *message processors* that have a channel cardinality of *one-to-one* (e.g., aggregator, splitter [2]). The remainder of the routing patterns are either *forks* with cardinality *one-to-many* (e.g., multicast), *conditional forks* with *one-to-many (cond.)* (e.g., content-based router [2]), or *joins* with cardinality *many-to-one* (e.g., join router [5]). We found no *many-to-many* patterns at all.

Message cardinality. Similar to the channel cardinality, the message cardinality specifies the number of incoming and outgoing messages. However, the message cardinality is relevant for specifying the data transfer between patterns in a composition. As summarised in Fig. 5, we found that most of the integration patterns receive or require one and emit one message (e.g., message translator, message signer). There are also patterns that require one message and emit several messages (e.g., splitter, multicast) and similarly receive many and emit only one (e.g., aggregator). Finally, there are patterns that require zero and emit one or vice versa (e.g., event-driven consumer or producer). Note that there are no patterns with a message cardinality of *one-to-many (cond.)* or *many-to-many*.

3.1.2. Optional Characteristics

We found that certain data flow characteristics are only present in some patterns. The number of patterns that have each identified characteristic can be found in Fig. 6.

Message Generating. A pattern is message generating, if it does not simply pass or alter the received message, but creates a completely new message (e.g., aggregator, splitter). The new message might preserve data and structure from a received message, but will have a new message identifier. If a newly generated message is composed out of several received messages, the lineage to the original messages is preserved by adding a message history [2]. We identified only three message generating patterns.

Changing Message Content. In some cases it might be interesting to distinguish between integration patterns that actually change the content of received messages (i.e., mostly, but not exclusively, message transformation patterns) and read-only patterns, which either read data from the message for evaluating a condition (e.g., content-based router) or do not look into the message at all (e.g., multicast [5], claim check [2]). We found 29 message changing patterns.

Conditions. A condition is a read-only, user-defined function that returns a Boolean valuation. Conditions are mostly used in routing patterns, which decide on route or no-route, when receiving a message. We found 40 patterns that require a condition to function.

Program/Complex Expressions. A program or complex expression is an arbitrary, user-defined function that might change the content of a message by local or remote program execution (incl. remote procedure and database calls), which we found in 34 patterns. As can be seen in Fig. 6, there are more patterns requiring

an expression than there are patterns changing the content of a message. Consequently, there is a small number of read-only patterns that require more complex processing, potentially with side-effects (e.g., load balancer [5]).

None. Since the identified characteristics are optional and non-exclusive, there is also a significant number of 36 patterns that do not have any of the characteristics (e.g., the detour [5], wire tap [2]).

3.1.3. Summary

Analysing all known application integration patterns from the literature [2, 4, 5], we identified and categorized several control and data flow characteristics, relevant for the structural composition of those patterns and their internal and inter-pattern data flow. Channel cardinality and message cardinality are relevant to all patterns, but other characteristics are optional and non-exclusive.

3.2. Static Optimization Strategies

We consider improvements very important in the context of EAI, and thus we briefly survey recent attempts to optimize composed EIPs, in order to motivate the need to formalize their semantics. As a result, we derive so far unexplored prerequisites for optimizing compositions of EIPs. A more detailed collection of optimization strategies can be found in a non-mandatory supplementary material [15].

3.2.1. Identifying Optimization Strategies

Since a formalization of the EAI foundations in the form of integration patterns for static optimization of "data-aware" pattern processing is missing [5], we conducted a horizontal literature search [35] to identify optimization techniques in related domains. For EAI, the domains of business processes, workflow management and data integration are of particular interest. The results of our analysis are summarized in Tab. 1. Out of the resulting 616 hits, we selected 18 papers according to the search criteria "data-aware processes", and excluded work on unrelated aspects. This resulted in the *seven* papers listed in Tab. 2. The mapping of techniques from related domains to EAI was done by for instance taking the idea of projection push-downs [17, 24, 26, 27, 36] and deriving the early-filter or early-mapping technique in EAI. We categorized the techniques according to their impact (e.g., structural or process, data-flow) in context of the objectives for which they provide solutions.

In the following subsections, we now briefly discuss the optimization strategies listed in Tab. 2, in order to derive prerequisites needed for optimizing compositions of EIPs. To relate to their practical relevance and coverage so far (in the form of evaluations on "real-world" integration scenarios), we also discuss existing "data-aware" message processing solutions for each group of strategies.

3.2.2. Process Simplification

We grouped together techniques whose main goal is reducing model complexity (i.e., number of patterns) under the heading of process simplification. The cost reduction of these techniques can be measured by pattern processing time (latency, i.e., time required per operation) and model complexity metrics [38]. Process simplification can be achieved by removing redundant patterns like *Redundant Subprocess Removal* (e.g., removing one of two identical sub-flows), *Combine Sibling Patterns* (e.g., removing one of two identical patterns), or *Unnecessary Conditional Fork* (e.g., removing redundant branching). As far as we know, the only practical study of combining sibling patterns can be found in Ritter et al. [11], showing moderate throughput improvements. The simplifications requires a formalization of patterns as a control graph structure, which helps to identify and deal with the structural changes. Previous work targeting process simplification include Böhm et al. [36] and Habib, Anjum and Rana [24], who use evolutionary search approaches on workflow graphs, and Vrhovnik et al. [27], who use a rule formalization on query graphs.

3.2.3. Data Reduction

The reduction of data can be facilitated by pattern push-down optimizations of message-element-cardinality-reducing patterns, which we call *Early-Filter* (for data; e.g., removing elements from the message content), *Early-Mapping* (e.g., applying message transformations), as well as message-reducing optimization patterns like *Early-Filter* (for messages; e.g., removing messages), *Early-Aggregation* (e.g., combining multiple messages), *Early-Claim Check* (e.g., storing content and claiming it later without passing it through the pipeline), and *Early-Split* (e.g., cutting one large message into several smaller ones). Measuring data reduction requires a cost model based on the characteristics of the patterns, as well as the data and element cardinalities. For example, the practical realizations for multimedia [37] and hardware streaming [11] show improvements especially for early-filter, split and aggregation, as well as moderate improvements for early-mapping. This requires a formalization that is able to represent data or element flow. Data reduction optimizations target message throughput improvements (i.e., processed messages per time unit), however, some have a negative impact on the model complexity. Previous work on data reduction include Getta [26], based on relational algebra, and Niedermann, Radeschütz and Mitschang [17], who define optimizations algorithmically for a graph-based model.

3.2.4. Parallelization

Parallelization of processes can be facilitated through transformations such as *Sequence to Parallel* (e.g., duplicating a pattern or sequence of pattern processing), or, if not beneficial, reverted, e.g., by *Merge Parallel*. For example, good practical results have been shown for vectorization [31] and hardware parallelization [11]. Formalizing such operations again require a control graph structure. Although the main focus of parallelization is message throughput, heterogeneous variants also improve latency. In both cases, parallelization requires additional patterns, which negatively impacts the model complexity, whereas merging parallel processes improves the model complexity and latency. Previous work on pattern parallelization include Zhang

Table 1: Optimizations in related domains — horizontal search

Keyword	hits	selected	Selection criteria	Selected Papers
Business Process Optimization	159	3	data-aware processes	survey [16], optimization patterns [17, 18]
Workflow Optimization	396	6	data-aware processes	instance scheduling [19, 20, 21], scheduling and partitioning for interaction [22], scheduling and placement [23], operator merge [24]
Data Integration Optimization	61	2	data-aware processes optimization, (no schema-matching)	instance scheduling, parallelization [25], ordering, materialization, arguments, algebraic [26]
Added	n/a	13	expert knowledge	business process [27], workflow survey [10, 28], data integration [29], distributed applications [30], EAI [4, 5, 11, 31], placement [32, 33], resilience [34]
Removed	-	1		classification only [16]
Overall	616	23		

Table 2: Optimization Strategies in the Context of Integration Processes

Strategy	Optimization	Throughput	Latency	Complexity	Practical Studies
OS-1: Process Simplification	Redundant Sub-process Removal [36]		↑	↑	-
	Combine Sibling Patterns [24, 36]		↑	↑	([11])
	Unnecessary conditional fork [27, 36]	↗	↑	↑	-
OS-2: Data Reduction	Early-Filter [17, 24, 26, 27, 36]	↑			[11]
	Early-Mapping [24, 26, 36]	↑			[11, 37]
	Early-Aggregation [24, 26, 36]	↑			[37]
	Claim Check [26, 36]	↑		↘	-
	Early-Split [11]	↑		↘	[11, 37]
OS-3: Parallelization	Sequence to parallel [17, 25, 27, 36]	↑		↘	[11, 31]
	Merge parallel sub-processes [17, 25, 27, 36]		↑	↑	[11]
	Heterogeneous parallelization [23]	↑		↘	-
OS-4: Pattern Placement	Pushdown to Endpoint (extending OS-2)	↑		(↑)	[31, 32, 33]
OS-5: Reduce Interaction	Ignore Failing Endpoints [4, 5, 34]		↑		-
	Reduce Requests [27]	↗	↑		-

↑: improvement, ↗: slight improvement, ↘: slight deterioration, ␣: no effect

et al. [25], who defines a service composition model, to which algorithmically defined optimizations are applied.

3.2.5. Pattern Placement

For all of the data reduction optimizations (cf. OS-2) "Pushdown to Endpoint" can be applied by extending the placement to the message endpoints, which improves message throughput and reduces the complexity of the integration process, while increasing the complexity at the message endpoints. For example, good practical results have been shown for vectorization [31] and cost efficient, security-aware placement [32, 33].

3.2.6. Reduce Interaction

The resilience and robustness of integration processes is crucial – especially in the cloud. Dependencies to resources used by an integration process (e.g., database, message queuing) and the message endpoints (e.g., mobile, cloud) has to be dealt with during the message processing. Optimizations like *Ignore Failing Endpoints* and *Reduce Requests* help dealing with potentially unreliable network communication, and allow for smart network usage and reaction to exceptional situations or unavailabilities. This requires a formalism that is able to represent data flow and time. These optimizations target more stable processes, improved latency and potentially higher throughput.

3.2.7. Summary

We found several optimizations in related domains like data integration, business process and workflow, and re-interpreted them in the context of EAI. The optimizations have different effects regarding relevant categories like throughput, latency and model complexity, which we used to categorize them into three disjoint classes of optimization strategies, namely process simplification, data reduction and parallelization. For most of the optimizations we identified implementations that report a successful application to problems in the EAI domain.

3.3. Discussion: Requirements for Formalizing Integration Pattern Compositions

Based on our analyses of characteristics of single patterns and optimization strategies for pattern compositions, we briefly discuss requirements for the formalization of pattern compositions and their improvements as optimizations. These requirements are listed and set into context to the closest related approaches known from the application and data integration do-

mains in Tab. 3. We also give examples of optimization strategies (cf. OS-x) that are co-enabled when fulfilling the requirements.

We found that a fundamental support for control flow is mandatory for pattern compositions, due to the pipes-and-filters nature of integration scenarios [2, 5] (e.g., co-enabling OS-1). Hence, a suitable formalism representing pattern compositions requires a formal representation (**REQ-1: Formal representation of control flow**), e.g., as found in first formalization attempts using Coloured Petri Nets by Fahland and Gierds [39] and our recent work on timed db-nets [6], which essentially denotes an improvement of previous work regarding formal representation and analysis properties like data flow, time, transactional data storage and exceptions. The work by Böhm et al. [36] stems from the data integration domain, and thus only has a weak notion of control flow and none of integration patterns. As also found in [6], the known integration patterns require different aspects of time like timeouts, delays, and time-based rate limits to be functional (**REQ-2: Formal treatment of time**), e.g., co-enabling OS-5.

The concept of pipes-and-filters also requires support for the flow and processing of messages (**REQ-3: Formal representation of data flow**), which is also supported by the closest known formalizations, using Coloured Petri nets in [39], plus an extension to db-nets [40] in [6], and a data dependency tree in [36] (e.g., co-enabling OS-2).

Another batch of requirements from related work [6] target properties that were only recently formally represented, and thus are not considered in [39, 36]. Let us briefly summarize these pattern-level requirements in the context of this work. To be operational, some of the patterns like commutative and idempotent receivers as well as aggregator require the ability to persistently store data (**REQ-4: Formal treatment of databases and transactions**), e.g., co-enabling OS-2. Finally, potential exceptions have to be handled and compensated for within a pattern and on a composition level (**REQ-5: Formal treatment of exceptions and compensations**).

To represent and reason about pattern compositions, an adequate formalism must include the ability to structurally and semantically compose patterns (**REQ-6: Formalism must be compositional**), co-enabling OS-1–5. This core requirement is only partially supported in related work: in the work on Petri nets [39, 6], composition is assumed through the composable nature of Petri nets, but no formal construction of such compositions is given. Similarly, [36] introduces a composition of data integration operations, but again without a formal construction.

Once patterns are composed, the compositions will be subject to frequent changes such as extensions, adaptation due to changing legal requirements, or improvements and optimizations. In this work we focus on optimizations representing a comprehensive set of change operations that introduce change to pattern compositions. For a formal analysis of such changes, the optimizations themselves shall be represented in a formal way, such that compositions and their change operations can be formally analyzed (**REQ-7: Formal treatment of improvements (of control and data flow)**), co-enabling OS-1–5. The notion of change is partially considered in data integration by [36], but not in recent application integration work [39, 6].

Finally, a suitable formal representation of pattern compositions shall allow for a structural and semantic analysis of the correctness of a composition. This requirement also contains the notion of remaining correct after applying changes to the compositions, e.g., in the form of optimizations (**REQ-8: Formal specification of preserving correctness (structurally and semantically) of compositions**), co-enabling OS-1–5. In the context of structural composition correctness, we identified the channel cardinality characteristic as decisive correctness criteria based on the control flow. For example, a content-based router with a cardinality of 1:n channels, can only be connected to 1 input and n output channels, otherwise the composition is structurally incorrect. The semantic correctness has to go deeper into the fundamental execution semantics of integration patterns as defined in [6], and thus we build on top of that formalism. However, neither of the current approaches [39, 6, 36] addresses the requirement of structural and semantic correctness for compositions, nor do they guarantee a general correctness-preserving property when changing compositions.

4. Graph-based Pattern Compositions

We now introduce a formalization of pattern compositions, and an abstract cost model for them. This is needed in order to rigorously reason about optimizations.

4.1. Integration Pattern Graphs

Taking requirement REQ-1 of having a formal representation of control flow from Sec. 3.3 into account, it is natural to model pattern compositions as extended control flow graphs [41], as we do in Definition 1. This gives a high level modelling language that is easy to use and understand, and is close to informal notation used by practitioners [4, 5]. To take requirement REQ-3 of having a formal representation of data flow into account, we will further enrich the vertices of the graph with additional information in Definitions 2 and 3. Requirements REQ-2, REQ-4 and REQ-5 are pattern local [6], and thus not relevant at the abstraction level of pattern compositions. They will become important when we consider the runtime semantics of compositions later in Sec. 5 and 6. Control flow graphs can easily be composed into larger graphs, and hence requirement REQ-6 of composability is fulfilled. Requirements REQ-7 and REQ-8 will be addressed in Sec. 7, of course building on the definitions in this section.

Compared to for example colored Petri Nets, integration pattern graphs represents pattern compositions at a higher and more specialised level of abstraction, which is more easily understood also for non-technical users.

Before we get to the definition of the kind of graph we need to model pattern compositions, let us fix some notation: a directed graph is given by a set of nodes P and a set of edges $E \subseteq P \times P$. For a node $p \in P$, we write $\bullet p = \{p' \in P \mid (p', p) \in E\}$ for the set of direct predecessors of p, and $p\bullet = \{p'' \in P \mid (p, p'') \in E\}$ for the set of direct successors of p.

Table 3: Formalization requirements

ID	Requirement	Fahland et al. [39]	Ritter et al. [6]	Böhm et al. [36]
REQ-1	Control flow (pipes and filter)	✓	✓	(✓)
REQ-2	Time		✓	
REQ-3	Data flow	✓	✓	✓
REQ-4	Database, Transactions		✓	
REQ-5	Exceptions, Compensations		✓	
REQ-6	Compositional	(✓)	(✓)	(✓)
REQ-7	Improvements (control and data flow)			(✓)
REQ-8	Preserving correctness			

✓: covered, (✓): partially covered, ⌴: not covered or out of scope

Definition 1 (Integration pattern type graph). *An* integration pattern typed graph (IPTG) *is a directed graph with set of nodes P and set of edges $E \subseteq P \times P$, together with a function type $: P \to T$, where $T = \{$start, end, message processor, fork, structural join, condition, merge, external call$\}$. An IPTG $(P, E, type)$ is* correct *if*

- *there exists p_1, $p_2 \in P$ with $type(p_1) =$ start and $type(p_2) =$ end;*
- *if $type(p) =$ start then $|\bullet p| = 0$, and if $type(p) =$ end then $|p \bullet| = 0$.*
- *if $type(p) \in \{$fork, condition$\}$ then $|\bullet p| = 1$ and $|p \bullet| > 1$, and if $type(p) =$ join then $|\bullet p| > 1$ and $|p \bullet| = 1$;*
- *if $type(p) \in \{$message processor, merge$\}$ then $|\bullet p| = 1$ and $|p \bullet| = 1$;*
- *if $type(p) \in \{$external call$\}$ then $|\bullet p| = 2$ and $|p \bullet| = 2$;*
- *The graph (P, E) is connected and acyclic.* □

In the definition, we think of P as a set of extended integration patterns that are connected by message channels represented as edges in E, as in a pipes and filter architecture. The function *type* records what type of pattern each node represents. The first correctness condition says that an integration pattern has at least one source and one target, while the next three states the cardinality of the involved patterns coincide with the in- and out-degrees of the nodes in the graph representing them. The last condition states that the graph represents one integration pattern, not multiple unrelated ones, and that messages do not loop back to previous patterns.

To represent the data flow, i.e., the basis for the optimizations and requirement REQ-3, the control flow has to be enhanced with (a) the data that is processed by each pattern, and (b) the data exchanged between the patterns in the composition. The data processed by each pattern (a) is described as a set of *pattern characteristics*:

Definition 2 (Pattern characteristics). *A* pattern characteristic *assignment for a graph (P, E) is a function $char : P \to 2^{PC}$, assigning to each vertex a subset of the set*

$$\begin{aligned} PC = {} & (\{MC\} \times \mathbb{N}^2) \cup (\{ACC\} \times \{ro, rw\}) \cup (\{MG\} \times \mathbb{B}) \cup \\ & (\{CND\} \times 2^{\mathsf{BExp}}) \cup (\{PRG\} \times \mathsf{Exp}) \cup \\ & (\{S\} \times \mathsf{Exp}) \cup (\{QRY\} \times 2^{\mathsf{Exp}}) \cup (\{ACTN\} \times 2^{\mathsf{Exp}}) \cup \\ & (\{TM\} \times (\mathbb{Q}^{\geq 0} \times (\mathbb{Q}^{\geq 0} \cup \{\infty\}))) \ , \end{aligned}$$

where $\mathbb{B}$ is the set of Booleans, BExp *and* Exp *the sets of Boolean and program expressions, respectively, and MC, ACC, MG, CND, PRG, S, QRY, ACTN, TM some distinct symbols.* □

The property and value domains in Definition 2 are based on the pattern characteristics identified in Sec. 3.1, and could of course be be extended if future patterns required it. We briefly explain the intuition behind the characteristics: the characteristic (MC, n, k) represents a message cardinality of n:k, (ACC, x) the message access, depending on if x is read-only ro or read-write rw, and the characteristic (MG, y) represents whether the pattern is message generating depending on the Boolean y. A $(\text{CND}, \{c1, ..., cn\})$ represents the conditions $c_1, \ldots, c_n$ used by the pattern to route messages, and (PRG, p) represents a program p used by the pattern (e.g., for message translation). The storage aspects are denoted by a schema $(\text{S}, (p_s))$ created by a program p_s, expressions $(\text{QRY}, \{q1, ..., qn\})$ denoting a set of distinct queries $q1, \ldots, qn$, and a set of actions $(\text{ACTN}, \{a1, ..., an\})$ with distinct $a1, \ldots, an$. Finally, $(\text{TM}, (\tau_s, \tau_e))$ represents a timing window from τ_s to τ_e.

Example 1. *The characteristics of a content-based router CBR is char(CBR) =*$\{$*(MC, 1:1), (ACC, ro), (MG, false), (CND,*$\{cnd_1, \ldots, cnd_{n-1}\}$*), (PRG,*`null`*), (S,* `null`*), (QRY,* $\emptyset$*), (ACTN,* $\emptyset$*), (TM,* $(0, 0)$*)*$\}$*, because of the workflow of the router: it receives exactly one message, then evaluates up to $n - 1$ routing conditions cnd_1 up to cnd_{n-1} (one for each outgoing channel), until a condition matches. The original message is then rerouted read-only (in particular, the router is not message generating) on the selected output channel, or forwarded to the default channel, if no condition matches. The router does not require programs, storage or time configurations.* ■

The data exchange between the patterns (b) is based on *input and output contracts* (similar to data parallelization contracts in [42]). These contracts specify how the data is exchanged in terms of required message properties of a pattern during the data exchange:

Definition 3 (Pattern contract). *A* pattern contract *assignment for a graph (P, E) is a function $contr : P \to CPT \times 2^{EL}$, assigning to each vertex a function of type*

$$CPT = \{signed, encrypted, encoded\} \to \{yes, no, any\}$$

and a subset of the set

$$EL = (\{HDR\} \times 2^D) \cup (\{PL\} \times 2^D) \cup (\{ATTCH\} \times 2^D) \ ,$$

where D is a set of data elements (the concrete elements of D will vary with the application domain). We represent the function of type CPT by its graph, leaving out the attributes that are sent to any, when convenient. □

The set CPT in a contract represents integration concepts, while the set EL represents data elements and the structure of the message: its headers (HDR, H), its payload (PL, Y) and its attachments (ATTCH, A). Each pattern will have an inbound and an outbound pattern contract, describing the format of the data it is able to receive and send respectively — the role of pattern contracts is to make sure that adjacent inbound and outbound contracts match.

Example 2. *A content-based router is not able to process encrypted messages. Recall that its pattern characteristics included a collection of routing conditions: these might require read-only access to message elements such as certain headers h_1 or payload elements e_1, e_2. Hence the input contract for a router mentioning these message elements is*

$$inContr(CBR) = (\{(encrypted, no)\}, \{(HDR, \{h_1\}), (PL, \{e_1, e_2\})\}) \ .$$

Since the router forwards the original message, the output contract is the same as the input contract. ■

Definition 4. *Let $(C, E) \in CPT \times 2^{EL}$ be a pattern contract, and $X \subseteq CPT \times 2^{EL}$ a set of pattern contracts. Write $X_{CPT} = \{C' \mid (\exists E')\ (C', E') \in X\}$ and $X_{EL} = \{E' \mid (\exists C')\ (C', E') \in X\}$. We say that (C, E)* matches *X, in symbols* $\mathsf{match}((C, E), X)$*, if following condition holds:*

$$(\forall x)(C(x) \neq any \implies (\forall C' \in X_{CPT})(C'(x) = C(x) \vee C'(x) = any)) \wedge$$
$$(\forall (m, Z) \in E)(Z = \bigcup_{(m,Z') \in \cup X_{EL}} Z') \ .$$

□

We are interested in an inbound contract K_{in} matching the outbound contracts $K_1, \ldots, K_n$ of its predecessors. In words, this is the case if (i) for all integration concepts that are important to K_{in}, all contracts K_i either agree, or at least one of K_{in} or K_i accepts any value (*concept correctness*); and (ii) together, $K_1, \ldots, K_n$ supply all the message elements that K_{in} needs (*data element correctness*).

Since pattern contracts can refer to arbitrary message elements, a formalization of an integration pattern can be quite precise. On the other hand, unless care is taken, the formalization can easily become specific to a particular pattern composition. In practice, it is often possible to restrict attention to a small number of important message elements (see Example 3 below), which makes the formalization manageable.

Putting everything together, we formalize pattern compositions as integration pattern typed graphs with pattern characteristics and inbound and outbound pattern contracts for each pattern:

Definition 5. *An* integration pattern contract graph *(IPCG) is a tuple*

$$(P, E, type, char, inContr, outContr)$$

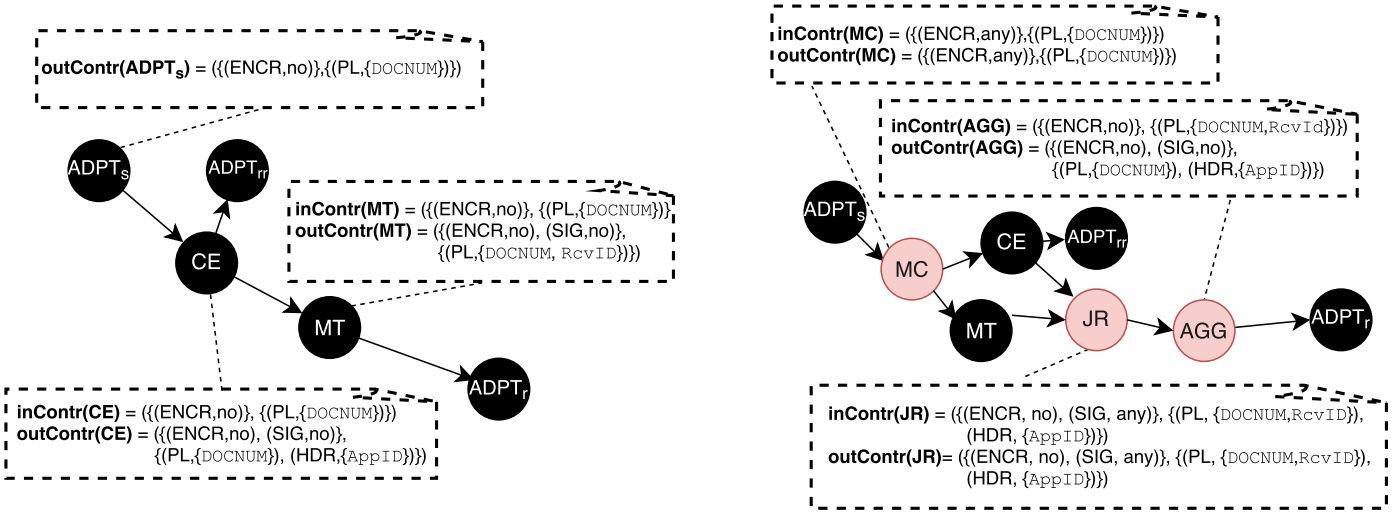
where $(P, E, type)$ is an IPTG, $char : P \to 2^{PC}$ is a pattern characteristics assignment, and $inContr : \prod_{p \in P} (CPT \times 2^{EL})^{|\bullet p|}$ and $outContr : \prod_{p \in P} (CPT \times 2^{EL})^{|p \bullet|}$ are pattern contract assignments — one for each incoming and outgoing edge of the pattern, respectively — called the inbound and outbound contract assignment respectively. It is correct*, if the underlying IPTG $(P, E, type)$ is correct, and inbound contracts matches the outbound contracts of the patterns' predecessors, i.e. if for every $p \in P$*

$$type(p) = start \vee \mathsf{match}(inContr(p), \{outContr(p') \mid p' \in \bullet p\}) \ .$$

Two IPCGs are isomorphic *if there is a bijective function between their patterns that preserves edges, types, characteristics and contracts.* □

Example 3. *Figures 7(a) and 7(b) show IPCGs representing an excerpt of the motivating example from the introduction. Figure 7(a) represents the IPCG of the original scenario with a focus on the contracts, and Fig. 7(b) denotes an already improved composition showing the characteristics and giving an indication on the pattern latency. In Fig. 7(a), the input contract $inContr(CE)$ of the content enricher pattern CE requires a non-encrypted message and a payload element* `DOCNUM`*. The content enricher makes a query to get an application ID* `AppID` *from an external system, and appends it to the message header. Hence the output contract $outContr(CE)$ contains $(HDR, \{\texttt{AppID}\})$. The content enricher then emits a message that is not encrypted or signed. A subsequent message translator MT requires the same message payload, but does not care about the appended header. It adds another payload* `RcvID` *to the message. Comparing inbound and outbound pattern contracts for adjacent patterns, we see that this is a correct IPCG.*

One improvement of this composition is depicted in Fig. 7(b), where the independent patterns CE and MT have been parallelized. To achieve this, a read-only structural fork with channel cardinality 1:n in the form of a multicast MC has been added. The inbound and outbound contracts of MC are adapted to fit into the composition. After the concurrent execution of CE and MT, a join router JR brings the messages back together again and feeds the result into an aggregator AGG that restores the format that $ADPT_r$ expects. We see that the resulting IPCG is still correct, so this would be a sound optimization. ■



(a) IPCG from the motivating example

(b) IPCG after "sequence to parallel" optimization

Figure 7: An IPCG of an excerpt of the motivating example

4.2. Abstract Cost Model

In order to decide if an optimization is an improvement or not, we want to associate abstract costs to integration patterns. We do this on the pattern level, similar to the work on data integration operators [43]. The cost of the overall integration pattern can then be computed as the sum of the cost of its constituent patterns. Costs are considered parametrized by the cardinality of data inputs $|d_{\text{in}_i}|$ ($1 \leq i \leq n$, if the pattern has in-degree n), data outputs $|d_{\text{out}_j}|$ ($1 \leq j \leq m$, if the pattern has out-degree m), and external resource data sets $|d_r|$. The costs can also refer to the pattern characteristics.

Definition 6 (Cost model). *A* cost assignment *for an IPCG* $G = (P, E, type, char, inContr, outContr)$ *is an function* $cost(p) : \mathbb{N}^n \times \mathbb{N}^k \times \mathbb{N}^r \to \mathbb{Q}$ *for each* $p \in P$*, where* p *has in-degree* n*, out-degree* k *and* r *external connections. The cost* $cost(G) : \mathbb{N}^N \times \mathbb{N}^K \times \mathbb{N}^R \to \mathbb{Q}$ *of the IPCG pattern graph* G*, where* N *is the sum of the in-degrees of its patterns,* K *the sum of their out-degrees, and* R *the sum of their external connections, is defined to be the sum of the costs of its constituent patterns:*

$$cost(G)(d_{\text{in}}, d_{\text{out}}, d_r) = \sum_{p \in P} cost(p)(|d_{\text{in}}(p)|, |d_{\text{out}}(p)|, |d_r(p)|) \ ,$$

where we suggestively have written $|d_{\text{in}}(p)|$ *for the projection from the tuple* d_{in} *corresponding to* p*, similarly for* $|d_{\text{out}}(p)|$ *and* $|d_r(p)|$. ■

We have defined the abstract costs of the patterns discussed in this work in Tab. 4 — these will be used in the subsequent evaluation. We now explain the reasoning behind them. Routing patterns such as content based routers, message filters and aggregators mostly operate on the input message, and thus have an abstract cost related to its element cardinality $|d_{\text{in}}|$. For example, the abstract cost of the CBR is $cost(CBR) = \frac{\sum_{i=0}^{n-1} |d_{\text{in},i}|}{2}$, since it evaluates on average $\frac{n-1}{2}$ routing conditions on the input message. More complex routing patterns such as aggregators evaluate correlation and completion conditions, as well as an aggregation function on the input message, and also on sequences of messages of a certain length from an external resource. Hence the cost of an aggregator is $cost(AGG) = 2 \times |d_{\text{in}}| + \frac{|d_{\text{in}}|+|d_r|}{avg(len(seq))}$, where $len(seq)$ denotes the length of a Message Sequence [2] as for example used by an aggregator pattern. In contrast, message transformation patterns like content filters and enrichers mainly construct an output message, hence their costs are determined by the output cardinality $|d_{\text{out}}|$. For example, content enrichers create a request message from the input message with cost $|d_{\text{in}}|$, conducts an optional resource query $|d_r|$, and creates and enriches the response with cost $|d_{\text{out}}|$. Finally, the cost of message creation patterns such as external calls, receivers, and senders arise from costs for transport, protocol handling, and format conversion, as well as decompression. Hence the cost depends on the element cardinalities of input and output messages $|d_{\text{in}}|$, $|d_{\text{out}}|$.

Example 4. *We return to the claimed improved composition in Example 3. The latency of the composition* G_1 *in Fig. 7(a), calculated from the constituent pattern latencies, is* $cost(G_1) = t_{CE} + t_{MT}$ *with latency* t_p *and pattern* p*. The latency improvement potential given by switching to the composition* G_2 *in Fig. 7(b) is given by* $cost(G_2) = \max(t_{CE}, t_{MT}) + t_{MC} + t_{JR} + t_{AGG}$*. Obviously it is only beneficial to switch if* $cost(G_2) < cost(G_1)$*, and this condition depends on the concrete values involved. At the same time, the model complexity increases by three nodes and edges.* ■

5. A Semantics Using Timed DB-nets

Integration pattern graphs model the structural composition of integration patterns, but not their dynamics, i.e. how data

Table 4: Abstract costs of relevant patterns

Pattern p	Abstract Cost $cost(p)$	Factors
Content-based Router [2]	$\frac{\sum_{i=0}^{n-1} \lvert d_{\text{in},i}\rvert}{2}$	n=#channel conditions, half of them evaluated on average
Message Filter [2]	$\lvert d_{\text{in}}\rvert$	input data condition $\lvert d_{\text{in}}\rvert$
Aggregator [2]	$2 \times \lvert d_{\text{in}}\rvert + \frac{\lvert d_{\text{in}}\rvert+\lvert d_r\rvert}{avg(len(seq))}$	correlation, and completion conditions $\lvert d_{\text{in}}\rvert$, aggregation function $\frac{\lvert d_{\text{in}}\rvert+\lvert d_r\rvert}{avg(len(seq))}$ and length of a sequence length(seq) >= 2, and (transacted) resource d_r
Claim Check [2]	$2 \times \lvert d_r\rvert$	resource insert and get $\lvert d_r\rvert$
Splitter [2]	$\lvert d_{\text{out}}\rvert$	output data condition $\lvert d_{\text{out}}\rvert$
Multicast, Join Router [5]	$\sum_{i=0}^{n} cost(\text{procunit}_i)$	costs of processing units $cost(\text{procunit}_i)$, e.g., threading in software, for n channels
Content Filter [2]	$\lvert d_{\text{out}}\rvert$	output data creation $\lvert d_{\text{out}}\rvert$
Mapping [2]	$\lvert d_{\text{in}}\rvert + \lvert d_{\text{out}}\rvert$	output data creation $\lvert d_{\text{out}}\rvert$ from input data $\lvert d_{\text{in}}\rvert$
Content Enricher [2]	$\lvert d_{\text{in}}\rvert+\lvert d_r\rvert+\lvert d_{\text{out}}\rvert$	request message creation on $\lvert d_{\text{in}}\rvert$, resource query $\lvert d_r\rvert$, response data enrich $\lvert d_{\text{out}}\rvert$
External Call [5]	$\lvert d_{\text{out}}\rvert + \lvert d_{\text{in}}\rvert$	request $\lvert d_{\text{out}}\rvert$ and reply data $\lvert d_{\text{in}}\rvert$
Receive [2]	$\lvert d_{\text{in}}\rvert$	input data $\lvert d_{\text{in}}\rvert$
Send [2]	$\lvert d_{\text{out}}\rvert$	output data $\lvert d_{\text{out}}\rvert$

actually flow through the system. To model this, we use timed db-nets [6], an extension of db-nets [40] with an explicit notion of time (addressing REQ-2). The formalism of db-nets in turn is a refinement of colored Petri nets [44] with primitives for the net to query and update persistent data stores (addressing REQ-4). Exceptions are built into the framework in the form of rollbacks (addressing REQ-5).

We choose to work with timed db-nets rather than just colored Petri nets because they meet the mentioned requirements of integration processes that we have identified, and balances the dimensions of persistence, data logic and control layer of a Petri net. Avoiding the heavier encoding of colored Petri nets, timed db-nets make the modeling more concise and tractable for the interpretation procedure defined in Sec. 6.

To make the definition compositional, we have to extend the notion of timed db-nets to timed db-nets with boundaries that can be reasoned about separately, and then plugged together to form larger timed db-nets.

5.1. Open Timed DB-nets

In this section, we formally define the mathematical structure we use to give a runtime semantics to pattern graphs. We first recall the definition of timed db-nets, and then extend them to open timed db-nets, in order to make the definition compositional.

5.1.1. Ordinary Timed DB-nets

A timed db-net has three layers: a persistence layer describing the underlying database of the net, a logic layer describing the queries that can be made of the persistence layer, and a control layer describing how tokens of data flow through the net, executing queries. See Ritter et al. [6] for motivation and a more gentle definition.

Definition 7 (timed db-net [6]). *A* timed db-net *is a tuple* $(\mathfrak{D}, \mathcal{P}, \mathcal{L}, \mathcal{N}, \tau)$*, where:*

- $\mathfrak{D}$ *is a type domain — a finite set of data types, each of the form* $D = (\Delta_D, \Gamma_D)$*, where* Δ_D *is a value domain, and* Γ_D *is a finite set of domain-specific predicate symbols.*
- $\mathcal{P}$ *is a* $\mathfrak{D}$*-typed* ***persistence layer****, i.e., a pair* $(\mathcal{R}, E)$*, where* $\mathcal{R}$ *is a* $\mathfrak{D}$*-typed database schema, and* E *is a finite set of first-order FO*$(\mathfrak{D})$ *constraints over* $\mathcal{R}$*.*
- $\mathcal{L}$ *is a* $\mathfrak{D}$*-typed* ***data logic layer*** *over* $\mathcal{P}$*, i.e., a pair* (Q, A)*, where* Q *is a finite set of FO*$(\mathfrak{D})$ *queries over* $\mathcal{P}$*, and* A *is a finite set of actions over* $\mathcal{P}$*.*
- $\mathcal{N}$ *is a* $\mathfrak{D}$*-typed* ***control layer*** *over* $\mathcal{L}$*, i.e., a tuple* $(P, T, F_{in}, F_{out}, F_{rb}, \texttt{color}, \texttt{query}, \texttt{guard}, \texttt{action})$*, where:*
  1. $P = P_c \uplus P_v$ *is a finite set of places, partitioned into so-called control places* P_c *and view places* P_v*,*
  2. T *is a finite set of transitions,*
  3. F_{in} *is an input flow from* P *to* T*,*
  4. F_{out} *and* F_{rb} *are respectively output and roll-back flows from* T *to* P_c*,*
  5. `color` *is a color assignment over* P *(mapping* P *to a Cartesian product of data types),*
  6. `query` *is a query assignment from* P_v *to* Q *(mapping the results of* Q *as tokens of* P_v*),*
  7. `guard` *is a transition guard assignment over* T *(mapping each transition to a formula over its input inscriptions), and*
  8. `action` *is an action assignment from* T *to* A *(mapping some transitions to actions triggering updates over the persistence layer).*
- $\tau : T \to \mathbb{Q}^{\geq 0} \times (\mathbb{Q}^{\geq 0} \cup \{\infty\})$ *is a timed transition guard, mapping each transition* $t \in T$ *to a pair of values* $\tau(t) = (v_1, v_2)$*, where* v_1 *is a non-negative rational number, and* v_2 *is a non-negative rational number with* $v_1 \leq v_2$*, or the special constant* ∞*.* □

We adopt the following graphical conventions for drawing the control layer of a timed db-net: places are depicted as round nodes — view places are labelled by a database icon with queries written in green — and transitions as rectangles. Roll-back arcs are depicted with an "x": . Actions are written in blue, and guards are written in square brackets next

to the transition, and we adopt the following conventions for a timed transition guard τ and a transition t: *(i)* if $\tau(t) = (0, \infty)$, then no temporal label is shown for t (this is often the default choice for $\tau(t)$); *(ii)* if $\tau(t)$ is of the form (v, v), we attach label "$@v$" to t; *(iii)* if $\tau(t)$ is of the form (v_1, v_2) with $v_1 \neq v_2$, we attach label "$@\langle v_1, v_2 \rangle$" to t.

Example 5. *Fig. 8 shows a timed db-net realisation of an aggregator. The intention is that messages arrive at the place ch_{in}. The database is then queried using the Q_{msgs} query via a view place, and if it already contains the message, it is updated via the UpdateSeq action at transition T_1. If it does not contain the message, the CreateSeq action is triggered at T_2 instead, and the sequence number gets passed to the ch_{timer} place, whose output transition T_3 will be enabled after 30 seconds, triggering the TimeoutSeq action. This will enable transition T_4, with the effect that a token containing the data from the completed sequence, queried via Q_{seqs} from the database, will move into ch_{out}.*

5.1.2. Open Timed DB-nets

We now describe timed db-nets that are open, in the sense that they have "ports" for communicating with the outside world: the idea being that tokens can be received and sent on these ports, similar to in the existing literature on open Petri nets [45, 46, 47].

Definition 8 (Open timed db-net). *An* open timed db-net *is a pair $A = (N_A, B_A)$, where $N_A = (\mathfrak{D}, \mathcal{P}, \mathcal{L}, \mathcal{N}, \tau)$ is a timed db-net with control layer*

$$\mathcal{N} = (P_c \uplus P_v, T, F_{in}, F_{out}, F_{rb}, \texttt{color}, \texttt{query}, \texttt{guard}, \texttt{action})$$

and $B_A = (I_A, O_A) \in \mathsf{List}\, P_c \times \mathsf{List}\, P_c$ are lists of control places, called the input and output boundaries respectively, such that $F_{in}(o, t) = \emptyset$ for every $o \in O_A$, and $F_{out}(t, i) = F_{rb}(t, i) = \emptyset$ for every $i \in I_A$. The input (output) boundary configuration of *A is given by the corresponding list of colours of the input (output) boundary places of A, and we write*

$$N_A : \texttt{color}(I_A) \to \texttt{color}(O_A)$$

(where $\texttt{color}(X) = [\texttt{color}(x) \mid x \in X]$) to indicate that $A = (N_A, (I_A, O_A))$ is an open timed db-net with the given boundary configurations.

Note in particular that an open timed db-net with empty boundaries is by definition an ordinary timed db-net.

Example 6. *Fig. 9 shows an open timed db-net realisation of a join router (joining messages containing integer data). The input boundary are the places ch_{in_1} and ch_{in_2} and the output boundary the place ch_{out}. We draw the input boundary using dashed places on the left of the image, and the output similarly on the right (in general, a place can be part of both the input and the output boundary, but this will not occur in any nets constructed from pattern graphs).*

Similarly, the timed db-net realisation of an aggregator in Fig. 8 from Example 5 can be made into an open timed db-net by declaring the boundaries to be ch_{in} and ch_{out} respectively.

5.2. Execution Semantics for Open Timed DB-nets

We define the execution semantics of a given open timed db-net as a labelled transition system, where the states are snapshots of the db-net and the labelled transitions are given by firings, as well as transitions that can create and consume tokens at the input and output boundary respectively. A snapshot of an open timed db-net $\mathcal{B}$ is a snapshot $(\mathcal{I}, m)$ of $\mathcal{B}$ considered as an ordinary timed db-net (i.e. we forget about the boundaries), which in turn consists of a compliant database instance $\mathcal{I}$ and a marking m; see [6] for the precise definitions.

Given an open timed db-net $\mathcal{B}$ with boundaries $(I_\mathcal{B}, O_\mathcal{B})$, and a $\mathcal{B}$-snapshot s_0 (the *initial $\mathcal{B}$-snapshot*), we construct a labelled transition system $\Gamma^\mathcal{B}_{s_0} = (S, s_0, \to)$ as follows: S is the infinite set of $\mathcal{B}$-snapshots, and $\to \subseteq S \times (I_\mathcal{B} \cup T \cup O_B) \times S$ is defined by the following clauses (we write $s \xrightarrow{\ell} s'$ for $(s, \ell, s') \in \to$):

- for a transition t and $\mathcal{B}$-snapshots s_1, s_2, if there is a binding σ such that t fires in s_1 with binding σ producing s_2 (see [6]), then $s_1 \xrightarrow{t} s_2$;
- for an input boundary place p_i and $\mathcal{B}$-snapshot $s = (\mathcal{I}, m)$, if $s' = (\mathcal{I}, m')$ where $m'(p_i) = m(p_i) + \{o\}$ for some $o \in \Delta_\mathfrak{D}$, and $m'(x) = m(x)$ for $x \neq p_i$, then $s \xrightarrow{p_i} s'$; and
- for an output boundary place p_o and $\mathcal{B}$-snapshot $s = (\mathcal{I}, m)$, if $s' = (\mathcal{I}, m')$ where $m'(p_i) = m(p_i) - \{o\}$ for some $o \in \Delta_\mathfrak{D}$, and $m'(x) = m(x)$ for $x \neq p_o$, then $s \xrightarrow{p_o} s'$.

That is, in addition to transitions in the LTS arising from transitions in the db-net firing, we also have transitions labelled by each boundary place that make one token appear or disappear at the boundary, depending on if the place is an input or output. Note that for a closed timed db-net, the boundaries are empty, and the LTS correspond exactly to the LTS of timed db-nets from [6].

5.3. Composition of Open Timed DB-nets

It is straightforward to compose timed db-nets in parallel, i.e. in such a way that there is no interaction between the component nets. Given open timed db-nets

$$\mathcal{A} : [c_1, \ldots, c_n] \to [d_1, \ldots, d_m]$$
$$\mathcal{B} : [c'_1, \ldots, c'_{n'}] \to [d'_1, \ldots, d'_{m'}]$$

with the same type domains, persistence layers and data logic layers[1], we define an open timed db-net

$$\mathcal{A} \otimes \mathcal{B} : [c_1, \ldots, c_n, c'_1, \ldots, c'_{n'}] \to [d_1, \ldots, d_m, d'_1, \ldots, d'_{m'}]$$

again with the same type domain, persistence layer data logic layer, but whose places and transitions are the disjoint union of the places and transitions in $\mathcal{A}$ and $\mathcal{B}$ respectively. This gives a tensor product or parallel composition of nets, with unit

[1] To compose timed db-nets with different underlying layers, we first rename any unintended clashing names, and then take the union of the layers and embed the nets into their now common layers.

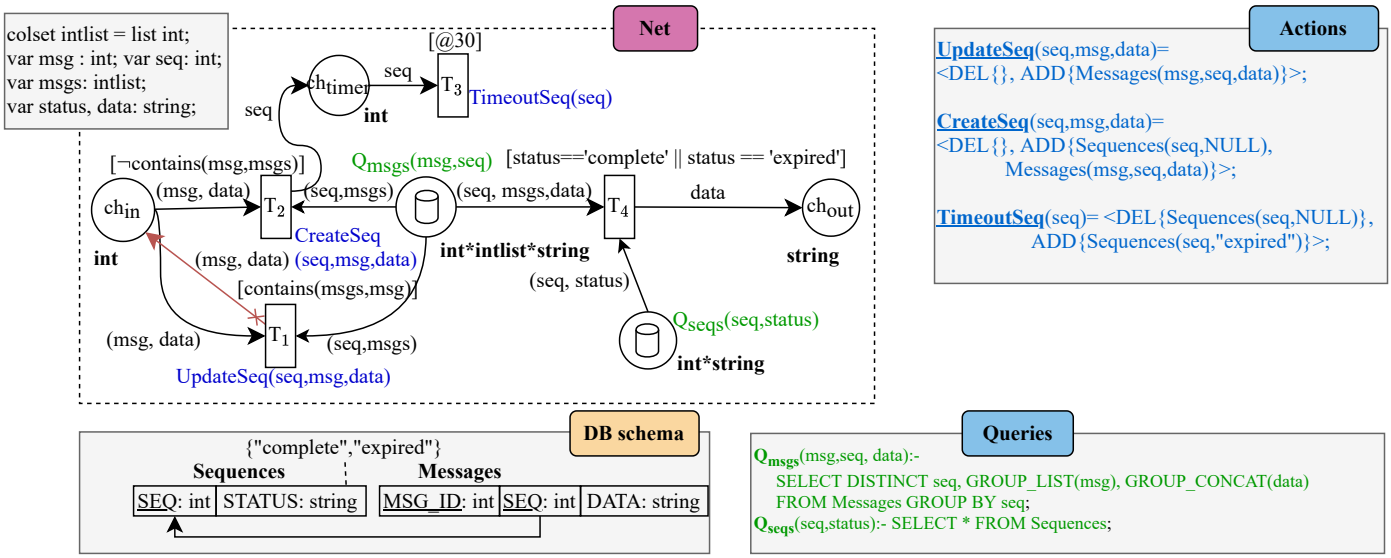


Figure 8: Timed db-net realization of an aggregator

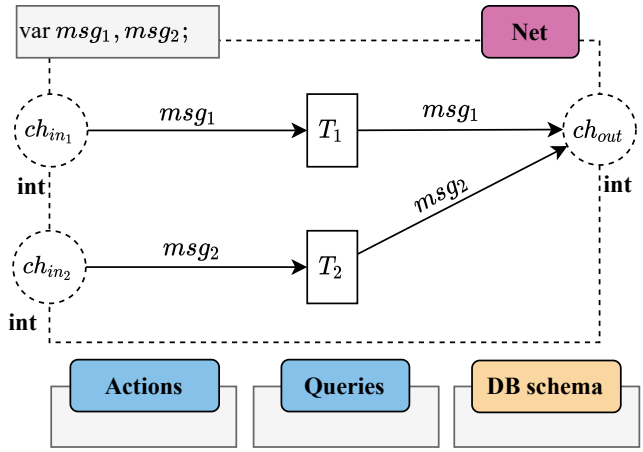


Figure 9: Timed db-net realization of a join router

$I : [] \to []$ the empty timed db-net (necessarily with empty boundary). Visually, we are stacking the control layers of $\mathcal{A}$ and $\mathcal{B}$ next to each other.

When the boundaries are compatible, i.e., when the input boundary configuration is the same as the output boundary configuration, we can also define a sequential composition of nets. This will be achieved by "gluing" the two nets together along their common boundary, formally expressed by quotienting the set of places in the construction of the composite net.

Definition 9 (Sequential composition of open nets). *Let* $\mathcal{A} : \texttt{color}(I_A) \to \texttt{color}(O_A)$ *and* $\mathcal{B} : \texttt{color}(I_B) \to \texttt{color}(O_B)$ *be open timed db-nets with the same type domains, persistence layers and data logic layers, and such that* $\texttt{color}(O_A) = \texttt{color}(I_B)$. *Write* $O_A = [o_1, \ldots, o_n]$ *and* $I_B = [i_1, \ldots, i_n]$ *— note that* O_A *and* I_B *must have the same length, since* $\texttt{color}(O_A) = \texttt{color}(I_A)$.

We define the composition

$$\mathcal{A} \fatsemi \mathcal{B} : \texttt{color}(I_A) \to \texttt{color}(O_B)$$

to again have the same type domain, persistence layer and data logic layer as $\mathcal{A}$ *and* $\mathcal{B}$, *and control layer*

$$\mathcal{N}_{\mathcal{A}\fatsemi\mathcal{B}} = (P, T, F_{in}, F_{out}, F_{rb}, \texttt{color}, \texttt{query}, \mathit{guard}, \texttt{action})$$

with

$$P = (P_A \uplus P_B)/\sim$$

where $\sim$ *is the equivalence relation generated by* $\mathsf{in}_{P_A}(o_k) \sim \mathsf{in}_{PB}(i_k)$ *for* $0 < k \leq n$,

$$T = T_A \uplus T_B$$

$$F_{in}(x,y) = \begin{cases} F^A_{in}(p,t) & \text{if } (x,y) = ([\mathsf{in}_{P_A}(p)], \mathsf{in}_{T_A}(t)) \\ F^B_{in}(p',t') & \text{if } (x,y) = ([\mathsf{in}_{P_B}(p')], \mathsf{in}_{T_B}(t')) \\ \emptyset & \text{otherwise} \end{cases}$$

$$F_{in}(x,y) = \begin{cases} F^A_{out}(p,t) & \text{if } (x,y) = (\mathsf{in}_{T_A}(t), [\mathsf{in}_{P_A}(p)]) \\ F^B_{out}(p',t') & \text{if } (x,y) = (\mathsf{in}_{T_B}(t'), [\mathsf{in}_{P_B}(p')]) \\ \emptyset & \text{otherwise} \end{cases}$$

$$F_{rb}(x,y) = \begin{cases} F^A_{rb}(t,p) & \text{if } (x,y) = (\mathsf{in}_{T_A}(t), [\mathsf{in}_{P_A}(p)]) \\ F^B_{rb}(t',p') & \text{if } (x,y) = (\mathsf{in}_{T_B}(t'), [\mathsf{in}_{P_B}(p')]) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\begin{aligned} \texttt{color} &= [\texttt{color}_A, \texttt{color}_B] \\ \texttt{query} &= [\texttt{query}_A, \texttt{query}_B] \\ \mathit{guard} &= [\mathit{guard}_A, \mathit{guard}_B] \\ \texttt{action}'' &= [\texttt{action}, \texttt{action}'] \\ \tau'' &= [\tau, \tau'] \end{aligned}$$

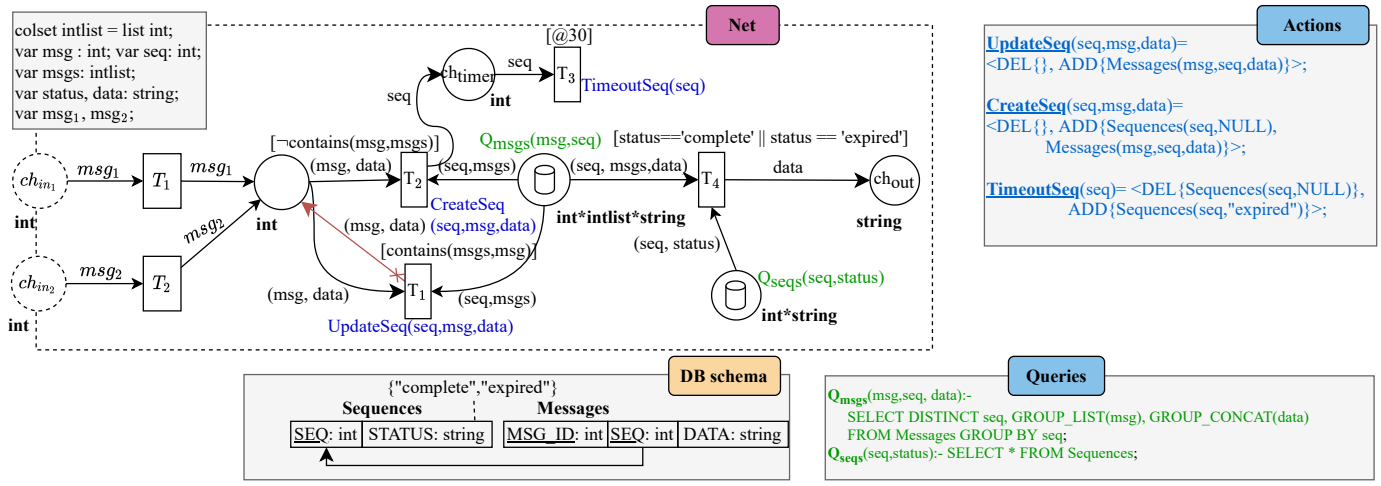


Figure 10: Sequential composition of timed db-net realizations of a join router and an aggregator

Note that the new colour assignment is well-defined on the quotient $(P_A \uplus P_B)/\sim$ since the two constituent nets have compatible boundaries, by assumption.

Example 7. *In Fig. 10, we see the sequential composition JoinRouter ⨟ Aggregator of the timed db-net realisation of a join router followed by an aggregator. The two nets have been glued together at their common boundary.*

Composition of nets behaves as expected: it is associative, and there is an identity net which is a unit for composition. Furthermore, the parallel and sequential compositions interact well, in the sense that we get the same result no matter if we first compose in parallel and then sequentially, or the other way around. All in all, this means that nets with boundaries are the morphisms of a strict monoidal category [48]:

Lemma 1. *For any open timed db-nets N, M, K with compatible boundaries, we have $N \fatsemi (M \fatsemi K) = (N \fatsemi M) \fatsemi K$, and for each boundary configuration $\vec{c} = [c_1, \ldots, c_n]$, there is an identity net $\mathsf{id}_{\vec{c}} : [c_1, \ldots, c_n] \to [c_1, \ldots, c_n]$ such that $\mathsf{id}_{\vec{c}} \fatsemi N = N$ and $\mathsf{id}_{\vec{c}} \fatsemi M = M$ for every M, N with compatible boundaries. Furthermore, for every N, M, K, we have $N \otimes (M \otimes K) = (N \otimes M) \otimes K$, and for compatible nets $N \fatsemi (M \otimes K) = (N \fatsemi M) \otimes (N \fatsemi K)$.*

Proof. Associativity for both $\fatsemi$ and $\otimes$ is straightforward. The identity net for $[c_1, \ldots, c_n]$ is the net with exactly n places $x_1, \ldots, x_n$ with $\texttt{color}(x_i) = c_i$, that are all both input and output boundaries. □

In particular, the lemma implies that we can compose nets sequentially and in parallel without worrying about how to bracket the compositions [48].

5.4. CPN Tools Prototype

We prototypically implemented our formalism so as to experiment with pattern compositions via simulation, following the idea described in [6, Sect. 5]. We have chosen CPN Tools v4.0.1[2] for modeling and simulation. As compared to other PN tools like Renew v2.5[3], CPN tools supports third-party extensions that can address the persistence and data logic layers of our formalism. Moreover, CPN Tools handles sophisticated simulation tasks over models that use the deployed extensions. To support db-nets, our extension[4]—denoting an unpublished part of the first author's PhD thesis [49]—adds support for defining view places together with corresponding SQL queries as well as actions, and realizes the full execution semantics of db-nets using Java and a PostgreSQL database.

6. Interpreting IPCGs as Open Timed DB-nets

In this section we define the interpretation of integration pattern contract graphs as timed db-nets with boundaries.

6.1. Interpretation of Single Patterns

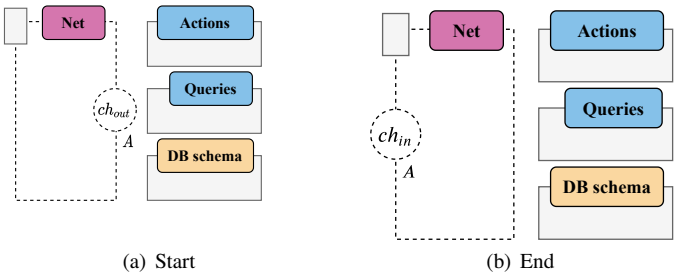
We assign an open timed db-net $[\![p]\!]$ for every node p in a integration pattern contract graph. Recall that an integration pattern contract graph has input and output contracts $inContr : \prod_{p \in P} (CPT \times 2^{EL})^{|\bullet p|}$ and $outContr : \prod_{p \in P} (CPT \times 2^{EL})^{|p \bullet|}$ respectively. If the cardinality of p is $k : m$, then the open timed db-net will be of the form

$$[\![p]\!] : \bigotimes_{i=1}^{k} inContr_i(p)_{EL} \to \bigotimes_{j=1}^{m} outContr_j(p)_{EL}$$

This incorporates the data elements of the input and output contracts into the boundary of the timed db-net, since these are essential for the dataflow of the net. In Sec. 6.2.1, we will also incorporate the remaining concepts from the contracts such as signatures, encryption and encodings into the interpretation.

The shape of the timed db-net $[\![p]\!]$ depends on $type(p)$ only, i.e., we give one interpretation for each pattern type:

[2] CPN Tools, visited 5/23: `https://cpntools.org/`
[3] Renew, visited 5/2023: `http://www.renew.de/`
[4] CPN Tools extension for timed db-net with boundaries and pattern models (i.e., mainly $*boundary*.cpn$, $*fusion*.cpn$) is available for download, visited 5/2023: `https://github.com/dritter-hd/db-net-eip-patterns`.



(a) Start (b) End

Figure 11: Start and end patterns

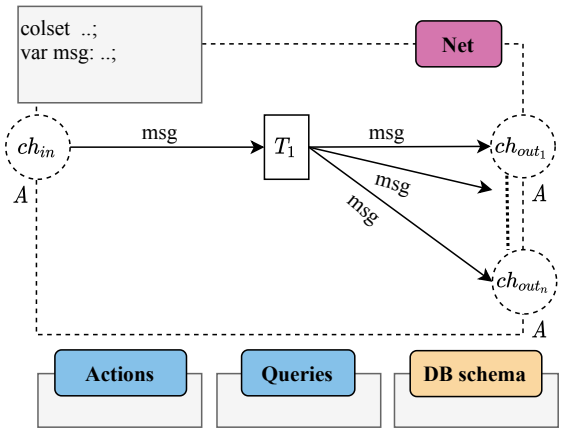


Figure 12: Interpretation of an unconditional fork pattern.

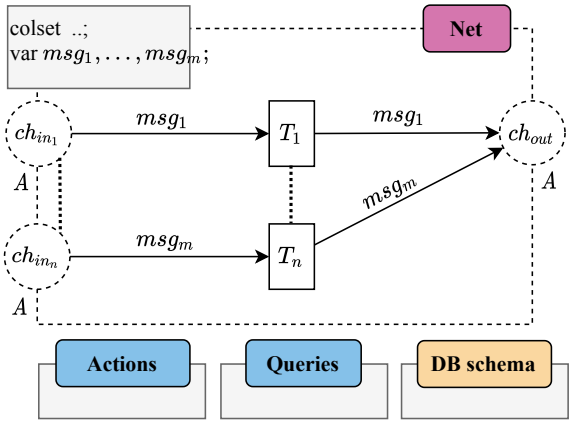


Figure 13: Interpretation of an unconditional join pattern

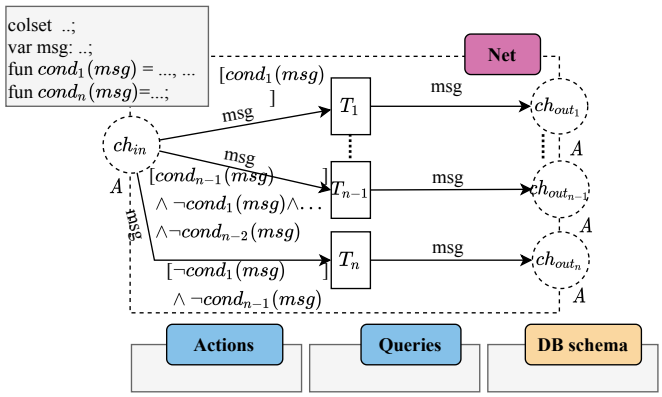


Figure 14: Interpretation of a conditional fork pattern

Start and end pattern types. We interpret a start pattern p_{start} as the open timed db-net $[\![p_{\text{start}}]\!] : I \to \text{color}_{\text{out}}(p_{\text{start}})$ shown in Fig. 11(a). Similarily, Fig. 11(b) shows the interpretation of an end pattern p_{end} as an open timed db-net $[\![p_{\text{end}}]\!] : \text{color}_{\text{in}}(p_{\text{end}}) \to I$.

Non-conditional fork patterns. We interpret a non-conditional fork pattern p_{fork} with cardinality $1 : n$ as the open timed db-net $[\![p_{\text{fork}}]\!] : \text{color}_{\text{in}}(p_{\text{fork}}) \to \bigotimes_{j=1}^{n} \text{color}_{\text{out}}(p_{\text{fork}})_j$ shown in Fig. 12.

Non-conditional join patterns. We interpret a non-conditional join pattern p_{join} with cardinality $m : 1$ as the open timed db-net $[\![p_{\text{join}}]\!] : \bigotimes_{j=1}^{m} \text{color}_{\text{in}}(p_{\text{join}})_j \to \text{color}_{\text{out}}(p_{\text{join}})$ shown in Fig. 13.

Conditional fork patterns. We interpret a conditional fork pattern p_{cfork} of cardinality $1 : n$ with conditions $cond_1, \ldots, cond_{n-1}$ in its pattern characteristic assignment as the open timed db-net $[\![p_{\text{cfork}}]\!] : \text{color}_{\text{in}}(p_{\text{cfork}}) \to \bigotimes_{j=1}^{n} \text{color}_{\text{out}}(p_{\text{cfork}})_j$ shown in Fig. 14. Note that net is constructed so that the conditions are evaluated in order — the transition corresponding to condition k will only fire if condition k is true, and conditions $1, \ldots, k-1$ are false. The last transition will fire if all conditions evaluate to false.

Message processor patterns. We interpret a message processor pattern p_{mp} with storage schema S, actions A, query Q, condition $cond$, time τ and program f in its pattern characteristic assignment as the open timed db-net $[\![p_{\text{mp}}]\!] : \text{color}_{\text{in}}(p_{\text{mp}}) \to \text{color}_{\text{out}}(p_{\text{mp}})$ shown in Fig. 15. If the condition $cond$ and timing window τ are satisfied, the incoming message possibly gets enriched by data from the query Q and action A might be triggered, before the program f transforms the data into possibly multiple messages, collected in a list. These get emitted one by one.

Of course, not all features need to be used by all message processor patterns (e.g., no storage for control-time delayer).

Merge patterns. We interpret a merge pattern p_{merge} with aggregation function f and timeout τ as the open timed db-net $[\![p_{\text{merge}}]\!] : \text{color}_{\text{in}}(p_{\text{merge}}) \to \text{color}_{\text{out}}(p_{\text{merge}})$ shown in Fig. 16, where `contains(msg, msgs)` is defined to be the function that checks if `msg` occurs in the list `msgs`. Briefly, the net works as follows: the first message in a sequence makes transition $T1$ fire, which creates a new database record for the sequence, and starts a timer. Each subsequent message from the same sequence gets stored in the database using transition $T2$, until τ seconds has elapsed, which will fire transition $T3$. The action associated to $T3$ will make the condition for the Aggregate transition true, which will retrieve all messages $msgs$ and then put $f(msgs)$ in the output place of the net.

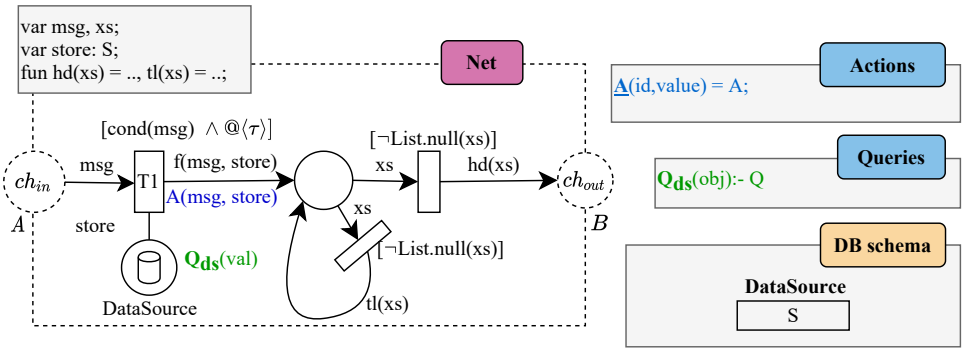


Figure 15: Interpretation of a message processor pattern.

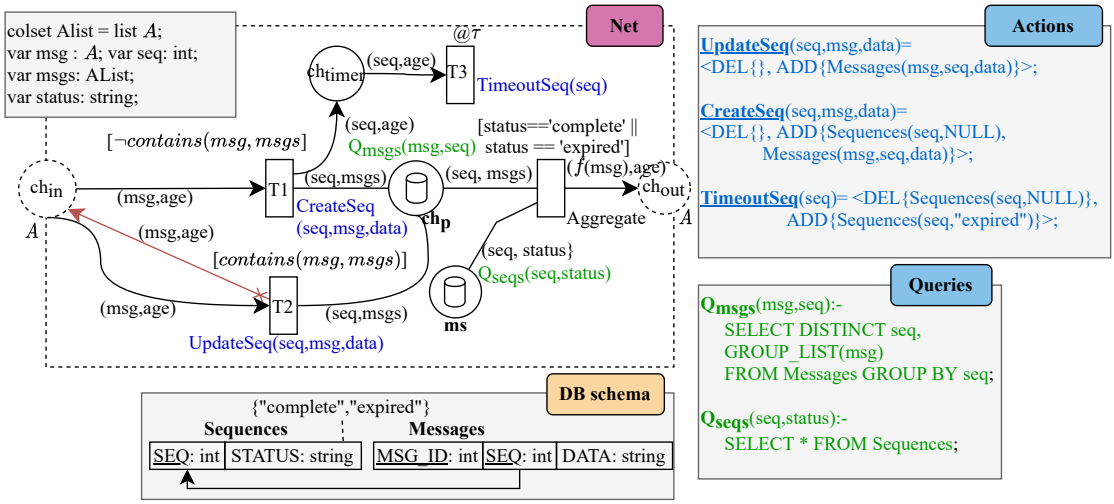


Figure 16: Interpretation of a merge pattern

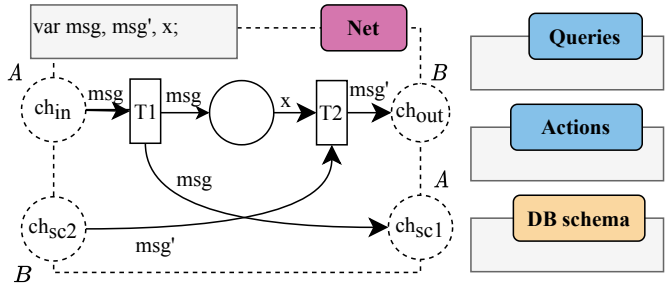


Figure 17: Interpretation of an external call pattern.

External call patterns. We interpret an external call pattern p_{call} to the timed db-net with boundaries $[\![p_{\text{call}}]\!] : A \otimes B \to B \otimes A$ shown in Fig. 17, where the boundary ports ch_{sc1} and ch_{sc2} are meant to be plugged into the pattern representing the external process called. Token x feeding into transition $T2$ ensures that the external process does not inject unasked for messages into the pattern.

6.2. *Interpreting integration pattern contract graphs*

We now show how to interpret not just individual nodes from an integration pattern contract graph, but how to also take edges into account. We first enrich the interpretations of single patterns with transitions and guards to enable and enforce the concepts from output and input contracts in Sec. 6.2.1, and then prove that composing the interpretation of individual patterns according to how they are connected in the graph gives rise to a well-formed timed db-net in Sec. 6.2.2.

6.2.1. *Taking contract concepts into account*

Recall that a pattern contract also represents concepts, i.e., properties of the exchanged data, such as if a pattern is able to process or produce signed, encrypted or encoded data. A message can only be sent from one pattern to another if their contracts match, i.e., if they agree on these properties. To reflect this in the timed db-nets semantics, we enrich all colorsets to also keep track of this information: given a place P with colorset C, we construct the colorset $C \times \{yes, no\}^3$, where the color $(x, b_{\text{sign}}, b_{\text{encr}}, b_{\text{enc}})$ is intended to mean that the data value x is respectively signed, encrypted and encoded or not according to the yes/no values b_{sign}, b_{encr}, and b_{enc}. To enforce the contracts, we also make sure that every token entering an input place c_{in} is guarded according to the input contract by creating a new place ch'_{in} and a new transition from ch'_{in} to ch_{in}, which conditionally forwards tokens whose properties match the contract. The new place ch'_{in} replaces ch_{in} as an input place. Dually, for each output

place ch_{out} we create a new place ch'_{out} and a new transition from ch_{out} to ch'_{out} which ensures that all tokens satisfy the output contract, via a new transition for each combination of output contract values. The new place ch'_{out} replaces ch_{out} as an output place. Formally, the construction is as follows:

Definition 10. *Let $\mathcal{X} : \otimes_{i<m} c_i \to \otimes_{i<n} c'_i$ be an open timed db-net, and $\vec{C} = IC_1, \ldots, IC_m, OC_1, \ldots, OC_n$ be a list of integration concepts with $IC_i, OC_j \in CPT$. Define the open timed db-net*

$$\mathcal{X}_{CPT(\vec{C})} : \otimes_{i<m}(c_i \times \{yes, no\}^3) \to \otimes_{i<n}(c'_i \times \{yes, no\}^3)$$

with the same type domains, persistence layers and data logic layers as $\mathcal{X}$, but with control layer

$$\mathcal{N}' = (P', T', F'_{in}, F'_{out}, F'_{rb}, \texttt{color}', \texttt{query}, \texttt{guard}', \texttt{action}')$$

with

$$P' = P \uplus \{ch'_{in,1}, \ldots, ch'_{in,m}, ch'_{out,1}, \ldots, ch'_{out,1}\}$$
$$T' = T \uplus T_{in} \uplus T_{out}$$

where $T_{in} = \{t_{in,1}, \ldots, t_{in,m},\}$ and

$$T_{out} = \{t_{out,1,\vec{b}}, \ldots, t_{out,n,\vec{b}} \mid \vec{b} \in \{yes, no\}^3\}$$

$$F'_{in}(x) = \begin{cases} F_{in}(p,t) & \text{if } x = (\mathsf{in}_P(p), \mathsf{in}_T(t)) \\ \{(y, y_{sign}, y_{encr}, y_{enc})\} & \text{if } x = (ch_{in,i}, t_{in,i}) \\ \{y\} & \text{if } x = (ch_{out,j}, t_{out,j,\vec{b}}) \\ & \text{and } \vec{b} = (b_{sign}, b_{encr}, b_{enc}) \\ & \text{with } (OC_j)_{CPT}(p) \in \{b_p, any\} \\ \emptyset & \text{otherwise} \end{cases}$$

$$F'_{out}(x) = \begin{cases} F_{out}(p,t) & \text{if } x = (\mathsf{in}_P(p), \mathsf{in}_T(t)) \\ \{y\} & \text{if } x = (ch_{in,i}, t_{in,i}) \\ \{(y, \vec{b})\} & \text{if } x = (ch_{out,j}, t_{out,j,\vec{b}}) \\ & \text{and } \vec{b} = (b_{sign}, b_{encr}, b_{enc}) \\ & \text{with } (OC_j)_{CPT}(p) \in \{b_p, any\} \\ \emptyset & \text{otherwise} \end{cases}$$

$$F'_{rb}(x) = \begin{cases} F_{rb}(p,t) & \text{if } x = (\mathsf{in}_P(p), \mathsf{in}_T(t)) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\texttt{guard}'(\mathsf{in}_T(t)) = \texttt{guard}(t)$$
$$\texttt{guard}'(t_{in,i}) = \bigwedge_{\{p \mid (IC_i)_{CPT}(p) \neq any\}} y_p = (IC_i)_{CPT}(p)$$
$$\texttt{guard}'(t_{out,j}) = \top$$
$$\texttt{action}' = [\texttt{action}, t_i \mapsto -, t'_j \mapsto -]$$
$$\tau' = [\tau, t_i \mapsto [0, \infty], t'_j \mapsto [0, \infty]]$$

The pattern contract construction in Definition 10, can again be realized as template translation on an inter pattern level, as shown in Fig. 18. On the input side, a token token $(y, b_{\text{sign}}, b_{\text{encr}}, b_{\text{enc}})$ only enables the transition $t_{\text{in},i}$ if $(b_{\text{sign}}, b_{\text{encr}}, b_{\text{enc}})$ fulfils the input contract IC_i, in which case the metadata is stripped and only the message y passed to the actual pattern. On the output side, any of the boundary transitions $t_{\text{out},i,(b_{\text{sign}},b_{\text{encr}},b_{\text{enc}})}$ may fire and enrich the data y with metadata $(b_{\text{sign}}, b_{\text{encr}}, b_{\text{enc}})$, ready to be passed to the next pattern.

Let us consider two examples to gain an understanding of the construction.

Example 8. *Figure 19 shows the translation of a message translator pattern MT with input contract $\{(ENC, no), (ENCR, no), (SIGN, any)\}$ and output contract $\{(ENC, no), (ENCR, no), (SIGN, any)\}$. The input transition T' hence checks the guard $[x, no, any, no]$, and if it matches, the token is forwarded to the actual message translator. After the transformation, the resulting message msg' is not encrypted, the signing is invalid, and not encoded, and thus emits (x, no, no, no).*

Example 9. *A join router structurally combines many incoming to one outgoing message channel without accessing the data. Consequently, both input and output contracts have any for all properties. Figure 20 shows the result of the boundary construction for the join router. The input boundary does not enforce CPT constraints, and thus no guards are defined for the transitions. The output boundary, however, supplies all 8 different permutations of {yes,no} for the three CPT properties.*

6.2.2. Synchronising Pattern Compositions and correctness of the translation

We are now in a position to define the full translation of a correct integration pattern contract graph G. For the translation to be well-defined, we need only data element correctness of the graph. Concept correctness is used to show that in the nets in the image of the translation, tokens can always flow from the translation of the start node to the translation of the end node.

Theorem 1. *Let a correct integration pattern contract graph G be given. For each node p, consider the timed db-net*

$$[\![p]\!]_{CPT(inContr(p),outContr(p))} : \bigotimes_{i=1}^{k} color_{in}(p)_i \to \bigotimes_{j=1}^{m} color_{out}(p)_j$$

Use the graphical language [48] enabled by Lemma 1 to compose these nets according to the edges of the graph. The resulting timed db-net is then well-defined, and has the option to complete, i.e., from each marking reachable from a marking with a token in some input place, it is possible to reach a marking with a token in an output place.

Proof. Since the graph is assumed to be correct, all input contracts match the output contracts of the nets composed with it, which by the data element correctness means that the boundary configurations match, so that the result is well-defined.

To see that the constructed net also has the option to complete, first note that the interpretations of basic patterns in

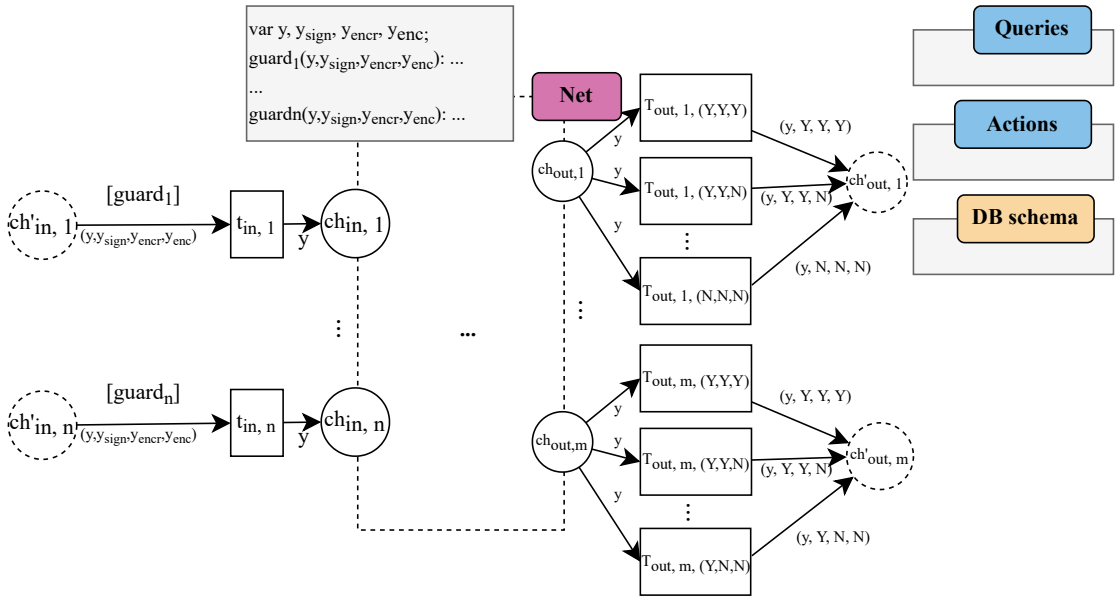


Figure 18: Boundary construction template.

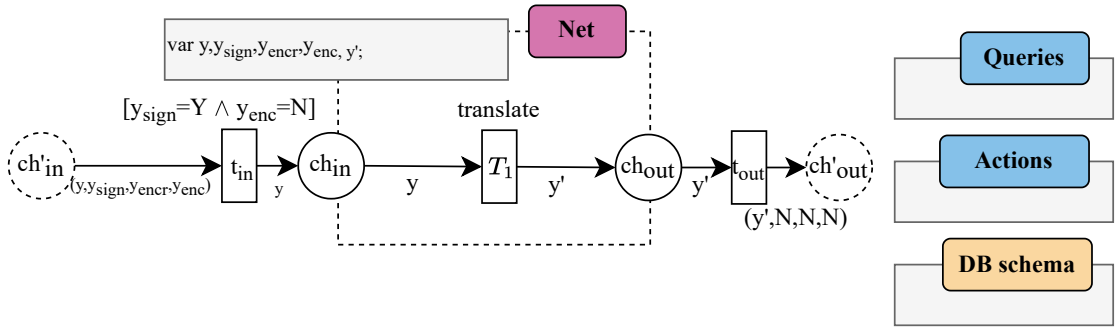


Figure 19: Example: message translator construction

Sec. 6.1 do (in particular, one transition is always enabled in the translation of a conditional fork pattern in Fig. 14, and the aggregate transition will always be enabled after the timeout in the translation of a merge pattern in Fig. 16). By the way the interpretation is defined, all that remains to show is that if N and N' have the option to complete, then so does $N_{CPT(\vec{C})} \circ N'_{CPT(\vec{C}')}$, if the contracts $\vec{C}$ and $\vec{C}'$ match. Assume a marking with a token in an input place of N'. Since N' has the option to complete, a marking with a token in an output place of N' is reachable, and since the contracts match, this token will satisfy the guard imposed by the $N_{CPT(\vec{C})}$ construction. Hence a marking with a token in an input place of N is reachable, and the statement follows, as N has the option to complete. □

6.3. Discussion

Solely giving an interpretation of pattern compositions as timed db-nets does not guarantee correctness. However, Theorem 1 gives confidence in the translation itself, as it shows that the output of the translation is structurally well-behaved (i.e., input cardinalities match output cardinalities), and also semantically well-behaved, in the sense that tokens flowing through the resulting timed db-net cannot get "stuck" (i.e., having the option to complete). Note that having a translation targeting timed db-nets means that one can now formulate formal conjectures and prove them, perhaps for classes of integration patterns.

7. Optimization Strategy Realization

In this section we formally define the optimizations from the different strategies identified in Tab. 2 in the form of a rule-based graph rewriting system (addressing REQ-7). This gives a formal framework in which different optimizations can be compared. We begin by describing the graph rewriting framework, and subsequently apply it to define the optimizations.

7.1. Graph Rewriting

Graph rewriting provides a visual framework for transforming graphs in a rule-based fashion. A graph rewriting rule is given by two embeddings of graphs $L \hookleftarrow K \hookrightarrow R$, where L represents the left hand side of the rewrite rule, R the right hand side, and K their intersection (the parts of the graph that should be preserved by the rule). A rewrite rule can be applied to a

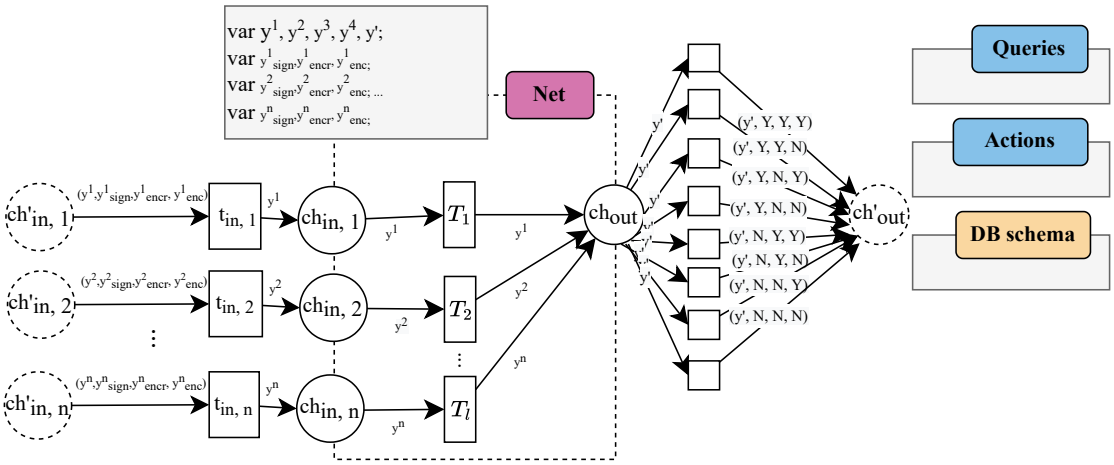
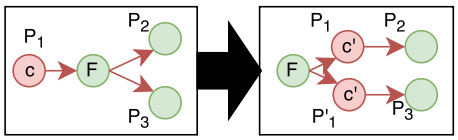


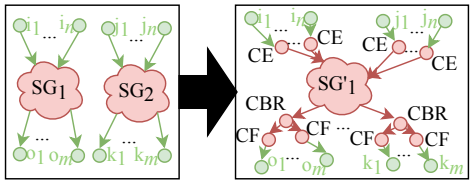
Figure 20: Join router construction

graph G after a match of L in G has been given as an embedding $L \hookrightarrow G$; this replaces the match of L in G by R. The application of a rule is potentially non-deterministic: several distinct matches can be possible [50]. Visually, we represent a rewrite rule by a left hand side and a right hand side graph colored green and red: green parts are shared and represent K, while the red parts are to be deleted in the left hand side, and inserted in the right hand side respectively. For instance, the following rewrite rule moves the node P_1 past a fork by making a copy in each branch, changing its label from c to c' in the process:

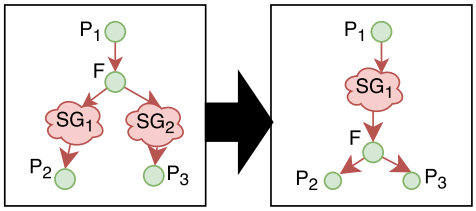


Formally, the rewritten graph is constructed using a double-pushout (DPO) [51] from category theory. We use DPO rewriting since rule applications are side-effect free (e.g., no "dangling" edges) and local (i.e., all graph changes are described by the rules). We additionally use Habel and Plump's relabeling DPO extension [52] to facilitate the relabeling of nodes in partially labeled graphs. In Fig. 7, we showed contracts and characteristics in dashed boxes, but in the rules that follow, we will represent them as (schematic) labels inside the nodes for space reasons.

In addition, we also consider rewrite rules parameterized by graphs, where we draw the parameter graph as a cloud (see e.g., Fig. 21(a) for an example). A cloud represents any graph, sometimes with some side-conditions that are stated together with the rule. When looking for a match in a given graph G, it is of course sufficient to instantiate clouds with subgraphs of G — this way, we can reduce the infinite number of rules that a parameterized rewrite rule represents to a finite number. Parameterized rewrite rules can formally be represented using substitution of hypergraphs [53] or by !-boxes in open graphs [54]. Since we describe optimization strategies as graph rewrite rules, we can be flexible with when and in what order we apply the strategies. We apply the rules repeatedly until a fixed point is reached, i.e., when no further changes are possible, making the process idempotent. Each rule application preserves IPCG correctness in the sense of Definition 5, because input contracts do not get more specific, and output contracts remain the same. Methodologically, the rules are specified by pre-conditions, change primitives, post-conditions and an optimization effect, where the pre- and post-conditions are implicit in the applicability and result of the rewriting rule.



(a) Redundant sub-process



(b) Combine sibling patterns

Figure 21: Rules for redundant sub-process and combine sibling patterns.

7.2. OS-1: Process Simplification

We first consider the process simplification strategies from Sec. 3.2 OS-1 to OS-3 that mainly strive to reduce the model complexity and latency.

7.2.1. Redundant sub-process

This optimization removes redundant copies of the same sub-process within a process.

Change primitives: The rewriting is given by the rule in Fig. 21(a), where $SG1$ and $SG2$ are isomorphic pattern graphs with in-degree n and out-degree m. The Content Enricher (CE) node is

a message processor pattern from Fig. 15 with a pattern characteristic $(PRG, (\text{addCtxt}, [0, \infty)))$ for an enrichment program addCtxt which is used to add content to the message (does it come from the left or right subgraph?). Similarly, the Content Filter (CF) is a message processor, with a pattern characteristic $(PRG, (\text{removeCtxt}, [0, \infty)))$ for an enrichment program removeCtxt which is used to remove the added content from the message again. Moreover, the Content-based Router (CBR) node is a conditional fork pattern from Fig. 14 with a pattern characteristic $(CND, \{\text{fromLeft?}\})$ for a condition fromLeft? which is used to route messages depending on their added context. In the right hand side of the rule, the CE nodes add the context of the predecessor node to the message in the form of a content enricher pattern, and the CBR nodes are content-based routers that route the message to the correct recipient based on the context introduced by CE. The graph SG'_1 is the same as SG_1, but with the context introduced by CE copied along everywhere. This context is stripped off the message by a content filter CF.

Effect: The optimization is beneficial for model complexity when the isomorphic subgraphs contain more than $n + m$ nodes, where n is the in-degree and m the out-degree of the isomorphic subgraphs. The latency reduction is by the factor of subgraphs minus the latency introduced by the additional n CE nodes, m CBR nodes and $2m$ CF nodes.

7.2.2. Combine sibling patterns

Sibling patterns have the same parent node in the pattern graph (e.g., they follow a non-conditional forking pattern) with channel cardinality of 1:1. Combining them means that only one copy of a message is traveling through the graph instead of two — for this transformation to be correct in general, the siblings also need to be side-effect free, i.e., no external calls.

Change primitives: The rule is given in Fig. 21(b), where SG_1 and SG_2 are isomorphic side-effect free pattern graphs, and F is a fork.

Effect: The model complexity and latency are reduced by the model complexity and latency of SG_2.

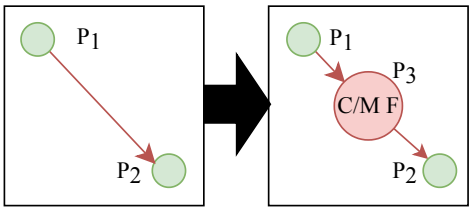
7.3. OS-2: Data Reduction

Now, we consider data reduction optimization strategies, which mainly target improvements of the message throughput (incl. reducing element cardinalities). These optimizations require that pattern input and output contracts are regularly updated with snapshots of element data sets EL_{in} and EL_{out} from live systems, e.g., from experimental measurements through benchmarks [55].

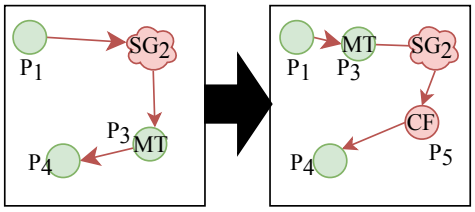
7.3.1. Early-Filter

A filter pattern can be moved to or inserted prior to some of its successors to reduce the data to be processed. The following types of filters have to be differentiated:

- A *message filter* removes messages with invalid or incomplete content. It can be used to prevent exceptional situations, and thus improves stability.



(a) Early Filter



(b) Early Mapping

Figure 22: Rules for early-filter and early-mapping.

- A *content filter* removes elements from messages, thus reduces the amount of data passed to subsequent patterns.

Both patterns are message processors in the sense of Fig. 15. The content filter assigns a filter function $(prg_1, ([0, \infty)) \rightarrow f(msg, value)$ to remove data from the message (i.e., without temporal information), and the message filter assigns a filter condition $\{(cond_1)\} \rightarrow g(msg)$.

Change primitives: The rule is given in Fig. 22(a), where P_3 is either a content or message filter matching the output contracts of P_1 and the input contract of P_2, removing the data not used by P_2.

Effect: Message throughput increases by the ratio of the number of reduced elements that are processed per second, unless limited by the throughput of the additional pattern.

7.3.2. Early-Mapping

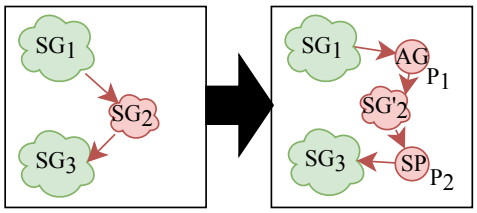
A mapping that reduces the number of elements in a message can increase the message throughput.

Change primitives: The rule is given in Fig. 22(b), where P_3 is an element reducing message mapping compatible with both SG_2, P_4, and P_1, SG_2, and where P_4 does not modify the elements mentioned in the output contract of P_3. Furthermore P_5 is a content filter, which ensures that the input contract of P_4 is satisfied. The Message Translator (MT) node is a message processor pattern from Fig. 15 with a pattern characteristic $(PRG, (prg, [0, \infty)))$ for some program prg which is used to transform the message.

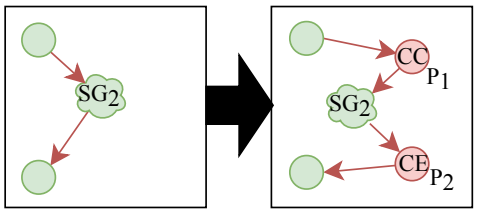
Effect: The message throughput for the subgraph subsequent to the mapping increases by the ratio of the number of unnecessary data elements processed.

7.3.3. Early-Aggregation

A micro-batch processing region is a subgraph which contains patterns that are able to process multiple messages combined to a multi-message [31] or one message with multiple segments with an increased message throughput. The optimal

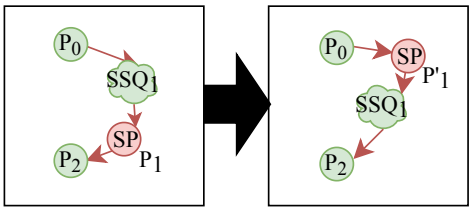


(a) Early-Aggregation

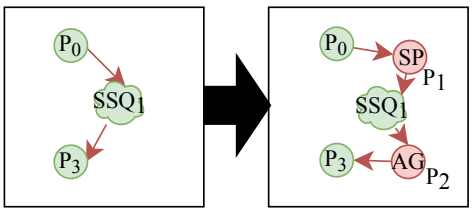


(b) Early-Claim Check

Figure 23: Rules for early-aggregation and early-claim check



(a) Early Split



(b) Early Split (inserted)

Figure 24: Rules for early split.

number of aggregations is determined by the highest batch-size for the throughput ratio of the pattern with the lowest throughput, if latency is not considered.

Change primitives: The rule is given in Fig. 23(a), where SG_2 is a micro-batch processing region, P_1 an aggregator, P_2 a splitter which separates the batch entries to distinct messages to reverse the aggregation, and SG'_2 finally is SG_2 modified to process micro-batched messages. The Aggregator (AG) node is a merge pattern from Fig. 16 with a pattern characteristic $\{(CND, \{cnd_{cr}, cnd_{cc}\}), (PRG, prg_{agg}, (v_1, v_2)\}$ for some correlation condition cnd_{cr}, completion condition cnd_{cc}, aggregation function prg_{agg}, and timeout interval (v_1, v_2). The Splitter (SP) node is a message processor from Fig. 15 with a pattern characteristic $(PRG, (prg, [0, \infty)))$ for some split function prg which is used to split the message into several ones.

Effect: The message throughput is the minimal pattern throughput of all patterns in the micro-batch processing region. If the region is followed by patterns with less throughput, only the overall latency might be improved.

7.3.4. Early Claim Check

If a subgraph does not contain a pattern with message access, the message payload can be stored intermediately persistently or transiently (depending on the quality of service level) and not moved through the subgraph. For instance, this applies to subgraphs consisting of data independent control-flow logic only, or those that operate entirely on the message header (e.g., header routing).

Change primitives: The rule is given in Fig. 23(b), where SG_2 is a message access-free subgraph, P_1 a claim check that stores the message payload and adds a claim to the message properties (and possibly routing information to the message header), and P_2 a content enricher that adds the original payload to the message. The Claim Check (CC) node is a message processor from Fig. 15 with a pattern characteristic $(PRG, (\_, [0, \infty)))$, which stores the message for later retrieval.

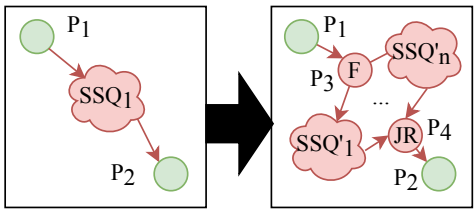
Effect: The main memory consumption and CPU load decreases, which could increase the message throughput of SG_2, if the claim check and content enricher pattern throughput is greater than or equal to the improved throughput of each of the patterns in the subgraph.

7.3.5. Early-Split

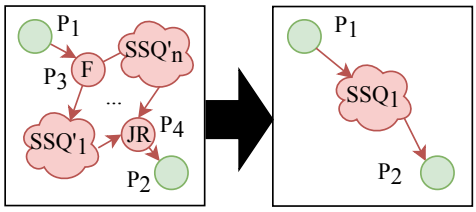
Messages with many segments can be reduced to several messages with fewer segments, and thereby reducing the processing per message. A segment is an iterable part of a message, such as a list of elements. When such a message grows bigger, the message throughput of a set of adjacent patterns might decrease, compared to the expected performance for a single segment; a phenomenon called *segment bottleneck sub-sequence*. Algorithmically, such bottlenecks can be found using max flow-min cut techniques based on workload statistics. The splitter (SP) node is a message processor from Fig. 15 with a pattern characteristic $(PRG, (prg, [0, \infty)))$, for some split program prg.

Change primitives: The rule is given in Fig. 24, where SSQ_1 is a segment bottleneck sub-sequence. If SSQ_1 already has an adjacent splitter, Fig. 24(a) applies, otherwise Fig. 24(b). In the latter case, SP is a splitter and P_2 is an aggregator that re-builds the required segments for the successor in SG_2. For an already existing splitter P_1 in Fig. 24(a), the split condition has to be adjusted to the elements required by the input contract of the subsequent pattern in SSQ_1. In both cases we assume that the patterns in SSQ_1 deal with single- and multi-segment messages; otherwise all patterns have to be adjusted.

Effect: The message throughput increases by the ratio of reduced number of message segments per message, if the throughput of the moved / added splitter (and aggregator) $\geq$ throughput of each of the patterns in the segment bottleneck sub-sequence after the segment reduction.



(a) Sequence to parallel



(b) Merge parallel

Figure 25: Rules for sequence to parallel variants.

7.4. OS-3: Parallelization

Parallelization optimization strategies increase message throughput, and again require experimentally measured message throughput statistics, e.g., from benchmarks [55].

7.4.1. Sequence to parallel

A bottleneck sub-sequence with channel cardinality 1:1 can also be handled by distributing its input and replicating its logic. The parallelization factor is the average message throughput of the predecessor and successor of the sequence divided by two, which denotes the improvement potential of the bottleneck sub-sequence. The goal is to not overachieve the mean of predecessor and successor throughput with the improvement to avoid iterative re-optimization. Hence the optimization is only executed, if the parallel sub-sequence reaches lower throughput than their minimum.

Change primitives: The rule is given in Fig. 25(a), where SSQ_1 is a bottleneck sub-sequence, P_2 a fork node, P_3 a join router, and each SSQ'_k is a copy of SSQ_1, for $1 \leq k \leq n$. The parallelization factor n is a parameter of the rule.

Effect: The message throughput improvement rate depends on the parallelization factor n, and the message throughput of the balancing fork and join router on the runtime. For a measured throughput t of the bottleneck sub-sequences, the throughput can be improved to $n \times t \leq$ average of the sums of the predecessor and successor throughput, which is limited by the upper boundary of the balancing fork or join router.

7.4.2. Merge parallel

The balancing fork and join router realizations can limit the throughput in some runtime systems, so that a parallelization decreases the throughput, e.g., when a fork or a join has smaller throughput than a pattern in the following sub-sequence.

Change primitives: The rule is given in Fig. 25(b), where P_3 and P_4 limit the message throughput of each of the n sub-sequence copies $SSQ'_1, \ldots, SSQ'_n$ of SSQ_1.

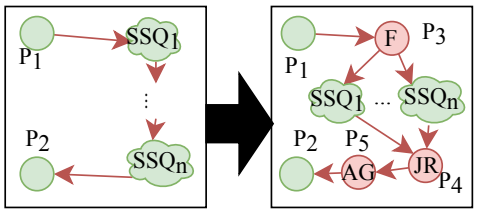


Figure 26: Heterogeneous sequence to parallel.

Effect: The model complexity is reduced by $(n-1)k-2$, where each SSQ'_i contains k nodes. The message throughput might improve, since the transformation lifts the limiting upper boundary of a badly performing balancing fork or join router implementations to the lowest pattern throughput in the bottleneck sub-sequence.

7.4.3. Heterogeneous Parallelization

A heterogeneous parallelization consists of parallel sub-sequences that are not isomorphic. In general, two subsequent patterns P_i and P_j can be parallelized, if the predecessor pattern of P_i fulfills the input contract of P_j, P_i behaves read-only with respect to the data element set of P_j, and the combined outbound contracts of P_i and P_j fulfill the input contract of the successor pattern of P_j.

Change primitives: The rule is given in Fig. 26, where the sequential sub-sequence parts SSQ_1, .., SSQ_n are side-effect free and can be parallelized, P_3 is a parallel fork, P_4 is a join router, and P_5 is an aggregator that waits for messages from all sub-sequence branches before emitting a combined message that fulfills the input contract of P_2.

Effect: Synchronization latency can be improved, but the model complexity increases by 3. The latency improves from the sum of the sequential pattern latencies to the maximal latency of all sub-sequence parts plus the fork, join, and aggregator latencies.

7.5. OS-4: Pattern Placement

All of the data reduction optimizations discussed in Sec. 7.3 can be applied in OS-4, i.e., "Pushdown to Endpoint", by extending the placement to the message endpoints, with contracts similar to our definition. However, due to our focus on the integration processes, we will not further elaborate on it in this work.

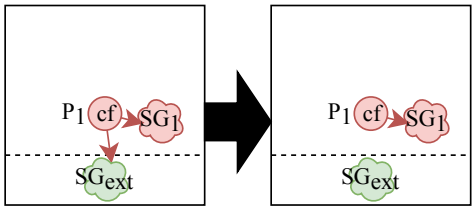
7.6. OS-5: Reduce Interaction

Optimization strategies that reduce interactions target a more resilient behavior of an integration process.

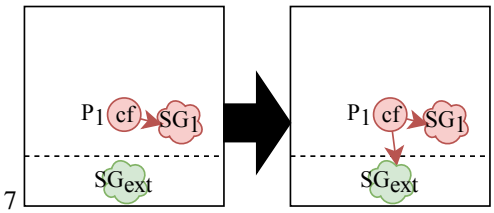
7.6.1. Ignore Failing Endpoints

When endpoints fail, different exceptional situations have to be handled on the caller side. This can come with long timeouts, which can block the caller and increase latency. Knowing that an endpoint is unreliable can speed up processing, by immediately falling back to an alternative.

Change primitives: The rule is given in Fig. 27(a), where SG_{ext} is a failing endpoint, SG_1 and SG_2 subgraphs, and P_1 is a

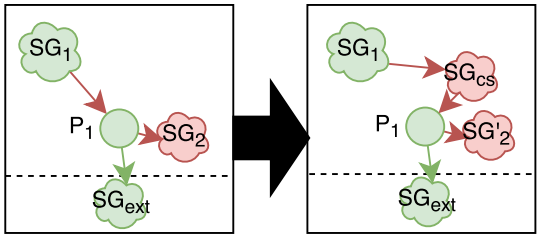


(a) Ignore Failing Endpoint



(b) Try Failing Endpoint

Figure 27: Rules for ignore failing endpoints.



(a) Reduce Requests

Figure 28: Rules for reduce requests.

service call or message send pattern with configuration cf. This specifies the collected number of subsequently failed delivery attempts to the endpoint or a configurable time interval. If one of these thresholds is reached, the process stops calling SG_{ext} and does not continue with the usual processing in SG_1, however, invokes an alternative processing or exception handling in SG_2.

Effect: Besides improved latency (i.e., average time to response from endpoint in case of failure), the integration process behaves more stable due to immediate alternative processing. To not exclude the remote endpoint forever, the rule in Fig. 27(b) is scheduled for execution after a period of time to try whether the endpoint is still failing. If not, the configuration is updated to cf' to avoid the execution of Fig. 27(a). The retry time is adjusted depending on experienced values (e.g., endpoint is down every two hours for ten minutes).

7.6.2. Reduce Requests

A *message limited* endpoint, i.e., an endpoint that is not able to handle a high rate of requests, can get unresponsive or fail. To avoid this, the caller can notice this (e.g., by TCP back-pressure) and react by reducing the number or frequency of requests. This can be done by employing a throttling or even sampling pattern [4], which removes messages. An aggregator can also help to combine messages to multi-messages [31].

Change primitives: The rewriting is given by the rule in Fig. 28(a), where P_1 is a service call or message send pattern, SG_{ext} a message limited external endpoint, SG_2 a subgraph with SG'_2 a re-configured copy of SG_2 (e.g., for vectorized message processing [31]), and SG_{cs} a subgraph that reduce the pace, or number of messages sent.

Effect: Latency and message throughput might improve, but this optimization mainly targets stability of communication. This is improved by configuring the caller to a message rate that the receiver can handle.

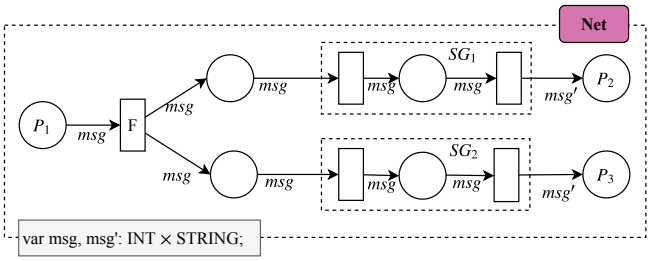
7.7. Optimization Correctness

We now show that the optimizations do not change the input-output behaviour of the pattern graphs in the timed db-nets semantics, i.e., if we have a rewrite rule $G \Rightarrow G'$ (cf. Sec. 7.1), then the constructed timed DB-net with boundaries $[\![G]\!]$ has the same observable behaviour as that of $[\![G']\!]$ (addressing REQ-8). More formally, we mean that the transition systems of the original and the rewritten graphs are bisimilar in a certain sense, as defined in Definition 11. At a high level, this means that G can simulate G' with respect to input-output behaviour, and vice versa. Recall that we associate a labelled transition system to each timed DB-net in Sec. 5.2.

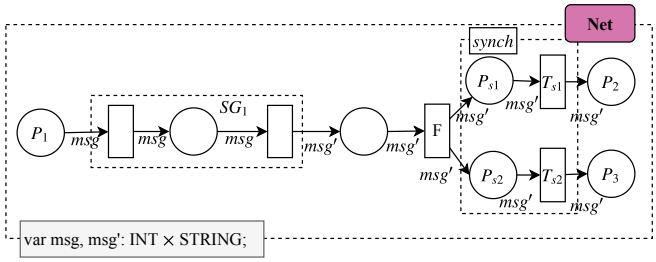
Definition 11 (Functional bisimulation). *Let B and B' be timed DB-nets with equal boundaries, and let $\Gamma^{\mathcal{B}}_{s_0} = \langle S, s_0, \rightarrow \rangle$ and $\Gamma^{\mathcal{B}'}_{s_0} = \langle S', s'_0, \rightarrow' \rangle$ be their associated labelled transition systems. We say that a B-snapshot (I, m) is* functionally equivalent *to a B'-snapshot (I', m'), $(I, m) \approx (I', m')$, if $I = I'$, and m and m' agree on output places except for age variables. Further we say that $\Gamma^{\mathcal{B}}_{s_0}$ is* functionally bisimilar *to $\Gamma^{\mathcal{B}'}_{s_0}$, $\Gamma^{\mathcal{B}}_{s_0} \sim \Gamma^{\mathcal{B}'}_{s_0}$, if whenever $s_0 \rightarrow^* (I, m)$ then there is (I', m') such that $s'_0 \rightarrow'^* (I', m')$, $(I, m) \approx (I', m')$, and $\Gamma^{\mathcal{B}}_{(I,m)} \sim \Gamma^{\mathcal{B}'}_{(I',m')}$, and similarly whenever $s'_0 \rightarrow^* (I', m')$ then there is (I, m) such that $s_0 \rightarrow^* (I, m)$, $(I', m') \approx (I, m)$, and $\Gamma^{\mathcal{B}'}_{(I',m')} \sim \Gamma^{\mathcal{B}}_{(I,m)}$.* ■

The notion of functional bisimulation captures the notion of having the same output behaviour, in the sense that transition system can reach a certain configuration of the output places if and only if the other one can. Note that this definition allows bisimilar transition systems to assign different token ages — what matters is not the exact age value, but that the corresponding transitions are always possible. Let us discuss an explanatory example of bisimulation.

Example 10. *Figure 29 shows the interpretation of a simple IPCG as a timed DB-net before and after applying the rewrite rule for the combining sibling patterns from Fig. 21(b) (for simplicity without boundaries). The improvement of the optimization is to move SG_1 (isomorphic to SG_2) in front of the forking pattern F and leave out SG_2, which reduces the modeling complexity on the right hand side (cf. Fig. 29(b)). The synchronization subnet* synch *is required to show bisimilarity between the original and the resulting net, since tokens might be moved independently in SG_1 and SG_2 before applying the optimisation. The subnet (essentially transitions T_{s1}, T_{s2}) compensates*



(a) Before applying the rewrite rule



(b) After applying the rewrite rule

Figure 29: Timed db-net translation of IPCGs before and after applying the "combine sibling patterns" rewrite rule

for that to ensure that places P_2 and P_3 can be reached independently as well. The timed DB-net B representing Fig. 29(a) and B′ Fig. 29(b) are bisimilar $\Gamma^{\mathcal{B}}_{(I,m)} \sim \Gamma^{\mathcal{B}'}_{(I,m')}$ for any database instance I, and any markings m and m′ with $m(P_1) = m'(P_1)$ and $m(p) = \emptyset = m'(p)$ for all other p. ■

We will need the following basic lemma to show the correctness of our optimizations, i.e., , that the right and left hand sides of the respective optimization rules are bisimilar.

Lemma 2. *The relation $\sim$ is an congruence relation with respect to composition of timed db-nets with boundaries, i.e., it is reflexive, symmetric and transitive, and if if $\Gamma^{\mathcal{B}_1}_{s_0} \sim \Gamma^{\mathcal{B}'_1}_{s_0}$ and $\Gamma^{\mathcal{B}_2}_{s_0} \sim \Gamma^{\mathcal{B}'_2}_{s_0}$ for all s_0 on the shared boundary of B_1 and B_2, then $\Gamma^{\mathcal{B}_1 \circ \mathcal{B}_2}_{s_0} \sim \Gamma^{\mathcal{B}'_1 \circ \mathcal{B}'_2}_{s_0}$.* □

The following lemma means that it makes sense to ask the question if the optimized version of an IPCG is bisimilar to the original IPCG or not.

Lemma 3. *Let G and G′ be IPCGs. For each optimisation rewrite rule $G \Rightarrow G'$, $[\![G]\!]$ and $[\![G']\!]$ have the same boundary.* □

We are now ready to state and prove our correctness theorem.

Theorem 2 (Change Correctness). *Let G and G′ be IPCGs such that $G \Rightarrow G'$ is an optimization rule. For every initial snapshot s_0 of both $[\![G]\!]$ and $[\![G']\!]$, with tokens in input places only, we have $\Gamma^{[\![G]\!]}_{s_0} \sim \Gamma^{[\![G']\!]}_{s_0}$.*

Proof. We verify the statement for each optimization $G \Rightarrow G'$. By Lemma 2, it is enough to show that the parts of the interpretation of the graphs which are actually modified by the rewrite are bisimilar.

Redundant Sub-Processes (Sec. 7.2.1). Each move on the left hand side of the optimization rule in Fig. 21(a) (on Page 20) either moves tokens into a cloud, out of a cloud, or inside a cloud. In the first two cases, this can be simulated by the right hand side by moving the token through the CE or CBR and CF respectively followed by a move into or out of the cloud, while in the latter case the corresponding token can be moved in SG'_1 up to the isomorphism between SG'_1 and the cloud on the left.

Similarly, a move on the right hand side into or out of the cloud can easily be simulated on the left hand side. Suppose a transition fires in SG'_1. Since all guards in SG'_1 have been modified to require all messages to come from the same enriched context, the corresponding transition can either be fired in SG_1 or SG_2.

Combining Sibling Patterns (Sec. 7.2.2). Suppose the left hand side of Fig. 21(b) (on Page 20) takes a finite number of steps and ends up with $m(P_2)$ tokens in P_2 and $m(P_3)$ tokens in P_3. There are three possibilities: (i) there are tokens of the same color in both P_2 and P_3; or (ii) there is a token in P_2 with no matching token in P_3; or (iii) there is a token in P_3 with no matching token in P_2. For the first case, the right hand side can simulate the situation by emulating the steps of the token ending up in P_2, and forking it in the end. For the second case, the right hand side can simulate the situation by emulating the steps of the token ending up in P_2, then forking it, but not moving one copy of the token across the boundary layer in the interpretation of the fork pattern. The third case is similar, using that SG_2 is isomorphic to SG_1.

The right hand side can easily be simulated by copying all moves in SG_1 into simultaneous moves in SG_1 and the isomorphic SG_2.

Early-Filter (Sec. 7.3.1). By construction, the filter removes the data not used by P_2, so if the left hand side of Fig. 22(a) (on Page 21) moves a token to P_2, then the same token can be moved to P_2 on the right hand side and vice versa.

Early-Mapping (Sec. 7.3.2). Suppose the left hand side of Fig. 22(b) (on Page 21) moves a token to P_4. The same transitions can then move the corresponding token to P_4 on the right hand side, with the same payload, by construction. Similarly, the right hand side can be simulated by the left hand side.

Early-Aggregation (Sec. 7.3.3). The interpretation of the subgraph SG_2 is equivalent to the interpretation of P_1 followed by SG'_2 followed by P_3, by construction in Fig. 23(a) (on Page 22), hence the left hand side and the right hand side are equivalent.

Early Claim Check (Sec. 7.3.4). Since the claim check CC + CE in Fig. 23(b) (on Page 22) simply stores the data and then adds it back to the message in the CE step, both sides can simulate each other.

Early-Split (Sec. 7.3.5). By assumption, P_1 followed by SSQ_1 (P_1 followed by SSQ_1 followed by P_2 for the inserted early split in Fig. 24(a) (on Page 22)) is equivalent to SSQ_1 followed by P_1, from which the claim immediately follows.

Sequence to Parallel (Sec. 7.4.1), Merge Parallel (Sec. 7.4.2). The left hand side of Fig. 25(a) (on Page 23) can be simulated by the right hand side by copying each move in SSQ_1 by a move

each in SSQ'_1 to SSQ'_n. If the right hand side moves a token to an output place, it must move a token through some SSQ'_i, and the same moves can move a token through SSQ_1 in the left hand side.

The same reasoning applies to the Merge Parallel transformation in Fig. 25(b), but in reverse.

Heterogeneous Sequence to Parallel (Sec. 7.4.3). By assumption, the sub-sequences SSQ_1 to SSQ_n are side-effect free. The right hand side of Fig. 26 (on Page 23) can simulate the left hand side as follows: if the left hand side moves a token to an output place, it must move it through all of SSQ_1 to SSQ_n. The right hand side can make the same moves in the same order. For the other direction, the left hand side can reorder the moves of the right hand side to first do all moves in SSQ_1, then in SSQ_2 and so on. This is still a valid sequence of steps because the subseqences can be parallelized.

Ignore, try failing endpoints (Sec. 7.6.1). Suppose the left hand side of Fig. 27(a) takes a finite amount of steps to move a token to an output place in SG_1, however, the transition to SG_{ext} does not produce a result due to an exceptional situation (i.e., no change of the marking in cf). Correspondingly, the right hand side moves the token, however, without the failing, and thus read-only transition to SG_{ext}, which ensures the equality of the resulting tokens on either side. Under the same restriction that the no exception context is returned from SG_{ext}, the right hand side can simulate the left hand side accordingly.

The situation for try failing endpoints in Fig. 27(b) is the same in reverse.

Reduce requests (Sec. 7.6.2). Since the only difference between the left hand side and the right hand side is the slow-down due to the insertion of the pattern CS, and simulation does not take the age of messages into account, the left hand side can obviously simulate the right hand side and vice versa. □

7.8. Discussion

Theorem 2 makes precise in which sense the proposed optimization rules are correct, in the sense that they preserve the "meaning" of integration patterns as defined in our translation to timed db-nets. We emphasise that this is a property proven by analysing the proposed optimization rules, and not a fact that we expect to be decidable for arbitrary rewrite rules. By interpreting integration patterns as timed db-nets, we however get access to a framework where it makes sense to prove individual optimizations correct in the above sense, since there is a formal definition of the behaviour of (the interpretation of) the integration pattern.

8. Evaluation

In this section, (a) we evaluate the impact of optimization strategies (i.e., OS-1–3) from Sec. 3.2 that are most relevant to this work, and (b) we study ReCO for real-world integration processes.

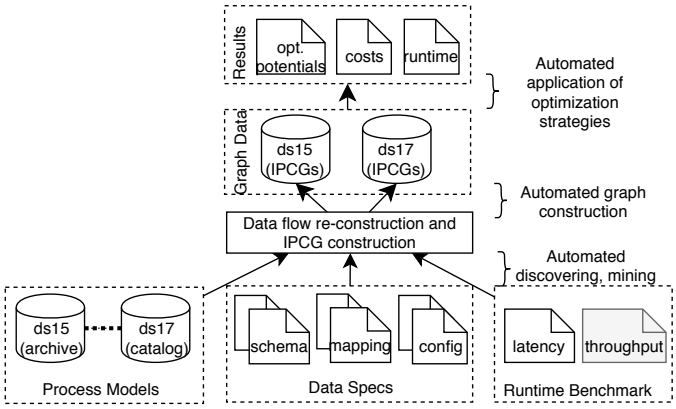


Figure 30: Pattern composition evaluation pipeline.

8.1. Optimization Strategies

For (a) we quantitatively analyze the effect of optimizations on two catalogs of integration processes regarding improvements of model complexity, throughput and latency. The catalogs have a two year difference to be able to study whether improvements were found by integration experts within that time span by themselves.

Then, we revisit our motivating example from Sec. 2 and study a more complex integration process regarding applicable optimization strategies.

8.1.1. Quantitative Analysis

We applied the optimization strategies OS-1–3 to 627 integration scenarios from the 2017 standard content of the SAP CPI (called ds17), and compared with 275 scenarios from 2015 (called ds15). Our goal is to show the applicability of our approach to real-world integration scenarios, as well as the scope and trade-offs of the optimization strategies. The comparison with a previous content version features a practical study on content evolution. To analyze the difference between different scenario domains, we grouped the scenarios into the following categories [5]: On-Premise to Cloud (OP2C), Cloud to Cloud (C2C), and Business to Business (B2B). Since hybrid integration scenarios such as OP2C target the extension or synchronization of business data objects, they are usually less complex. In contrast native cloud application scenarios such as C2C or B2B mediate between several endpoints, and thus involve more complex integration logic [5]. The process catalog also contained a small number of simple Device to Cloud scenarios; none of them could be improved by our approach.

Setup: Construction and analysis of IPCGs For the analysis, we constructed an IPCG for each integration scenario following the workflow sketched in Fig. 30. Notably, the integration scenarios are stored as process models in a BPMN-like notation [4]. The process models reference data specifications such as schemas (e.g., XSD, WSDL), mapping programs, selectors (e.g., XPath) and configuration files. For every pattern used in the process models, runtime statistics are available from benchmarks [55]. The data specifications are picked up from the 2015 content archive and from the current 2017 content catalog, while

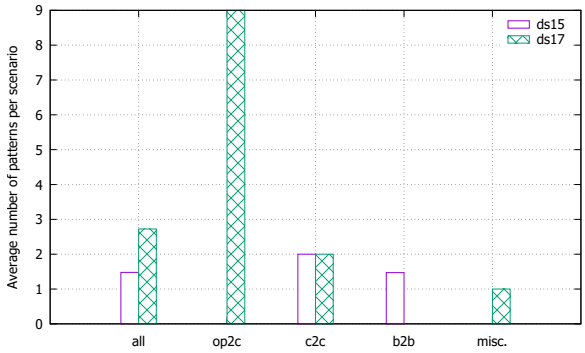


Figure 31: Pattern reduction per scenario.

the runtime benchmarks are collected using the open-source integration system *Apache Camel* [56][5] as used in SAP CPI. The mapping and schema information is automatically mined and added to the patterns as contracts, and the rest of the collected data as pattern characteristics. For each integration scenario and each optimization strategy, we determine if the strategy applies, and if so, if the cost is improved. This analysis runs in about two minutes in total for all 902 scenarios on our workstation.

We now discuss the improvements for the different kinds of optimization strategies identified in Sec. 3.2.

Improved Model Complexity: Process Simplification (OS-1). The relevant metric for the process simplification strategies from OS-1 is the average reduction in model complexity, shown in Fig. 31.

Results. Although all scenarios were implemented by integration experts, who are familiar with the modeling notation and the underlying runtime semantics, there is still a small amount of patterns per scenario that could be removed without changing the execution semantics. On average, the content reduction for the content from 2015 and 2017 was 1.47 and 2.72 patterns/IPCG, respectively, with significantly higher numbers in the OP2C domain.

Conclusions. (1) Even simple process simplifications are not always obvious to integration experts in scenarios represented in a control-flow-centric notation (e.g., current SAP CPI does not use BPMN Data Objects to visualize the data flow); and (2) the need for process simplification does not seem to diminish as experts gain more experience.

Improved Bandwidth: Data Reduction (OS-2). Data reduction impacts the overall bandwidth and message throughput [11]. To evaluate data reduction strategies from OS-2, we leverage the data element information attached to the IPCG contracts and characteristics, and follow their usages along edges in the graph, similar to "ray tracing" algorithms [57]. We collect the data elements that are used or not used, where possible — we do not have sufficient design time data to do this for user defined functions or some of the message construction patterns, such as request-reply. Based on the resulting data element usages, we calculate two metrics: the comparison of used vs. unused elements in Fig. 32(a), and the savings in abstract costs on unused data elements in Fig. 32(b).

Results. There is a large amount of unused data elements per scenario for the OP2C scenarios; these are mainly web service communication and message mappings, for which most of the data flow can be reconstructed. This is because the predominantly used EDI and SOA interfaces (e.g., SAP IDOC, SOAP) for interoperable communication with on-premise applications define a large set of data structures and elements, which are not required by the cloud applications, and vice versa. In contrast, C2C scenarios are usually more complex, and mostly use user defined functions to transform data, which means that only a limited analysis of the data element usage is possible.

When calculating the abstract costs for the scenarios with unused fields, there is an immense cost reduction potential for the OP2C scenarios as shown in Fig. 32(b). This is achieved by adding a content filter to the beginning of the scenario, which removes unused fields. This results in a cost increase $|d_{in}|$ = #unused elements for the content filter, but reduces the cost of each subsequent pattern up to the point were the elements are used.

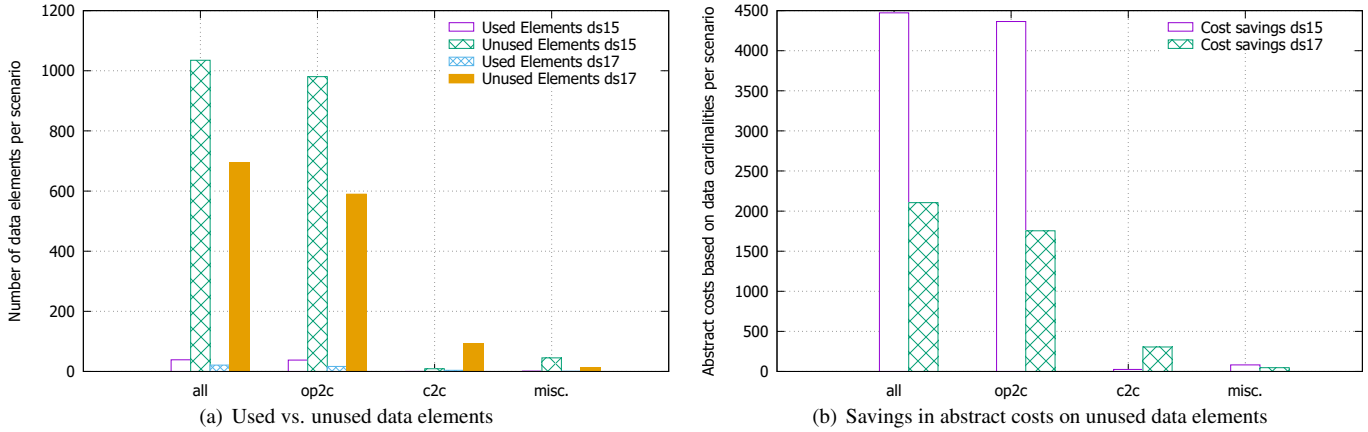
Conclusions. (3) Data flows can best be reconstructed when design time data based on interoperability standards is available; and (4) a high number of unused data elements per scenario indicates where bandwidth reductions are possible.

Improved Latency: Parallelization (OS-3). For the sequence-to-parallel optimization strategies from OS-3, the relevant metric is the processing latency of the integration scenario. Because of the uncertainty in determining whether a parallelization optimization would be beneficial, we first report on the classification of parallelization candidates in Fig. 33(a). We then report both the improvements according to our cost model in Fig. 33(b), as well as the actual measured latency in Fig. 33(c).

Results. Based on the data element level, we classify scenario candidates as `parallel`, definitely `non parallel`, or `potentially parallel` in Fig. 33(a). The uncertainty is due to sparse information. From the 2015 catalog, 81% of the scenarios are classed as `parallel`, or `potentially parallel`, while the number for the 2017 catalog is 53%. In both cases, the OP2C and B2B scenarios show the most improvement potential. Figure 33(b) shows the selection based on our cost model, which supports the pre-selection of all of these optimization candidates. The actual, average improvements per impacted scenario are shown in Fig. 33(c). The average improvements of up to 230 milliseconds per scenario must be understood in the context of the average runtime per scenario, which is 1.79 seconds. We make two observations: (a) the cost of the additional fork and join constructs in Java are high compared to those implemented in hardware [11], and the improvements could thus be even better, and (b) the length of the parallelized pattern sequence is usually short: on average 2.3 patterns in our scenario catalog.

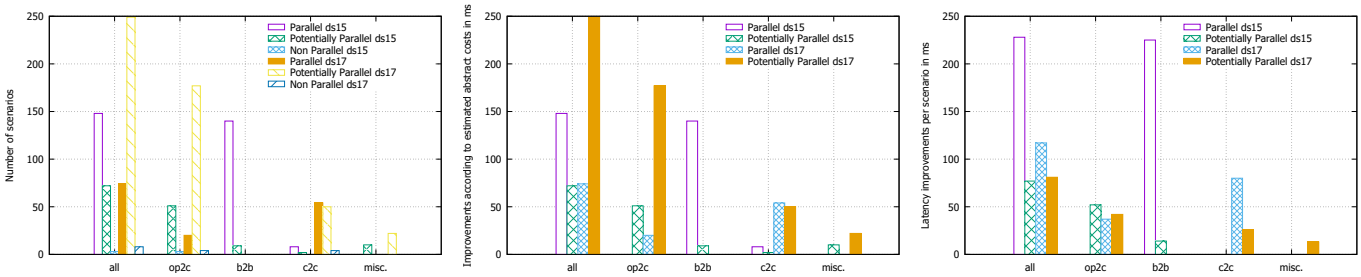
Conclusions. (5) The parallelization requires low cost fork and join implementations; and (6) better runtime improvements

[5]All measurements were conducted on a HP Z600 workstation, equipped with two Intel X5650 processors clocked at 2.67GHz with a 12 cores, 24GB of main memory, running a 64-bit Windows 7 SP1 and a JDK version 1.7.0, with 2GB heap space.



(a) Used vs. unused data elements (b) Savings in abstract costs on unused data elements

Figure 32: Unused elements in integration scenarios.



(a) Parallelization scenario candidates (b) Optimized scenarios based on costs (c) Actual latency improvements

Figure 33: OS-3 "Sequence to parallel" optimization candidates on (a) integration flows, (b) optimization selection based on abstract cost model, and (3) actual latency improvements.

might be achieved for scenarios with longer parallelizable pattern sequences.

8.1.2. Case Studies

We apply, analyze and discuss the proposed optimization strategies in the context of two case studies: the Replicate Material on-premise to cloud scenario from Fig. 3 in Sec. 2, as well as an SAP eDocument invoicing cloud to cloud scenario. These scenarios are part of the SAP CPI standard, and thus several users (i.e., SAP's customers) benefit immediately from improvements. For instance, we additionally implemented a content monitor pattern [5] that allowed analysis of the SAP CPI content. This showed the Material Replicate scenario was used by 546 distinct customers in 710 integration processes copied from the standard into their workspace — each one of these users is affected by the improvement.

Replicate Material (revisited) Recall from Sec. 2 that the Replicate Material scenario is concerned with enriching and translating messages coming from a CRM before passing them on to a Cloud for Customer service, as in Fig. 3. As already discussed, the content enricher and the message translator can be parallelized according to the sequence to parallel optimization from OS-3. The original and resulting IPCGs are shown in Figs. 7(a) and 7(b). No throughput optimizations apply.

Latency improvements. The application of this optimization can be considered, if the latency of the resulting parallelized process is smaller than the latency of the original process, i.e. if

$$\begin{aligned} &cost(MC) + \max(cost(CE), cost(MT)) \\ &\quad + cost(JR) + cost(AGG) \\ &\quad < cost(CE) + cost(MT) \end{aligned}$$

Subtracting $\max(cost(CE), cost(MT))$ from both sides of the inequality, we are left with

$$\begin{aligned} &cost(MC) + cost(JR) + cost(AGG) \\ &\quad < \min(cost(CE), cost(MT)) \end{aligned}$$

If we assume that the content enricher does not need to make an external call, its abstract cost becomes

$$cost(CE)(|d_{in}|, |d_r|) = |d_{in}|,$$

and plugging in experimental values from a pattern benchmark [55], we arrive at the inequality (with latency costs in seconds)

$$0.01 + 0.002 + 0.005 \not< \min(0.005, 0.27)$$

which tells us that the optimization is not beneficial in this case — the additional overhead is larger than the saving. However, if the

content enricher does use a remote call, $cost(CE)(|d_{in}|, |d_r|) = |d_{in}| + |d_r|$, and the experimental values now say $cost(CE) = 0.021$. Hence the optimization is worthwhile, as

$$0.01 + 0.002 + 0.005 < \min(0.021, 0.27) \ .$$

Model Complexity. Following Sánchez-González et al. [38], we measure the model complexity by node count. In this case, the optimization increases the complexity by 3.

Conclusions. (7) Pattern characteristics are important when deciding if an optimization should be applied (e.g., local vs. remote enrichment); and (8) there are conflicts between different objectives, as illustrated by the trade-off between latency reduction and model complexity increase.

eDocuments: Italy Invoicing. The Italian government accepts electronic invoices from companies, as long as they follow regulations — they have to be correctly formatted, signed, and not be sent in duplicate. Furthermore, these regulations are subject to change. This can lead to an ad-hoc integration process such as in Fig. 34 (simplified). Briefly, the companies' *Fattura Electronica* is used to generate a *factorapa* document with special header fields (e.g., *Paese*, *IdCodice*), then the message is signed and sent to the authorities, if it has not been sent previously. The multiple authorities respond with standard *Coglienza*, *Risposta* acknowledgments, that are transformed to a *SendInvoiceResponse*. We transformed the BPMN model to an IPCG, tried to apply optimizations, and created a BPMN model again from the optimized IPCG.

Model Complexity. Our heuristics for deciding in which order to try to apply different strategies are "simplification before parallelization" and "structure before data", since this seems to enable the largest number of optimizations. Hence we first try to apply OS-1 strategies: the *combine siblings* rule matches the sibling Message Signers, since the preceding content-based router is a fork. (The signer is also side-effect free, so applying this rule will not lead to observably different behavior.)

Latency Improvements. Next we try OS-3 strategies. Although *heterogeneous parallelization* matches for the CE and the Message Encoder, it is not applied since

$$cost(MC) + cost(JR) + cost(AGG) \not< \min(cost(CE), cost(ME)),$$

i.e., the overhead is too high, due to the low-latency, local CE. Finally, the *early-filter* strategy from OS-2 is applied for the Content Filter, inserting it between the Content Enricher and the Message Encoder. No further strategies can be applied. The resulting integration process translated back from IPCTG to BPMN is shown in Fig. 35.

Conclusions. (9) The application order OS-1, OS-3, OS-2 seems most beneficial ("simplification before parallelization", "structure before data"); (10) an automatic translation from IPCGs to concepts like BPMN could be beneficial for connecting with existing solutions.

8.2. Case Studies: Responsible Pattern Composition

For (b), we evaluate the translation in two case studies of real-world integration scenarios: the replicate material scenario from Fig. 3, and a predictive machine maintenance scenario. The former is an example of hybrid integration, and the latter of IOT device integration.

For each of the scenarios, we give an integration pattern contract graph with matching contracts, translate it to a timed DB-net with boundaries, and show how its execution can be simulated. The scenarios are both taken from the SAP Cloud Platform Integration solution catalog of reference integration scenarios, and are frequently used by customers [14]. For the simulation we use the CPN Tools timed DB-net prototype from Sec. 5.4 with the extension for hierarchical PN composition. In CPN Tools hierarchies, the patterns can be represented as sub-groups and pages with explicit in- and out-port type definitions [58], which we use as part of the boundaries defined in Sec. 5. Thereby the synchronization is checked based on the CPN color sets of the port types. The other boundary checks are performed during the simulation according to the constructed boundaries (see construction mechanism in Definition 10 in Sec. 6.2).

8.2.1. Hybrid Integration: Replicate Material

An IPCG representing an integration process for the replication of material from an enterprise resource planning or customer relationship management system to a cloud system was given in Fig. 7(a) in Sec. 4.1. We now add slightly more data in the form of the pattern characteristics, which provides sufficient information for the translation to timed DB-nets with boundaries. Figure 36 depicts the enriched IPCG. The adapters are actually message processors, however, for simplicity they are represented as start and end pattern types, $ADPT_s$ denoting *erp* and $ADPT_r$ representing *cod*. The characteristics of the CE node includes the tuple $(PRG, (prg1, [0, \infty)))$, with enrichment function $prg1$ which assigns the *DOCNUM* payload to the new header field *AppID*. Similarly, the characteristics of the MT nodes includes a tuple $(PRG, (prg2, \_))$ with mapping program $prg2$, which maps the *EDI_DC40-DOCNUM* payload to the *MMRR-BMH-ID* field (the Basic Message Header ID of the Material Mass Replication Request structure), and the *EPM-PRODUCT_ID* payload to the *MMRR-MAT-ID* field (the Material ID of the Material Mass Replication Request structure).

Translation to a Timed DB-Nets with Boundaries. First we translate each single pattern from Fig. 36 according to the construction in Sec. 6.1. The integrati on adapter nodes $ADPT_s$ and $ADPT_r$ are translated as the start and end patterns in Fig. 11(a) and Fig. 11(b), respectively. The content enricher CE node and message translator MT node are message processors without storage, and hence translated as in Fig. 15 with $< f >_{CE} = prg1$ and $< f >_{MT} = prg2$ (no database values are required). Since no database table updates are needed for either translation, the database update function parameter $< g >$ can be chosen to be the identity function in both cases.

In the second step, we refine the timed DB-net with boundaries to also take contract concepts into account by the construction in Definition 10. The resulting net is shown in Fig. 37. This ensures the correctness of the types of data exchanged between patterns, and follows directly from the correctness of the corresponding IPCG. Other contract properties such as encryption

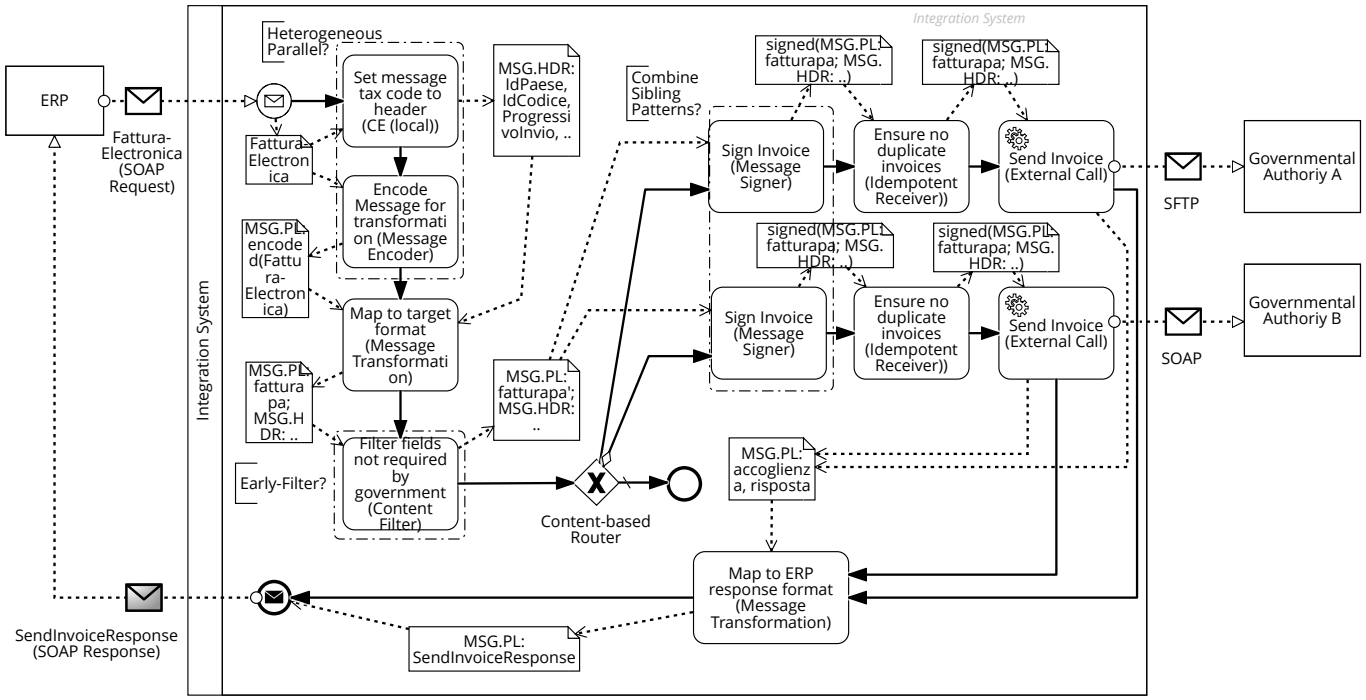


Figure 34: Country-specific invoicing (potential improvements as BPMN Group)

encr, encodings *enc*, and signatures *sign* are checked through transition guards.

Simulation. We test the composition construction of the material replicate scenario in Fig. 37 through simulation in the form of a hierarchical timed DB-net model, shown in Fig. 38. Thereby, the CE and MT patterns are represented by CPN Tool Subpage elements that are annotated with subpage tags *enricher*, *translator*, respectively.

On arrival of the request *msg* from the ERP system, the boundary configuration is appended to the message in place *erpToCe*. In the replicate material scenario the data is received unencrypted, uncoded and unsigned, leading to a boundary (msg,no,no,no) (which is encoded as *(msg,false,false,false)* in our prototype). The extended message *erp_msg* is then moved to the boundary place *ch0* by transition *CheckCeBoundary*, if the *[encr=false]* guard holds, and thus ensures the correctness of the data exchange between patterns. Subsequently only the actual message without the boundary data is moved to place *ch0*, that is linked to the input place *ch0* of the enricher, as in Fig. 38. We recall, that the *in* port type ensures that the synchronization on the CPN color set level are correct. After the *enricher* processing, the *out* port type ensures the correctness of the synchronization on the CPN color set level and the resulting message *emsg* is moved to the linked output place *ch4*. The constructed outbound boundary, represented by transition *SetCeBoundary* sets the boundary properties of the enricher to (msg,false,false,false) for the following pattern. On the input boundary side of the *translator*, transition *CheckMtBoundary* evaluates its guard, before moving the message without the boundary data to the boundary place *ch5*, which proceeds similar to the enricher.

Note that our boundary construction mechanism from Definition 10 generated the input boundary, e.g., denoted by place *erpToCe* and transition *CheckCeBoundary*, as well as the output boundary, e.g., transition *SetCeBoundary* and place *ceToMt*, including the transition guards, colorsets, variables, and port type configurations, for the validation by simulation.

Discussion. Notably, constructing an IPCG requires less technical knowledge such as particularities of timed DB-nets but still enables correct pattern compositions on an abstract level. While the *CPT* part of the pattern contracts (e.g., encrypted, signed) could be derived and translated automatically from a scenario in a yet to be defined modeling language, many aspects like their elements *EL* as well as the configuration of the characteristics by enrichment and mapping programs requires a technical understanding of IPCGs and the underlying scenarios. As such IPCGs can be considered a suitable intermediate representation of pattern compositions. The user might still prefer a more appealing graphical modeling language on top of IPCGs. The simulation capabilities of the constructed timed DB-net with boundaries allow for the experimental validation of real-world pattern compositions. However, the complexity of the construction highlights the importance of an automation of the construction.

Conclusions. (11) IPCG and timed DB-net with boundaries can be shown correct with respect to composition and execution semantics; (12) timed DB-nets with boundaries are even more complex than timed DB-nets; (13) IPCGs are more comprehensible than timed DB-nets, and expressive enough for current integration scenarios.

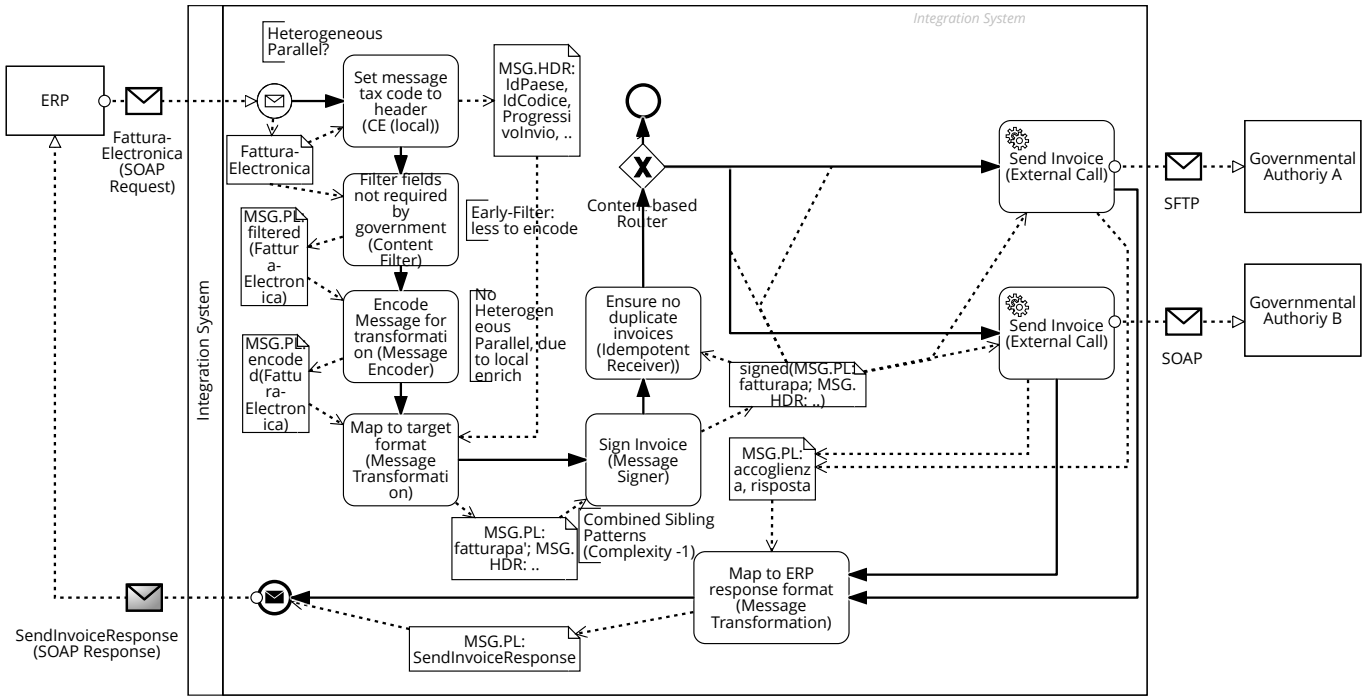


Figure 35: Invoice processing from Fig. 34 after application of strategies OS-1–3.

8.2.2. *Internet of Things: Predictive Maintenance and Service (PDMS)*

The IPCG representing the predictive maintenance create notification scenario that connects machines with enterprise resource planning (ERP) and PDMS systems is given in Fig. 39. We add all pattern characteristics and data, which provides sufficient information for the translation to timed DB-nets with boundaries. Figure 39 depicts the corresponding IPCG. The characteristics of the CE_1 node includes an enrichment function $prg1$ that adds further information about the machine in the form of the *FeatureType* to the message that contains machine *ID* and *UpperThresholdWarningValue*. This data is leveraged by the UDF_1 *predict* node, which uses a prediction function $prg2$ about the need for maintenance and adds the result into the *MaintenanceRequestById* field. Before the data is forwarded to the ERP system (simplified by an *End*), the single machine predictions are combined into one message by the AGG_1 node with correlation cnd_{cr} and completion cnd_{cc} conditions as well as the aggregation function prg_3 and completion timeout (v_1, v_2) as pattern characteristics $\{(\{cnd_{cr}, cnd_{cc}\}), (PRG, prg_4, (v_1, v_2))\}$.

Translation to Timed DB-Nets with Boundaries. Again, we translate each single pattern from Fig. 39 according to the construction in Sec. 6.1. The *Start* and *End* nodes are translated as the start and end pattern in Fig. 11(a) and Fig. 11(b) respectively. The CE CE_1 and user-defined function UDF_1 nodes are message processors, and hence translated as in Fig. 15 with $< f >_{CE_1} = prg1$ and $< f >_{UDF_1} = prg2$. Since no table updates are needed for either translation, the database update function parameter $< g >$ can be chosen to be the identity function in all cases. The aggregator AGG_1 node is a merge pattern type, and thus translated as in Fig. 16 with $(v_1, v_2) \rightarrow [\tau_1, \tau_2]$, $prg_{agg} \rightarrow f(msgs)$. Moreover, the correlation condition $cnd_{cr} \rightarrow g(msg, msgs)$ and the completion condition $cnd_{cc} \rightarrow complCount$.

In the second step, we refine the timed DB-net with boundaries to also take contract concepts into account by the construction in Definition 10. The resulting net is shown in Fig. 40. This ensures the correctness of the types of data exchanged between patterns, and follows directly from the correctness of the corresponding IPCG. Other contract properties such as encryption, signatures, and encodings are checked through the transition guards.

Simulation. We illustrate the composition construction of the predictive maintenance scenario in Fig. 40 through simulation in the form of a hierarchical timed DB-net model, shown in Fig. 41. Again, all timed DB-net patterns are hierarchically represented by CPN Tool Subpage elements that are annotated with subpage tags *enricher*, *message_aggregator*, respectively, and the user-defined function *predict* is denoted by a transition. The boundaries are constructed from Fig. 40 by inserting *SetPdmsBoundary* and *pdmsToCe* as output boundary of *get report*, which matches the input boundary of the subsequent *enricher*, denoted by the *CheckCeBoundary* transition. Transition *SetCeBoundary* and place *ceToPredict* represent the output boundary of the enricher, which again match the input boundary of the *predict* user-defined function pattern through transition *CheckPredictBoundary*. Finally, the output boundary of the predict step is ensured by transition *SetPredictBoundary* and place *predictToAgg*. Again, it can be easily seen that the input boundary of the aggre-

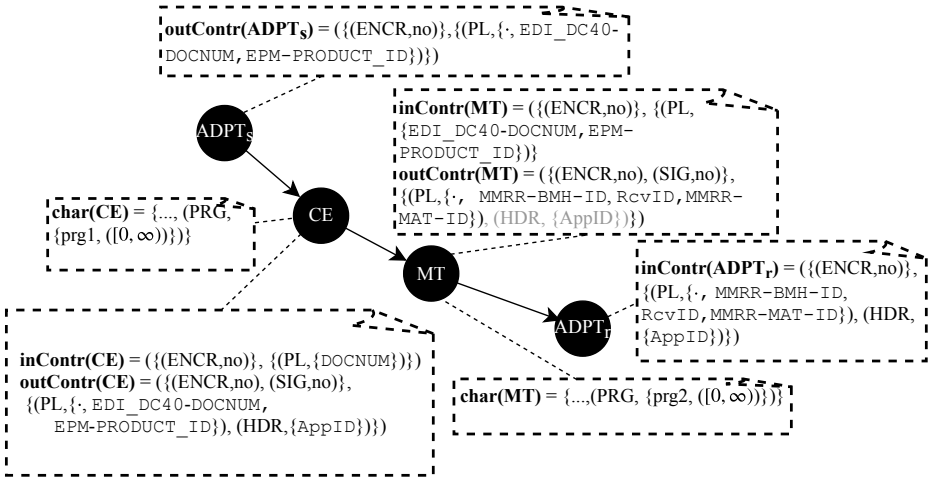


Figure 36: Complete integration pattern contract graph of the replicate material scenario

gator in the form of the *CheckAggBoundary* transition matches, and thus the overall composition is correct. Consequently, the simulation of the timed DB-net with these boundaries in Fig. 41 results in the same, correct output with the timed DB-nets without boundaries in Fig. 41.

Discussion. In this slightly more complex scenario, it becomes more obvious that the constructed IPCGs are quite technical as well and require a careful construction of pattern characteristics and contracts. While this seems to be an ideal representation for checking the structural correctness of compositions, this should be no manual task for a user. Especially for more complex scenarios, we found that the re-configurable pattern type-based translation works well. However, the construction of the timed DB-nets with boundaries corresponding to an IPCG would benefit from an automatic translation (e.g., through tool support).

Conclusions. (14) IPCGs are still quite technical, especially for more complex scenarios; (15) a tool support for automatic construction and translation is preferable.

8.3. Discussion

The evaluation on the optimization strategies on IPCGs (cf. (a)) resulted into several interesting conclusions, i.e., emphasizing on the importance of a pattern composition and optimization formalization, which are relevant even for experienced integration experts (conclusions 1–2), with interesting choices (conclusions 3–4, 6), implementation details (conclusions 5, 10) and trade-offs (conclusions 7–9). The contract graphs provide a rich composition context, which might help the user when composing patterns with built-in structural correctness guarantees.

The second major aspect of this work — besides process optimizations — concerns the responsible composition of processes out of integration patterns (cf. (b)) that can be automated (cf. conclusion 10). The evaluation of two case studies — following ReCO — resulted into further interesting conclusions, i.e., the suitability of our approach for pattern compositions (cf. conclusions (11,13)), model complexity considerations (cf. conclusions (12,14)) and desirable extensions like automatic translation (cf. conclusion (15)). However, while IPCGs based on timed DB-nets with boundaries denote the first comprehensive definition of application integration scenarios with built-in functional correctness and compositional correctness validation and verification, it might not give an appealing modeling language for (non-technical) users (cf. conclusions (12,14)). We envision a novel modeling language and tool support that facilitates a translation from that language to IPCGs (cf. conclusion (15)), which we consider as future work. Based on such a language infrastructure, more advanced compositional aspects like modeling guidelines on the different layers (i.e., language, intermediate IPCG, and simulation timed DB-net with boundaries) could be studied.

Table 5: Optimization Strategies in context of the objectives

Approach	Formal model	Optimizations	Correctness
EAI	👍	👎	👍
BPM	👍	👎	👍
SPC	👍	👍	👍
AOS	👎	👍	👎
GT	👍	👎	👍
ReCO	👍	👍	👍

👍: covered, 👍: partially covered, 👎: not covered

9. Related Work

We presented related optimization techniques in Sec. 3.2. We now briefly situate our work within the context of other formalizations, beyond the already discussed BPMN [4] and PN [39] approaches, as summarized in Tab. 5.

Enterprise Application Integration (EAI) Similar to the BPMN and PN notations, several domain-specific languages (DSLs) have been developed that describe integration scenarios. Apart from the EIP icon notation [2], there is also the Java-based Apache Camel DSL [56], and the UML-based Guaraná DSL [59]. However, none of these languages aim to be optimization-friendly formal integration scenario representations. Conversely, we do

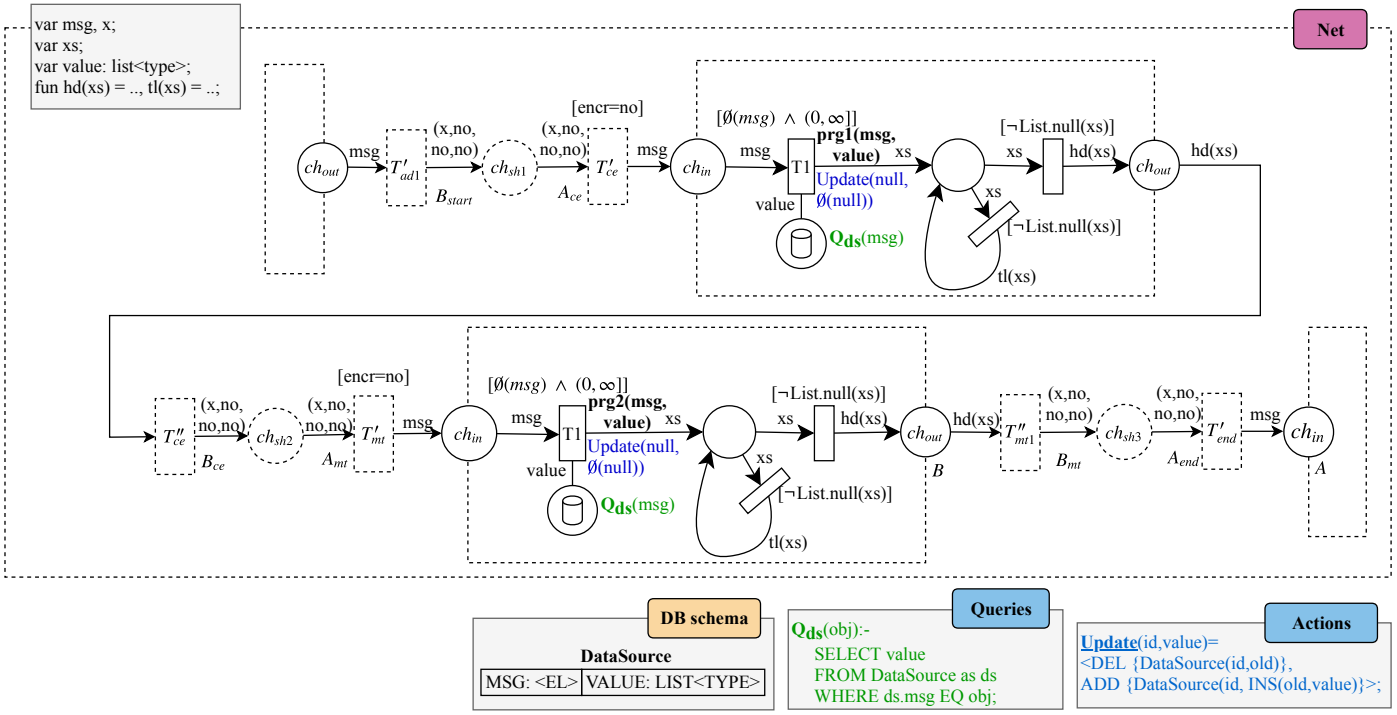


Figure 37: Material replicate scenario as a timed DB-nets with boundaries

not strive to build another integration DSL. Instead we claim that all of the integration scenarios expressed in such languages can be formally represented in our formalism, so that optimizations can be determined that can be used to rewrite the scenarios.

There is work on formal representations of integration patterns, e.g. Mederly et al. [60] represents messages as first-order formulas and patterns as operations that add and delete formulas, and then applies AI planning to find an process with a minimal number of components. While this approach shares the model complexity objective, our approach applies to a broader set of objectives and optimization strategies. For the verification of service-oriented manufacturing systems, Mendes et al. [61] uses "high-level" Petri nets as a language instead of integration patterns, similar to the approach of Fahland and Gierds [39].

Business Process Management (BPM) Sadiq and Orlowska [62] applied reduction rules to workflow graphs for the visual identification of structural conflicts (e.g., deadlocks) in business processes. Compared to process control graphs, we use a similar base representation, which we extend by pattern characteristics and data contracts. Furthermore, we use graph rewriting for optimization purposes. In Cabanillas et al. [63], the structural aspects are extended by a data-centered view of the process that allows to analyze the life cycle of an object, and check data compliance rules. This adds a view on the required data, but does not propose optimizations for the EIPs. The main focus is rather on the object life cycle analysis of the process.

Semantic Program Correctness (SPC) Semantic correctness plays a bigger role in the compiler construction and analysis domain. For example, Muchnick [64] provides an exhaustive catalog of optimizing transformations and states that the proof of the correctness of rewritings must be based on the (execution) semantics, and Nielson [65] provides semantic correctness proofs using data-flow analysis, while Cousot [66] provides a general framework for designing program transformations by analyzing abstract interpretations. Although far simpler than general programming language transformations, our translation of IPCGs to timed db-nets with boundaries can be seen as a concretization in the sense of an abstract interpretation, and thus giving a similar notion of semantic correctness.

Analysis and Optimization Structures (AOS) Transformation techniques for optimization have been employed by compiler construction, e.g., for parallel [67] or pipeline processing [68], where dependence graph representations become especially useful. For example, Kuck et al. [69] construct dependence graphs with output, anti, and flow dependencies as a foundation for optimizing transformations. These kind of dependence graphs were also used by Böhm et al. [36], however, they are "linearized" in the form of our pattern contracts. This makes the decision of the optimization "local" and does not require dependence graph abstractions like intervals [70] or scoping [71]. More recently these techniques have been applied for business process optimization by Sadiq [62], Niedermann et al. [17, 18] or reductions to process tree structures [72] with incremental transformations [73]. In our case the scope of the analysis is a local match of pattern contracts.

Graph Transformations (GT) Similar to our approach, graph transformations have been used in related domains, e.g., formalizing parts of the BPMN semantics by Dijkman et al. [74], who specify the execution semantics as graph rewrites. Conformance is checked experimentally and verification is left for

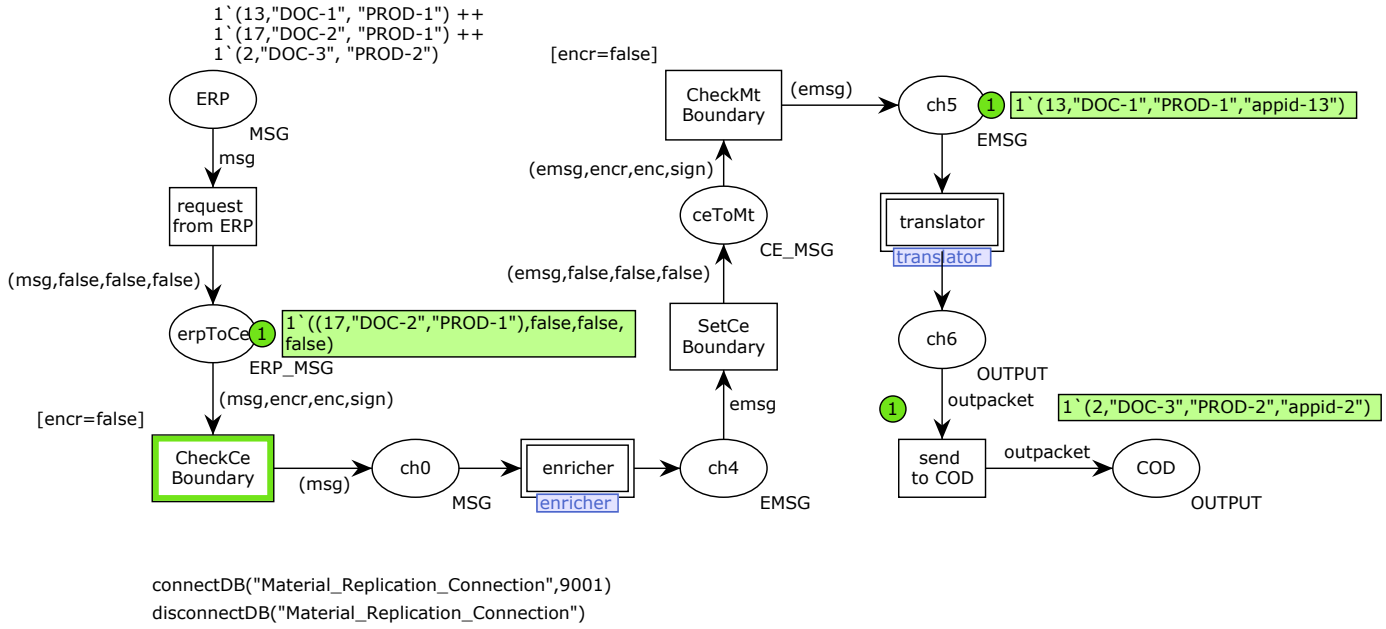


Figure 38: Material replicate scenario simulation

future work. For the optimizations, we use the same visual notation and double-pushout rule application approach. However, our execution semantics are given as timed db-net and can be formally analyzed.

10. Conclusions

This work addresses an important shortcoming in EAI research, namely the lack of means for responsible or correct integration pattern compositions and the application of changes, e.g., as optimization strategies, which preserve structural and semantic correctness, and thus ends the informality of descriptions of pattern compositions and optimizations (cf. Q1–Q3).

We approached the questions along a responsible pattern composition and optimization process (short ReCO), and started by compiling catalogs of integration pattern characteristics as well as optimization strategies from the literature. We then developed a formalization of pattern compositions in order to precisely define optimizations as pattern contract graphs. Then we extended the timed db-nets formalism, covering integration pattern semantics, into timed db-nets with boundaries, which resemble the contracts in the pattern graphs, and defined a mechanism to interpret the pattern graphs by them.

With the resulting formal framework, we proved that all defined optimizations preserve the meaning of compositions as timed db-nets. We evaluated the framework on data sets containing in total over 900 real world integration scenarios, and two brief case studies. The responsible pattern composition part in ReCO was then studied for two integration processes down to the execution semantics, essentially showing ReCO from a modeling perspective.

We conclude that formalization and optimizations of integration processes in the form of integration pattern compositions — using pattern contract graphs — are relevant even for experienced integration experts (conclusions 1–2), with interesting choices (concls. 3–4, 6), implementation details (conclusions 5, 10) and trade-offs (concls. 7–9). In the two additional case studies, we showed the suitability of our interpretation of pattern contract graphs in the newly defined timed db-nets with boundaries for pattern compositions (concls. (11,13)), model complexity considerations (concls. (12,14)) and desirable extensions like automatic translation (concl. (15)).

References

[1] T. Jeske, M. Würfels, F. Lennings, M. Weber, S. Stowasser, Achievements and opportunities of digitalization in productivity management, in: AHFE, Vol. 1207 of Advances in Intelligent Systems and Computing, Springer, 2020, pp. 17–24.

[2] G. Hohpe, B. Woolf, Enterprise integration patterns: Designing, building, and deploying messaging solutions, Addison-Wesley, 2004.

[3] D. Ritter, M. Holzleitner, Integration adapter modeling, in: CAiSE, 2015, pp. 468–482.

[4] D. Ritter, J. Sosulski, Exception handling in message-based integration systems and modeling using BPMN, Int. J. Cooperative Inf. Syst 25 (2) (2016) 1–38.

[5] D. Ritter, N. May, S. Rinderle-Ma, Patterns for emerging application integration scenarios: A survey, Inf. Syst. 67 (2017) 36 – 57.

[6] D. Ritter, S. Rinderle-Ma, M. Montali, A. Rivkin, Formal foundations for responsible application integration, Inf. Syst. 101 (2021) 101439.

[7] D. Ritter, N. May, F. Nordvall Forsberg, S. Rinderle-Ma, Optimization strategies for integration pattern compositions, in: ACM DEBS, 2018, pp. 88–99.

[8] S. Abiteboul, M. Arenas, P. Barceló, M. Bienvenu, D. Calvanese, C. David, R. Hull, E. Hüllermeier, B. Kimelfeld, L. Libkin, W. Martens, T. Milo, F. Murlak, F. Neven, M. Ortiz, T. Schwentick, J. Stoyanovich, J. Su, D. Suciu, V. Vianu, K. Yi, Research directions for principles of data management (abridged), SIGMOD Record 45 (4) (2017) 5–17.

[9] D. Eyers, A. Gal, H.-A. Jacobsen, M. Weidlich, Integrating Process-Oriented and Event-Based Systems (Dagstuhl Seminar 16341), Dagstuhl Reports 6 (8) (2017) 21–64.

[10] G. Kougka, A. Gounaris, Optimization of data-intensive flows: Is it needed? Is it solved?, in: DOLAP, ACM, 2014, pp. 95–98.

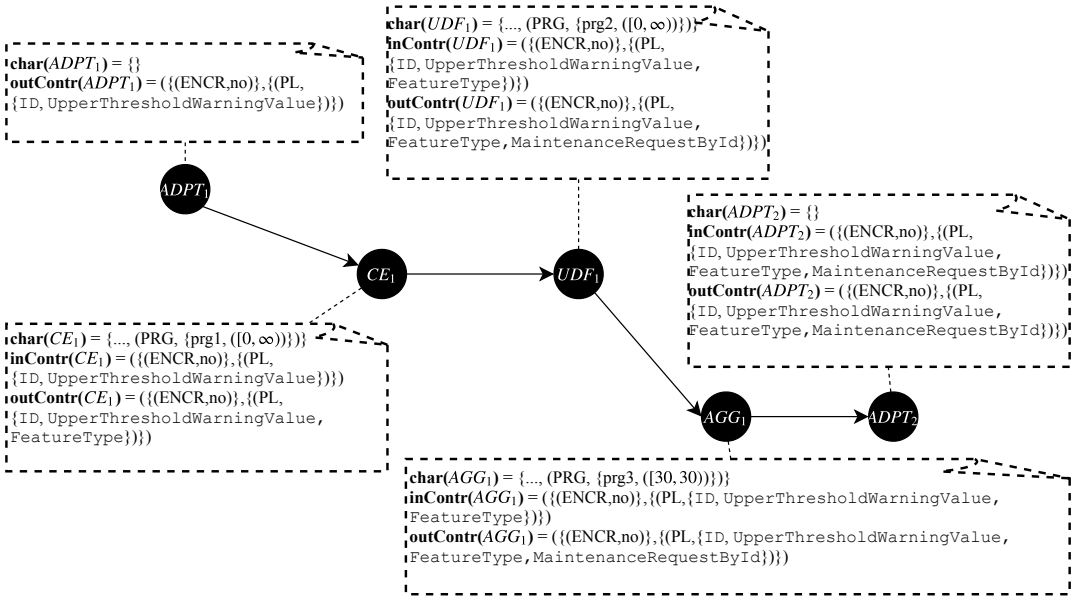


Figure 39: Integration pattern contract graph of the predictive maintenance scenario

[11] D. Ritter, J. Dann, N. May, S. Rinderle-Ma, Hardware accelerated application integration processing: Industry paper, in: ACM DEBS, 2017, pp. 215–226.

[12] D. Ritter, Experiences with business process model and notation for modeling integration patterns, in: ECMFA, 2014, pp. 254–266.

[13] K. Peffers, T. Tuunanen, M. A. Rothenberger, S. Chatterjee, A design science research methodology for information systems research, JMIS 24 (3) (2007) 45–77.

[14] SAP SE, SAP API Business Hub – Prepackaged cloud integration content, https://api.sap.com/ (2023).

[15] D. Ritter, F. Nordvall Forsberg, S. Rinderle-Ma, N. May, Catalog of optimization strategies and realizations for composed integration patterns, CoRR abs/1901.01005 (2019).

[16] K. Vergidis, A. Tiwari, B. Majeed, Business process analysis and optimization: Beyond reengineering, IEEE Transactions on SMC, Part C 38 (1) (2008) 69–82.

[17] F. Niedermann, S. Radeschütz, B. Mitschang, Business process optimization using formalized patterns, BIS (2011).

[18] F. Niedermann, H. Schwarz, Deep business optimization: Making business process optimization theory work in practice, in: BPMDS / EMMSAD, Springer, 2011, pp. 88–102.

[19] K. Agrawal, A. Benoit, L. Magnan, Y. Robert, Scheduling algorithms for linear workflow optimization, in: IPDPS, 2010, pp. 1–12.

[20] L. F. Bittencourt, E. R. M. Madeira, Hcoc: a cost optimization algorithm for workflow scheduling in hybrid clouds, Journal of Internet Services and Applications 2 (3) (2011) 207–227.

[21] T. Tirapat, O. Udomkasemsub, X. Li, T. Achalakul, Cost optimization for scientific workflow execution on cloud computing, in: ICPADS, 2013, pp. 663–668.

[22] S. G. Ahmad, C. S. Liew, M. M. Rafique, E. U. Munir, S. U. Khan, Data-intensive workflow optimization based on application task graph partitioning in heterogeneous computing systems, in: IEEE BdCloud, 2014, pp. 129–136.

[23] A. Benoit, M. Coqblin, J.-M. Nicod, L. Philippe, V. Rehn-Sonigo, Throughput optimization for pipeline workflow scheduling with setup times., in: Euro-Par Workshops, 2012, pp. 57–67.

[24] I. Habib, A. Anjum, R. Mcclatchey, O. Rana, Adapting scientific workflow structures using multi-objective optimization strategies, TAAS 8 (1) (2013) 4.

[25] P. Zhang, Y. Han, Z. Zhao, G. Wang, Cost optimization of cloud-based data integration system, in: WISA, 2012, pp. 183–188.

[26] J. R. Getta, Static optimization of data integration plans in global information systems, in: ICEIS, 2011, pp. 141–150.

[27] M. Vrhovnik, H. Schwarz, O. Suhre, B. Mitschang, V. Markl, A. Maier, T. Kraft, An approach to optimize data processing in business processes, in: VLDB, 2007, pp. 615–626.

[28] G. Kougka, A. Gounaris, A. Simitsis, The many faces of data-centric workflow optimization: a survey, Int. J. Data Sci. Anal. 6 (2) (2018) 81–107.

[29] M. Böhm, U. Wloka, D. Habich, W. Lehner, Model-driven generation and optimization of complex integration processes., in: ICEIS (1), 2008, pp. 131–136.

[30] A. Böhm, C. Kanne, Demaq/transscale: Automated distribution and scalability for declarative applications, Inf. Syst. 36 (3) (2011) 565–578.

[31] D. Ritter, Database processes for application integration, in: BICOD, 2017, pp. 49–61.

[32] D. Ritter, Cost-efficient integration process placement in multiclouds, in: IEEE EDOC, IEEE, 2020, pp. 115–124.

[33] D. Ritter, Cost-aware process modeling in multiclouds, Inf. Syst. 108 (2022) 101969.

[34] M. Nygard, Release It!: Design and Deploy Production-Ready Software, Pragmatic Bookshelf, 2007.

[35] B. Kitchenham, Procedures for performing systematic reviews, Keele, UK, Keele University 33 (2004) (2004) 1–26.

[36] M. Böhm, D. Habich, S. Preissler, W. Lehner, U. Wloka, Cost-based vectorization of instance-based integration processes, Inf. Syst. 36 (1) (2011) 3–29.

[37] D. Ritter, S. Rinderle-Ma, Toward application integration with multimedia data, in: IEEE EDOC, 2017, pp. 103–112.

[38] L. Sánchez-González, F. García, J. Mendling, F. Ruiz, M. Piattini, Prediction of business process model quality based on structural metrics, in: ER, 2010, pp. 458–463.

[39] D. Fahland, C. Gierds, Analyzing and completing middleware designs for enterprise integration using coloured petri nets, in: CAiSE, 2013, pp. 400–416.

[40] M. Montali, A. Rivkin, DB-Nets: On the marriage of colored petri nets and relational databases, T. Petri Nets and Other Models of Concurrency 12 (2017) 91–118.

[41] F. E. Allen, Control flow analysis, SIGPLAN Notices 5 (7) (1970) 1–19.

[42] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, D. Warneke, Nephele/PACTs: a programming model and execution framework for web-scale analytical processing, in: SoCC, 2010, pp. 119–130.

[43] M. Böhm, D. Habich, W. Lehner, U. Wloka, Systemübergreifende Kostennormalisierung für Integrationsprozesse, in: BTW, 2009, pp. 67–86.

[44] K. Jensen, Coloured Petri nets: basic concepts, analysis methods and practical use, Vol. 1, Springer Science & Business Media, 2013.

[45] P. Baldan, A. Corradini, H. Ehrig, R. Heckel, Compositional semantics for open petri nets based on deterministic processes, Mathematical Structures in Computer Science 15 (1) (2005) 1–35.
[46] P. Sobociński, Representations of petri net interactions, in: International Conference on Concurrency Theory, Springer, 2010, pp. 554–568.
[47] J. C. Baez, J. Master, Open petri nets, Mathematical Structures in Computer Science 30 (3) (2020) 314–341.
[48] P. Selinger, A survey of graphical languages for monoidal categories, in: B. Coecke (Ed.), New Structures for Physics, Vol. 813 of Lecture Notes in Physics, Springer, 2011, pp. 289–355.
[49] D. Ritter, Application integration patterns and their compositions, Ph.D. thesis, University of Vienna (2019).
URL http://othes.univie.ac.at/58436/
[50] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, Monographs in Theoretical Computer Science, Springer, 2006.
[51] H. Ehrig, M. Pfender, H. J. Schneider, Graph-grammars: An algebraic approach, in: Switching and Automata Theory, 1973, pp. 167–180.
[52] A. Habel, D. Plump, Relabelling in graph transformation, in: ICGT, Vol. 2505, Springer, 2002, pp. 135–147.
[53] D. Plump, A. Habel, Graph unification and matching, in: TAGT, 1994, pp. 75–88.
[54] A. Kissinger, A. Merry, M. Soloviev, Pattern graph rewrite systems, in: DCM, 2012, pp. 54–66.
[55] D. Ritter, N. May, K. Sachs, S. Rinderle-Ma, Benchmarking integration pattern implementations, in: ACM DEBS, 2016, pp. 125–136.
[56] C. Ibsen, J. Anstey, Camel in Action, Manning, 2010.
[57] A. S. Glassner, An introduction to ray tracing, Elsevier, 1989.
[58] K. Jensen, L. M. Kristensen, L. Wells, Coloured Petri nets and CPN Tools for modelling and validation of concurrent systems, International Journal on Software Tools for Technology Transfer 9 (3-4) (2007) 213–254.
[59] R. Z. Frantz, A. M. Reina Quintero, R. Corchuelo, A domain-specific language to design enterprise application integration solutions, Int. J. Cooperative Inf. Syst 20 (02) (2011) 143–176.
[60] P. Mederly, M. Lekavỳ, M. Závodský, P. Navra, Construction of messaging-based enterprise integration solutions using AI planning, in: CEE-SET, 2009, pp. 16–29.
[61] J. M. Mendes, P. Leitão, A. W. Colombo, F. Restivo, High-level petri nets for the process description and control in service-oriented manufacturing systems, IJPR 50 (6) (2012) 1650–1665.
[62] W. Sadiq, M. E. Orlowska, Analyzing process models using graph reduction techniques, Inf. Syst. 25 (2) (2000) 117–134.
[63] C. Cabanillas, M. Resinas, A. Ruiz-Cortés, A. Awad, Automatic generation of a data-centered view of business processes, in: CAiSE, Springer, 2011, pp. 352–366.
[64] S. Muchnick, Advanced compiler design implementation, Morgan Kaufmann, 1997.
[65] F. Nielson, Semantic foundations of data flow analysis, DAIMI Report Series 10 (131) (1981).
[66] P. Cousot, R. Cousot, Systematic design of program transformation frameworks by abstract interpretation, in: Symposium on Principles of Programming Languages (POPL), ACM, 2002, pp. 178–190.
[67] D. J. Kuck, Y. Muraoka, S.-C. Chen, On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup, IEEE Transactions on Computers 100 (12) (1972) 1293–1310.
[68] D. Kuck, R. Kuhn, B. Leasure, M. J. Wolfe, Analysis and transformation of programs for parallel computation, in: COMPSAC, IEEE, 1980, pp. 709–715.
[69] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, M. Wolfe, Dependence graphs and compiler optimizations, in: POPL, ACM, 1981, pp. 207–218.
[70] J. Cocke, Global common subexpression elimination, in: Symposium on Compiler Optimization, ACM, 1970, pp. 20–24.
[71] M. V. Zelkowitz, W. G. Bail, Optimization of structured programs, Software: Practice and Experience 4 (1) (1974) 51–57.
[72] J. Vanhatalo, H. Völzer, J. Koehler, The refined process structure tree, Data & Knowledge Engineering 68 (9) (2009) 793–818.
[73] R. F. Hauser, M. Friess, J. M. Kuster, J. Vanhatalo, An incremental approach to the analysis and transformation of workflows using region trees, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 38 (3) (2008) 347–359.
[74] R. M. Dijkman, P. V. Gorp, BPMN 2.0 execution semantics formalized as graph rewrite rules, in: BPMN Workshop, 2010, pp. 16–30.

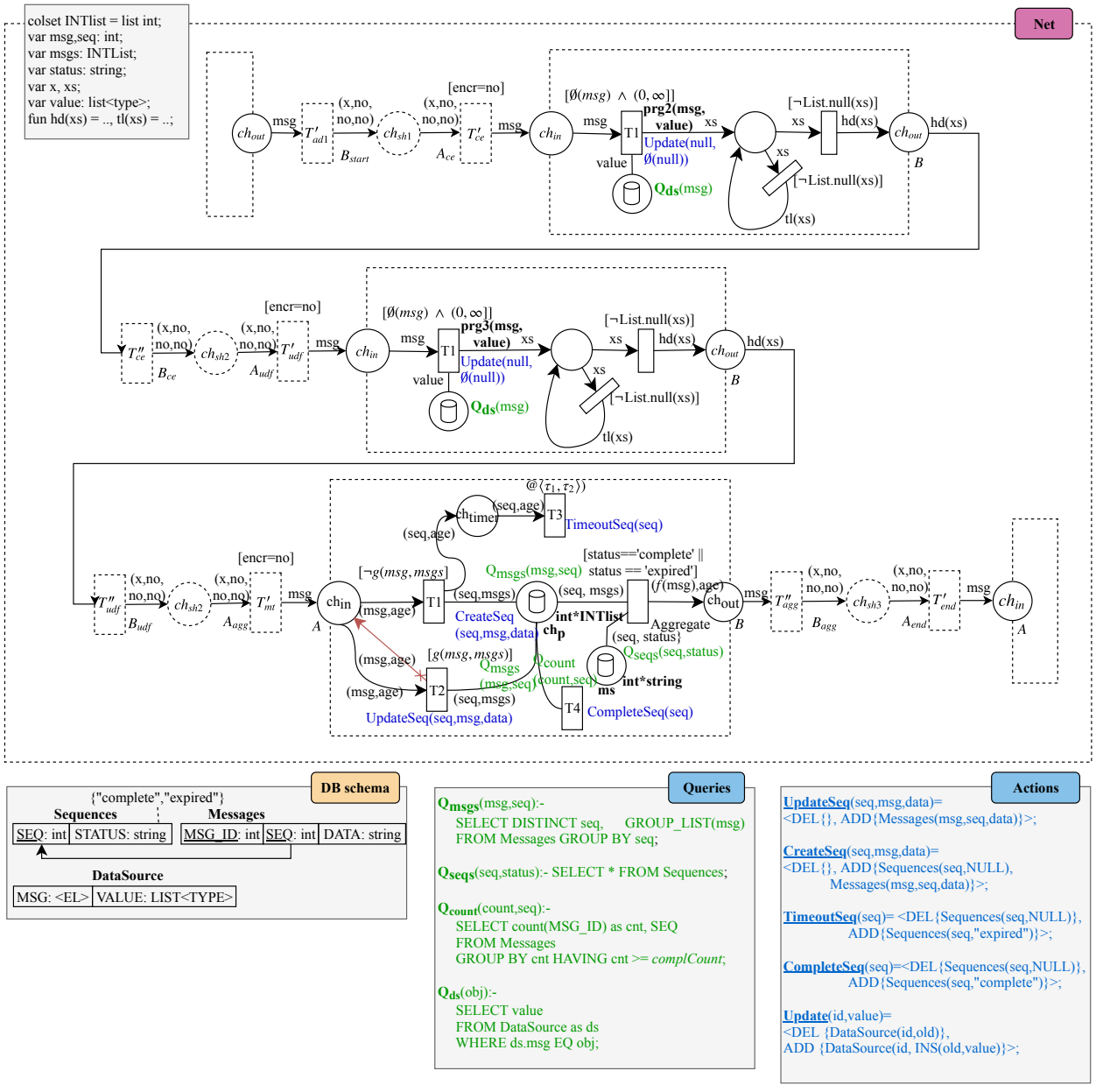


Figure 40: Predictive maintenance scenario as a timed DB-nets with boundaries

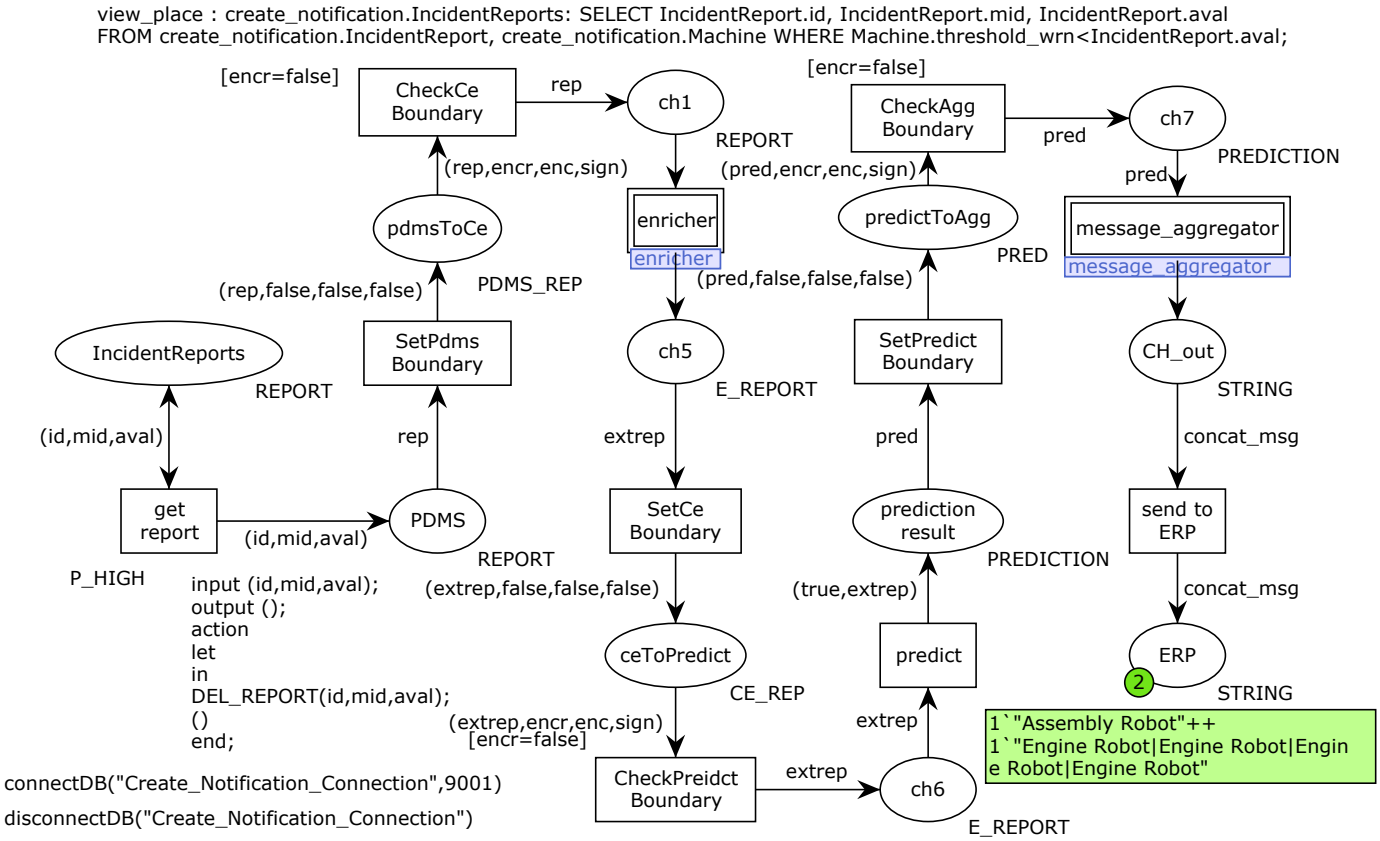


Figure 41: Predictive maintenance scenario simulation