Information Systems 101 (2021) 101439

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Formal foundations for responsible application integration

Daniel Ritter [a,b,*], Stefanie Rinderle-Ma [b], Marco Montali [c], Andrey Rivkin [c]

[a] SAP, Technology and Innovation, Germany
[b] University of Vienna, Faculty of Computer Science, Austria
[c] Free University of Bozen-Bolzano, Faculty of Computer Science, Italy

ARTICLE INFO

Article history:
Received 14 March 2019
Received in revised form 13 July 2019
Accepted 10 September 2019
Available online 18 September 2019
Recommended by Gottfried Vossen

Keywords:
Enterprise application integration
Enterprise integration patterns
Petri nets
Responsible programming
Trustworthy application integration

ABSTRACT

Enterprise Application Integration (EAI) constitutes the cornerstone in enterprise IT landscapes that are characterized by heterogeneity and distribution. Starting from established Enterprise Integration Patterns (EIPs) such as Content-based Router and Aggregator, EIP compositions are built to describe, implement, and execute integration scenarios. The EIPs and their compositions must be correct at design and runtime in order to avoid functional errors or incomplete functionalities. However, current EAI system vendors use many of the EIPs as part of their proprietary integration scenario modeling languages that are not grounded on any formalism. This renders correctness guarantees for EIPs and their composition impossible. Thus this work advocates responsible EAI based on the formalization, implementation, and correctness of EIPs. For this, requirements on an EIP formalization are collected and based on these requirements an extension of db-net, i.e., *timed db-net*, is proposed, fully equipped with execution semantics. It is shown how EIPs can be realized based on *timed db-nets* and how the correctness of these realizations can be shown. Moreover, the simulation of EIP realizations based on *timed db-nets* is enabled which is essential for later implementation. The concepts are evaluated in many ways, including a proof-of-concept implementation and case studies. The EIP formalization based on *timed db-nets* constitutes the first step towards responsible EAI.

1. Introduction

With the growing number of cloud and mobile applications, the importance of Enterprise Application Integration (EAI) [1] has immensely increased. Integration scenarios — essentially compositions of Enterprise Integration Patterns (EIPs) [2] and their recent extensions [3–6] — describe typical concepts in designing messaging systems as used for EAI (e.g., the communication between these applications). Due to the increasing heterogeneity of endpoints and their distribution, *trust* into productive integration solutions becomes even more essential. This, in turn, requires to provide means for a *responsible* development of integration solutions (cf. "responsible programming" [7]) in order to avoid design flaws such as functional errors or incomplete functionality, starting with the EIPs.

Similar to Cerf [7] we say that users who define integration solutions should have a clear sense of responsibility for their reliable operation as well as their resistance to compromise and error. This implies that the integration logic can only be trusted if, in principle, it is possible to prove that it behaves correctly. In turn, this means that users should be able to express and specify what their integration solution should do. We call a method that takes this into account a *responsible* development of integration solutions. The main properties of this principle are a formal treatment of integration patterns (i.e., their formalization within a framework with clearly defined syntax and semantics, and that supports formal analysis of its models), simulation and validation. We conjecture that this will become one of the key principles for the development in shared responsibility environments like cloud or mobile computing. Moreover, it will constitute the foundation for functionally correct compositions and correctness-preserving improvements (e.g., optimizations).

Contemporary EAI system vendors use many of the EIPs as part of their proprietary integration scenario modeling languages [3]. However, these languages are not grounded on any formalism and, hence, may produce models that are subject to the design flaws mentioned above. Due to the missing formal definition, currently the detection and analysis of these flaws are by large performed manually. This results in system engineers putting in huge effort that may potentially lead to mistakes. Hence, nowadays, EIPs should be regarded as a set of informal design solutions rather than a collection of models produced by a formal language and whose correctness can be verified, thus leaving the EAI vendors with their own proprietary semantics and not

* Corresponding author at: University of Vienna, Faculty of Computer Science, Austria.

E-mail addresses: danielr81@unet.univie.ac.at (D. Ritter), stefanie.rinderle-ma@univie.ac.at (S. Rinderle-Ma), montali@inf.unibz.it (M. Montali), rivkin@inf.unibz.it (A. Rivkin).

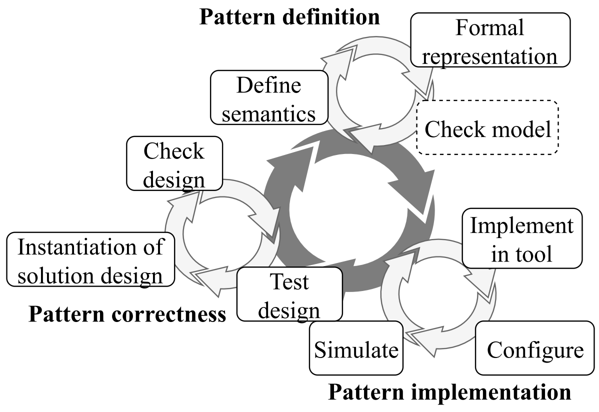


Fig. 1. *Responsible* pattern formalization process.

allowing for a responsible development. Our recent survey [3] identified a first attempt towards formalization of some EIPs using colored Petri nets (CPNs) [8]. Although the CPN colors abstractly account for data types, and CPNs support the control flow through control threads (i.e., tokens) progressing through the net, carrying data conforming to colors, they cannot be used to model, query, update, and reason on requirements inherent to the extended EIPs [2–6] such as persistent data or timings.

To overcome these limitations, a *responsible* development of integration solutions, i.e., solutions that can be thoroughly tested for their correctness at design time, requires the formalization of its pattern foundations. Therefore we adopt a responsible pattern formalization process that covers the following objectives: (i) defining the execution sematnics of EIPs and (ii) their realization, (iii) simulation of the EIP realizations as well as (iv) their validation and verification. Fig. 1 shows this process with its three main steps that we subsequently discuss: pattern formalization, pattern implementation, pattern correctness.

Definition The definition of a pattern starts with capturing and specifying its semantics. With a thorough understanding of the pattern and its variations, it can be formally represented. The resulting formal pattern model can be analyzed and verified (i.e., model checking). With model checking capabilities, errors in patterns can be found and either their semantics or formal representation is revisited.

Implementations If model checking is not possible or difficult, the formal patterns can be implemented, configured and simulated in a suitable tool. The simulation not only bridges the implementation gap, but allows for an experimental validation of a pattern.

Correctness The correctness of a pattern can be decided according to its semantics, when put into the context of a dedicated, scenario-specific configuration, and a test design, which specifies the desired properties like the expected output of a pattern simulation, for a given input. This test design is instantiated and checked during the simulation of the pattern. Any flaws found during this step can result into another round of formal or implementation adjustments.

We argue that existing approaches do not fully support a responsible development and hence the following research questions are formulated to guide the design and development of an EIP formalization process living up to objectives (i)–(iv):

Q1 Which EAI requirements are relevant for the formal definition of EIPs? To which extent are they met by existing approaches?
Q2 How to design an EIP formalization to meet relevant EAI requirements and objectives (i)–(iv)?
Q3 How to realize the EIPs and real-world integration scenarios?
Q4 How to accomplish correctness testing and simulation of EIP realizations?

The conference paper [9] that is extended in this work has provided the foundations for Q1–Q2 (and partially Q3). The formalization of EIPs is based on db-nets [10] as a database-centric extension of CPNs (with atomic transactions). In [9], db-nets have been extended by EAI requirements such as time, resulting in so called *timed db-nets*. It has been shown that with *timed db-nets* a responsible development of integration solutions becomes possible. The following example leverages *timed db-nets*, illustrating selected EIPs and their requirements on a formalization.

Example 1. A commonly used stateful Aggregator pattern [2] is designed to combine a number of input messages into a single output message. In the stateful version, the aggregator also stores collected messages in a persistent storage. The messages can be collected according to a Message Sequence [2], which identifies a group of coherent messages. Fig. 2 shows how the stateful aggregator pattern can be represented using timed db-nets. While the formal aspects will be specified throughout this work (e.g., data, queries, actions, transactions, time), the aggregator's semantics is subsequently described in timed db-net terms.

The aggregator starts with a number of input messages in its input channel that is represented with a place ch_{in}. Every message consists of a unique message identifier *msg* and an information block *data*, and will be eventually put into the database (cf. **DB schema** in Fig. 2 for more detailed database schema description). To collect messages from the input channel and assign them to correct sequences, the net correlates every incoming $(msg, data)$ token to those in place ch_p, that, in turn, stores pairs of sequences *seq* and lists of messages *msgs* that have been already collected and stored in the persistence storage.[1] If the message is the first in a sequence, new entries, one containing information about the message and another containing data about the referenced sequence, are added to tables called **Messages** and **Sequences**, respectively. This is achieved by checking guard condition $isFirst(msg, msgs) = \mathsf{true}$ attached to transition T1, and, if the condition holds, by firing that transition and executing action **CreateSeq** attached to it. The firing of T1 also starts a timer (cf. ch_{timer}) that in 30 time units can expire the sequence status (this is done by firing transition $\mathtt{T}_3$ that changes the sequence status in table **Sequences** to expired). If a message is not new in a sequence, then it is inserted into **Messages** by firing T2 and executing **UpdateSeq** (see a more detailed definition of updates in the **Actions** block in Fig. 2). However, the update by **UpdateSeq** may fail if a message is already in the database or a referenced sequence has already been aggregated due to a timeout (i.e., status is expired). In this case the net switches to an alternative roll-back flow (a directed arc from T2 to ch_{in}) and puts the message back to the input message channel ch_{in}.

The aggregation of a sequence of messages happens based either on the completion condition (i.e., the sequence status is complete) or on time-out of 30 time units (i.e., sequence status is expired), and is realized by firing transition `Aggregate`. Since the sequence completion logic is defined depending on a specific pattern application scenario, we use a subnet (a cloud element in the **Net** block of Fig. 2) denoting a configurable part of the model.

[1] To be more precise, ch_p is a special place that can be understood as a view over the database defined by a query $\mathtt{Q}_{\mathtt{msgs}}$ (cf. **Queries** in Fig. 2) and that can be accessed only by reading tokens from it.

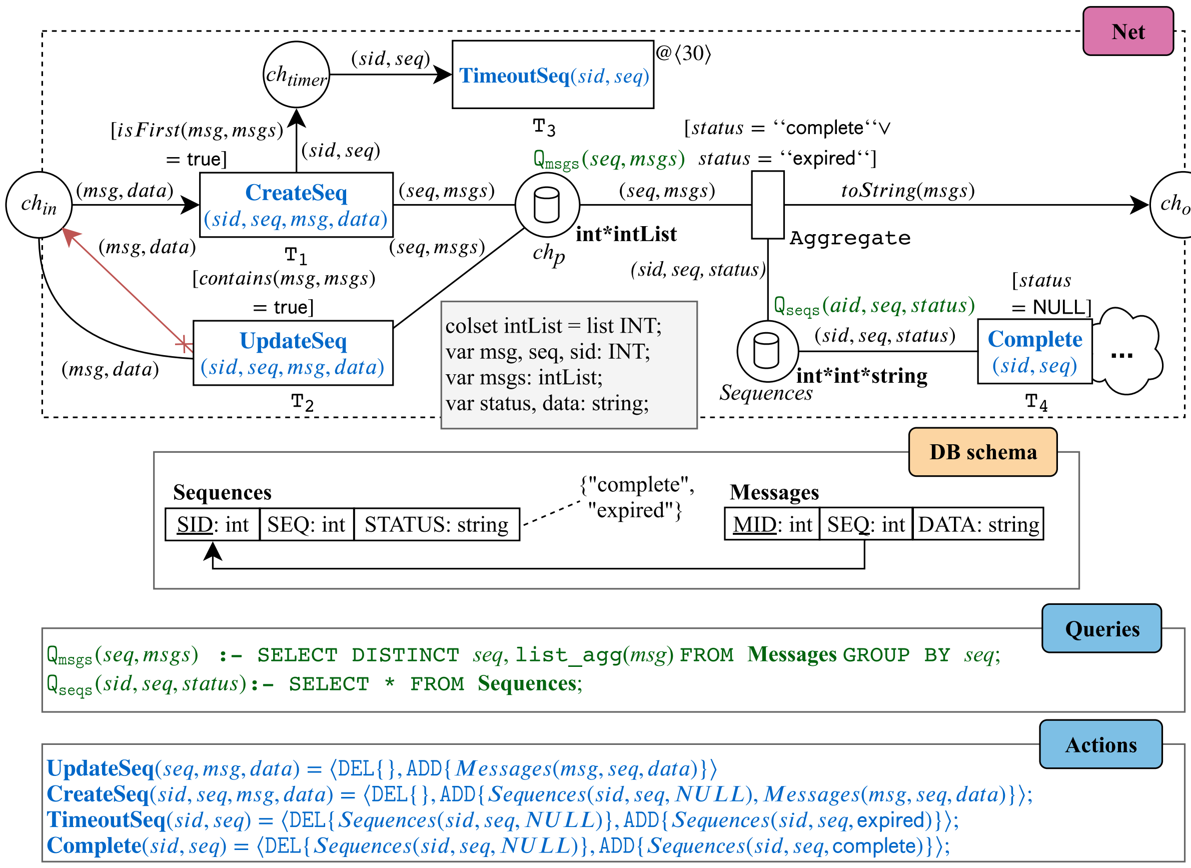

Net
ch_{timer}
(sid, seq)
TimeoutSeq(sid, seq)
@⟨30⟩
T_3
$[isFirst(msg, msgs) = true]$
(sid, seq)
$[status = ``complete`` \vee status = ``expired``]$
ch_{in}
$(msg, data)$
CreateSeq $(sid, seq, msg, data)$
$(seq, msgs)$
$Q_{msgs}(seq, msgs)$
$(seq, msgs)$
$toString(msgs)$
ch_{out}
T_1
int*intList
ch_p
Aggregate
$(msg, data)$
$(seq, msgs)$
$[contains(msg, msgs) = true]$
$(sid, seq, status)$
$Q_{seqs}(aid, seq, status)$
$[status = NULL]$
UpdateSeq $(sid, seq, msg, data)$
colset intList = list INT;
var msg, seq, sid: INT;
var msgs: intList;
var status, data: string;
$(sid, seq, status)$
Complete (sid, seq)
...
int*int*string
$(msg, data)$
T_2
Sequences
T_4
DB schema
Sequences
SID: int | SEQ: int | STATUS: string
{"complete", "expired"}
Messages
MID: int | SEQ: int | DATA: string
Queries
$Q_{msgs}(seq, msgs)$:- SELECT DISTINCT seq, list_agg(msg) FROM **Messages** GROUP BY seq;
$Q_{seqs}(sid, seq, status)$:- SELECT * FROM **Sequences**;
Actions
UpdateSeq$(seq, msg, data)$ = ⟨DEL{ }, ADD{$Messages(msg, seq, data)$}⟩
CreateSeq$(sid, seq, msg, data)$ = ⟨DEL{ }, ADD{$Sequences(sid, seq, NULL)$, $Messages(msg, seq, data)$}⟩;
TimeoutSeq(sid, seq) = ⟨DEL{$Sequences(sid, seq, NULL)$}, ADD{$Sequences(sid, seq,$ expired$)$}⟩;
Complete(sid, seq) = ⟨DEL{$Sequences(sid, seq, NULL)$}, ADD{$Sequences(sid, seq,$ complete$)$}⟩;

Fig. 2. Aggregator pattern variant as a timed db-net.

Note that the logic must always be realized in the designated subnet since we explicitly guard it with transition T_4 that executes an update (using action **Complete**) changing a given sequence state. ■

In this work we first devise a collection of EAI requirements on the EIP formalization process including a comprehensive assessment and elaboration on the selection of existing approaches, i.e., Petri nets (↦ Q1). Based on this, a formalism of db-nets is selected and equipped with missing EAI requirements, most prominently time, resulting in *timed db-nets* (↦ Q2). The extension also includes a study of suitable time formalisms (including the one we have chosen). The verification of reachability for *timed db-nets* is shown to be decidable based on an elaborated proof sketch. Next, an instructive catalog of pattern realizations is provided (↦ Q3). It is shown how to test the correctness of various formal EIP realizations based on their execution traces (↦ Q4). Finally, a prototypical implementation on top of CPN Tools [11] is used to develop a concept to experimentally test the correctness of EIPs realised in *timed db-net*.

The contributions to research questions Qx are elaborated following the principles of the design science research *methodology* described in [12]: *"Activity 1: Problem identification and motivation"* is based on literature and assessment of vendor-driven solutions (e.g., [3]) as well as on the harvested EAI requirements (i.e., existing catalogs with 166 integration patterns) in a quantitative analysis (cf. Q1). *"Activity 2: Define the objectives for a solution"* is addressed by formulating objectives (i)–(iv). For *"Activity 3: Design and development"* several artifacts are created to answer questions Q1–Q4 and meet objectives (i)–(iv), including the formalism for EIPs, the catalog of realizations of EIP formalizations, and the foundations for correctness testing. *"Activity 4: Demonstration"* and *"Activity 5: Evaluation"* are conducted based on a prototypical implementation and case studies.

The paper is structured as follows. In Section 2, pattern requirements are harvested from literature. The formalism for EIPs *timed db-nets* is presented in Section 3. An instructive catalog of formalized patterns is provided in Section 4. The foundations for correctness testing are lied in Section 5. In Section 6, we elaborate on the comprehensiveness of the formalism in a quantitative study, show a prototypical db-net realization for testing correctness, and discuss the general applicability of PNs and, in particular, timed db-nets for the composition of integration patterns in a real-world example. We conclude by discussing related work in Section 7 and outlining the main results and future research directions in Section 8.

2. Formalization requirements analysis

In this section, we collect the EAI requirements relevant for the formalization of the EIPs by analyzing the existing pattern catalogs [2–6] (cf. Q1). Then we briefly discuss which of them can be represented by the means of CPNs or db-nets, and which require further extensions.

2.1. Pattern analysis and categories

The EIP formalization requirements are derived by an analysis of the pattern descriptions based on the integration pattern catalogs from 2004 [2] (as `original`) and recent extensions [3–6] (as `extended`) that consider emerging EAI scenarios (e.g., cloud, mobile and internet of things). Together, the catalogs describe 166 integration patterns, of which we consider 139 due to their relevance for this work (e.g., excluding abstract concepts like

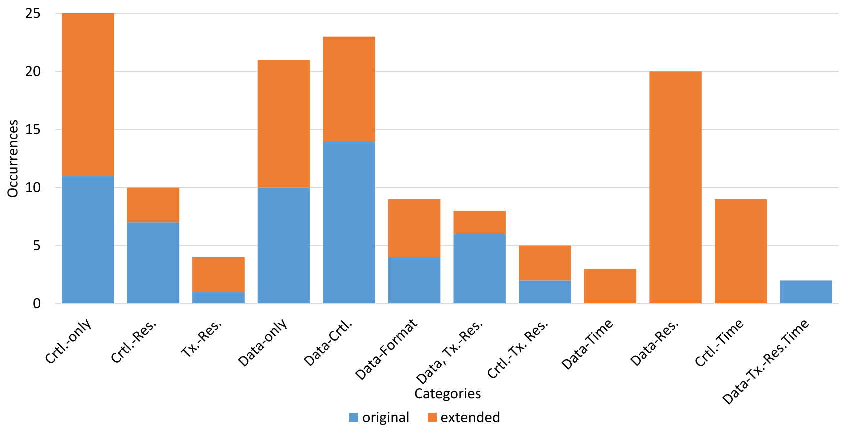


Fig. 3. EIP requirement categories (with Control (Crtl.), Resource (Res.), Transaction (Tx.)).

Canonical Data Model [2] or Messaging System [2]). During the analysis, we manually collected characteristics from the textual pattern descriptions (e.g., data, time) and created new categories, if not existent.

The reoccurring characteristics found in this work allow for a categorization of patterns, as summarized in Fig. 3, to systematically pinpoint relevant EAI requirements into general categories (with more than one pattern). Most of the patterns require (combinations of) *Data* flow, *Control* (Crtl.) flow, and *(Transacted) Resource* ((Tx.) Res.) access. While the control flow denotes the routing of a message from pattern to pattern via channels (i.e., ordered execution), the data flow describes the access of the actual message by patterns (including message content, headers, attachments). Notably, most of the patterns can be classified as control (Crtl.-only; e.g., Wire Tap [2]) and data only (Data-only; e.g., Splitter [2]) or as their combination (Data-Crtl.; e.g., Message Filter [2]), which stresses on the importance of data-aspects of the routing and transformation patterns. In addition, resources denote data from an external service that are not in the message (e.g., Data Store [3]). The EIP extensions add new categories like combinations of data and {time, resources} (Data-Time like Message Expiration [2,3], Data-Res. like Encryptor [3]) and control and time (Crtl.-Time; e.g., Throttler [3]). For instance, the motivating example in Fig. 2 is classified as *Data-Tx.-Res.-Time*. The different categories are disjoint with respect to patterns.

2.2. From categories to requirements

We assume that the control requirement **REQ-0 "Control flow"** is inherently covered by any PN approach, and thus it is by CPN and db-net. However, there are two particularities in the routing patterns that we capture in requirement **REQ-1 "Msg. channel priority, order"**: (a) the ordered evaluation of Msg. channel conditions or guards of sibling PN transitions, required for the Content-based Router pattern, (b) the enablement or firing of a PN transition according to a ratio for the realization of a Load Balancer [3]. In both cases, neither execution priorities nor ratios are trivially covered by CPNs or db-nets.

Furthermore, there are 77 patterns in the catalogs with data and 10 with message format aspects, which require an expressive CPN token representation (e.g., for encodings, security, complex message protocols), for which we add a second requirement **REQ-2 "Data, format"** that has to allow for the formal analysis of the data. Although CPNs and db-nets have to be severely restricted (e.g., finite color domains, pre-defined number of elements) for that, db-nets promise a relational representation that can be formally analyzed [10].

Table 1
Formalization requirements (covered √, partially (√), not –).

ID	Requirement	CPN	db-net
REQ-0	Control flow (pipes and filter)	√	√
REQ-1	(a) Msg. channel **priority**	(√)	(√)
	(b) Msg. channel **distribution**	–	(√)
REQ-2	**Data, format** including message protocol with encoding, security	(√)	√
REQ-3	(a) **Timeout** on message, operation	–	–
	(b) **Expiry date** on message	–	–
	(c) **Delay** of message, operation	–	–
	(d) **Msg./time ratio**	–	–
REQ-4	(a) **CRUD operations** on (external) resources	–	√
	(b) **Transaction semantics** on (external) resources (incl. roll-back)	–	√
REQ-5	**Exceptions, compensation** similar to roll-back in REQ-4	–	√

We capture the 11 patterns with time-related requirements as **REQ-3 "time"**: (a) *Timeout*: numerical representation of fixed, relative time (i.e., no global time); (b) *Expiry date*: discrete point in time according to a global time (i.e., based on existing message content); (c) *Delay*: numerical, fixed time value to wait or pause until continued (e.g., often used in a redelivery policy); (d) *Message/time ratio*: number of messages that are sent during a period of time. Consequently, a quantified, fixed time delay or duration semantics is required.

The 49 patterns with resources **REQ-4 "(external) Resources"** require: (a) create, retrieve, update and delete (**CRUD**) access to external services or resources, and (b) **transactional** semantics on a pattern level. Similarly, exception semantics are present in 28 patterns as **REQ-5 "Exceptions"**, which require **compensations** and other post-error actions. Consequently, a PN definition that allows for reasoning over time aspects and structured (persistent) data access is required.

2.3. Requirements summary

Table 1 summarizes the formalization requirements for timed db-nets by setting the coverage of the CPN [8] and db-net [10] approaches into context. While CPNs provide a solid foundation for control (cf. REQ-0) and a simple data flow representation (cf. REQ-2), db-nets extend it towards more complex data structures — message protocols in our case (cf. REQ2), and add CRUD operations (cf. REQ-4(a)), transactional semantics (cf. REQ-4(b)), and exception handling (cf. REQ-5), suitable for working with external, transactional resources. In CPNs, message channel distributions cannot be represented and priorities require explicit modeling, leading to complex models. In this work we build

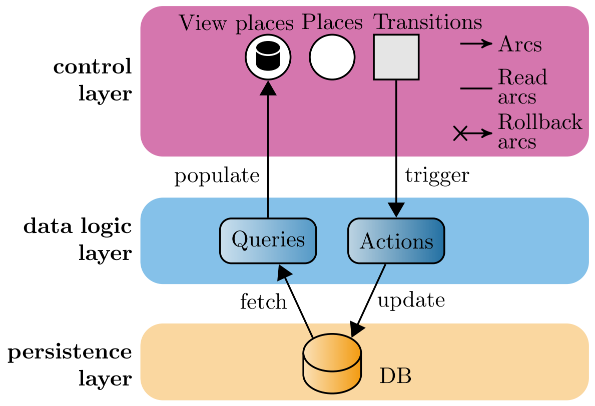


Fig. 4. The conceptual components of db-nets.
Source: From [10].

upon the CPN approach by subsequently defining timed db-nets in Section 3 for the time-related requirements (cf. REQ-3(a)–(d)) and provide (less complex) realizations for message channel priority execution (cf. REQ-1(a)) and load balancing (cf. REQ-4(b)) in Section 4.

3. Integration pattern formalization

We recall the main characteristics of db-nets [10], and then extend them with temporal features, obtaining a formalism called timed db-nets. We then show how decidability results on the formal analysis of db-nets can be lifted to timed db-nets (cf. Q2).

3.1. The db-net Framework

When facing the problem of formalizing multi-perspective models that suitably account for the dynamics of a system (i.e., the process perspective) and how it interacts with data (i.e., the data perspective), several design choices can be made. In the Petri net tradition, the vast majority of formal models striving for this integration approaches the problem by enriching execution threads (i.e., tokens) with complex data. Notable examples within this tradition are data nets [13] and ν-nets [14], Petri nets with nested terms [15], nested relations [16], and XML documents [17].

While all of the approaches treat data subsidiary to the control-flow dimension, the EIPs require data elements attached to tokens being connected to each other by explicitly represented global data models (cf. Section 2). Consequently, they do not allow for reasoning on persistent, relational data such as tree or graph structured message formats [18].

Db-net. The recently proposed framework of db-nets [10] aims at conceptually establishing this connection through a formal model that consists of three layers (cf. Fig. 4). On the one hand, a db-net separately represents a *persistence* storage (constituted by a full-fledged relational database with constraints) and a *control* (captured as a CPN with additional, specific constructs) flow. On the other hand, it explicitly handles their interplay through a *data logic* intermediate layer, which provides the control layer with queries and database operations (such as `trigger`, `update`, `read`, `bind`). Updates are transactional, that is, are only committed if the resulting instance of the persistence layer satisfies the database constraints. The control layer is informed about the outcome of an update, and can consequently compensate in case of a roll-back.

We select db-nets (see Definition 2) as a foundation of timed db-nets for three main reasons: (*i*) ability to represent relational data (cf. REQ-2: "data", "format"); (*ii*) built-in support for transactional CRUD operations (cf. REQ-4); (*iii*) exception handling and a corresponding compensation mechanism (cf. REQ-5). In addition, since db-nets are based on CPNs, it is possible to lift existing simulation techniques from CPNs to db-nets [10].

In the remainder of this section, we recall the definition of a db-net and its execution semantics.

Definition 2 ([10]). A *db-net* is a tuple $\langle \mathfrak{D}, \mathcal{P}, \mathcal{L}, \mathcal{N} \rangle$, where:

- $\mathfrak{D}$ is a type domain — a finite set of data types, each of the form $D = \langle \Delta_D, \Gamma_D \rangle$, where Δ_D is the value domain of D, and Γ_D is a set of domain-specific (rigid) predicates.
- $\mathcal{P}$ is a $\mathfrak{D}$-typed **persistence layer**, i.e., a pair $\langle \mathcal{R}, E \rangle$, where $\mathcal{R}$ is a $\mathfrak{D}$-typed database schema, and E is a finite set of first-order FO($\mathfrak{D}$) constraints over $\mathcal{R}$.[2]
- $\mathcal{L}$ is a $\mathfrak{D}$-typed **data logic layer** over $\mathcal{P}$, i.e., a pair $\langle Q, A \rangle$, where Q is a finite set of FO($\mathfrak{D}$) queries over $\mathcal{P}$, and A is a finite set of actions over $\mathcal{P}$. Each action in A is parameterized, and uses its parameters to express a series of insertions and deletions over $\mathcal{P}$.
- $\mathcal{N}$ is a $\mathfrak{D}$-typed **control layer** $\mathcal{L}$, i.e., a tuple $(P, T, F_{in}, F_{out}, \mathtt{color}, \mathtt{query}, \mathtt{guard}, \mathtt{act})$, where:
  - $P = P_c \cup P_v$ is a finite set of places partitioned into control places P_c and so-called view places P_v,
  - T is a finite set of transitions,
  - F_{in} is an input flow from P to T
  - F_{out} and F_{rb} are respectively an output and rollback flow from T to P_c
  - `color` is a color assignment over P (mapping P to a cartesian product of data types),
  - `query` is a query assignment from P_v to Q (mapping the results of Q as tokens of P_v),
  - `guard` is a transition guard assignment over T (mapping each transition to a formula over its input inscriptions), and
  - `act` is an action assignment from T to A (mapping some transitions to actions triggering updates over the persistence layer). ■

Input and output/roll-back flows contain inscriptions that match the components of colored tokens present in the input and output/roll-back places of a transition. Such inscriptions consist of tuples of (typed) variables, which then can be mentioned in the transition guard as well as in the action assignment (to bind the updates induced by the action to the values chosen to match the inscriptions), and also, in case of the output flow, the inscriptions may contain rigid predicates. Specifically, given a transition t, we denote by $InVars(t)$ the set of variables mentioned in its input flows, by $OutVars(t)$ the set of variables mentioned in its output flows, and by $Vars(t) = InVars(t) \cup OutVars(t)$ the set of variables occurring in the action assignment of t (if any). Fresh variables $FreshVars(t) = OutVars(t) \setminus InVars(t)$ denote those output variables that do not match any corresponding input variables, and are consequently interpreted as external inputs. While input inscriptions are used to match tokens from the input places to $InVars(t)$, the output expressions that involve rigid predicates operate over $OutVars(t)$. In case of numerical types, these expressions can be used to compare values, or to arithmetically operate over them. We call *plain* a db-net that employs matching output inscriptions only (i.e., does not use expressions).

Intuitively, each view place is used to expose a portion of the persistence layer in the control layer, so that each token

[2] These constraints capture typical database constraints such as key and foreign key dependencies.

represents one of the answers produced by the query attached to the place. Such tokens are not directly consumed, but only read by transitions, so as to match the input inscriptions with query answers. A transition in the control layer may bind its input inscriptions to the parameters of data logic action attached to the transition itself, thus providing a mechanism to trigger a database update upon transition firing (and consequently indirectly change also the content of view places). If the induced update commits correctly, the transition emits tokens through its output arcs, whereas if the update rolls back, the transition emits tokens through its rollback arcs.

The terms message and (db-net, CPN) token will be used synonymously hereinafter.

Db-net execution semantics. We briefly recall the execution semantics of db-nets. A state of a db-net captures at once a state of the persistence layer (i.e., an instance of the database), and that of the control layer (i.e., a net marking, where the content of view places must be compatible with that of the database instance). More technically, in each moment (called *snapshot*) the persistence layer is associated to a database instance I, and the control layer is associated to a marking m aligned with I via `query` (for what concerns the content of view places). The corresponding snapshot is then simply the pair $\langle I, m\rangle$. Tokens in m have to carry data compatible with the color of the places and the marking of a view place P_v must correspond to the associated queries over the underlying database instance.

Similar to CPNs, the firing of a transition t in a snapshot is defined by a binding that maps the value domains of the different layers, if several properties are guaranteed, e.g., the guard attached to t is satisfied. More specifically, we have the following.

Definition 3 (*Transition Enablement [10]*). Let B be a db-net $\langle \mathfrak{D}, \mathcal{P}, \mathcal{L}, \mathcal{N}, \rangle$, and t a transition in $\mathcal{N}$. Let σ be a binding for t, i.e., a substitution $\sigma : Vars(t) \to \Delta_{\mathfrak{D}}$.[3] A transition $t \in T$ is *enabled* in a B-snapshot $\langle I, m\rangle$ with binding σ, if:

- For every place $p \in \mathcal{P}$, $m(p)$ provides enough tokens matching those required by inscription $w = F_{in}(\langle p, t\rangle)$, once w is grounded by σ, i.e., $\sigma(w) \subseteq m(p)$;
- the guard $\mathtt{guard}(t)\sigma$ evaluates to true;
- σ is injective over $FreshVars(t)$, thus guaranteeing that fresh variables are assigned to pairwise distinct values of σ, and for every fresh variable $v \in FreshVars(t)$, $\sigma(v) \notin (Adom_{\mathtt{type}(v)}(I) \cup Adom_{\mathtt{type}(v)}(m))$.[4] ■

Firing an enabled transition has the following effects: (*i*) all matching tokens in control places P_c are consumed; (*ii*) the action instance `action` — induced by the firing — is applied on the current database instance in an atomic transaction (and rolled back, if not successful); (*iii*) accordingly, tokens on output places F_{out} or rollback places F_{rb} (i.e., those connected via rollback flow) are produced. Technically, we have the following.

Definition 4 (*Transition Firing [10]*). Let B be a db-net $\langle \mathfrak{D}, \mathcal{P}, \mathcal{L}, \mathcal{N}\rangle$, and $s_1 = \langle I_1, m_1\rangle$, $s_2 = \langle I_2, m_2\rangle$ be two B-snapshots. Fix a transition t of $\mathcal{N}$ and a binding σ such that t is enabled in s_1 with σ (cf. Definition 3). Let $I_3 = apply(action_\sigma(t), I_1)$ be the database instance resulting from the application of the action attached to t on database instance I_1 with binding σ for the action parameters. For a control place p, let $w_{in}(p, t) = F_{in}(\langle p, t\rangle)$, and $w_{out}(p, t) = F_{out}(\langle p, t\rangle)$ if I_3 is compliant with $\mathcal{P}$, or $w_{out}(p, t) = F_{rb}(\langle p, t\rangle)$ otherwise. We say that t *fires* in s_1 with binding σ producing s_2, written $s_1[t, \sigma\rangle s_2$, if:

- if I_3 is compliant with $\mathcal{P}$, then $I_2 = I_3$, otherwise $I_2 = I_1$;
- for each control place p, m_2 corresponds to m_1 with the following changes: $\sigma(w_{in}(p, t))$ tokens are removed from p, and $\sigma(w_{out}(p, t))$ are added to p. In formulas: $m_2(p_c) = (m_1(p_c) - \sigma(w_{in}(p, t))) + \sigma(w_{out}(p, t))$. ■

All in all, the complete execution semantics of a db-net is captured by a possibly infinite-state transition system where each transition represents the firing of an enabled transition in the control layer of the net with a given binding, and each state is a snapshot. The infinity comes from the presence of external inputs, and the fact that value domains may contain infinitely many elements. It is important to notice that the resulting transition system may be infinite even if the control layer is bounded in the classical Petri net sense.

Example 5. The aggregator in Fig. 2 requires a view place ch_p (denoted by –⊜) for storing and updating the message sequences as well as rollback arc ($\mathtt{T}_2$, ch_{in}) to manage compensation tasks (represented as ←×). The graphical notation is in line with [10]. ■

3.2. Timed db-nets

We now extend the db-net model so as to account for an explicit notion of time. While the implicit temporal support in PNs (i.e., adding places representing the current time) is rather counterintuitive [19], the temporal semantics of adding timestamps to tokens [19], timed places [20], arcs [21] and transitions [22] are well studied and naturally capture different facets of time in dynamic systems. The temporal requirements in REQ-3 demand a quantified, fixed or discrete time representation by timed transitions or places, representing the delay induced by a transition firing. This is currently missing in db-nets. So, in the spectrum of timed extensions to PNs, we extend the db-net control layer $\mathcal{N}$ with a temporal semantics that achieves a suitable trade-off: it is expressive enough to capture the requirements in REQ-3, and at the same time it allows us to transfer the existing technical results on the verification of db-nets to the timed extension.

We start by explaining the intuition behind the approach, and then provide the corresponding formalization. We assume that there is a global, continuous notion of time. The firing of a transition is instantaneous, but can only occur in certain moments of time, while it is inhibited in others, even in presence of the required input tokens. Every *control token*, that is, token assigned to a control place, carries a (local) *age*, indicating how much time the token is spending in that control place. This means that when a token enters into a place, it is assigned an age of 0. The age then increments as the time flows and the token stays put in the same place. View places continuously access the underlying persistence layer, and consequently their (virtual) tokens do not age. Each transition is assigned to a pair of non-negative (possibly identical) rational numbers, respectively describing the minimum and maximum age that input tokens should have when they are selected for firing the transition. Thus, such numbers identify a relative time window that expresses a delay and a deadline on the possibility of firing.

Definition 6. A *timed db-net* is a tuple $\langle \mathfrak{D}, \mathcal{P}, \mathcal{L}, \mathcal{N}, \tau\rangle$ where $\langle \mathfrak{D}, \mathcal{P}, \mathcal{L}, \mathcal{N}\rangle$ is a db-net with the control layer $\mathcal{N}$, and $\tau : T \to \mathbb{Q}^{\geq 0} \times (\mathbb{Q}^{\geq 0} \cup \{\infty\})$ is a *timed transition guard* mapping each transition $t \in T$ to a pair of values $\tau(t) = \langle v_1, v_2\rangle$, such that: (*i*) v_1 is a non-negative rational number; (*ii*) v_2 is either a non-negative rational number equal or greater than v_1, or the special constant ∞. ■

[3] We assume σ to be naturally extended to arc inscriptions. In case when an arc inscription contains an expression, σ will be applied to its variables.

[4] $Adom_D(X)$ is the set of values of type D explicitly contained in X.

The default choice for τ is to map transitions to the pair $\langle 0, \infty\rangle$, which corresponds to a standard db-net transition.

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

Example 7. The aggregator in Fig. 2 defines a timed transition T3, that can be fired precisely after 30 time units (here seconds) from the moment when a new sequence seq has been created. Upon firing, T_3 enables the Aggregate transition, by updating the sequence's status on the database to expired using the **TimeoutSeq** action. ■

Timed db-net execution semantics. The execution semantics of timed db-net builds on the one for standard db-nets, extended with additional conditions on the flow of time and the temporal enablement of transitions. The management of bindings, guards, and database updates via actions, is kept unaltered. What changes is that, in a snapshot, each token now comes with a corresponding age, represented as a number in $\mathbb{Q}^{\geq 0}$.

As customary in several temporal extensions of Petri nets, we consider two types of evolution step. The first type deals with *time lapses*: it indicates that a certain amount of time has elapsed with the net being quiescent, i.e., not firing any transition. This results in incrementing the age of all tokens according to the specified amount of time.

The second type deals with a transition firing, which refines that of db-nets by checking that the chosen binding selects tokens whose corresponding ages are within the delay window attached to the transition. Specifically, let B be a timed db-net $\langle \mathfrak{D}, \mathcal{P}, \mathcal{L}, \mathcal{N}, \tau\rangle$, t a transition in $\mathcal{N}$ with $\tau(t) = \langle v_1, v_2\rangle$, and σ a binding for t. We say that t is enabled in a given B snapshot with binding σ if it is so according to Definition 3 and, in addition, all the tokens selected by σ have an age that is between v_1 and v_2. Firing an enabled transition is identical to the case of standard db-nets (cf. Definition 4), with the only addition that for each produced token, its age is set to 0 (properly reconstructing the fact that it is entering into the corresponding place).

The execution semantics of a timed db-net then follows the standard construction (using the refined notions of enablement and firing), with the addition that each snapshot may be subject to an arbitrary time lapse. This is done by imposing that every B-snapshot $\langle I, m\rangle$ is connected to every B-snapshot of the form $\langle I', m'\rangle$ where:

- $I' = I$ (i.e., the database instances are identical);
- m' is identical to m except for the ages of tokens, which all get incremented by the same, fixed amount $x \in \mathbb{Q}$ of time.

Given two B-snapshots s and s', we say that s *directly leads* to s', written $s \rightarrow s'$, if there exists a direct transition from s to s' in the transition system that captures the execution semantics of B. This means that s' results from s because of a transition firing or a certain time lapse. We extend this notion to finite execution traces $s_0 \rightarrow \ldots \rightarrow s_n$. We also write $s \overset{*}{\rightarrow} s'$ if s directly or indirectly leads to s'. If this is the case, we say that s' *reachable* from s.

Example 8. To complete the aggregator, when the persisted sequence in the aggregator is complete or the sequence times out, then the enabled Aggregate transition fires by reading the sequence number seq and snapshot of the sequence messages, and moving an aggregate msg' to ch_{out}. Notably, the Aggregate transition is invariant to which of the two causes led to the completion of the sequence. ■

3.3. Checking reachability over timed db-nets

Checking fundamental correctness properties such as *safety/ reachability* is of particular importance for timed db-nets, in the light of the subsequent discussion in Section 6.2 on reachable goal states. We consider here, in particular, the following relevant REACH-TEMPLATE problem:

Input: (*i*) a timed db-net B with set P_c of control places, (*ii*) an initial B-snapshot s_0, (*iii*) a set $P_{empty} \subseteq P_c$ of *empty control places*, (*iv*) a set $P_{filled} \subseteq P_c$ of *nonempty control places* such that $P_{empty} \cap P_{filled} = \emptyset$.

Output: *yes* if and only if there exists a finite sequence of B-snapshots of the form $s_0 \rightarrow \ldots \rightarrow s_n = \langle I_n, m_n\rangle$ such that for every place $p_e \in P_{empty}$, we have $|m_n(p_e)| = 0$, and for every place $p_f \in P_{filled}$, we have $|m_n(p_f)| > 0$.

Checking the emptiness of places in the target snapshot is especially relevant in the presence of timed transitions, so as to predicate over runs of the systems were tokens are consumed within the corresponding temporal guards. For example, by considering transition T3 in Fig. 2, asking for the $\mathtt{ch}_{\mathtt{timer}}$ place to be empty guarantees that T3 indeed triggered whenever enabled.

Since timed db-nets build on db-nets, reachability is highly undecidable, even for nets that do not employ timed transitions, have empty data logic and persistence layers, and only employ simple string colors. As pointed out in [10], this setting is in fact already expressive enough to capture ν-nets [13,14], for which reachability is undecidable. Similar undecidability results can be obtained by restricting even more the control layer, but allowing for the insertion and deletion of arbitrarily many tuples in the underlying persistence layer.

However, when controlling the size of information maintained by the control and persistence layers in each single snapshot, reachability and also more sophisticated forms of temporal model checking become decidable for db-nets using string and real data types (without arithmetics).

In particular, decidability has been shown for *bounded*, *plain* db-nets. Technically, a db-net B with initial snapshot s_0 is:

- **width-bounded** if there is $b \in \mathbb{N}$ s.t., for every B-snapshot $\langle I, m\rangle$, if $s_0 \overset{*}{\rightarrow} \langle I, m\rangle$, then the number of distinct data values assigned by m to the tokens residing in the places of B is bounded by b;
- **depth-bounded** if there is $b \in \mathbb{N}$ s.t., for every B-snapshot $\langle I, m\rangle$, if $s_0 \overset{*}{\rightarrow} \langle I, m\rangle$, then the number of appearances of each distinct token assigned by m to the places of B is bounded by b;
- **state-bounded** if there is $b \in \mathbb{N}$ s.t., for every B-snapshot $\langle I, m\rangle$, if $s_0 \overset{*}{\rightarrow} \langle I, m\rangle$, we have $|\cup_{D\in\mathfrak{D}} Adom_D(I)| \leq b$.

We say that a db-net is *bounded*, if it is at once width-, depth-, and state-bounded. Intuitively, a db-net is bounded if it does not accumulate unboundedly many tokens in a place, and guarantees that the number of data objects used in each database instance does not exceed a pre-defined bound.

The decidability of reachability for bounded db-nets does not imply decidability of reachability for bounded timed db-nets. In fact, ages in timed db-nets are subject to comparison and (global) increment operations that are not expressible in db-nets. However, we can prove decidability by resorting to a *separation* argument: the two dimensions of infinity respectively related to the infinity of the data domains and of the flow of time can in fact be tamed orthogonally to each other. In particular, we get the following.

Theorem 1. *The* REACH-TEMPLATE *problem is decidable for bounded and plain timed db-nets with initial snapshot.*

Proof sketch. Consider a bounded timed db-net B with initial snapshot s_0, empty control places P_{empty}, and filled control places P_{filled}. Using the faithful data abstraction techniques presented in [10, Thm 2], one obtains a corresponding timed db-net B' enjoying two key properties. First, B' is bisimilar to B, with a data-aware notion of bisimulation that takes into account both the dynamics induced by the net, as well as the correspondence between data elements. Such a notion of bisimulation captures reachability as defined above, and consequently REACH-TEMPLATE(B, s_0, P_{empty}, P_{filled}) returns *yes* if and only if REACH-TEMPLATE(B', s_0, P_{empty}, P_{filled}) does so. Second, the only source of infinity, when characterizing the execution semantics of B', comes from the temporal aspects, and in particular the unboundedness of token ages. This means that B' can be considered as a "standard" temporal variant of a CPN with bounded colors that, in turn, boils down to a temporal variant of an (uncolored) P/T net. In particular, one can easily see that B' corresponds to a specific type of bounded Timed-Arc Petri Net (TAPN) [21], a classical P/T net with continuous time, where the tokens carry an age and arcs between places and transitions are labeled with time intervals that restrict the age of tokens:

- whenever B' contains a transition t with $\tau(t) = \langle v_1, v_2\rangle$, its corresponding TAPN labels each arc entering in t with the same interval $[v_1, v_2]$;
- each transition-place arc is labeled with the interval $[0, 0]$.

Consequently, the infinity of B' can be tamed using standard techniques known for *bounded* TAPNs, which indeed enjoy decidability of reachability for the queries tackled by REACH-TEMPLATE [23, 24]. In particular, notice that REACH-TEMPLATE does not explicitly express constraints on the expected token ages when reaching the final state. □

It is interesting to notice that TAPNs have a more expressive mechanism to specify temporal guards in the net. In fact, TAPNs attach temporal guards to arcs, not transitions, and can therefore express different age requirements for different places, as well as produce tokens with an age nondeterministically picked from a specified interval. Hence, this more refined temporal semantics can be seamlessly introduced in our timed db-net model without compromising Theorem 1.

4. Formal pattern realizations

In this section we discuss (formal) pattern realizations using timed db-nets. Due to the high number of patterns, the formalization and corresponding in-detail description of all of them seems impractical. However, thanks to the fact that patterns can be classified into disjoint categories (see the requirement categories in Section 2), it suffices to discuss the most representative ones from each of such categories. We call this an *instructive pattern formalization*, which strives to formalize the patterns and, at the same time, offers modeling guidelines for other patterns of the respective categories using the provided examples. The structure of subsequent sections is as follows: first we give a brief description of the pattern and its aim in the context of the requirements, then we discuss its relevance as candidate, and finally specify a realization.

4.1. Control flow: Load balancer

Control flow only patterns are those that route the token flow without looking inside the actual content of the message.

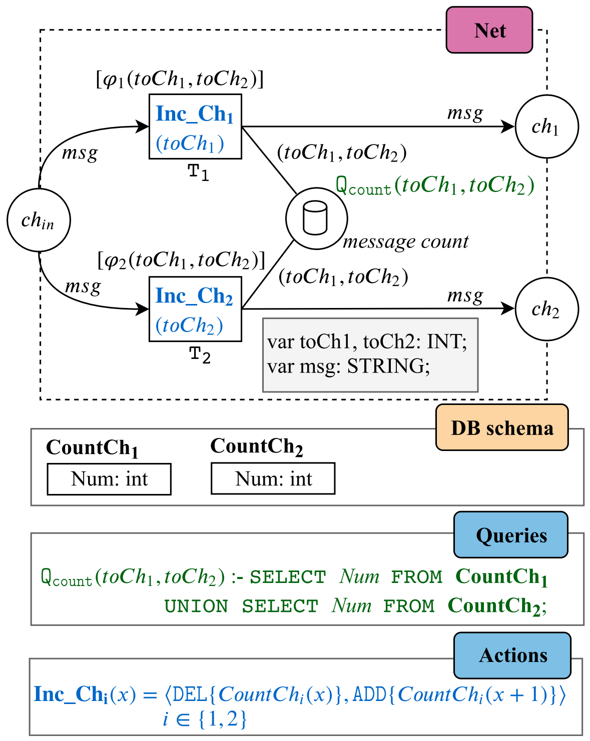

Net
$[\varphi_1(toCh_1, toCh_2)]$
Inc_Ch$_1$ $(toCh_1)$
T$_1$
msg
ch_1
msg
$(toCh_1, toCh_2)$
ch_{in}
$Q_{count}(toCh_1, toCh_2)$
message count
$[\varphi_2(toCh_1, toCh_2)]$
$(toCh_1, toCh_2)$
msg
Inc_Ch$_2$ $(toCh_2)$
msg
ch_2
T$_2$
var toCh1, toCh2: INT;
var msg: STRING;
DB schema
CountCh$_1$
Num: int
CountCh$_2$
Num: int
Queries
$Q_{count}(toCh_1, toCh_2)$:- SELECT Num FROM CountCh$_1$ UNION SELECT Num FROM CountCh$_2$;
Actions
Inc_Ch$_i(x) = \langle$DEL$\{CountCh_i(x)\}$, ADD$\{CountCh_i(x+1)\}\rangle$
$i \in \{1, 2\}$

Fig. 5. Load balancer realization in timed db-net.

Candidate Selection. To demonstrate the *control flow only* pattern (cf. requirement REQ-0 in Table 1), we have chosen the Load Balancer pattern [3]. Interestingly, this pattern also covers the message channel distribution requirement (cf. requirement REQ-1(b) in Table 1) and thus can be considered as a relevant candidate of this category as well.
Candidate Description. In a nutshell, the balancer distributes the incoming messages to a number of receivers based on a criterion that uses some probability distribution or ratio defined on the sent messages.
Pattern Realization. To realize the probability- or ratio-based criterion in Petri nets, one could adopt the stochastic Petri nets [25, 26] or extend the db-net transition guards definition with an ability to sample probability values from a probability distribution (e.g., [27]). While the latter would extend the db-net persistence layer, it is unclear whether the decidability results discussed in the previous section will still hold. Hence, we opted for the ratio criterion that, as shown in Fig. 5, is realized using a persistence storage and transition guards with a simple balancing scheme. Specifically, a message *msg* in channel ch_{in} leads to a lookup of the current ratio by accessing the current message counts per output channel in the database (via the view place *message count*) and evaluating guards assigned to one of the two transitions based on the extracted values. The ratio criterion is set up with two (generic) guards $\varphi_1(toCh_1, toCh_2)$ and $\varphi_2(toCh_1, toCh_2)$ respectively assigned to T$_1$ and T$_2$. If one of the guards holds, the corresponding transition fires by moving the message to its output place as well as updating the table by incrementing the corresponding channel count. The latter is done by executing action **Inc_Ch$_\mathbf{i}$**$(toCh_i)$ that consecutively performs $Inc\_Ch_i \cdot \mathtt{del} = \{CountCh_i(toCh_i)\}$ and $Inc\_Ch_i \cdot \mathtt{add} = \{NumberCh_i(toCh_i + 1)\}$ (for $i \in \{1, 2\}$).[5]

[5] Note that in db-nets an update is realized by first deleting a tuple and then adding its modified version.

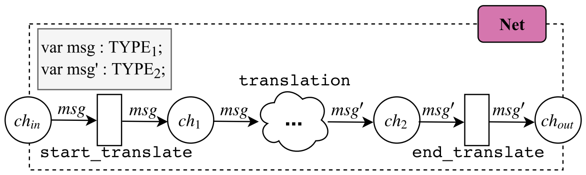


Fig. 6. Message translator realization in timed db-net.

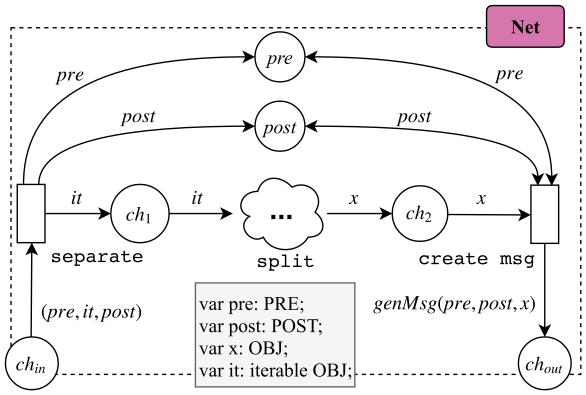


Fig. 7. Splitter realization in timed db-net.

4.2. Data flow: Message translator, splitter

Data flow only patterns are those that access the actual content of the message mostly for data filtering and transformation purposes.

Candidate Selection. The stateless Message Translator [2] is the canonical example of a *data flow only* pattern (cf. requirement REQ-2 in Table 1).

The (iterative) Splitter [2] is an example of a non-message transformation *data flow only* pattern, which is also required for a case study scenario in Section 6.

Candidate Description. The translator works on the data representation level (i.e., data flow) and transforms incoming messages of type TYPE$_1$ into (output) messages of type TYPE$_2$. The translation mechanism is defined a-priori by the user.

The splitter represents a complex routing mechanism that, given an iterable collection of input messages *it* together with two objects *pre* and *post*, is able to construct for each of its elements a new message of the form $[\langle pre\rangle][\langle it : msg_i\rangle][\langle post\rangle]$ with optional *pre* and *post* parts.

Pattern Realizations. A timed db-net representing a message translator is shown in Fig. 6. As in the pattern's definition, the corresponding net performs the message transformation from one type to another. Specifically, an incoming message of type TYPE$_1$ from input channel place ch_{in} is consumed (after firing `start_translate`) by subnet `translation` that, in turn, produces (by firing `end_translate`) a new message of type TYPE$_2$ into the receiver place ch_{out}.

As for the Splitter pattern, we show in Fig. 7 that, under certain restrictions assumed for the type of the iterable collection at hand, the pattern can be fully realized using only colored Petri nets. The entering message payloads in ch_0 are separated into its parts: *pre*, *post* and *it*. While the first two are remembered during the processing, the iterable *it* is iteratively split into parts according to some criterion realized in the `split` subnet, which represents a custom split logic and thus is intentionally left unspecified (in Fig. 7 it is marked with a cloud symbol).

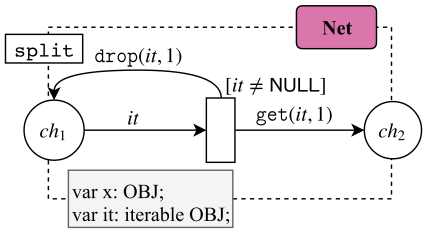


Fig. 8. Sample `split` subnet realization.

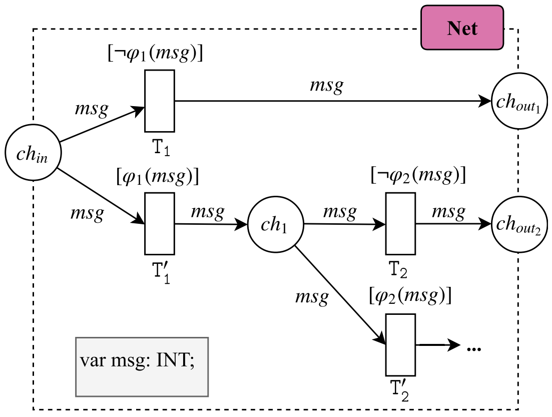


Fig. 9. Content-based router realization in timed db-net.

Example 9. The `split` subnet can be adapted to the message format and the requirements of a specific scenario. Fig. 8 demonstrates a possible implementation of the subnet. Here, functions `get` and `drop` are used to read and remove the *n*-th element of an iterable object. In our case, we alternate their applications to the *iterable* object *it* from place ch_1 in order to extract and delete its first element that is then placed into ch_2. Such a procedure is repeated until *it* is empty (i.e., it is NULL). ■

Each of extracted elements from *it*, together with the information about *pre* and *post*, is then used to create a new message (by calling function *genMsg*) that is passed to the output channel ch_{out} (Fig. 8).

4.3. Data and control flow: Content-based router

Data and control flow patterns consider the actual content of the message for routing a token through the net.

Candidate Selection. The Content-based Router pattern [2] is the canonical candidate for *data and control flow* patterns and also one of the mostly used integration patterns (cf. [28]).

Candidate Description. The Content-Based Router examines the message content and routes the message onto a different channel based on data contained in the message. The routing conditions have to be executed in a pre-defined order (cf. requirement REQ-1(a) in Table 1).

Pattern Realization. The realization of the Content-based Router pattern with conditions φ_1, φ_2 is shown in Fig. 9. Given that the router can be realized using various strategies imposed on the conditions (for example, using priority function similarly to [25]), we opted for more explicit realization where conditions have to be evaluated strictly in-order (cf. requirement REQ-1(a) in Table 1), using pair-wise negated db-net transition guards. In our pattern realization, a message from input channel place ch_{in} is first evaluated against condition φ_1. Based on the evaluation result, the message is moved either to ch_1 or ch_2. In case it has

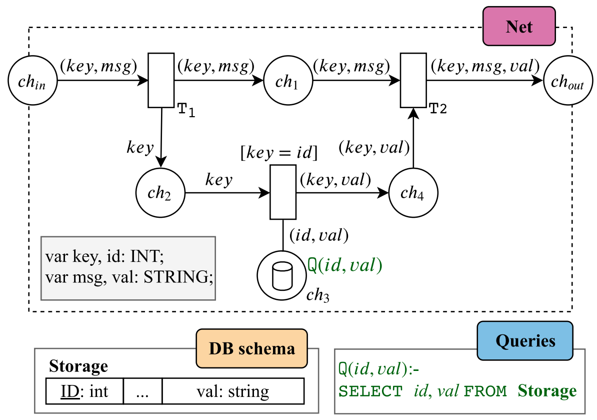


Fig. 10. Content enricher realization in timed db-net.

been moved to ch_2, the net proceeds with the subsequent evaluation of other conditions using the same pattern. When none of the guards can be evaluated, a non-guarded, low-priority default transition fires (not shown). The provided realization covers the router's semantics, however, requires $(k \times 2) + 1$ transitions (i.e., condition, negation, and only one default), with the number of conditions k.

4.4. Data flow with transacted resources: Content enricher, resequencer

Data flow with transacted resource patterns consider the actual content of the message as well as additional information from transacted sources mostly for message transformations.

Candidate Selection. The first pattern chosen for this category, such that it includes a data flow with *transacted resources*, is the Content Enricher [2] (cf. requirements REQ-2, REQ-4 in Table 1), which mainly reads from the transacted resource. Hence we add the stateful Resequencer [2], which writes to a transacted resource.

Candidate Description. The content enricher requires accessing external resources (e.g., relational database) based on *data* from an incoming message. Such data are then used to enrich the content of the message.

The stateful Resequencer is a pattern that ensures a certain order imposed on messages in (asynchronous) communication by persistently storing intermediate message sequences (cf. requirements REQ-2, REQ-4 in Table 1).

Pattern Realizations. A Content Enricher db-net realization is shown in Fig. 10. The message enriching processes starts with consuming a message from the input channel place ch_{in}. The message is represented by two elements: a message identifier *key* and message content *msg*. We use request–reply transitions $\mathtt{T}_1$ and $\mathtt{T}_2$ to direct the net flow towards extracting message-relevant data from an external resource. The extraction is performed by matching the message identifier *key* with the one in the storage. While the stateless enriching part is essentially a colored Petri net, we need db-nets in order to access a stateful resource in ch_3 one needs to specify and perform queries on the external storage (cf. REQ-4(a,b)). In addition to the specific pattern requirements, the message processing semantics of the EIPs describes one message (or token) at a time. Thus we assume that the represented net model always deals with a single message as well.

Fig. 11 shows how the resequencer can be represented in db-nets. We assume that incoming message *msg* comes with the information about sequence *seq* it belongs to and certain order *ord*, and all such data are eventually persisted in the database. The information about stored messages can be accessed through the view place ch_p. For the first incoming message in a sequence (i.e., the guard of $\mathtt{T}_1$ evaluates to true), a corresponding sequence entry with a unique identifier value bound to *sid*[6] will be created in the persistent storage (sequences can be accessed in the view place *ms*) using action **CreateSeq**, whereas for all subsequent messages of the same sequence (i.e., when the guard of $\mathtt{T}_2$ holds), the messages are simply stored in the database via updating action **UpdateSeq**. As soon as the sequence is complete, i.e., all messages of that sequence have arrived, the messages of this sequence are queried from the database in ascending order of their *ord* component (see the view place $ch_{p'}$ and its corresponding query) using transition Fetch. The query result is represented as a list that is forwarded to ch_{out}. Note that, similarly to the aggregator in Section 6, the completion condition can be extended by a custom logic, indicated by a subnet connected to $\mathtt{T}_3$.

4.5. Control flow with transacted resource and time: Circuit breaker

Control flow with transacted resource and time patterns require a transacted resource to route a message to their receivers, while involving time aspects.

Candidate Selection. To demonstrate a family of patterns that are based on a control flow with transacted resources and time, we selected as its representative the Circuit Breaker pattern [3] (cf. requirements REQ-0, REQ-3, REQ-4(a)).

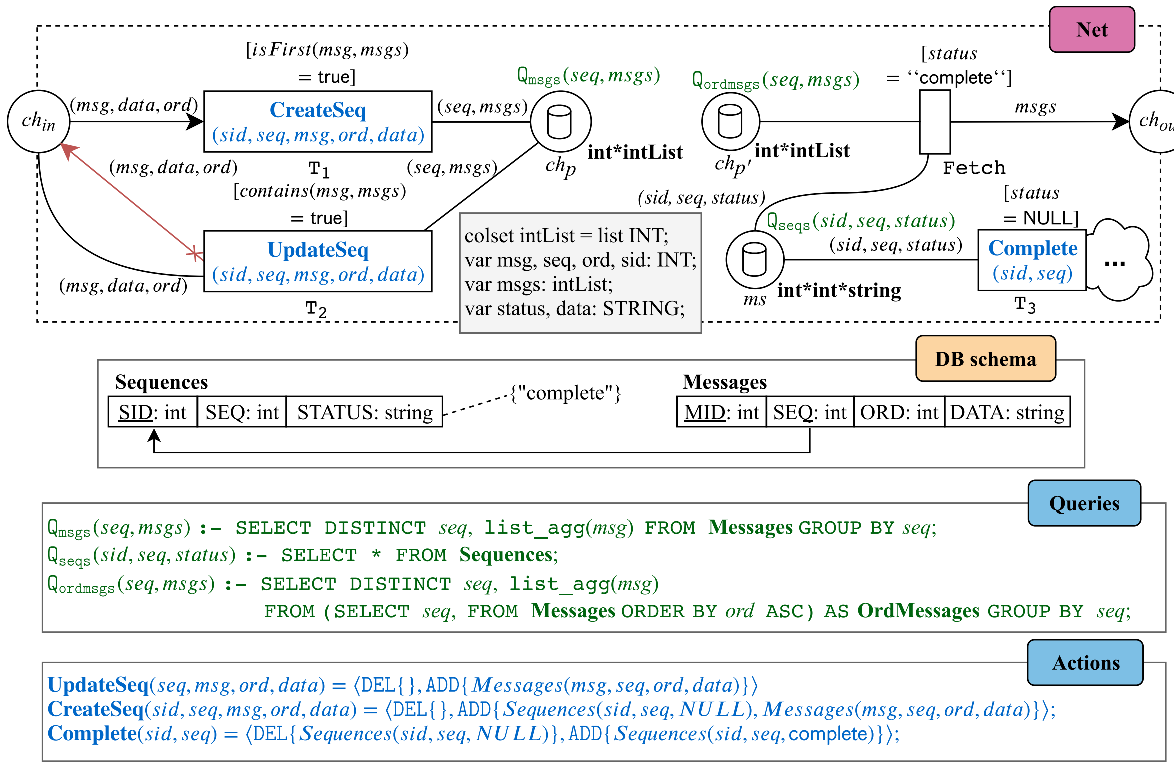
Candidate Description. The Circuit Breaker addresses failing or hung up remote communication, which impacts the control flow of the request–reply pattern [2] by using transacted access to external resources. Thereby the request–reply pattern is just one out of many control flow only pattern examples, for which the circuit breaker can be used.

Pattern Realization. Fig. 12 shows a representation of the request–reply pattern in timed db-nets, extended by a circuit breaker "wrapper" that protects the remote call. At the beginning, every (endpoint-dedicated) circuit[7] in the circuit breaker is closed (that is, its status in table **Circuit** is initially set to closed), thus allowing to pass the input message from input place ch_{in} to the Request–Reply pattern part (represented with a dedicated subnet) by firing Send_Req. If the request–reply pattern executes normally (i.e., transition Receive_Resp has been executed), the resulting message is placed in ch_{out}. Otherwise, in case when an exception has been raised, the information about the failed endpoint is first placed in place ep_{exec}, and then, by firing $\mathtt{T}_1$ and action **UpdCount** assigned to it, can be stored both in a special place *ch exec* and the **Endpoints** table of the persistent storage.[8] Such a table contains all endpoints together with the number of failures that happened at them. If the number of failures reaches a certain limit (e.g., $num > 5$), the circuit trips using transition $\mathtt{T}_2$ and updates its status in the corresponding entry of the Circuit relation to open using action **TripCircuit**. This in turn immediately blocks the communication process that, however, can be resumed (i.e., the circuit is again set to open and the failure count is set to 0) after 40 time units have been passed and transition $\mathtt{T}_3$ can be fired. Note that whenever at least one circuit remains opened, the messages from ch_{in} will be immediately redirected to *ch_exec*.

[6] Note that since *sid* is not bound to variables in the input flow of $\mathtt{T}_i$ ($i \in \{1, 2, 3\}$), it can be treated as a fresh variable [10] that, whenever the transition is executed, gets a unique value of a corresponding type assigned to it.

[7] For simplicity, every endpoint is identified with a unique number EPID.

[8] To be more precise, **UpdCount** only updates the count of exceptions occurred at a given endpoint.



```
Qmsgs(seq, msgs) :- SELECT DISTINCT seq, list_agg(msg) FROM Messages GROUP BY seq;
Qseqs(sid, seq, status) :- SELECT * FROM Sequences;
Qordmsgs(seq, msgs) :- SELECT DISTINCT seq, list_agg(msg)
                          FROM (SELECT seq, FROM Messages ORDER BY ord ASC) AS OrdMessages GROUP BY seq;
```

```
UpdateSeq(seq, msg, ord, data) = ⟨DEL{}, ADD{Messages(msg, seq, ord, data)}⟩
CreateSeq(sid, seq, msg, ord, data) = ⟨DEL{}, ADD{Sequences(sid, seq, NULL), Messages(msg, seq, ord, data)}⟩;
Complete(sid, seq) = ⟨DEL{Sequences(sid, seq, NULL)}, ADD{Sequences(sid, seq, complete)}⟩;
```

Fig. 11. Resequencer realization in timed db-net.

4.6. Control flow with time: Throttler, delayer

Control flow with time patterns are those that route tokens depending on the time.

Candidate Selection. The message expiration (cf. REQ-3(b)) can be modeled implicitly in the timed db-net tokens. The patterns that represent most natural candidates for requirements of delaying the messaging or an operation (cf. requirement REQ-3(c)) and processing only a certain number of messages per time (cf. requirement REQ-3(d)) are the Delayer [3] and Throttler [3] patterns.

Candidate Description. The first pattern is the Throttler. It helps to ensure that a specific receiver does not get overloaded by regulating the number of transferred messages. A slightly different pattern of this category is the Delayer that uses a timer to reduce the frequency of messages sent to the receiving channel.

Pattern Realizations. The representative patterns of this group mostly require control flow and time aspects, and thus can be represented using timed colored Petri nets (e.g., [19]).

Fig. 13 shows the realization of a throttler that emits at most n messages per 60 time units to the receiving place ch_{out}. To ensure that the number of messages taken from input channel place ch_{in} does not exceed the predefined bound, we introduce place *cap* that has n “simple”, black tokens assigned to it with an initial marking. Given that every token’s age in ch_{in} is initially set to zero, one can fire T_1 until the time elapses to the point that the timed guard of T_2 is satisfied. Note that every time T_1 is fired, message token *msg* is getting appended to collection of messages *msgs* using special function *add*.[9]

A slightly different pattern of this category is the Delayer that uses a timer to reduce the frequency of messages sent to the receiving place ch_{out}. As shown in Fig. 14, the sent message from the input channel place ch_{in} is first placed into intermediate place ch_1, and then gets delayed by 60 time units (so as to make the timed guard of T_2 satisfiable).

[9] For example, in CPN Tools one can define the *add* function on lists.

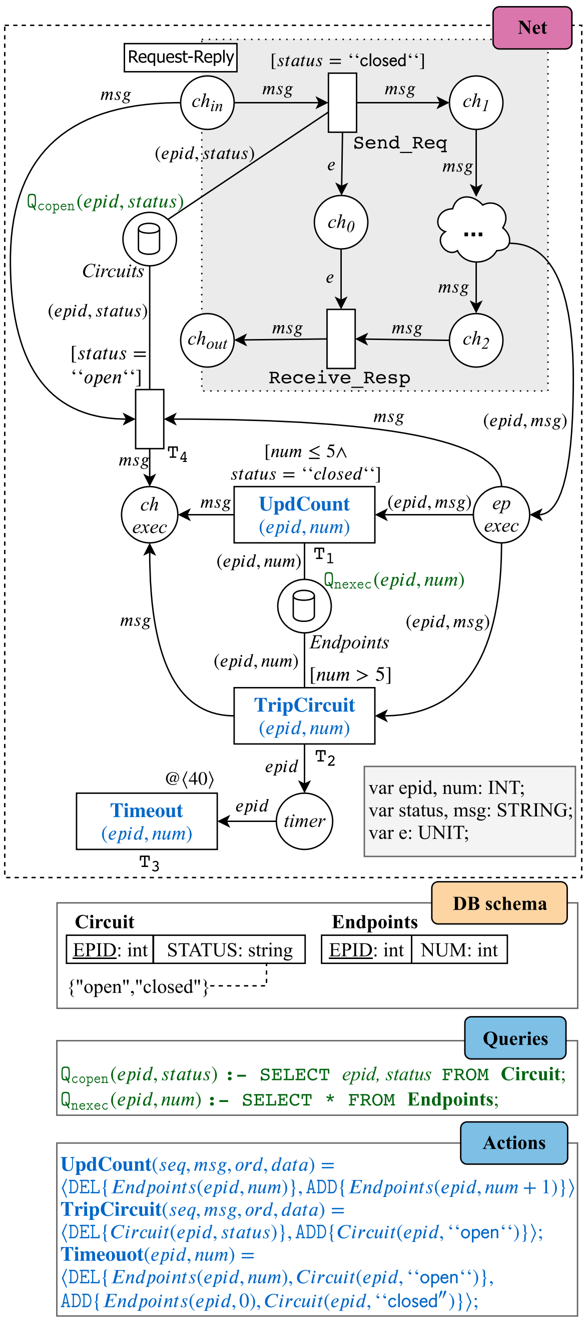
4.7. Data flow with transacted resources time: Aggregator

The combination of data, transacted resources and time aspects in patterns makes them the semantically most complex ones.

Candidate Selection. One of the mostly used patterns in this category (cf. [28]) is our leading pattern example, the Aggregator. To be fully functional it requires access to the data (cf. requirement REQ-2), the configuration of a timeout (cf. requirement REQ-3(a)), CRUD operations on transacted external resources (cf. requirement REQ-4), and compensation (cf. requirement REQ-5).

Candidate Description. In a nutshell the aggregator is a stateful filter that combines several incoming message into one outgoing message. Since the aggregator is our running example throughout this work, more elaborate descriptions can be found in Examples 1, 5, 7 and 8.

Pattern Realization. Fig. 2 specifies the semantics of a commonly used stateful Aggregator [2] pattern. The aggregator persistently collects messages, that can be seen in a dedicated view place ch_p, and aggregates them using the `Aggregate` transition based on a completion condition (i.e., a sequence that the message is related to is complete) or on a sequence completion timeout. For this an incoming message *msg* is correlated to an existing sequence based on a sequence label *seq* attached to it. If the message is first in a sequence (i.e., the guard of T_1 is satisfied), a new sequence is created in the **Sequences** table and the message together with a reference to this sequence is recorded in the **Messages** table in the persistent storage using action **CreateSeq**. If a message correlates to an existing sequence *seq* (i.e., the guard of T_2 is satisfied), which has been aggregated due to a timeout (i.e., T_3 has fired and updated the sequence by assigning to it the expired status in the **Sequences** table), the update fails. This results in the roll-back behavior: the database instance is restored to its previous state, while the net uses the roll-back arc to put the message back to the message channel ch_{in}. This message can be then added as the first one to another newly created sequence



```
Q_copen(epid, status) :- SELECT epid, status FROM Circuit;
Q_nexec(epid, num) :- SELECT * FROM Endpoints;
```

```
UpdCount(seq, msg, ord, data) =
⟨DEL{Endpoints(epid, num)}, ADD{Endpoints(epid, num + 1)}⟩
TripCircuit(seq, msg, ord, data) =
⟨DEL{Circuit(epid, status)}, ADD{Circuit(epid, ''open'')}⟩;
Timeouot(epid, num) =
⟨DEL{Endpoints(epid, num), Circuit(epid, ''open'')},
ADD{Endpoints(epid, 0), Circuit(epid, ''closed'')}⟩;
```

Fig. 12. Circuit breaker realization in timed db-net.

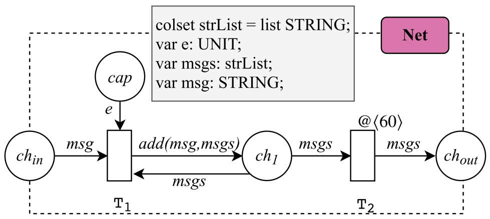


Fig. 13. Throttler realization in timed db-net.

seq. To aggregate the messages one needs to fire `Aggregate` that, in turn, can be fired if the sequence completion criterion has been met and transition T_4 has fired. Note that the completion

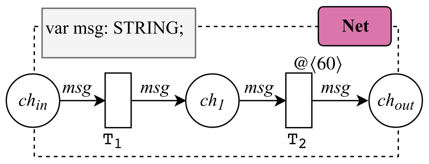


Fig. 14. Delayer realization in timed db-net.

criterion is user-definable and thus we leave out its concrete implementation in this pattern realization.

4.8. Discussion

The db-net foundation implicitly covers REQs-2,4 in form of a relational formalization with database transactions. Together with the realizations of the Content-based Router, Load Balancer (cf. REQ-1(a), (b)) and Aggregator Fig. 2 (cf. REQs-3(a), REQ-4 and REQ-5) we showed realizations for all of the requirements from Section 2. The expiry of tokens, depending on time information within the message, can be represented using CPNs and db-nets by modeling it as part of the token's color set and transition guards (similar to [19]). Nevertheless, to model the transition timeouts (cf. REQ-3(a)) and delays (cf. REQ-3(c)) one needs to resort to more refined functionality realized in timed transitions provided by the timed extension of db-nets. Similarly, the msg/time ratio (cf. REQ-3(d)) can be represented (see Throttler pattern in Fig. 13).

The categorization of patterns according to their characteristics allows for an *instructive* formalization based on candidates of these categories and shows that even complex patterns can be defined in timed db-nets. This, in turn, allowed us not to discuss candidates of all the categories from Fig. 3, since they can be seamlessly derived by the introduced patterns from other categories. For example, *control* and *data* with resource patterns do not require transacted resources, and can thus be realized similar to their transacted resource cases by substituting view places with normal ones. The building blocks for the realization of *transacted resource* as well as *data flow with time* patterns can be derived from, e.g., the Resequencer or Aggregator patterns. Finally, the *data flow with format* patterns can be represented using CPNs, and thus not further discussed here.

Thanks to our model checking results presented in the previous section, the correctness of the realization of each pattern can be formally verified. However, due to the absence of a model checker for (timed) db-nets, the formal analysis (cf. [10]) of such cannot be automatically performed. Nevertheless, as an alternative to the model checking approach, it is possible to perform the correctness testing using the experimental validation via (repeated) simulation of db-net models. We discuss this approach in the following section.

5. Correctness testing

The correctness of an integration pattern realization represented in timed db-nets can be validated by evaluating the execution traces of such models (e.g., similar to the *state-oriented* testing scheme by Zu and He [29]), where at each step, an execution trace contains a B-snapshot representing a current state of the persistence layer together with a control layer marking. According to the timed db-net execution semantics (see Section 3), a consecutive, finite enactment of a pattern model starting from an initial B-snapshot $s_1 = \langle I_1, m_1 \rangle$ produces several B-snapshots $s = \langle I, m \rangle$ that, depending on the number of enactment steps, generates a finite execution trace $s_1 \rightarrow \ldots \rightarrow s_{n+1}$ for some $n \in \mathbb{N}$.

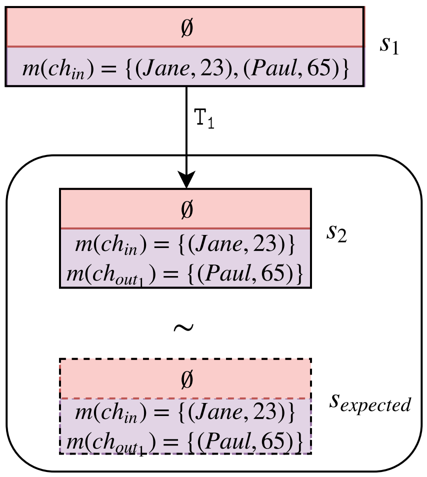

∅
$m(ch_{in}) = \{(Jane, 23), (Paul, 65)\}$
s_1
T_1
∅
$m(ch_{in}) = \{(Jane, 23)\}$
$m(ch_{out_1}) = \{(Paul, 65)\}$
s_2
$\sim$
∅
$m(ch_{in}) = \{(Jane, 23)\}$
$m(ch_{out_1}) = \{(Paul, 65)\}$
$s_{expected}$

Fig. 15. A finite db-net execution of a Content-based Router.

Example 10. Consider a Content-based Router model B from Fig. 9 with an initial B-snapshot s_1 containing markings for messages of two employees with their age (Jane, 23), and (Paul, 65). Since the router has no persistence, the database snapshot is empty. Fig. 15 shows a possible, finite execution of B starting from s_1. In order to reach s_2 from s_1, transition T_1 with routing condition $[age \leq 25]$ has to be fired. The resulting marking only contains the message of Jane that has satisfied the condition, while the message of Paul has not been forwarded. ■

In the router example, given the initial marking {(Jane, 23), (Paul, 65)} analyzed against the guard of T_1, the marking $s_{expected}$ in Fig. 15 denotes the only allowed final (or expected) state that can be reached from s_1 in case of a properly working router. Indeed, it is evident that the generated state s_2 is identical to the expected one and thus one could conjecture the correctness of the presented execution. More generally this is defined in Definition 11, which, together with a comparison operator $\sim$, allow for a configurable, correctness criterion definition over finite traces induced by pattern models.

Definition 11 (*Correctness Criterion*). Let $s_1 \rightarrow \dots \rightarrow s_{n+1}$ be a finite execution trace of some pattern model M and C be a set of reference B-snapshots that define a set of correct, desired states. We say that a pattern execution is *correct*, if for all $c \in C$ it holds that $c \sim s_i$, for $i \in \{1, \dots, n+1\}$. The operator $\sim$ typically denotes equality, but can also correspond to more sophisticated comparison operators for relating reached and desired snapshots. ■

For example, it is easy to see, that in Fig. 15 $s_2 \sim s_{expected}$.

Note that the definition still captures the situation where target snapshots are enumerated explicitly. Other forms of validation (e.g., based on statistical goals formulated over the exhibited behaviors of the system) would require a more fine-grained approach able to aggregate snapshots and traces. This is matter of future work.

Next we discuss the application of this correctness criterion for three different requirement categories: control flow, data flow together with format and (transacted) resource, and timed patterns.

5.1. Control flow patterns

To test control flow patterns for correctness, the operator $\sim$ can be defined so as to compare the number of tokens in

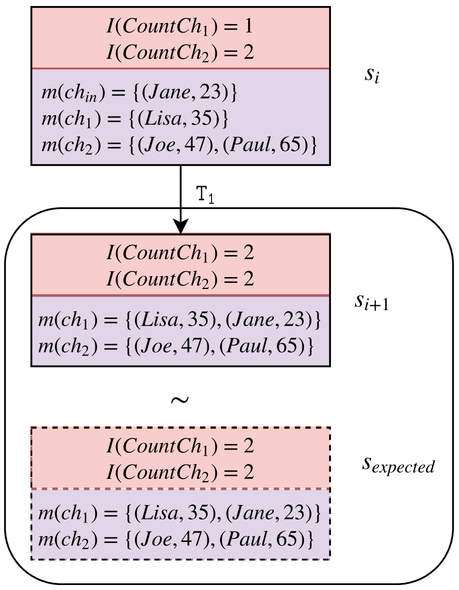

$I(CountCh_1) = 1$
$I(CountCh_2) = 2$
$m(ch_{in}) = \{(Jane, 23)\}$
$m(ch_1) = \{(Lisa, 35)\}$
$m(ch_2) = \{(Joe, 47), (Paul, 65)\}$
s_i
T_1
$I(CountCh_1) = 2$
$I(CountCh_2) = 2$
$m(ch_1) = \{(Lisa, 35), (Jane, 23)\}$
$m(ch_2) = \{(Joe, 47), (Paul, 65)\}$
s_{i+1}
$\sim$
$I(CountCh_1) = 2$
$I(CountCh_2) = 2$
$m(ch_1) = \{(Lisa, 35), (Jane, 23)\}$
$m(ch_2) = \{(Joe, 47), (Paul, 65)\}$
$s_{expected}$

Fig. 16. A finite execution trace of a load balancer in timed db-net. Here we use *Empl* to define an input place with employees and *count* as a function that counts a number of tokens in marking *m*.

the correct, final snapshot. Nevertheless, there are control flow patterns whose correctness testing puts additional requirements on $\sim$. For example, the Load Balancer pattern (cf. REQ-1) denotes a special case, since it requires a sequence of input tokens, which then have to produce data entries in the output instances that fit the probability values and distribution of the balancer (e.g., Kolmogorov–Smirnov test [30]). Therefore, the $\sim$ operator has to check whether the number of tokens in the desirable states follows a probability distribution.

Example 12. Consider a Load Balancer in timed db-net B from Fig. 5 with an initial $B-snapshot$ s_1 containing a marking m with several messages composed of employee names and ages m = {(Jane, 23), (Lisa, 35), (Joe, 47), (Paul, 65)}. In the example shown in Fig. 16, three of the tokens have been already distributed. This means that the current, observable snapshot s_i has a database instance with two tuples $CountCh1(1)$ and $CountCh2(2)$, and a marking with one token in the input state $m(ch_{in})$ = {(Jane, 23)}, one token in the first channel $m(ch_1)$ = {(Lisa, 35)} and two tokens in the second channel $m(ch_2)$ = {(Joe, 47), (Paul, 65)}. Assuming that in the current state φ_1 holds, we can fire transition T_1. This, in turn, generates a new state s_{i+1}. In order to check whether s_{i+1} is an expected state, we run a correctness test that is performed on the number of messages sent to the channels. Such a test allows us to see whether a final, desired message ratio is produced by the model. For example, knowing that the bandwidth of the second channel is considerably greater than the one of the first channel, we may expect that the final ratio of $\frac{ch_1}{ch_2}$ is not greater than 0.7. Our $\sim$ operator can be accordingly adopted to perform such a test. ■

The example shows that, even though the correctness testing of control flow patterns is feasible, there are cases in which such tasks may require extra workload in form of input data, mainly on the configuration of the testing setup.

5.2. Data flow and (transacted) resource patterns

In order to test the correctness of patterns that meet requirements REQ-2 and REQ-4 (cf. Table 1), one needs to consider

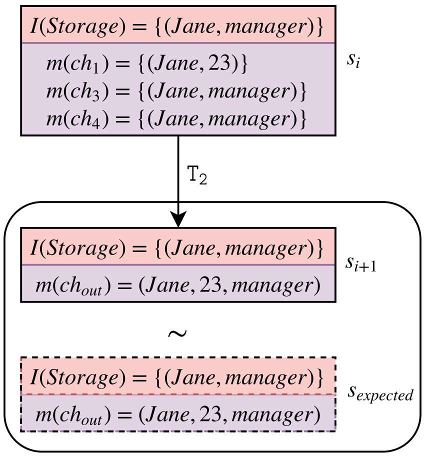

$I(Storage) = \{(Jane, manager)\}$
$m(ch_1) = \{(Jane, 23)\}$
$m(ch_3) = \{(Jane, manager)\}$
$m(ch_4) = \{(Jane, manager)\}$
s_i
T_2
$I(Storage) = \{(Jane, manager)\}$
$m(ch_{out}) = (Jane, 23, manager)$
s_{i+1}
$\sim$
$I(Storage) = \{(Jane, manager)\}$
$m(ch_{out}) = (Jane, 23, manager)$
$s_{expected}$

Fig. 17. A partial execution of a content enricher timed db-net. For simplicity we omit the view place marking in s_{i+1} and $s_{expected}$.

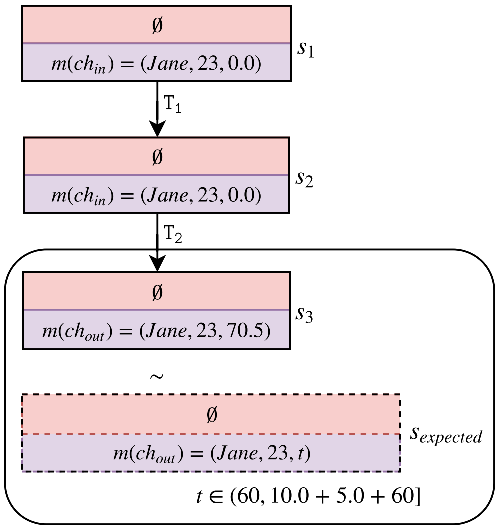

$\emptyset$
$m(ch_{in}) = (Jane, 23, 0.0)$
s_1
T_1
$\emptyset$
$m(ch_{in}) = (Jane, 23, 0.0)$
s_2
T_2
$\emptyset$
$m(ch_{out}) = (Jane, 23, 70.5)$
s_3
$\sim$
$\emptyset$
$m(ch_{out}) = (Jane, 23, t)$
$s_{expected}$
$t \in (60, 10.0 + 5.0 + 60]$

Fig. 18. A partial execution of a delayer timed db-net.

testing not only the marking, as it is done in the case of control flow patterns, but also to compare states of the persistent storage. Specifically, for a given initial snapshot s_1 with an instance I_1, either an expected state s_n with an instance I_n or an expected error state s_j must be produced by the pattern. Otherwise the pattern is considered incorrect.

Example 13. Let us consider a timed db-net B for the Content Enricher in Fig. 10 with an initial B-snapshot s_1 that contains a marking m with a message composed of an employee name and age such that $m(ch_{in}) = (\mathtt{Jane}, 23)$. In Fig. 17 the database of the content enricher model stores information about employees and their positions in a relation called *Empl*. To reach the final state s_{i+1} from s_i, we need to fire T_2. The fact that we have s_i with a marking containing two tokens (one with the employee's name and age, and another one with the same name and position) shows that, a few steps before, the employee token $(\mathtt{Jane}, 23)$ was matched to the corresponding entry in the persistent storage and extra information about her position was extracted. If it was not the case, the execution trace, which is partially represented in Fig. 17, would not contain such two tokens that, in turn, would mean that the transition used for accessing the view-place could not fire since no matches were found. Thus, T_2 does not fire and the final state does not fulfill the correctness criterion for the given initial snapshot. ■

Note that, however, in this example the internal database state does not play the main role when testing the correctness. The correctness checking is done on the markings which are populated from the database based on the matching condition assigned to the transition inspecting the view place.

5.3. Timed patterns

Finally, a timed pattern can be validated by extending database schemas with extra attributes for storing timestamps (as "on-insert timestamps" in actual databases) or by adding such timestamps to tokens, indicating the token creation time. This allows for checking delays, e.g., by comparing the insert timestamps $time(I_1)$, $time(I_n)$ of data to instance I_1 and those of the final instance I_n, or the timestamps in the tokens, respectively. With this, a numeric delay interval $d = (d_1, d_2]$ can be checked, with $d_1 = \tau$ being the delay configured in the pattern and $d_2 = \tau + avg(t_p) + var(t_p)$ being a sum of τ, the average time t_p and the variance the pattern requires for the internal transition firings without the configured delay. Since the delay τ is an interval itself, its upper value is taken for the application of the correctness criterion.

Example 14. Consider a timed db-net B for the Delayer in Fig. 14 with an initial B-snapshot s_1 that has a marking m with a message composed of an employee's name and age $m(ch_{in}) = (\mathtt{Jane}, 23)$. In Fig. 18 transition T_2 fires with a time delay of 60 time units. Since the delayer does not require a database state, the correctness of the timestamps is checked on the markings. In this example, we assume that the average time $avg(t_{delayer})$ is 10.0 and the variance $var(t_{delayer})$ is 5.0 without the delay. This results in a desirable marking $(\mathtt{Jane}, 23, \mathtt{t})$ in $s_{expected}$ with $t \in (60, 75]$ that, in turn, can be checked against the one in s_3 using $\sim$ that, on top of comparing the states by equality, also compares whether the time stamp belongs to a desired time interval. ■

Note that the token ages cannot be used for checking delays, since they are reset, when inserted into a timed db-net place.

5.4. Erroneous patterns

The main sources of errors during the responsible pattern formalization process in Fig. 1 are the conceptual work on defining the formal representation of a pattern, as well as the model to implementation step, in which the formal model is implemented and configured. Subsequently, we briefly describe these types of errors by example.

Pattern Description to Model Errors. The formal representation of a pattern depends on different challenging factors concerning the quality and comprehensiveness of the pattern description as well as the clarity of its variations, and the complexity of the formalism. Consequently, the process of formalizing a pattern can introduce flaws due to the wrong understanding of the complex task at hand.

Example 15 (*Content-based Router*)**.** While the Content-based Router in Fig. 9 represents the pattern correctly, one could go wrong with the ordered execution (cf. REQ-1(a)), e.g., through transition guards at T_1 and T_1'. If these guards were set with overlapping conditions, then several tokens are produced in different output places, i.e., $m(ch_{out_1} = \{(Paul, 65)\})$, $m(ch_{out_2} = \{(Paul, 65)\})$, which does not match the desired state in the example in Fig. 15. ■

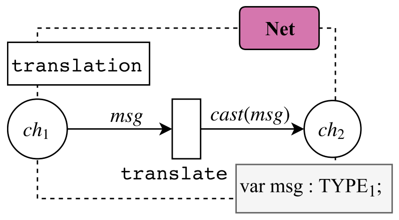


Fig. 19. Sample `translation` subnet realization.

Pattern Model to Implementation Errors. The model to implementation gap specifies the difficulties that can arise during the implementation of a formalized pattern. With the model on one side and the tool-specificities on the other, errors can occur during the translation and configuration. While translation-related errors target particularities of the chosen tool or language, configuration errors can occur in user-defined subnets.

Example 16 (*Message Translator*)**.** The Message Translator in Fig. 6 allows for the configuration of a user-defined subnet that translates an input message *msg* of type TYPE$_1$ into an output message of TYPE$_2$ using a special casting function *cast*. In case of an expected final state $m(ch_{out} = \{(Paul, 65, London)\})$ for an input state $m(ch_{out} = \{(Paul, 65, EC4M)\})$, a configuration of the subnet, as shown in Fig. 19, would not suffice due to the resulting final state $m(ch_{out} = \{(Paul, EC4M)\})$. ■

It is important to note that in this work timed db-nets are used to formalize various enterprise integration patterns and integration scenarios involving them. This, in turn, facilitates the usage of model-driven development of EIP realizations (e.g., Apache Camel [31]) as well as their combinations (most likely while running a Petri net-based engine in the background of some modeling suite that uses more conventional graphical EIP language like the EIP icon notation [2]) since the generated interfaces are having clear syntax and semantics, and thus are less prone to errors caused by users at design time. On top of that, designed application integration infrastructures can be tested against various correctness guarantees using conventional techniques such as validation through simulation. In particular, while testing software systems can be considered more program-based testing (e.g., [32,33]), the validation of Petri nets allows for specification-based testing, when developing concurrent systems like integration solutions. This enables more elaborate testing methods like structural [34], place- or transition-oriented [35–37], or state-oriented testing [29,36], as used in this work. Those techniques go beyond the classical program-based testing used for software systems by providing deeper insights, e.g., into the actual concurrency graphs for structural testing [34] or the states of the transition system representing the software at hand [29,36]. Note also that the validation approach used in this paper is similar to (Big Bang) *Integration Testing* [38] that aims at checking correctness of interfaces between various software components against a (correct) software design. Yet, when modeled with timed db-nets, the whole testing process is facilitated with the graphical, design-time representation of the integration scenario "topology", and thus allows for faster and more precise detection of test cases.

Given that on top of the formalism of timed db-nets one can enforce certain composition rules for designing integration scenarios for EIPs, it would be even possible to reduce the effort for responsible development of integration scenarios in which manual errors are minimized. In other words, by restricting the modeling language and giving it a blocked structure (i.e., integration scenarios can be represented only as certain compositions of EIPs), one can facilitate user design tasks.

6. Evaluation

In this section, we quantitatively evaluate the comprehensiveness of the timed db-net formalism against the real-world integration scenarios (including pattern composition cases), show the correctness of the formal pattern realizations for the requirements discussed in Section 2.2 via the simulation, and qualitatively study the application of timed db-nets to one hybrid integration (i.e., "on-premise to cloud" (OP2C)) and one internet of things integration scenario (i.e., "device to cloud" (D2C)) (cf. Q3).

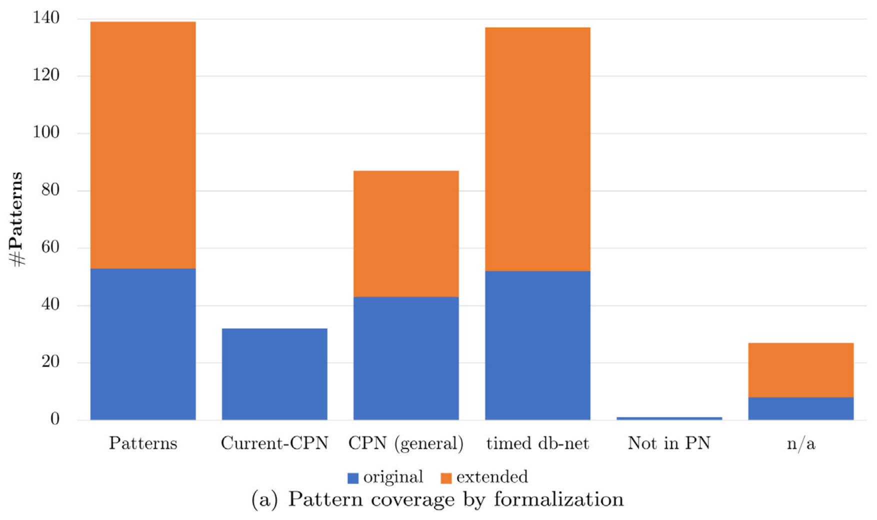
6.1. Comprehensiveness of timed db-nets

The comprehensiveness of timed db-nets is evaluated with respect to coverage of the patterns in the catalogs depicted in Fig. 20(a). Here we compare the applicability of the existing CPN-based formalization [8] (*Current-CPN*), colored Petri nets in general (*CPN (general)*) and timed db-nets (*timed db-net*). While the formalization proposed in [8] covers only some of the EIPs from [2], many more EIPs as well as the recently extended patterns can be represented by colored Petri nets. Now, as we have indicated in the previous sections, one can formalize nearly all of the EIPs using timed db-nets. The only exception is one pattern, namely Dynamic Router, whose requirements cannot be represented using Petri net classes discussed in this work. In fact, in order to represent such a pattern one would need to employ a formalism that, on the one hand, subsumes db-nets and thus covers all the requirements discussed in Table 1 and, on the other hand, supports extra requirements (i.e., dynamically added or removed channels during runtime [8]) that, in turn, extend the expressiveness of the formalism with the ability to generate arbitrary topologies. To allow for such a functionality one may opt for an approach similar to the one in [39], where the authors enrich classical Petri nets with tokens carrying Linear Logic formulas. This, however, would require further investigations.

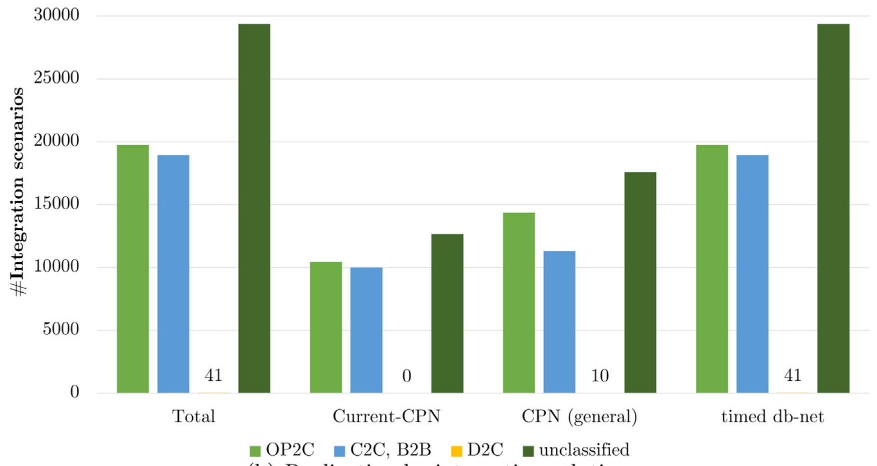
After having analyzed the pattern coverage per formalism, we now consider the relevance of such formalisms against real-world integration scenarios. For this we implemented a Content Monitor pattern [3], which allows for the analysis of the actually deployed integration scenarios that are, for example, running on SAP Cloud Platform Integration (SAP CPI)[10]. Fig. 20(b) shows the coverage of the formalisms grouped by the following integration scenario domains, taken from [3]: On-Premise to Cloud (short OP2C, also known as hybrid integration), Cloud to Cloud or Business Network (native cloud applications C2C, B2B), and Device to Cloud (D2C, including Mobile, IoT and Personal Computing) integration. The results show that the current approach by Fahland and Gierds [8] is only partially sufficient to cover the OP2C, C2C and B2B scenarios. With a more general CPN approach, more than 70% of more conventional OP2C communication patterns can be covered. The more recent and complex cloud, business network and device integration requires timed db-nets to a larger extent, which covers all analyzed scenarios. Note that the Dynamic Router with arbitrary topologies was not practically required for these scenarios, and thus seems to be rather of theoretical relevance.

Conclusions. (1) timed db-nets are sufficient to represent most of the EIPs; (2) EIPs that are generating arbitrary topologies are not covered by considered PN classes; (3) hybrid integration requires less complex semantics and thus is largely in CPN; (4) timed db-nets cover all of the current integration scenarios in SAP CPI.

[10] SAP Cloud Platform Integration, visited 03/2019: https://api.sap.com/shell/integration.



(a) Pattern coverage by formalization



(b) Realization by integration solution

Fig. 20. Timed db-net comprehensiveness.

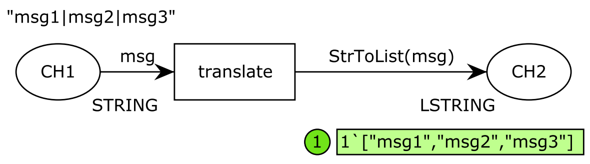


Fig. 21. Message translator in CPN tools.

6.2. *Simulation: Pattern correctness testing*

We prototypically implemented the db-net formalism so as to experimentally test the correctness of the pattern realizations via simulation, following the idea described in Section 5. In order to test the correctness, we simply generate a finite execution trace, starting in an initial B-snapshot s_1 and finishing in s_n, using the prototype and inspect the generated marking together with the database instance. If s_n corresponds to an expected state according to Definition 11, then the test is considered to be successful. Since the inner workings of a pattern can differ between various pattern implementations (e.g., the implementation generates some intermediate states, which are not related to the actual pattern model, but are used, for example, for collecting statistics), the correctness can be also checked at any step of such pattern's finite execution trace.

Prototype. In this work we have chosen CPN Tools v4.0 [11] (CPN Tools, visited 03/2019:) for the modeling and simulation. As compared to other PN tools like Renew v2.5 (Renew, visited 03/2019: http://www.renew.de/), CPN tools supports third-party extensions that can address the persistence and data logic layers of db-nets. Moreover, CPN Tools handles sophisticated simulation tasks over models that use the deployed extensions. To support db-nets, our extension[11] 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.

Simulation. We illustrate the correctness for the majority of the formalized patterns from Section 4 using the simulation in our CPN Tool extension. We focus on the following case studies: Message Translator, Splitter, Content Enricher and Aggregator. In addition, we discuss the case of a flawed example of the Content-based Router pattern from Example 15. Together, these patterns denote the most frequently used patterns in practice according to [28] and cover patterns from five out of seven categories discussed in Section 4 (excluding "control flow only" and "control flow with transacted resources").

Message Translator, Splitter. The realization of a variant of the message translator from Fig. 6 is shown in Fig. 21. Here, as input,

[11] CPN Tools extension for timed db-net and pattern models is available for download (visited 03/2019): https://github.com/dritter-hd/db-net-eip-patterns.

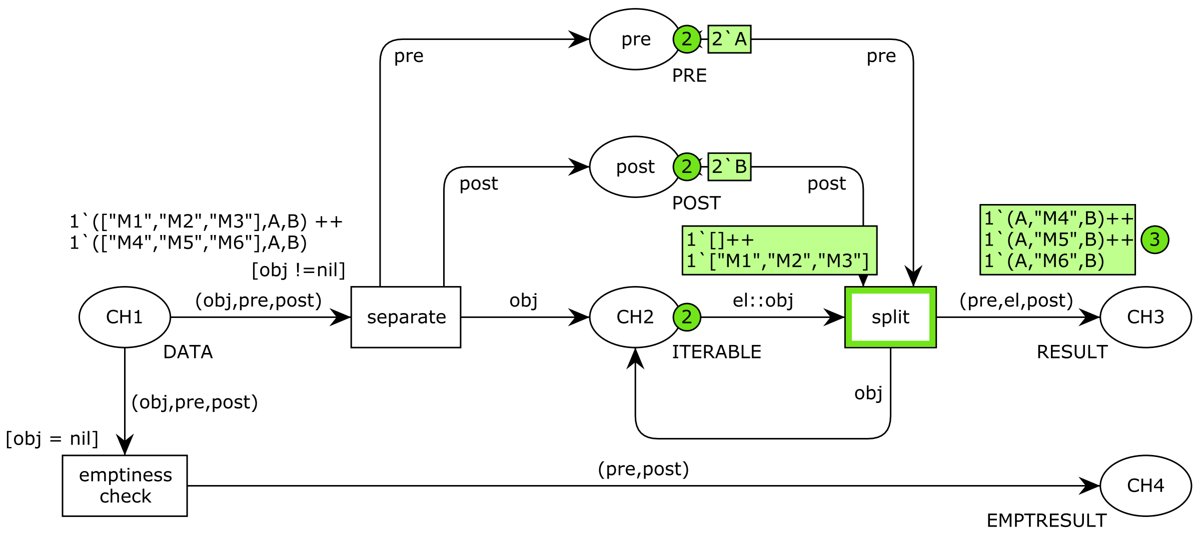


Fig. 22. Splitter in CPN tools.

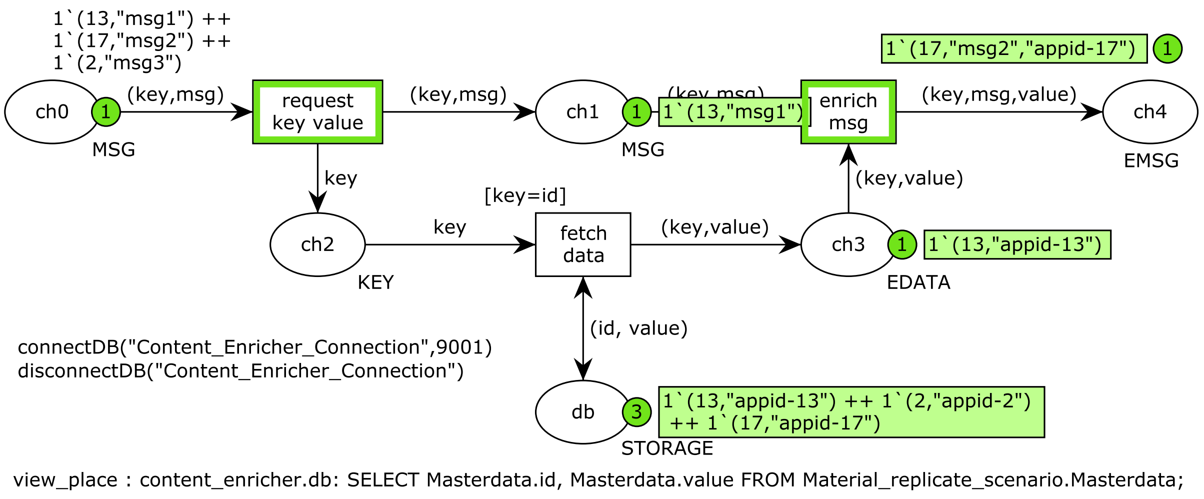


Fig. 23. Content enricher in CPN tools.

the pattern receives a delimiter-separated string and translates it into a list of strings using a special function *StrToList* defined in CPN Tools. The final marking of the net shows the expected state $s_{expected} = \langle \mathcal{I}_{exp}, m_{exp} \rangle$ in which the database instance is empty (thus not shown) and the net is having only $CH2$ marked such that $m_{exp}(CH2) = \{(\text{“msg1”}), (\text{“msg2”}), (\text{“msg3”})\}$.

The Splitter from Fig. 7 is implemented as shown in Fig. 22. In this model, we have two input messages ([“M1”, “M2”, “M3”], A, B) and ([“M4”, “M5”, “M6”], A, B) consisting of iterable objects of size three each as well as pre and post data values A and B. These two messages are then split into six single objects of a shape (A, “Mi”, B), for $i \in \{1, \ldots, 6\}$. The partial execution of the Splitter in Fig. 22 demonstrates the second message to be already split (see the three output messages in place $CH3$), whereas the first message is ready to be split (i.e., the `split` transition is enabled[12]). Note that the current marking of the net can be already intermediately tested against the expected state $s_{expected} = \langle \mathcal{I}_{exp}, m_{exp} \rangle$ in which the database instance is empty and the marking is having only $CH3$ marked such that $m_{exp}(CH3) =$

$$\{(\text{A, “M1”, B}), (\text{A, “M2”, B}), (\text{A, “M3”, B}),$$

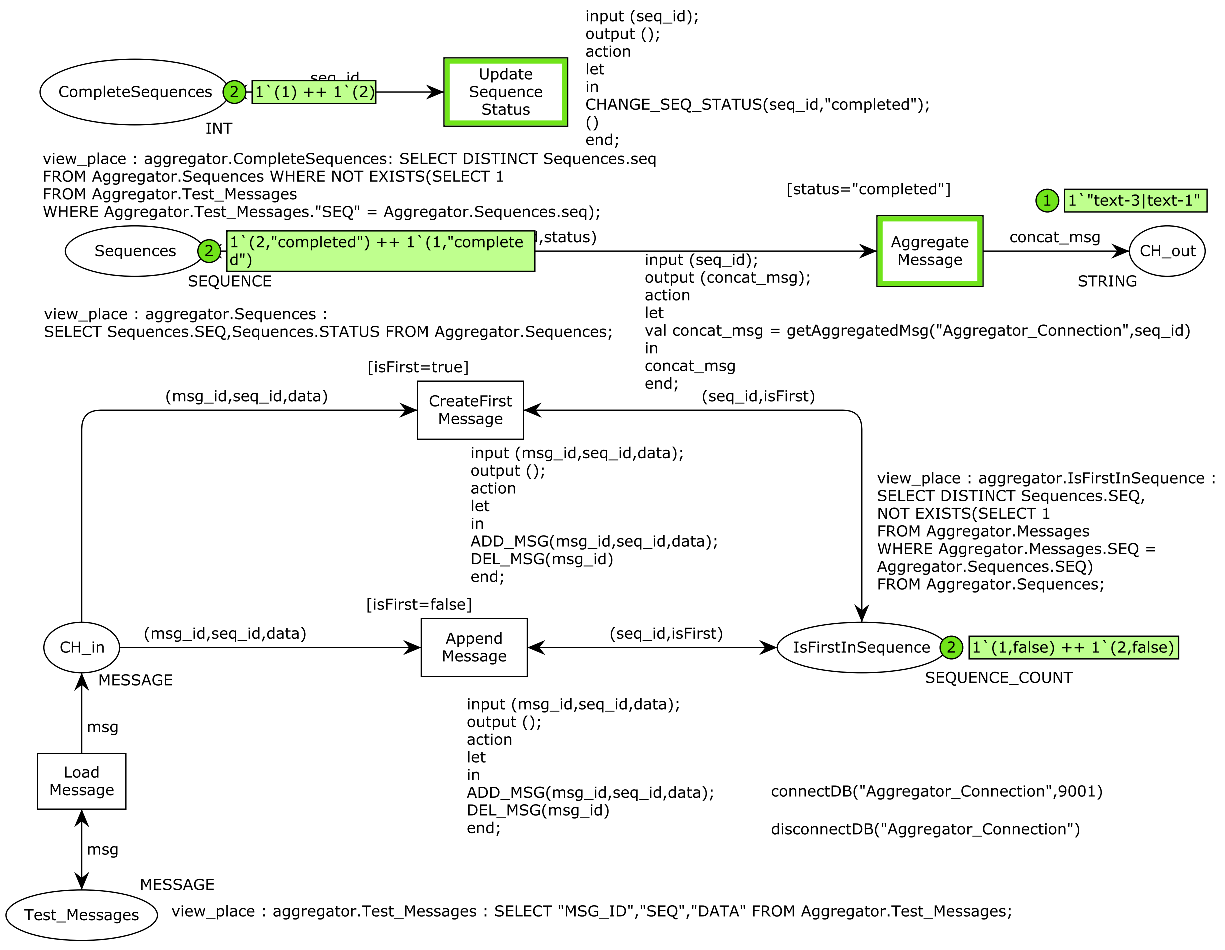
$$(\text{A, “M4”, B}), (\text{A, “M5”, B}), (\text{A, “M6”, B})\}.$$

Indeed, it is easy to see that $m(CH3) \sim m_{exp} \setminus \{(\text{A, “M1”, B}), (\text{A, “M2”, B}), (\text{A, “M3”, B})\}$, indicating that elements of the second message have been correctly processed, by duly adding pre and post data values. The correctness of the splitter implementation, as it is defined in Definition 11, naturally follows.

Content Enricher. The Content Enricher from Fig. 10 can be realized as shown in Fig. 23.

The demonstrated net has three messages (namely, (13, “*msg*1”), (17, “*msg*2”) and (2, “*msg*3”)) in its initial marking and in its current state has already enriched message *msg*2 by adding to a corresponding token an extra data value “appid-17” from the storage (see place *ch*4), that is accessed through the view place called *db*. The data in *db* is stored in a shape of key–value pairs which are then matched with messages by their keys (that is, first components of the pairs). One can see that the net is ready to enrich *msg*1: the `enrich msg` transition is already enabled and the data from the storage that match the key of the token carrying *msg*1 had been fetched from *db* and placed in *ch*3. While the type of data used in different applications may require to reconfigure the query on the storage as well as to use a different enrichment function, the topology of the net representing the enricher remains the same. To test the correctness, we assume

[12] Graphically, enabled transitions are highlighted by a green frame, indicating that they are ready to fire.



```
input (seq_id);
output ();
action
let
in
CHANGE_SEQ_STATUS(seq_id,"completed");
()
end;
```

```
view_place : aggregator.CompleteSequences: SELECT DISTINCT Sequences.seq
FROM Aggregator.Sequences WHERE NOT EXISTS(SELECT 1
FROM Aggregator.Test_Messages
WHERE Aggregator.Test_Messages."SEQ" = Aggregator.Sequences.seq);
```

```
input (seq_id);
output (concat_msg);
action
let
val concat_msg = getAggregatedMsg("Aggregator_Connection",seq_id)
in
concat_msg
end;
```

```
view_place : aggregator.Sequences :
SELECT Sequences.SEQ,Sequences.STATUS FROM Aggregator.Sequences;
```

```
input (msg_id,seq_id,data);
output ();
action
let
in
ADD_MSG(msg_id,seq_id,data);
DEL_MSG(msg_id)
end;
```

```
view_place : aggregator.IsFirstInSequence :
SELECT DISTINCT Sequences.SEQ,
NOT EXISTS(SELECT 1
FROM Aggregator.Messages
WHERE Aggregator.Messages.SEQ =
Aggregator.Sequences.SEQ)
FROM Aggregator.Sequences;
```

```
input (msg_id,seq_id,data);
output ();
action
let
in
ADD_MSG(msg_id,seq_id,data);
DEL_MSG(msg_id)
end;
```

```
connectDB("Aggregator_Connection",9001)

disconnectDB("Aggregator_Connection")
```

```
view_place : aggregator.Test_Messages : SELECT "MSG_ID","SEQ","DATA" FROM Aggregator.Test_Messages;
```

Fig. 24. Aggregator in CPN tools.

an expected state with a partial marking only. Specifically, we are interested in $m_{exp}(ch4) =$

$$\{(13, \text{“msg1”}, \text{“appid-13”}),\\ (17, \text{“msg2”}, \text{“appid-17”}),\\ (2, \text{“msg3”}, \text{“appid-2”})\}.$$

Given that the current net demonstrates the enricher being in its intermediate state and having processed only message one out of three, with its current marking in $ch4$ we have that $m(ch4) \sim m_{exp} \setminus \{(13, \text{“msg1”}, \text{“appid-13”}), (2, \text{“msg3”}, \text{“appid-2”})\}$, and thus can conjecture that the given pattern realization works as expected.

Aggregator. The Aggregator pattern in Fig. 2 can be realized using our CPN Tool extension as it is shown in Fig. 24.

Here we neglect the timed completion condition due to the differing temporal semantics in the tool. For the ease of simulation, we added a table *Test_Messages* containing four test messages $(1, 1, \text{“text-1”})$, $(2, 2, \text{“text-2”})$, $(3, 1, \text{“text-3”})$, $(4, 2, \text{“text-4”})$, with ids from $\{1, \ldots, 4\}$, two sequences $\{1, 2\}$ and a textual payload. The completion condition is configured to aggregate after two messages of the same sequence and the aggregation function concatenates the message payloads separated by '|'. The expected result in the output place *CH_out* for the first sequence is one message with both payloads aggregated $(1, \text{“text-3–text-1”})$.

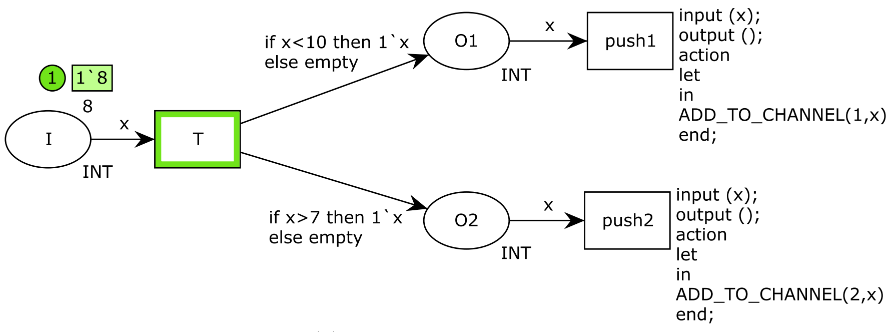
Now, when establishing a connection to the database and to the CPN Tools extension server, the data from the connected database tables are queried and the net is initialized with the data from the database in place *CH_in*. We simulated the aggregator realization in Fig. 24 for the two test sequences, until one sequence was complete. The intermediate marking in $m(CH\_out) \sim m_{exp} \setminus \{(\text{“text-4”}|\text{“text-2”})\}$, for $m_{exp}(CH\_out) =$

$$\{(\text{“text-3”}|\text{“text-1”}), (\text{“text-4”}|\text{“text-2”})\},$$

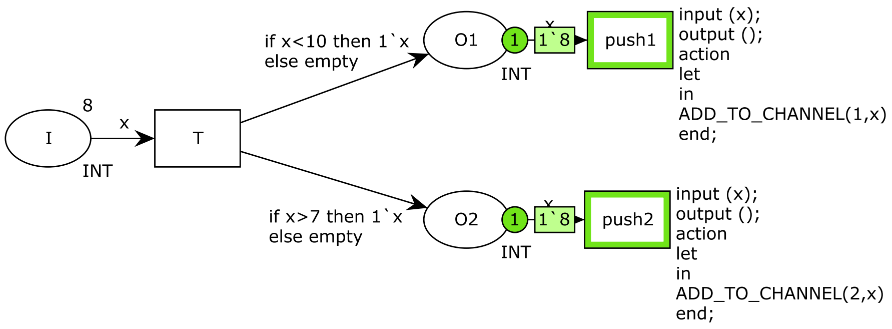
will eventually result into the expected outcome in *CH_out* and the database.

Flawed Content-based Router. While the previously discussed pattern implementations are correct, we added a *flawed* implementation of the Content-based Router, which is not required for the subsequent case studies, so as to demonstrate how the simulation could be used to detect an erroneous design. Content-based Router is a pattern that takes one input message and passes it to exactly one receiver without changing its content. This is done by evaluating a condition per recipient on the content of the message. Fig. 25(a) shows one out of many router implementations, which may look correct, but, however, its process layer violates the correct design.

For the evaluation we use the aforementioned method for "data and (transacted) resource-bound patterns", which is based on the reachability of a correct database state. Such a correct state would be a database instance with one entry in table $Channel_1$ and an empty table $Channel_2$. This should happen due to the fact that the logical expressions on the arcs outgoing from T are expected to be disjoint. Now, let us explore the inner workings of the flawed pattern realization. In Fig. 25(a), transition T reads the token in place I and then conditionally inserts it to the two subsequent places. Since the value of the token matches all conditions, both output places O_1 and O_2 receive a copy of the token



(a) Transition T is enabled



(b) Messages duplicated according to the conditions on the arc inscriptions

Fig. 25. Flawed Content-based Router in CPN tools.

Channel_1

ID : int	data : int
10	8

Channel_2

ID : int	data : int
11	8

Fig. 26. Database instance after flawed Content-based Router PN was executed "successfully".

as it is shown in Fig. 25(b). In terms of application integration, this could mean that two companies receive a payment request or a sales order that was actually meant for only one of them. In the net, the two subsequent transitions $\texttt{push}_1$ and $\texttt{push}_2$ are enabled and fire by executing the database inserts defined in the *ADD_TO_CHANNEL*(i, x) function, where i is being an index of one of the `Channel` tables and x is a data value to be inserted. From the net alone (i.e., in the initial state in Fig. 25(a)), the pattern realization seems to be correct. However, after its execution, we can see that no correct state has been reached. Indeed, after the tokens have been processed on the control layer, the database instance contains two entries (cf. Fig. 26), one in each table, that, in turn, would mean that the logical expressions that are meant to guard two different outputs are not disjoint, and by executing T we populated both *O*1 and *O*2 (instead of generating a token in only one of them).

Note that, when assuming one input token in I and a precedence of `push1` over `push2`, and considering that $I_{exp} = \{Channel_1(8)\}$, the final database instance $I(\mathsf{Channel}_1)$ comes out to be as expected (that is, $I(\mathsf{Channel}_1) \sim I(\mathsf{Channel}_1)_{exp}$), whereas $I(Channel_2) \not\sim I_{exp}(\mathsf{Channel}_2)$. It is easy to see that, knowing the control-flow and data aspects, a given timed db-net allows for detecting flaws in a pattern realizations as well as provide richer information for fixing them.

Conclusions. (5) The CPN Tools extension allows for EIP simulation and correctness testing; (6) model checking implementations beyond correctness testing are desirable.

6.3. Applicability: Case studies

The single patterns can be composed to represent integration scenarios, for which we study the formalism with respect to its applicability to two scenarios from the analysis: one hybrid OP2C and one D2C scenario.

6.3.1. Hybrid integration: Replicate material

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. 27. A *CRM Material* is sent from the CRM system via EDI (more precisely SAP IDOC transport protocol) to an integration process running on SAP Cloud Platform Integration (SAP CPI).[13] 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.

Formalization. For this study, we manually encoded the BPMN scenario into a timed db-net as shown in Fig. 28. Since the obtained timed db-net is mainly a composition of two patterns, namely the Content Enricher in Fig. 10 and the Message Translator in Fig. 6, we just represent an abstract net topology, indicating with places connecting input and output channels of the patterns and leaving nets representing patterns as white boxes. The database schema together with queries from the data logic layer are omitted, since they are identical to those used for the content enricher in Section 4.4. We would like to mention that, while the message translator is close to the currently existing CPN solution

[13] SAP CPI, visited 03/2019: https://api.sap.com/. The pattern compositions in this catalog are represented in a BPMN model (e.g., [4]), and thus we subsequently represent our examples in this way — this is also more expressive in terms of message and data representation than the EIP icon notation [2].

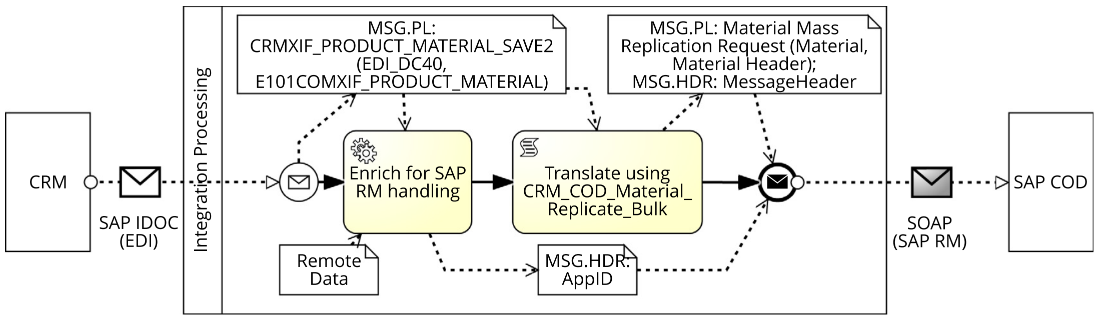

MSG.PL: CRMXIF_PRODUCT_MATERIAL_SAVE2 (EDI_DC40, E101COMXIF_PRODUCT_MATERIAL)
MSG.PL: Material Mass Replication Request (Material, Material Header); MSG.HDR: MessageHeader
Integration Processing
CRM
SAP IDOC (EDI)
Enrich for SAP RM handling
Translate using CRM_COD_Material_Replicate_Bulk
Remote Data
MSG.HDR: AppID
SOAP (SAP RM)
SAP COD

Fig. 27. SAP hybris cloud replicate material from sap business suite (a "hybrid integration" scenario).

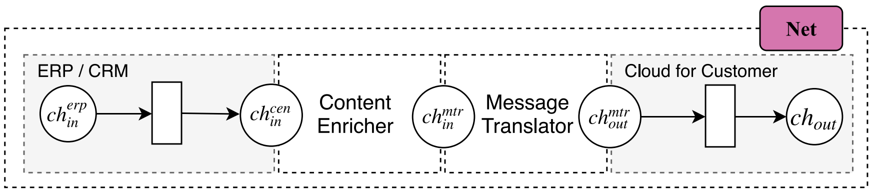


Fig. 28. Replicate material scenario translated into its timed db-net representation (schematic).

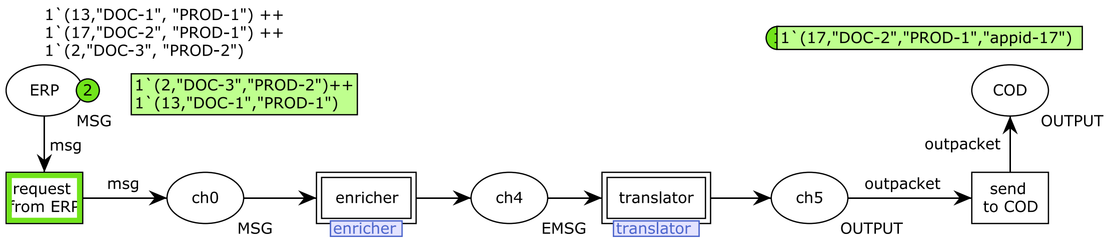


Fig. 29. A pattern composition for the replicate scenario as a hierarchical net in CPN Tools.

in [8], the content enricher (including the need of querying the persistent storage) should be represented using timed db-nets. Consequently, the enricher is a pattern not covered before, for which neither soundness nor correctness could be checked. It is also noteworthy that, given that hybrid integration usually denotes data movement between on-premise and cloud applications, which do not require complex integration logic (see [3]), the timed db-net representation for such rather straightforward hybrid integration scenarios gives richer insight into the data stored in the database as well as their manipulation (as opposed to, for example, BPMN), while the models remain still intuitively understandable.

Simulation. The replicate material scenario in a timed db-net (cf. Fig. 28) is implemented as hierarchical net with our CPN Tools extension in Fig. 29, which references the pattern implementations of the enricher from Fig. 23 and translator from Fig. 21, annotated with `enricher` and `translator`, respectively. In the hierarchical model representing this scenario, the `MSG` message from the ERP system is enriched with master data. The derived enriched message of type `EMSG` is then sent to the translator that maps the intermediate message format to the one understood by the COD system, thus generating a new message of type `OUTPUT`. Note that here the arc inscriptions abstractly account for messages without revealing their concrete structure.

In order to check the correctness of the given scenario, one has to keep in mind that, in general, the composition of the single patterns in timed db-nets requires a careful, manual alignment of the "shared" control places (e.g., $ch0$, $ch4$ and $ch5$) with respect to the exchanged data and the characteristics of the neighboring patterns. Thus it is required to carefully consider various pattern characteristics together with input and output message types to ensure its correctness. Assume that the expected marking in out case is $m_{exp}(COD) =$

$$\{(13, \text{“DOC-1”}, \text{“PROD-1”}, \text{“appid-13”}),\\ (17, \text{“DOC-2”}, \text{“PROD-1”}, \text{“appid-17”}),\\ (2, \text{“DOC-3”}, \text{“PROD-2”}, \text{“appid-2”})\}.$$

Then, given the intermediate marking in COD, we can see that $m(COD) \sim m_{exp} \setminus \{(13, \text{“DOC-1”}, \text{“PROD-1”}, \text{“appid-13”})\}(2, \text{“DOC-3”}, \text{“PROD-2”}, \text{“appid-2”})$ and thus conjecture that the scenario is correct. Note that, while the composition in Fig. 29 denotes a correct implementation of the replicate material scenario, the general question of composition correctness remains open.

Conclusions. (7) timed db-net representations allow for an understandable, sound and comprehensive representation of single patterns and their compositions; (8) the correctness of the compositions requires further considerations.

6.3.2. *Internet of things: Predictive Maintenance and Service (PDMS)*

In the context of digital transformation, an automated maintenance of industrial machinery is imperative and requires the communication between the machines, the machine controller and ERP systems that orchestrate maintenance and service tasks. Integrated maintenance is realized by one of the analyzed D2C scenarios in Section 6.1, which helps to avoid production outages and to track the maintenance progress. Thereby, notifications are usually issued in a PDMS solution as shown in Fig. 30 from SAP CPI, represented in BPMN according to [4].

Although we simplified the scenario, the relevant aspects are preserved. Industrial manufacturing machines, denoted by `Machine`, measure their own states and observe their environment

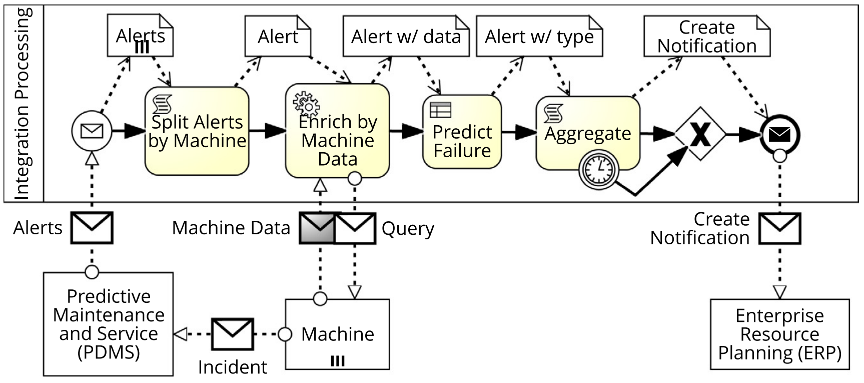


Fig. 30. Predictive maintenance — create notification scenario as modeled by a user (an "internet of things" scenario).

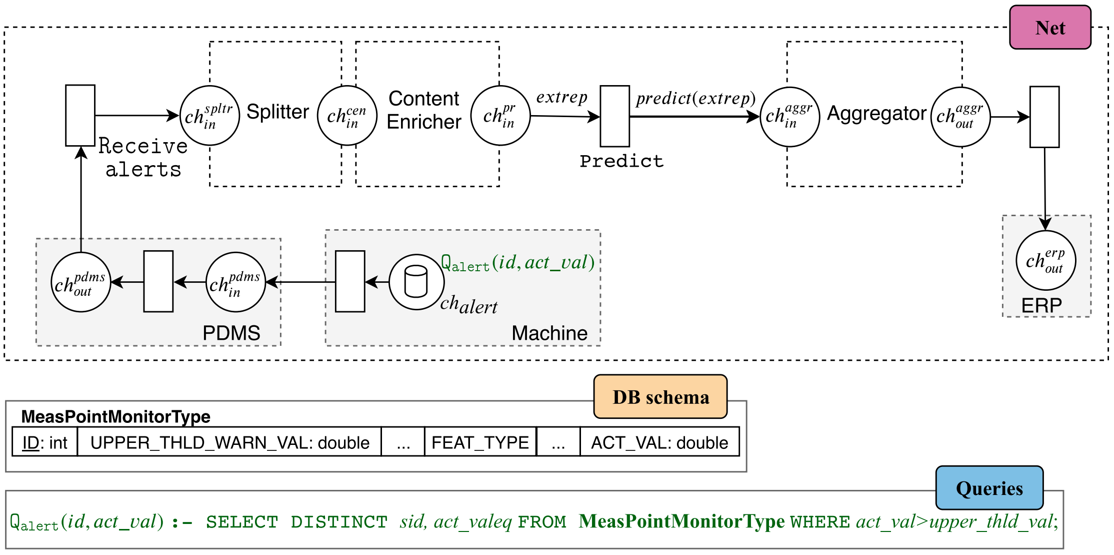
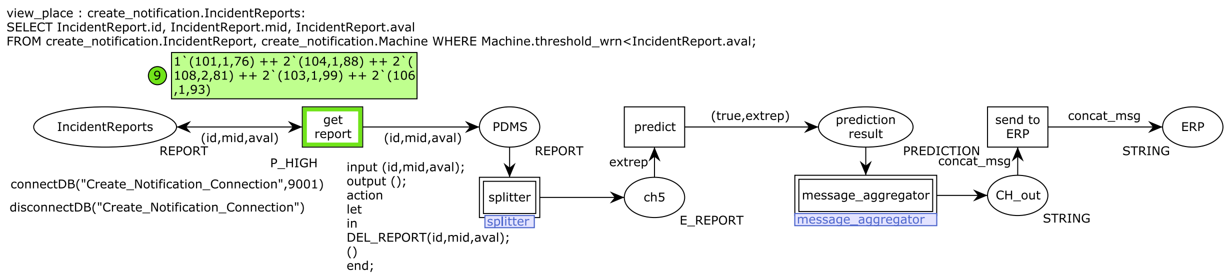


Fig. 31. Create notification scenario translated into its timed db-net representation (schematic).

with sensors in a high frequency. When they detect an unexpected situation (e.g., parameter crosses threshold), they send an incident to a local endpoint (e.g., IoT edge system), the PDMS, indicating that a follow-on action is required. The PDMS system creates alerts for the different machines and forwards them to a mediator, connecting the PDMS to the ERP system. To throttle the possibly high frequent alerts, several incidents are collected (not shown) and sent as list of alerts. Before the ERP notification can be created, additional data from the machines are queried based on the split and single alerts, and then enriched with information that adds the feature type. The information of the single alerts is used to predict the impact by value and machine type, and then gets aggregated to be sent to ERP. In case the notification has been created successfully in ERP, the PDMS gets notified including the service task identifier and thus stops sending the alert (not shown).

Formalization. The BPMN scenario from Fig. 30 has been manually translated into a (schematically represented) timed db-net in Fig. 31. Since the scenario mainly relies on the patterns previously discussed in the paper, we omit their explicit representation and put instead white boxes surrounded with input and output channel places of such patterns so as to indicate the connection points between them. Like that, we hide the Splitter (Fig. 7), Content Enricher (Fig. 10) and Aggregator (Fig. 2) proviso that the queries defined on top of the persistent storage are changed according to the database schema adopted in this scenario. Namely, the content enricher will need to query machine states in order to get data (more precisely, additional information about feature types) for enriching the messages, whereas the aggregator will deal with sequences based on the machine identifiers and as the result will produce messages of concatenated machine names. We reveal the only query, namely Q_{alert}, that is used to create alerts in the PDMS system and populates the view place it is assigned to (ch_{alert}) with pairs containing device identifiers (ID) and corresponding critical values (ACT_VAL). We also add a part representing the predictor pattern. With transition `Predict` we consume messages (*extrep*) produced by the enricher and generate new ones together with the prediction by calling a function *predict*.

Simulation. The predictive maintenance scenario in timed db-nets (cf. Fig. 30) is implemented as hierarchical net with our CPN Tools extension in Fig. 33, which references the pattern implementations of the enricher from Fig. 23 and translator from Fig. 24, annotated with `enricher`, `aggregator`, respectively. In the original scenario, the PDMS sends lists of incidents to the integration system to reduce the number of requests as shown in Fig. 32. The incidents have an incident ID, a machine ID, and the actual critical incident value (e.g., $(101, 1, 76)$). Unfortunately, due to the fact that CPN Tools does not support third party extensions with complex data types like lists, it was decided to make the PDMS component emit single messages using the `get report` transition together with its outgoing place *PDMS* (see Fig. 32). Consequently, the splitter is not required for separating the single incidents, but the incident messages



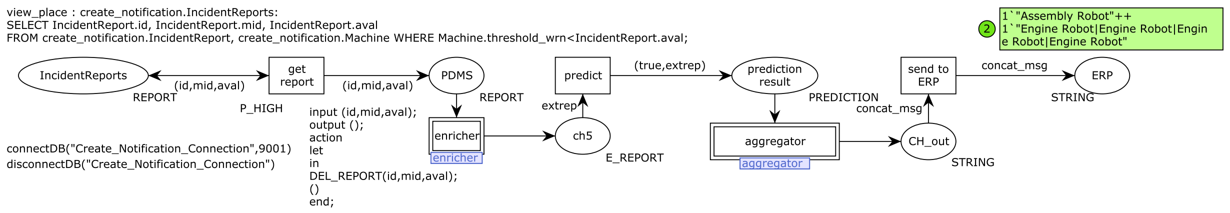
```
view_place : create_notification.IncidentReports:
SELECT IncidentReport.id, IncidentReport.mid, IncidentReport.aval
FROM create_notification.IncidentReport, create_notification.Machine WHERE Machine.threshold_wrn<IncidentReport.aval;
```

```
9  1`(101,1,76) ++ 2`(104,1,88) ++ 2`(
108,2,81) ++ 2`(103,1,99) ++ 2`(106
,1,93)
```

```
connectDB("Create_Notification_Connection",9001)
disconnectDB("Create_Notification_Connection")
```

```
input (id,mid,aval);
output ();
action
let
in
DEL_REPORT(id,mid,aval);
()
end;
```

Fig. 32. Create notification pattern composition as a hierarchical net in CPN tools (in the initial state).



```
view_place : create_notification.IncidentReports:
SELECT IncidentReport.id, IncidentReport.mid, IncidentReport.aval
FROM create_notification.IncidentReport, create_notification.Machine WHERE Machine.threshold_wrn<IncidentReport.aval;
```

```
2  1`"Assembly Robot"++
1`"Engine Robot|Engine Robot|Engin
e Robot|Engine Robot"
```

```
connectDB("Create_Notification_Connection",9001)
disconnectDB("Create_Notification_Connection")
```

```
input (id,mid,aval);
output ();
action
let
in
DEL_REPORT(id,mid,aval);
()
end;
```

Fig. 33. Create notification pattern composition as a hierarchical net in CPN tools (in the final state).

of type REPORT are immediately enriched by the enricher. After master data has been added to the message, a new one of type E_REPORT has been produced. The net then immediately proceeds with predicting the impact using transition predict, which usually assesses the probability of a timely machine error based on previous experiences with the particular machine type. Here, for simplicity, the prediction is always set to true and the results are placed into *prediction result*. Tokens in this place are then used to aggregate several incidents by machine, where, for simplicity, we use machine identifiers to identify aggregator's sequences. The aggregated incident messages are then sent to the ERP system. With the final marking in $m(ERP)$ and $m_{exp}(ERP) = \{$"Assembly Robot", "Engine Robot"|"Engine Robot"| "Engine Robot"$\}$, for the three incidents from machine Engine Robot and one from Assembly Robot, we can see that $m(ERP) \sim m_{exp}$ and thus conjecture that the scenario is correct.

Although the resulting timed db-net provides so far unmatched insights into the different aspects of integration scenarios, the complexity of the composed patterns increased even more, when using hierarchical nets.

Conclusions. (9) timed db-net representations allow for an explicit modeling of all data aspects in complex data-aware scenarios (e.g., roll-back, queries); (10) the formalism's technical complexity might prevent non-technical users from using it on a regular basis.

6.4. Discussion

With the timed db-net formalization, it is possible to model and reason about EAI requirements like data, transacted resources and time (cf. conclusions (1), (5)), going beyond the simple hybrid integration scenarios (cf. conclusion (3)). Thereby the pattern realizations are self-contained, can be composed into complex integration scenarios (e.g., Fig. 31; cf. conclusion (7)) and analyzed (cf. conclusions (4)), while leaving the extension of our tool prototype to model checking as well as a formal treatment of pattern compositions as future work (cf. conclusions (6), (8), respectively). The composition is facilitated through "sharing" control places, preventing unwanted side-effects between patterns.

However, there are some limitations that we briefly discuss next. PN classes considered in this work fall short when it comes to generation of places, transitions or arcs (cf. conclusion (2)). For example, Dynamic Router requires a proper representation of dynamically added or removed channels. Further, the deep insights into data-aware patterns and scenarios lead to the trade-off between sufficient information and model complexity (cf. conclusion (9)). The complexity of PN models compared to their BPMN counterparts in Fig. 30 might not allow for modeling by non-technical users (cf. conclusion (10)). Hence, we propose modeling in a less technical modeling notation, which can be then encoded into PN models, e.g., for verification. Further, while the PN formalism closes the conceptual vs. implementation gap by simulation, we leave a translation of existing EIP implementations to timed db-nets for verification as future work.

In summary, timed db-nets allow to represent patterns not covered before (e.g., the stateful aggregator with a timeout or the content enricher with external resources) and check their soundness and correctness. Note that for more complex scenarios the timed db-net representation might become very complex, e.g., compared to a BPMN representation, and thus might be more suitable as formalism and not as modeling language for the average user (e.g., integration developer).

7. Related work

We discussed related Petri net approaches in Sections 2 and 3. We now briefly situate our work within the context of further Petri net formalisms, and formalizations from EAI and related domains.

7.1. Petri net formalizations

Using PNs, van Hee et al. [40] define an alternative approach for representing and reasoning on database transactions using special token vectors with identifiers and inhibitor nets. While this could also be used similar to db-nets, we build our formalism on db-nets due to their more comprehensive focus on (relational) data, operations, and persistent storage. Furthermore, there is work on ITCPN by van der Aalst [19] and stochastic PNs by Zenie [25] that are either too restricted by time intervals with a single global time in case of ITCPN or hard to practically reason as in case of the stochastic nets. However, both works helped during the specification of the timed db-net models. Stochastic PNs [25] define a priority function, whose execution semantics however does not suffice in representing the required ordering in REQ-1(a).

7.2. Enterprise application integration

We found [3] that the only existing formalization of EIPs is provided in [8] using CPNs. In particular, Fahland et al. [8] define messages as colored tokens and uses PN transition guards as conditions. However, it does not cover all requirements we singled out, and hence we employ db-nets [10] as an extension of CPNs that covers all but one of the EIPs as discussed before. In the business process domain PNs were successfully used to model and reason about workflow nets [41] and some resource- [42] and data-aware [43]) extensions, without however tackling EIP requirements. Although our create notification and predictive maintenance scenarios have been captured in BPMN [4], we do not consider BPMN as a suitable formalism for our objectives (i)–(iii), however, build on a formalization by PNs, which were employed to define the BPMN control-flow semantics [44].

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 by Ibsen et al. [31], and the UML-based Guaraná DSL by Frantz et al. [45]. However, none of these languages aim to be verification-friendly formal integration scenario representations. Conversely, we do 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 formal analysis can be applied to their scenarios.

There are also other works on formal representations of integration patterns, e.g., Mederly et al. [46] represents messages as first-order formulas and patterns as operations that add and delete formulas, and then applies AI planning to find a process with a minimal number of components. While this approach shares the formalization objective, our approach applies to a broader set of objectives (e.g., formal analysis, simulation). Furthermore, the data, transactional database and time semantics are not covered (e.g., cf. REQ-2, REQ-3, REQ-4). Mendes et al. [47] use "high-level" Petri nets as a language for the verification of service-oriented manufacturing systems, that is similar to the approach of Fahland and Gierds [8] for the integration patterns.

7.3. Interaction and architecture patterns

In the related service-oriented architecture domain, service interactions and service interaction patterns were formalized. Works on service interactions largely target formalizations of service orchestration and choreographies (i.e., similar to compositions of patterns). Those, e.g., for web services [48–50] are mainly based on process algebras that account for our time requirements, but, however, lack database transaction semantics (cf. REQ-4). The same holds true for approaches using π-calculus (e.g., by Decker et al. [51]) or coming from the workflow domain (e.g., by Puhlmann et al. [52]).

The approaches to formalize object-oriented, architectural patterns, or component-based systems (e.g., Alencar et al. [53] or Allen et al. [54]) focus on pattern descriptions up to runtime instantiation. Nevertheless, they do not cover, e.g., time, transaction and execution semantics (cf. REQ-3, REQ-4).

7.4. Business process management

An early algorithmic work by Sadiq and Orlowska [55] applied reduction rules to workflow graphs for the visual identification of structural conflicts (e.g., deadlocks) in business processes. From a control flow perspective, we use a similar base representation, which we extend by data, transacted database semantics, and time (e.g., cf. REQ-2, REQ-3, REQ-4). Furthermore, we use graph rewriting for optimization purposes. In Cabanillas et al. [56], 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. Although this provides a view over the required data, it neither covers transactional database or time aspects (e.g., cf. REQ-3, REQ-4) nor proposes formal analysis capabilities for the extended EIPs. All in all, the main focus is rather left on the object life cycle analysis of the process.

In the workflow interaction domain, again workflow nets and workflow modules are used, e.g., by van der Aalst and Weske [57], and Martens [58], respectively. Furthermore, service interaction patterns are formalized using the composition capabilities of Petri nets provided by open nets (e.g., by van der Aalst et al. [59] or as open workflow nets by Massuthe et al. [60]) that will become of interest for formalizing compositions of timed db-nets, which is left as future work.

8. Conclusion

This work aims at providing the formal underpinning for responsible EAI along research questions Q1–Q4. Responsible EAI means to ground EIPs as basic building blocks on a formalization that meets relevant EAI requirements with respect to control flow, data, time, and transactional properties ($\mapsto$ Q1). Q1 could be sufficiently addressed by a thorough analysis of EAI requirements in comparison with existing formalisms and the development of the *timed db-net* formalism that adds the crucial, yet missing time requirement to existing formalism of db-nets [10]. Moreover, the formalism is supposed to be equipped with full execution semantics which was achieved for *timed db-net* in this work ($\mapsto$ Q2). With this the formalization and execution of EIPs becomes possible. In order to bridge the EIP formalization to EIP realizations, an instructive catalog of realizations of EIPs as *timed db-net* models was provided ($\mapsto$ Q3). With the verification results and the possibility of correctness testing based on execution traces, validation of EIPs realizations is enabled ($\mapsto$ Q4). This is complemented with the possibility to simulate EIP realizations through a prototypical implementation ($\mapsto$ Q4). In summary, the research questions and objectives set out in the beginning of the paper could be addressed.

This work focuses on the formalization and realization of EIPs in an isolated manner. EAI solutions, however, often require the composition of EIPs. Such EIP composition necessitate formal treatment as well. Putting the responsible design of EIPs at stake through their informal compositions is counterproductive. Hence, future work will address the formalization and realization of EIP compositions in interplay with the formalization and realization of EIPs as proposed in this work.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This research has been partially supported by the project *Planning for WORkflow Management* (PWORM), funded through the 2017 call issued by the Research Committee of the Free University of Bozen-Bolzano, Italy.

References

[1] D.S. Linthicum, Enterprise Application Integration, Addison-Wesley, 2000.
[2] G. Hohpe, B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2004.
[3] D. Ritter, N. May, S. Rinderle-Ma, Patterns for emerging application integration scenarios: A survey, Inf. Syst. 67 (2017) 36–57.
[4] D. Ritter, J. Sosulski, Exception handling in message-based integration systems and modeling using BPMN, Int. J. Coop. Inf. Syst. 25 (2) (2016).
[5] D. Ritter, S. Rinderle-Ma, Toward a collection of cloud integration patterns, arXiv preprint arXiv:1511.09250, 2015.
[6] D. Ritter, M. Holzleitner, Integration adapter modeling, in: International Conference on Advanced Information Systems Engineering (CAiSE), Springer, 2015, pp. 468–482.
[7] V.G. Cerf, Responsible programming, Commun. ACM 57 (7) (2014) 7–7.
[8] D. Fahland, C. Gierds, Analyzing and completing middleware designs for enterprise integration using coloured petri nets, in: International Conference on Advanced Information Systems Engineering (CAiSE), 2013, pp. 400–416.
[9] D. Ritter, S. Rinderle-Ma, M. Montali, A. Rivkin, A. Sinha, Formalizing application integration patterns, in: 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC), IEEE, 2018, pp. 11–20.
[10] M. Montali, A. Rivkin, Db-nets: On the marriage of colored petri nets and relational databases, T. Petri Nets Other Models Concurr. 12 (2017) 91–118.
[11] K. Jensen, L.M. Kristensen, L. Wells, Coloured petri nets and cpn tools for modelling and validation of concurrent systems, Int. J. Softw. Tools Technol. Transf. 9 (3–4) (2007) 213–254.
[12] K. Peffers, T. Tuunanen, M.A. Rothenberger, S. Chatterjee, A design science research methodology for information systems research, JMIS 24 (3) (2007) 45–77.
[13] S. Lasota, Decidability border for petri nets with data: WQO dichotomy conjecture, in: Application and Theory of Petri Nets and Concurrency - 37th International Conference (ICATPN), Springer, 2016, pp. 20–36.
[14] F. Rosa-Velardo, D. de Frutos-Escrig, Decidability and complexity of petri nets with unordered data, Theoret. Comput. Sci. 412 (34) (2011) 4439–4451.
[15] M. Triebel, J. Sürmeli, Homogeneous equations of algebraic petri nets, arXiv preprint arXiv:1606.05490, 2016.
[16] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, J. Van den Bussche, DFL: A dataflow language based on petri nets and nested relational calculus, Inf. Syst. 33 (3) (2008) 261–284.
[17] E. Badouel, L. Hélouët, C. Morvan, Petri nets with structured data, in: Application and Theory of Petri Nets and Concurrency - 36th International Conference (ICATPN), 2015, pp. 212–233.
[18] D. Ritter, Database processes for application integration, in: British International Conference on Databases (BICOD), Springer, 2017, pp. 49–61.
[19] W.M. van der Aalst, Interval timed coloured petri nets and their analysis, in: Application and Theory of Petri Nets and Concurrency - 14th International Conference (ICATPN), 1993, pp. 453–472.
[20] J. Sifakis, Use of petri nets for performance evaluation, Acta Cybernetica 4 (2) (1980) 185–202.
[21] L. Jacobsen, M. Jacobsen, M.H. Møller, J. Srba, Verification of Timed-Arc Petri Nets, in: SOFSEM 2011: Theory and Practice of Computer Science - 37th Conference on Current Trends in Theory and Practice of Computer Science, Springer, 2011, pp. 46–72.
[22] W. Zuberek, D-timed petri nets and modeling of timeouts and protocols, Trans. Soc. Comput. Simul. 4 (4) (1987) 331–357.
[23] B. Berthomieu, M. Diaz, Modeling and verification of time dependent systems using time petri nets, IEEE Trans. Softw. Eng. 17 (3) (1991) 259–273.
[24] S. Akshay, B. Genest, L. Hélouët, Decidable classes of unbounded petri nets with time and urgency, in: Application and Theory of Petri Nets and Concurrency - 37th International Conference (ICATPN), Springer, 2016, pp. 301–322.
[25] A. Zenie, Colored stochastic petri nets, in: International Workshop on Timed Petri Nets, 1985, pp. 262–271.
[26] G. Balbo, Introduction to stochastic petri nets, in: Lectures on Formal Methods and Performance Analysis: First EEF/Euro Summer School on Trends in Computer Science (FMPA), Springer, 2001, pp. 84–155.
[27] Y. Hu, S. Sundara, J. Srinivasan, Supporting time-constrained sql queries in oracle, in: VLDB, 2007, pp. 1207–1218.
[28] D. Ritter, N. May, K. Sachs, S. Rinderle-Ma, Benchmarking integration pattern implementations, in: Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems, ACM, 2016, pp. 125–136.
[29] H. Zhu, X. He, A methodology of testing high-level petri nets, Inf. Softw. Technol. 44 (8) (2002) 473–489.
[30] A. Kolmogorov, Sulla determinazione empirica di una leggi di distribuzione, Inst. Ital. Attuari Giorn. 4 (1933) 83–91.
[31] C. Ibsen, J. Anstey, Camel in Action, Manning, 2010.
[32] B. Beizer, Software Testing Techniques, Dreamtech Press, 2003.
[33] G.J. Myers, T. Badgett, T.M. Thomas, C. Sandler, The Art of Software Testing, Vol. 2, Wiley Online Library, 2004.
[34] R.N. Taylor, D.L. Levine, C.D. Kelly, Structural testing of concurrent programs, IEEE Trans. Softw. Eng. 18 (3) (1992) 206–215.
[35] S. Morasca, M. Pezze, Using high-level petri nets for testing concurrent and real-time systems, Real-Time Syst.: Theory Appl. 132 (1990) 119–131.
[36] D. Lee, M. Yannakakis, Principles and methods of testing finite state machines-a survey, Proc. IEEE 84 (8) (1996) 1090–1123.
[37] R.M. Hierons, Checking states and transitions of a set of communicating finite state machines, Microprocess. Microsyst. 24 (9) (2001) 443–452.
[38] B. Beizer, Software Testing Techniques, second ed., Van Nostrand Reinhold Co., New York, NY, USA, 1990.
[39] B. Farwer, I.A. Lomazova, A systematic approach towards object-based petri net formalisms, in: Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference (PSI), 2001, pp. 255–267.
[40] K.M. Van Hee, N. Sidorova, et al., Generation of database transactions with petri nets, Fund. Inf. 93 (1–3) (2009) 171–184.
[41] W.M. Van der Aalst, The application of petri nets to workflow management, J. Circuits Syst. Comput. 8 (01) (1998) 21–66.
[42] M. Martos-Salgado, F. Rosa-Velardo, Dynamic soundness in resource-constrained workflow nets, in: Formal Techniques for Distributed Systems, Springer, 2011, pp. 259–273.
[43] R. De Masellis, C. Di Francescomarino, C. Ghidini, M. Montali, S. Tessaris, Add data into business process verification: Bridging the gap between theory and practice, in: Proceedings of the 31st Conference on Artificial Intelligence (AAAI), 2017, pp. 1091–1099.
[44] R.M. Dijkman, M. Dumas, C. Ouyang, Formal semantics and analysis of bpmn process models using petri nets, Tech. rep., Queensland University of Technology, 2007.
[45] R.Z. Frantz, A.M. Reina Quintero, R. Corchuelo, A domain-specific language to design enterprise application integration solutions, Int. J. Coop. Inf. Syst. 20 (02) (2011) 143–176.
[46] 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.
[47] 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, Int. J. Prod. Res. 50 (6) (2012) 1650–1665.
[48] A. Brogi, C. Canal, E. Pimentel, A. Vallecillo, Formalizing web service choreographies, Electron. Notes Theor. Comput. Sci. 105 (2004) 73–94.
[49] R. Gorrieri, C. Guidi, R. Lucchi, Reasoning about interaction patterns in choreography, in: Formal Techniques for Computer Systems and Business Processes, Springer, 2005, pp. 333–348.
[50] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, G. Zavattaro, Choreography and orchestration: A synergic approach for system design, in: 3rd International Conference on Service-Oriented Computing (ICSOC), Springer, 2005, pp. 228–240.
[51] G. Decker, F. Puhlmann, M. Weske, Formalizing service interactions, in: 4th International Conference on Business Process Management (BPM), Springer, 2006, pp. 414–419.
[52] F. Puhlmann, M. Weske, Using the π-calculus for formalizing workflow patterns, in: 2nd International Conference on Business Process Management (BPM), Springer, 2005, pp. 153–168.
[53] P.S. Alencar, D.D. Cowan, C.J.P.d. Lucena, A formal approach to architectural design patterns, in: 3rd International Symposium of Formal Methods Europe (FME), Springer, 1996, pp. 576–594.
[54] R. Allen, D. Garlan, A formal basis for architectural connection, ACM Trans. Softw. Eng. Methodol. (TOSEM) 6 (3) (1997) 213–249.
[55] W. Sadiq, M.E. Orlowska, Analyzing process models using graph reduction techniques, Inf. Syst. 25 (2) (2000) 117–134.
[56] C. Cabanillas, M. Resinas, A. Ruiz-Cortés, A. Awad, Automatic generation of a data-centered view of business processes, in: 23rd International Conference on Advanced Information Systems Engineering (CAiSE), Springer, 2011, pp. 352–366.
[57] W.M. van der Aalst, M. Weske, The p2p approach to interorganizational workflows, in: 13th International Conference on Advanced Information Systems Engineering (CAiSE), Springer, 2001, pp. 140–156.
[58] A. Martens, Analyzing web service based business processes, in: 8th International Conference on Fundamental Approaches To Software Engineering (FASE), Springer, 2005, pp. 19–33.
[59] W.M. van der Aalst, A.J. Mooij, C. Stahl, K. Wolf, Service interaction: Patterns, formalization, and analysis, in: International School on Formal Methods for the Design of Computer, Communication and Software Systems, Springer, 2009, pp. 42–88.
[60] P. Massuthe, W. Reisig, K. Schmidt, An Operating Guideline Approach to the SOA, Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät II, Institut für Informatik, 2005.