Catalog of Optimization Strategies and Realizations for Composed Integration Patterns

Daniel Ritter\textsuperscript{1,3}, Fredrik Nordvall Forsberg\textsuperscript{2}, Stefanie Rinderle-Ma\textsuperscript{3}, Norman May\textsuperscript{1}

\textsuperscript{1} SAP SE, Walldorf, Germany, \{firstname.lastname\}@sap.com
\textsuperscript{2} University of Strathclyde, Glasgow, UK, fredrik.nordvall-forsberg@strath.ac.uk
\textsuperscript{3} University of Vienna, Vienna, Austria, stefanie.rinderle-ma@univie.ac.at

Abstract. The discipline of Enterprise Application Integration (EAI) is the centrepiece of current on-premise, cloud and device integration scenarios. However, the building blocks of integration scenarios, i.e., essentially a composition of Enterprise Integration Patterns (EIPs), are only informally described, and thus their composition takes place in an informal, ad-hoc manner. This leads to several issues including a currently missing optimization of application integration scenarios.

In this work, we collect and briefly explain the usage of process optimizations from the literature for integration scenario processes as catalog.

1 Introduction

Although Enterprise Application Integration (EAI) \cite{21} plays an important role in current IT infrastructures, the development of integration scenarios -- essentially a composition of the Enterprise Integration Patterns (EIPs) \cite{15} and extensions \cite{31} -- remains a challenging task for the users. During the modeling of these scenarios, which is currently done with vendor-specific languages and tools, the users have hardly any support or guidance, when it comes to consistency and correction checks of their models, not to mention optimization possibilities \cite{31}.

For instance, Fig. 1 shows one possible implementation of the Italy Invoicing Scenario (country-specific), which allows organizations to communicate with the Italian authorities -- in Business Process Model and Notation (BPMN) with BPMN Message and BPMN Data Object representing the message data flow \cite{27,30,34}. An organization sends the invoice data, e.g., from its ERP system, to the integration process, in which data from the message body $b.x$ to set the required tax data header of the organization required by the governmental authorities using a Content Enricher \cite{15}. Then the invoice data is mapped to the required target format (Message Transformation \cite{15}) by a mapping program that requires the message in a special encoding (Message Encoder \cite{31}). These operations are parallelized by a Multicast \cite{31} and structurally joined by a Join Router \cite{31}. The unnecessary fields of the mapped invoice are removed from the message by a Content Filter \cite{15} resulting to a message content $b.z'$. The message
is routed to one of two receiving governmental authorities, depending on the type of inquiry, by a Content-based Router [15] or discarded (incl. content enricher). Both governmental authorities require signed messages to verify the authenticity and identity of the sending organization using a Message Signer [31]. Duplicate invoices have to be removed using a (stateful) Idempotent Receiver pattern [15]. The invoice messages are sent to the authority by an External Call [31] pattern and the response is combined with the original message using a content enricher. The combined data is then mapped to the response format of the ERP sender using a message transformation.

While this example is syntactically and semantically correct, it is not optimal with respect to control and data flow. Even with an explicit data flow, it is difficult for the user, e.g., to recognize the redundant control flows and sibling patterns as well as the potential for pattern parallelization or the benefits of pattern re-ordering for the message throughput. Furthermore, the modeling complexity and processing latency reduction potential of the patterns remain hidden to the user (cf. compare with Fig. 16 in Sect. 5).

In this work, we discuss EAI optimization objectives in Sect. 2 and collect relevant optimization techniques from the literature of related domains of business process, workflow and data integration in an optimization catalog and specify them for EAI processes in Sect. 3. We discuss the realization of optimization strategies in Sect. 4. Then we show the applicability to an extended scenario Sect. 5 and conclude in Sect. 6.
2 Optimization Objectives and Strategies

In this section we collect and discuss EAI optimization objectives in the context of classical EAI \[21,15\] and emerging application integration scenarios \[31\]. The latter results to new EAI challenges and solutions, which are represented in this work by our studies on “data-aware” message processing solution spaces: dealing with high velocity and increasing message volume through table-centric processing \[28\] and streaming on dataflow (hardware) architectures \[29\], as well as new message format variety aspects in terms of multimedia integration \[33\]. Figure 2 a high-level view on the classical system architecture (based on \[30\], evolved by new components for multimedia integration (from \[33\]). Subsequently, we define and discuss optimization objectives along the different architectural components.

**Message Throughput and Latency** The message throughput and the request-response latency are the most important optimization objectives of EAI systems. While the message throughput is the number of messages sent and processed from a sender to a receiver (through the whole system), the latency is the time a sender has to wait until it receives a response from the receiver. We consider message throughput and latency on a message processing pattern or a composition of patterns (e.g., including Adapter Runtime, Integration Process Engine). The work on vectorized integration patterns \[28\] illustrates the trade-off of immense message throughput gains, when processing sets of messages in contrast to a reduced overall latency (throughput $\rightarrow$ Vectorization). Furthermore the message throughput can be increased, through processing messages in multiple parallel sub-processes, e.g., separate hardware resources \[29\] ($\rightarrow$ Parallelization).
message stream \cite{29} and multimedia integration \cite{33} showed decreasing message throughput for increasing message sizes. While message indexing reached its limits for increasing multimedia data \cite{33}, keeping message sizes smaller helped throughout the experiments (\rightarrow \textbf{Data Reduction}). This could be achieved through process restructurings. When the structural process changes reduce the number of process elements or complex fork or join patterns, the latency and process complexity could be improved \cite{29} (\rightarrow \textbf{Process Simplification}).

**Energy Efficiency and Costs** While the objectives of energy efficiency and costs are out of scope for this work, some of the named solutions can help to improve both. The study on application integration on a dataflow hardware architecture \cite{29} showed the trade-off between message throughput and energy consumption as well as costs. Even in modern data centers with virtualized hardware, the resource and energy costs play a crucial role \cite{29}. While the FPGA hardware reached a high throughput with low energy consumption and costs, the software solutions required more energy on one machine and could only reach the same throughput by adding more computing units, and thus increased the costs (\rightarrow \textbf{Data flow architecture}). On the common von Neumann architectures a reduction of message processors can help to improve the energy consumption – in CPU-bound messaging scenarios – which also includes the reduction of parallelized processing units (\rightarrow \textbf{Process Simplification}).

**Resilience and Robustness** Since integration systems are central infrastructure components in current IT infrastructures, they are realized as distributed, high-available systems. However, as shown in \textbf{Fig. 2} the dependencies to internal resources (e.g., database, message queuing, machine learning) and the growing number of frequently changing message endpoints (e.g., mobile and cloud endpoints) \cite{31} has to be dealt with during the message processing. Not only because network communication is a little less reliable than process-to-process communication \cite{15}, smart network usage and reaction to exceptional situations or unavailabilities are crucial (\rightarrow \textbf{Reduce Interaction}).

**Context-Awareness and Separation of Duties** While smarter network usage and resilience improves the stability of integration scenarios, even more distributed processing in context-aware scenarios close to the data producers allows for a short circuit processing and potentially a reduction of message sizes. Furthermore – especially in foreign-hosted, cloud computing environments – the aspect of secure and private execution of integration processes with user defined functions is crucial \cite{31}. For instance, a process of one cooperation must not be able to interfere with one process of another cooperation. This requires an integration process fragmentation and placement on heterogeneous runtime systems – with a focus on reduced interactions (\rightarrow \textbf{Pattern Placement}).

**Modeling Complexity** As seen in \textbf{Fig. 1} even moderately simple integration scenarios can become complex. One objective could be to build smarter
editors that support the users, during modeling, configuration, navigating and understanding an integration scenario. The understanding and navigating could be approached by simpler integration scenarios (→ **Process Simplification**) as well as guided modeling (→ **Smart Guidance**). In addition, the modeling context, e.g., “query by sketch” support on visual queries for multimedia data integration [33], could improve the interaction on the integration process (→ **Contextualized Modeling**).

**Objectives and Strategies Summary** Let us summarize the optimization objectives and strategies relevant for EAI in the context of emerging applications. The objectives are:

- Optimization on static design time and dynamic runtime / workload data
- Process / Control flow, Data flow and Interaction optimization (not pattern processing efficiency)
- Optimization of Message throughput, latency as well as resilience and Separation of Duties (minor focus on energy efficiency and costs)
- (Multi-objective optimization)

We do not focus on the objectives of pattern solution efficiency, energy efficiency and costs. Some parts of the reduce modeling complexity objective will be indirectly addressed, due to our focus on integration process related optimizations. The resulting solution optimization strategies (OSx) that we want to consider are:

- **OS1**: Process simplification
- **OS2**: Data reduction
- **OS3**: Parallelization
- **OS4**: Pattern placement
- **OS5**: Reduce interaction

We do not consider Vectorization, the architecture shift to Data flow architectures, since they partially target the pattern solution efficiency and we already addressed them in the context of EAI in our recent work [28,29]. Furthermore, we consider smart guidance and contextualized modeling out of scope, due to our focus on integration process optimizations.

Subsequently, we collect optimizations from related domains, classify them according to our objectives and check, whether they help to realize the optimization strategies.

### 3 Optimization Catalog

In this section we collect relevant optimization techniques for EAI processes from related domains. Therefore we briefly discuss the selected domains and literature and then list the derived optimization techniques again with brief descriptions. Furthermore, we discuss the significance of each technique in the context of the optimization objectives from Sect. 2.
3.1 Domain and Literature Selection

For EAI, the domains of business processes, workflow management and data integration are of particular interest. Therefore, we conducted a study based on [17] using scholar.google.com with the following keywords: business process optimization, workflow optimization, data integration optimization (allintitle, no patents, accessible, from 2004 as the EIPs [15]).

Table 1: Optimizations in related domains - horizontal search

| Keyword                | hits selected | Selection criteria                                                                 | Selected Systems                                                                 |
|------------------------|---------------|------------------------------------------------------------------------------------|----------------------------------------------------------------------------------|
| Business Process       | 159           | 3 data-aware processes survey [39], optimization patterns [22,23]                  |                                                                                  |
| Workflow Optimization  | 396           | 6 data-aware processes instance scheduling [14,15], scheduling and partitioning     | for interaction [2], scheduling and placement [3], operator merge [13]            |
| Data Integration       | 61            | 2 data-aware processes instance scheduling, parallelization, (no schema-matching)   |                                                                                  |
| Added                  | n/a           | 9 expert knowledge                                                                  |                                                                                  |
| Removed                | -             | 1                                                                                    |                                                                                  |
| Overall                | 616           | 19                                                                                  |                                                                                  |

3.2 Optimization Approach Summaries

We summarize and classify the found optimization techniques along the objectives of control and data flow as well as design time (static) and runtime or workload (dynamic) optimizations (similar to [8]). Furthermore, we rate these techniques according to their relevance for the introduced system implementations in the context of emerging application integration.

Design Time Optimizations The design time optimizations denote process improvement techniques that can be identified only on “static” process models. Subsequently, we summarize the found static optimization techniques listed in Tab. 2 and discuss them in the context of the objectives and strategies.

The static control (SCx) optimization techniques mostly support the optimization strategy of process simplification (cf. OS1). Thereby Redundant Path
### Table 2: Static Control (SCx) and Data Flow (SDx) Optimizations

| ID   | Optimization                                      | OS1 | OS2 | OS3 | OS4 | OS5 | other | References                                                                 |
|------|---------------------------------------------------|-----|-----|-----|-----|-----|-------|----------------------------------------------------------------------------|
| SC1  | Redundant Path Removal                            | ✓   | -   | -   | -   | -   | ✓     | - Remove redundant control flow [8]                                        |
| SC2  | Dead Path Removal                                 | ✓   | -   | -   | -   | -   | ✓     | - Unreachable Sub-graph Elimination [8]                                    |
| SC3  | Sub-process Inlining                              | ✓   | -   | -   | -   | -   | ✓     | - Local Sub-process Inlining [8]                                           |
| SC4  | Sub-process Generation                            | ✓   | -   | -   | ✓   | -   | ✓     | (√) (Work-) flow partitioning [2,4,6]                                     |
| SD1  | Resource Squander                                 | -   | -   | -   | -   | -   | ✓     | - Double Variable Assignment [8]                                           |
| SD2  | Unnecessary Resource Allocation                   | -   | -   | -   | -   | -   | ✓     | - Unnecessary Variable Declaration / Assignment [8]; Eliminate-Unused-Variable, SQL simplifications (e.g., Eliminate-Redundant-Attributes, Eliminate-Unused-Attributes, Eliminate-Redundant-Predicates) [10] |
| SD3  | Combine Sibling Patterns                          | ✓   | -   | -   | -   | -   | ✓     | - Two Sibling Translation Operation / Validation merging [8]; Operator Merge [11] |
| SD4  | Unnecessary conditional fork                      | ✓   | -   | -   | ✓   | -   | ✓     | - Unnecessary Switch-Path [8]; similar to Eliminate-Unused-Partner [16]   |
| SD5  | Algebraic                                         | -   | -   | -   | -   | -   | ✓     | - Algebraic Optimization / Simplification [8,12,46]                       |

Covers ✓, partially covers (√), does not cover -; relevant optimization techniques “gray”.
Removal and Dead Path Removal techniques identify and remove identical, redundant and non-reachable branches in an integration process. The Sub-process Inlining technique helps to simplify modular processes and reduce the number of sub-process calls, however, reduces modularization and fragmentation. In contrast Sub-process Generation allows for the partitioning of a process, which could simplify the main process, helps to place patterns (cf. OS4), while reducing the need for communications (cf. OS5).

The techniques from the static data (SDx) optimizations relevant for integration processes Combine Sibling Patterns and Unnecessary conditional fork again mostly support the optimization strategy of process simplification (cf. OS1). Thereby (redundant) sibling patterns and unnecessary conditional fork require additional data to decide that the processed elements are the same, the conditional fork statistics indicate no usage, respectively. The latter can occur in the variant Eliminate-Unused-Partner, which even reduces the interaction with message endpoints (cf. OS5). The found techniques in the areas of Resource Squander and Unnecessary Resource Allocation target the resource consumption on a “source code” and not process level. Hence, they will not be further considered in the context of this work. Similarly, the Algebraic optimizations and simplifications focus on the integration process configuration or the communication between the integration process pipeline and other integration system resources (e.g., datastore).

Runtime Optimizations The runtime optimizations denote process improvement techniques that require (continuous) runtime / workload data. However, in some cases, abstract costs or experimentally collected runtime data is sufficient, e.g., latency, message throughput. Subsequently, we summarize the found static optimization techniques listed in Tab. 3 and discuss them in the context of the objectives and strategies.

The dynamic control (DCx) optimization techniques mostly support process parallelization (cf. OS3) and data reduction (cf. OS2). The most important one is Sequence to parallel / merge parallel processes, which can be considered as technique that elastically scales and reduces parallel patterns or sub-processes. However, the Distribute Message technique can help to reduce interaction with a specific endpoint. The Message Indexing can help to reduce the data, however, rather targets more efficient access data access and the Parallel Process Re-Scheduling allows for a different start order of integration processes. Due to our focus on integration process optimization and our recent work [33], these techniques are not further considered. The same applies to Loop to parallel processes, since unfoldable loops are considered a special case in integration processes that would mostly impact patterns like Message Redelivery on Exception [34].

The dynamic data (DDx) optimization techniques mostly target data reduction (cf. OS2), e.g., Early-Filter, Early-Mapping, Early-Aggregation, and reduced interaction (cf. OS5), e.g., Ignore Failing Endpoints and Reduce Requests. The data reducing optimizations go along with a process re-ordering, which can even lead to an off-loading to a message endpoint, e.g., Execution Pushdown to End-
Table 3: Dynamic Control (DCx) and Data Flow (DDx) Optimizations

| ID   | Optimization                      | OS1 | OS2 | OS3 | OS4 | OS5 | other | References                                                                 |
|------|-----------------------------------|-----|-----|-----|-----|-----|-------|-----------------------------------------------------------------------------|
| DC1  | Message Indexing                  |     |     |     |     |     | ✓     | 2-layer-hash-index [8]; ONC [?] ; Feature Selector [33]; message slices [56] |
| DC2  | Distribute Messages               |     | ✓   |     |     |     | ✓     | Heterogeneous load balancing [8]                                             |
| DC3  | Parallel Process Re-Scheduling    |     |     | ✓   |     |     | ✓     | instance scheduling [11,35]; Rescheduling Start of Parallel Flows [51,2]   |
| DC4  | Sequence to parallel / merge parallel processes |     | ✓   |     |     |     | ✓     | Rewriting Sequences to Parallel Flows, Merging Parallel Flows [52,54,12]; similar to SQL-Parallelization in [10] |
| DC5  | Loop to parallel processes        |     | ✓   |     |     |     | ✓     | Rewriting Iterations to Parallel Flows [5]                                 |
| DD1  | Reordering / Merging Conditional Paths | ✓   |     |     |     |     | ✓     | Reordering / Merging Switch-Paths [8]                                     |
| DD2  | Pushdown to Endpoints             |     | ✓   |     |     |     |       | mentioned in [8]; Web-Service-Pushdown to DB [40]; Email Selection [31,15,22]; similar to Predicate-Pushdown in [10] |
| DD3  | Early-Filter                      |     | ✓   |     |     |     | ✓     | Early Projection [31,21,14]; Early GroupBy [31,21,14]                     |
| DD4  | Early-Mapping                     | ✓   |     |     |     |     | ✓     | Materialization Point Insertion / Elimination [31,2]                      |
| DD5  | Early-Aggregation                 | ✓   |     |     |     |     | ✓     | Orderby Insertion / Removal, Join-Type Selection, Join Enumeration [8]; similar to Join-Optimization [40]; message slices [36] |
| DD6  | Claim Check                       | ✓   |     |     |     |     | ✓     | Setoperation-Type Selection, process vectorization [57]; Tuple-to-Set operations like Insert-Tuple-To-Set, Update-Tuple-To-Set, Delete-Tuple-To-Set [10]; ONC [?] |
| DD7  | Algebraic                         |     |     |     |     |     | ✓     | all Activity-Merging optimizations from [40]; Insert-Update and Update-Insert merging on workload, Select-Merging, Select-Into-Merging; Coarse-Grained-Optimierung [20] |
| DD8  | Process Vectorization             |     |     |     |     |     | ✓     | Multi-Query-Optimierung techniques [11,33,23,35]; Detector Region [33]     |
| DD9  | Pattern Split / Merge             | ✓   |     |     |     |     | ✓     | Operator merge [11,33,19,22]; all Activity-Merging optimizations from [40]; Insert-Update and Update-Insert merging on workload, Select-Merging, Select-Into-Merging; Coarse-Grained-Optimierung [20] |
| DD10 | Pre-computation of Values         |     |     |     |     |     | ✓     | Multi-Query-Optimierung techniques [11,33,23,35]; Detector Region [33]     |
| DD11 | Ignore Failing Endpoints          |     |     | ✓   |     |     | ✓     | - (Resilience Patterns [24,31])                                           |
| DD12 | Reduce Requests                   | ✓   |     |     |     |     | ✓     | Workflow-Database optimization [41]                                        |

covers ✓, partially covers (✓), does not cover -; relevant optimization techniques “gray”.  

points (cf. OS-4). The interactions can be reduced, e.g., by ignoring unreachable, non-responsive endpoints or frequently failing endpoints, as well as reducing the number of requests.

3.3 Optimizations Relevant for Application Integration

In this section, we assess the practical relevance or the found optimizations for EAI. We select only those optimizations contributing to strategies OS1–5, which makes the analysis comprehensive only for optimizations on processes (not “other”). Then we conduct an impact analysis of the selected optimizations by setting them into context of the objectives as depicted in Tab. 4.

Table 4: Relevant Optimization Techniques in the Context of the Objectives

| ID  | Optimization                          | Throughput | Latency | Resilience | Context | Complexity |
|-----|--------------------------------------|------------|---------|------------|---------|------------|
| SC1 | Redundant Path Removal                | -          | √       | (✓)        | -       | ✓          |
| SC2 | Dead Path Removal                     | -          | ✓       | (✓)        | -       | (✓)        |
| SC3 | Sub-process Inlining                  | -          | ✓       | -          | (✓)    | ✓          |
| SC4 | Sub-process Generation                | -          | ✓       | -          | ✓       | ✓          |
| SD3 | Combine Sibling Patterns              | -          | ✓       | (✓)        | -       | ✓          |
| SD4 | Unnecessary conditional fork          | ✓          | ✓       | (✓)        | -       | ✓          |
| DC2 | Distribute Messages                   | ✓          | -       | (✓)        | (✓)    | -          |
| DC6 | Sequence to parallel / merge parallel processes | ✓          | ✓       | (✓)        | -       | (✓)        |
| DD1 | Merging of Switch-Paths               | -          | ✓       | (✓)        | -       | ✓          |
| DD2 | Pushdown to Endpoints                 | ✓          | ✓       | (✓)        | -       | ✓          |
| DD3 | Early-Filter                          | ✓          | ✓       | ✓          | ✓       | ✓          |
| DD4 | Early-Mapping                         | ✓          | ✓       | ✓          | ✓       | ✓          |
| DD5 | Early-Aggregation                     | ✓          | ✓       | ✓          | ✓       | ✓          |
| DD6 | Claim Check                           | ✓          | ✓       | ✓          | ✓       | ✓          |
| DD9 | Splitting / Merging Patterns          | ✓          | ✓       | -          | -       | ✓          |
| DD11| Ignore Failing Endpoints              | -          | ✓       | ✓          | -       | -          |
| DD12| Reduce Requests                       | -          | ✓       | ✓          | -       | -          |

improves ✓; possible improvement (✓), does not improve or makes it worse -.

In summary, the static control flow optimization techniques act as Process Simplification (cf. OS1), which improves the latency and reduce the modeling complexity. The sub-process generation technique allows for a more optimal separation of duties. The static data flow optimization techniques are useful to reduce the latency and modeling complexity. The dynamic control flow optimization techniques are part of the parallelization strategy (cf. OS3) and vastly impact the message throughput. Finally, the dynamic data flow optimization techniques are data reducing (cf. OS2), and thus mostly increase the message throughput. Some of the optimizations might improve the latency and the modeling complexity. The ignore failing endpoints and reduce requests optimizations improve the resilience of the integration process.
4 Realization of Optimization Strategies

In this section, we show the realization of the optimizations strategies by discussing the integration patterns that can be used for realizing them. We also discuss the execution order of the optimization strategies and a decision tree that helps to identify the fitting strategy as practical aspects of the realization.

4.1 Realization of Optimization Strategies

In this section we formally define the optimization techniques from the different identified optimization strategies OS1–OS5 in the form of a rule-based graph rewriting system. We begin by describing the graph rewriting framework, and subsequently apply it to define the optimizations.

Graph Rewriting

Graph rewriting provides a visual framework for transforming graphs in a rule-based fashion. A graph rewriting rule is given by two embeddings of graphs $L \leftarrow K \rightarrow R$, where $L$ represents the left hand side of the rewrite rule, $R$ the right hand side, and $K$ their intersection (the parts of the graph that should be preserved by the rule). A rewrite rule can be applied to a graph $G$ after a match of $L$ in $G$ has been given as an embedding $L \rightarrow G$; this replaces the match of $L$ in $G$ by $R$. The application of a rule is potentially non-deterministic: several distinct matches can be possible. Visually, we represent a rewrite rule by a left hand side and a right hand side graph colored green and red: green parts are shared and represent $K$, while the red parts are to be deleted in the left hand side, and inserted in the right hand side respectively. For instance, the following rewrite rule replaces forks with straight edges:

Formally, the rewritten graph is constructed using a double-pushout (DPO) from category theory. We use DPO rewriting since rule applications are side-effect free (e.g., no “dangling” edges) and local (i.e., all graph changes are described by the rules). We additionally use Habel and Plump’s relabelling DPO extension to facilitate the relabelling of nodes in partially labelled graphs. A relabelling is shown in the example, where a property $p$ is rewritten to $p'$.

In addition, we also consider rewrite rules parameterised by graphs, where we draw the parameter graph as a cloud (see e.g. Fig. 3(a) for an example). A cloud represents any graph, sometimes with some side-conditions that are stated together with the rule. When looking for a match in a given graph $G$, it is of course sufficient to instantiate clouds with subgraphs of $G$ — this way, we can reduce the infinite number of rules that a parameterised rewrite rule represents to a finite number. Parameterised rewrite rules can formally be represented using substitution of hypergraphs or by $!$-boxes in open
Since we will describe optimisation strategies as graph rewrite rules, we can be flexible with when and in what order we apply the strategies. We apply the rules repeatedly until a fixed point is reached, i.e., when no further changes are possible. Methodologically, the rules are specified in a pattern-like formalism with pre-conditions, change primitives, post-conditions and an optimization effect. The pre- and post-conditions are part of the re-writing rule representation.

**OS-1: Process Simplification** We first consider the process simplification optimization strategies from Sect. 2 that mainly strive to reduce the model complexity and latency.

*Redundant sub-process* This optimisation removes redundant copies of the same sub-process within a process.

*Change primitives*: The rewriting is given by the rule in Fig. 3(a), where $SG_1$ and $SG_2$ are isomorphic pattern graphs with in-degree $n$ and out-degree $m$. In the right hand side of the rule, the $CE$ nodes add the context of the predecessor node to the message in the form of a content enricher pattern, and the $CBR$ nodes are content-based routers that route the message to the correct recipient based on the context introduced by $CE$. The graph $SG_1'$ is the same as $SG_1$, but with the context introduced by $CE$ copied along everywhere.

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

*Combine sibling patterns* Sibling patterns have the same parent node in the pattern graph (e.g., they follow a non-conditional forking pattern) with implied channel cardinality of 1:1.

*Change primitives*: The rule is given in Fig. 3(b), where $SG_1$ and $SG_2$ are isomorphic pattern graphs that are side-effect free, and $F$ is an unconditional fork.

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

---

![Fig. 3: Rules for redundant sub-process and combine sibling patterns](image)

(a) Redundant sub-process  
(b) Combine sibling patterns
Unnecessary Fork Paths A fork path is unnecessary, if the patterns on the path are side-effect free without a transitive connection to a message endpoint, or additionally read-only, when transitively connected to an endpoint.

Change primitives: The rules are given in Fig. 4 where $SG_2$ denotes a side-effect free and read-only subgraph, and $JR$ is a join router.

Effect: The model complexity and latency are reduced by the number and costs of the patterns in the subgraph.

OS-2: Data Reduction, OS-4: Pattern Placement Now, we consider data reduction optimization strategies, which mainly target improvements of the message throughput (incl. reducing element cardinalities). These optimizations require that pattern input and output contracts are regularly updated with snapshots of element data sets $EL_{IC}$ and $EL_{OC}$ from live runtime systems, e.g., from experimental measurements through benchmarks [32].

All of the data reduction optimizations discussed in this section can also be applied as OS-4 pattern placement strategies (“Pushdown to Endpoint”), by extending the placement to the message endpoints. Due to our focus on integration processes, we will not further elaborate on this here.

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

- A message filter removes messages with invalid or incomplete content. It can be used to prevent exceptional situations, and thus improves stability.
- A content filter removes elements from messages. It can be used to reduce the amount of data passed to subsequent patterns.

Change primitives: The rule is given in Fig. 5(a) where $C/MF$ is either a content or message filter matching the output contracts of $SG_1$ and the input contracts of $P_1$, and filtering out the data not used by $P_1$.

Effect: The message throughput increases by the ratio of the number of reduced data elements that are processed per second, if not limited by the throughput of the additional pattern.

Fig. 4: Rules for unnecessary fork paths
Early Filter

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

Change primitives: The rule is given in Fig. 5(b) where $\mathcal{P}_1$ is an element reducing message mapping compatible with both $\mathcal{SG}_2$, $\mathcal{SG}_3$, and $\mathcal{SG}_1$, $\mathcal{SG}_2$, and where no pattern in $\mathcal{SG}_3$ modifies the elements mentioned in the output contract of $\mathcal{P}_1$. Furthermore $\mathcal{P}_2$ is a content filter, which ensures that the input contracts of the subsequent patterns in $\mathcal{SG}_3$ are satisfied.

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

Early Mapping

A micro-batch processing region is a subgraph which contains patterns that are able to process multiple messages combined to a multi-message \[28\] or one message with multiple segments with an increased message throughput. The optimal number of aggregated messages is determined by the highest batch-size for the throughput ratio of the pattern with the lowest throughput, if latency is not considered.

Change primitives: The rule is given in Fig. 5(a) where $\mathcal{SG}_2$ is a micro-batch processing region, $\mathcal{P}_1$ an aggregator, $\mathcal{P}_2$ a splitter which separates the batch entries to distinct messages to reverse the aggregation, and $\mathcal{SG}_2'$ finally is $\mathcal{SG}_2$ modified to process micro-batched messages.
Effect: The message throughput is the minimal pattern throughput of all patterns in the micro-batch processing region. If the region is followed by patterns with less throughput, only the overall latency might be improved.

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

Change primitives: The rule is given in Fig. 6(b), where $SG_2$ is a message access-free subgraph, $P_1$ a claim check that stores the message payload and adds a claim to the message properties (and possibly routing information to the message header), and $P_2$ a content enricher that adds the original payload to the message.

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

Early-Splitter: Messages with many segments can be reduced to several messages with fewer segments, reducing complexity. Given scenario workload statistics $w_{sk}$ for scenarios $k$, and an assignment $M(p, ts)$ of number of segments for each process $p$ and throughput statistics $ts$, a segment bottleneck subsequence consists of a set of adjacent patterns $\{p_1, \ldots, p_m\}$ such that their $w_{sk}$ is significantly lower than of their preceding and succeeding patterns. The patterns in the sequence must have a similar optimal segment size. Algorithmically, bottlenecks could be found using max flow-min cut techniques.

Change primitives: The rule is given in Fig. 7 where $SSQ_1$ is a segment bottleneck sub-sequence. If $SSQ_1$ already has an adjacent splitter Fig. 7(a) applies, otherwise Fig. 7(b). In the latter case, $SP$ is a splitter and $P_2$ is an aggregator that re-builds the required segments for the successor in $SG_2$. For an already existing splitter $P_1$ in Fig. 7(a) the split condition has to be adjusted to the elements required by the input contract of the subsequent pattern in Fig. 7(b).
SSQ\textsubscript{1}. In both cases we assume that the patterns in SSQ\textsubscript{1} deal with single- and multi-segment messages; otherwise all patterns have to be adjusted as well.

**Effect:** The message throughput increases by the ratio of increased throughput on less message segments minus, if the throughput of the moved or added splitter (and aggregator) \( \geq \) message throughput of each of the patterns in the segment bottleneck sub-sequence after the segment reduction.

**OS-3: Parallelization** Parallelization optimization strategies increase message throughput. Again, these optimizations are not completely decidable on static integration scenarios according to Sect. 2 but require experimentally measured message throughput statistics \( MT_c \) (e.g., benchmarks \[ ?? \]).

**Sequence to parallel** A bottleneck sub-sequence with channel cardinality 1:1 can also be handled by parallelizing it. The parallelization factor is the average message throughput of the predecessor and successor of the sequence divided by two, which denotes the improvement potential of the bottleneck sub-sequence. The goal is to not overachieve the mean of predecessor and successor message throughput with the improvement to avoid iterative re-optimization.

**Change primitives:** The rule is given in Fig. 8(a), where SSQ\textsubscript{1} is a bottleneck sub-sequence, \( P_2 \) a fork node, \( P_3 \) a join router, and each SSQ\textsubscript{k} is a copy of SSQ\textsubscript{1}, for \( 1 \leq k \leq n \). The parallelization factor \( n \) is a parameter of the rule.

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

**Merge parallel** The balancing fork and join router realizations can limit the throughput in some runtime systems, so that a parallelization decreases the throughput. This is called a limiting parallelization, and is defined as when a fork or a join has smaller throughput than a pattern in the following sub-sequence.

![Fig. 8: Rules for sequence to parallel and merge parallel.](image-url)
Heterogeneous Parallelization

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

Change primitives: The rule is given in Fig. 9(a) where the sequential sub-sequence parts $SSQ_1$, .., $SSQ_n$ can be parallelized, $P_3$ is a parallel fork, $P_4$ is a join router, and $P_5$ is an aggregator that waits for messages from all sub-sequence part branches before emitting a combined message that fulfills the input contract of $P_2$. Similarly a rule for a heterogeneous merge is given in Fig. 9(b) (see Merge parallel above).

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

OS-5: Reduce Interaction

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

Ignore Failing Endpoints

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

**Change primitives:** The rule is given in Fig. 10(a) where \(SG_{ext}\) is a failing endpoint, \(SG_1\) and \(SG_2\) subgraphs, and \(P_1\) is a service call or message send pattern with configuration \(cf\). This specifies the collected number of subsequently failed delivery attempts to the endpoint or a configurable time interval. If one of these thresholds is reached, the process stops calling \(SG_{ext}\) and does not continue with the usual processing in \(SG_1\), however, invokes an alternative processing or exception handling in \(SG_2\).

**Effect:** Besides and improved latency (i.e., average time to response from endpoint in case of failure), the integration process behaves more stable due to immediate alternative processing. To not exclude the remote endpoint forever, the rule in Fig. 10(b) is scheduled for execution after a period of time to try whether the endpoint is still failing. If not, the configuration is updated to \(cf'\) to avoid the execution of Fig. 10(a). The retry time is adjusted depending on experienced values (e.g., endpoint is down every two hours for ten minutes).

**Reduce Requests** A *message limited* endpoint, i.e., an endpoint that is not able to handle a high rate of requests, can get unresponsive or fail. To avoid this, the caller can notice this (e.g., by TCP back-pressure) and react by reducing the number or frequency of requests. This can be done by employing a throttling or even sampling pattern \([34]\), which removes messages. An aggregator can also help to combine messages to multi-messages \([28]\).
Change primitives: The rewriting is given by the rule in Fig. 11(a), where $P_1$ is a service call or message send pattern, $SG_{ext}$ a message limited external endpoint, $SG_2$ a subgraph with $SG'_2$ a re-configured copy of $SG_2$ (e.g., for vectorized message processing [28]), and $SG_{cs}$ a subgraph that reduce the pace, or number of messages sent.

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

4.2 Realization by Patterns

One way of realizing the optimization strategies is to use the integration patterns themselves. The patterns denote general categories of solutions for message processing, and thus cover a large solution space for the realization of optimizations. Subsequently, we assign and discuss the applicability of integration patterns to realize optimization strategies denoted in Sect. 2.

Table 5: Optimization Strategies realized by Integration Patterns

| Strategy ID | Realization Categories |
|-------------|-----------------------|
| OS-1        | - (structural criteria only) |
| OS-2        | Permanent message removal (cf. Early-Filter messages, Early-Mapping) / temporal message removal (cf. Claim Check), message combination (cf. Early-Aggregation), message content filtering (cf. Early-Filter data) / size reduction (cf. Compression) |
| OS-3        | - (supporting Sequence to Parallel, Merge Parallel through non-conditional forking and joining patterns; Heterogeneous Load Balancing through forking and format handling) |
| OS-4        | cf. OS-2 categories, Early-Quality of Service and Early-Operation (cf. pushdown), Late-Quality of Service and Right-Place (cf. push-back) |
| OS-5        | resilience, tolerance (cf. Ignore Failing Endpoints), filtering and flow control (cf. Reduce Requests), prevention (cf. Distribute Messages) |

The basic characteristics of pattern compositions (i.e., soundness and reachability) as well as process simplifications (cf. OS-1) denote structural criteria on the pattern graph, and thus do not require patterns for their realization. There are 16 out of 153 patterns with potential of data reduction (cf. OS-2) can be categorized along message and message content reducing patterns. The message reducing patterns, reduce the amount of data through their potential to remove messages permanently (e.g., Content-Based Router, Message Filter, Selective Consumer, Idempotent Receiver, Channel Purger from [15], Message Cancellation, Message Expiration, Validate Message, Message Sampler from [31]) or temporarily (e.g., Claim Check [15]), combine messages (e.g., Aggregator, Composed Msg. Processor from [15]), or reduce their message content by filtering (e.g., Splitter, Content Filter from [15]), or data size reduction (e.g., Compress Content, Image Resizer from [31]).
Furthermore, there are 6 out of 153 patterns supporting the parallelization of the control flow (cf. OS-3). The realization of parallel patterns and sequences, requires non-conditional forking (e.g., Multicast \[31\], Scatter-Gather, Wire Tap from \[15\]) and joining (e.g., Join Router \[31\]) patterns. In addition, heterogeneous load balancing can be realized structurally by a Load Balancer \[31\] and content-wise by an Aggregator pattern.

Since the pattern placement (cf. OS-4) denotes an extension of OS-2, all OS-2 patterns are applicable as well (i.e., 16+12 out of 153). The patterns that can be additionally pushed-down to the sender are categorized as early-quality of service (e.g., Guaranteed Delivery \[15\]) and Early-Operation (e.g., Adapter Flow, Content Sort, Custom Script, Find and Replace, Language Translator, Message Interceptor, Type Converter from \[31\]). The patterns for push-back are according to their categories right-place (e.g., Content Enricher \[15\]) and late-quality of service (e.g., Idempotent Receiver, Resequencer from \[15\], Commutative Endpoint \[31\]).

The reduction of interactions between endpoints (cf. OS-5), for which 13 out of 153 patterns can be used, can be categorized into flow control (e.g., Back Pressure, Message Sampler, Message Throttler, Pause Operation, Timeout from \[31\]), resilience (e.g., Circuit Breaker \[31\]), prevention (e.g., Load Balancer), tolerance (e.g., Delayed Redelivery, Failover Request Handler from \[31\]), filtering (e.g., Request Collapsing, Request Caching, Request Partitioning from \[31\], Aggregator). Thereby most of the patterns from the flow control category reduce the number of requests per time, but not the general number of messages that lead to requests.

4.3 Pattern Stratification

The catalog of optimization strategies from Sect. 2 contains optimizations that have a similar effect, revert, support or inhibit each other. Hence, we analyzed the dependencies between the optimizations and propose an order of their execution. Figure 12 shows the order, in which the optimization strategies can be executed. The order is determined based on the conditions and impact of the optimization strategies. According to that DC6 might lead to SC1, while SC1 might revert DC6, hence SC1 before DC6. The dynamic control optimizations DC6 and DC8 revert each other, and thus require an order. This order implicitly comes from the observation that DC8, SC1, SD4,5 have a similar effect, and are thus related. Since we know that SC1 comes before DC6, optimizations \{SC1, SD4,5, DC8\} have to be executed before DC6 DC8 should be executed before SD4,5, since it might produce SD4,5. The static data optimization SD4,5 might inhibit DC8 in some cases, hence DC8 before SD4,5. As an intermediate summary, we can formulate the general rule “structure before data”: strategies OS-1 and OS-3, before OS-2.

Now, the data reduction strategy optimizations DD6, DD4 have a similar effect, and thus can be applied in arbitrary order. The same is valid for DD14, DD4. Since DD14 might work on too many messages, DD6 should be before DD14 to reduce the number of messages.
The pattern placement strategy OS-4 might extend the reach of OS-2, hence OS-2 before OS-4.

The pattern placement strategy OS-4 might extend the reach of OS-2, hence OS-2 before OS-4. The OS-5 strategies profit from all previous strategies and thus come last.

5 Case Study: Italy Invoice Scenario (revisited)

In this section, we study the application of the optimization strategies to the case of our motivating scenario. Thereby the strategies are applied iteratively according to the sequence from Fig. 12 until no further graph re-writing can be found.
Simplify the Integration Process

The algorithm starts on the integration process (Fig. 1) by matching and applying the process simplification (OS-1) optimizations, whose sequence can be arbitrary. For example, first, the redundant control flows (cf. SC1) starting from Encode Message for transformation and ending with Map to target format are identified and rewritten to a sequence without the multicast and join router patterns. Then the dead paths (SC2) Content Enricher to End Event is removed. And finally, the sibling patterns Sign Invoice and Ensure no duplicate invoices are identified and merged. They are placed before the Content-based Router, thus pulled out of the conditional fork! The resulting integration process is shown in Fig. 13. The original modeling complexity of 15 patterns was reduced to 10 patterns.

Scale the Sub-Processes of the Integration Process

Now the OS3 parallelization strategies are applied on Fig. 13 - transformed sequence to parallel

Place Integration Sub-process to where they belong

Figure 16 shows the original integration process from Fig. 1 after the application of strategies OS-1 to 4 in the order denoted in Fig. 12. Notably, the Message Filter could be pushed down to the sender, which sends only relevant data for the receiver plus the data needed by the integration process. The combination of Encode Message and Message Transformation can be executed in parallel to the Content Enricher.
Then the invoice is signed and sent to the authorities, which recognize duplicates and discard them. Further OS-5 strategies could be applied, if necessary.

6 Discussion

In this work we collect and briefly explain a catalog of optimizations techniques from related domains that we transferred to EAI optimization strategies. We formalize these optimizations and discuss their stratification as well as show the applicability for an extended scenario.

References

1. K. Agrawal, A. Benoit, L. Magnan, and Y. Robert. Scheduling algorithms for linear workflow optimization. In IPDPS, pages 1–12, 2010.
2. S. G. Ahmad, C. S. Liew, M. M. Rafique, E. U. Munir, and S. U. Khan. Data-intensive workflow optimization based on application task graph partitioning in heterogeneous computing systems. In IEEE BdCloud, pages 129–136, 2014.
3. A. Benoit, M. Coqblin, J.-M. Nicod, L. Philippe, and V. Rehn-Sonigo. Throughput optimization for pipeline workflow scheduling with setup times. In Euro-Par Workshops, pages 57–67, 2012.
4. L. F. Bittencourt and E. R. M. Madeira. Hcoc: a cost optimization algorithm for workflow scheduling in hybrid clouds. Journal of Internet Services and Applications, 2(3):207–227, 2011.
5. A. Böhm. Building Scalable, Distributed Applications with Declarative Messaging. PhD thesis, University of Mannheim, 2010.
Fig. 15: Integration process from Fig. 14 after application of OS2 strategies.

6. A. Böhm and C. Kanne. Demaq/transscale: Automated distribution and scalability for declarative applications. *Information Systems*, 36(3):565–578, 2011.
7. M. Böhm, D. Habich, S. Preissler, W. Lehner, and U. Wloka. Cost-based vectorization of instance-based integration processes. *Information Systems*, 36(1):3–29, 2011.
8. M. Böhm, U. Wloka, D. Habich, and W. Lehner. Model-driven generation and optimization of complex integration processes. In *ICEIS (1)*, pages 131–136, 2008.
9. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006.
10. H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. In *Switching and Automata Theory*, pages 167–180, 1973.
11. S. Finkelstein. Common expression analysis in database applications. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’82, pages 235–245, New York, NY, USA, 1982. ACM.
12. J. R. Getta. Static optimization of data integration plans in global information systems. In *ICEIS*, pages 141–150, 2011.
13. A. Habel and D. Plump. Relabelling in graph transformation. In *ICGT*, volume 2505, pages 135–147. Springer, 2002.
14. I. Habib, A. Anjum, R. Mcclatchey, and O. Rana. Adapting scientific workflow structures using multi-objective optimization strategies. *TAAS*, 8(1):4, 2013.
15. G. Hohpe and B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley, 2004.
16. A. Kissinger, A. Merry, and M. Soloviev. Pattern graph rewrite systems. In *DCM*, pages 54–66, 2012.
17. B. Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26, 2004.
Fig. 16: Integration process from Fig. 15 after application of OS4 strategies.

18. G. Kougka and A. Gounaris. Optimization of data-intensive flows: Is it needed? Is it solved? In DOLAP, pages 95–98. ACM, 2014.
19. G. Kougka, A. Gounaris, and A. Simitis. The many faces of data-centric workflow optimization: A survey. CoRR, abs/1701.07723, 2017.
20. T. Kraft, H. Schwarz, et al. Coarse-Grained Optimization: Techniques for Rewriting SQL Statement Sequences. In VLDB, pages 488–499, 2003.
21. D. S. Linthicum. Enterprise Application Integration. Addison-Wesley, 2000.
22. F. Niedermann, S. Radeschütz, and B. Mitschang. Business process optimization using formalized patterns. BIS, 2011.
23. F. Niedermann and H. Schwarz. Deep business optimization: Making business process optimization theory work in practice. In Enterprise, Business-Process and Information Systems Modeling, pages 88–102. Springer, 2011.
24. M. Nygard. Release It!: Design and Deploy Production-Ready Software. Pragmatic Bookshelf, 2007.
25. J. Park and A. Segev. Using common subexpressions to optimize multiple queries. In Proceedings of the Fourth International Conference on Data Engineering, pages 311–319, Washington, DC, USA, 1988. IEEE Computer Society.
26. D. Plump and A. Habel. Graph unification and matching. In TAGT, pages 75–88, 1994.
27. D. Ritter. Experiences with business process model and notation for modeling integration patterns. In ECMFA, pages 254–266, 2014.
28. D. Ritter. Database processes for application integration. In BICOD, pages 49–61, 2017.
29. D. Ritter, J. Dann, N. May, and S. Rinderle-Ma. Hardware accelerated application integration processing: Industry paper. In ACM DEBS, pages 215–226, 2017.
30. D. Ritter and M. Holzleitner. Integration adapter modeling. In CAiSE, pages 468–482, 2015.
31. D. Ritter, N. May, and S. Rinderle-Ma. Patterns for emerging application integration scenarios: A survey. *Information Systems*, 67:36 – 57, 2017.

32. D. Ritter, N. May, K. Sachs, and S. Rinderle-Ma. Benchmarking integration pattern implementations. In *ACM DEBS*, pages 125–136, 2016.

33. D. Ritter and S. Rinderle-Ma. Toward application integration with multimedia data. In *IEEE EDOC*, pages 103–112, 2017.

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

35. P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’00, pages 249–260, New York, NY, USA, 2000. ACM.

36. T. K. Sellis. Global query optimization. *SIGMOD Rec.*, 15(2):191–205, June 1986.

37. T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.

38. T. Tirapat, O. Udomkasemsub, X. Li, and T. Achalakul. Cost optimization for scientific workflow execution on cloud computing. In *ICPADS*, pages 663–668, 2013.

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

40. M. Vrhovnik. *Optimierung datenintensiver Workflows: Konzepte und Realisierung eines heuristischen, regelbasierten Optimierers*. PhD thesis, University of Stuttgart, 2011.

41. M. Vrhovnik, H. Schwarz, O. Suhre, B. Mitschang, V. Markl, A. Maier, and T. Kraft. An approach to optimize data processing in business processes. In *VLDB*, pages 615–626, 2007.

42. P. Zhang, Y. Han, Z. Zhao, and G. Wang. Cost optimization of cloud-based data integration system. In *WISA*, pages 183–188, 2012.