Attributed Graph Rewriting for Complex Event Processing Self-Management

Wilson A. Higashino, Cédric Eichler, Miriam Capretz, Luiz F. Bittencourt, Thierry Monteil

Attributed Graph Rewriting for Complex Event Processing Self-Management

WILSON A. HIGASHINO, Western University/University of Campinas
CÉDRIC EICHLER, INSA CVL
MIRIAM A. M. CAPRETZ, Western University
LUIZ F. BITTENCOURT, University of Campinas
THIERRY MONTEIL, LAAS-CNRS/INSA Toulouse

The use of Complex Event Processing (CEP) and Stream Processing (SP) systems to process high-volume, high-velocity Big Data has renewed interest in procedures for managing these systems. In particular, self-management and adaptation of runtime platforms have been common research themes, as most of these systems run under dynamic conditions. Nevertheless, the research landscape in this area is still young and fragmented. Most research is performed in the context of specific systems, and it is difficult to generalize the results obtained to other contexts. To enable generic and reusable CEP/SP system management procedures and self-management policies, this research introduces the Attributed Graph Rewriting for Complex Event Processing Management (*AGeCEP*) formalism. *AGeCEP* represents queries in a language- and technology-agnostic fashion using attributed graphs. Query reconfiguration capabilities are expressed through standardized attributes, which are defined based on a novel classification of CEP query operators. By leveraging this representation, *AGeCEP* also proposes graph rewriting rules to define consistent reconfigurations of queries. To demonstrate *AGeCEP* feasibility, this research has used it to design an autonomic manager and to define a selected set of self-management policies. Finally, experiments demonstrate that *AGeCEP* can indeed be used to develop algorithms that can be integrated into diverse CEP systems.

CCS Concepts: • **Information systems** → **Data streaming**; • **Computer systems organization** → **Self-organizing autonomic computing**; *Cloud computing*; • **Theory of computation** → *Rewrite systems*; • **Software and its engineering** → *Specialized application languages*

Additional Key Words and Phrases: Complex event processing, autonomic computing, self-management, attributed graph, graph rewriting

ACM Reference Format:
Wilson A. Higashino, Cédric Eichler, Miriam A. M. Capretz, Luiz F. Bittencourt, and Thierry Monteil. 2016. Attributed graph rewriting for complex event processing self-management. ACM Trans. Auton. Adapt. Syst. 11, 3, Article 19 (September 2016), 39 pages.
DOI: http://dx.doi.org/10.1145/2967499

1. INTRODUCTION

Current technological trends such as the Internet of Things and mobile computing are causing a massive explosion in the volume and velocity of data generated every

This work was partially supported by an NSERC CRD at Western University (CRDPJ 453294-13).
Authors' addresses: W. A. Higashino and M. A. M. Capretz, Dept. of Electrical and Computer Engineering, Thompson Engineering Building, Western University, London ON, Canada, N6A 5B9; emails: {whigashi, mcapretz}@uwo.ca; C. Eichler, INSA Centre Val de Loire, LIFO, 88 boulevard Lahitolle CS 60013 18022 Bourges, France; email: cedric.eichler@insa-cvl.fr; T. Monteil, CNRS, LAAS, 7 avenue du Colonel Roche, F-31400 Toulouse, France / Université de Toulouse, CNRS, INSA, F-31400 Toulouse, France; email: monteil@laas.fr; L. F. Bittencourt, Institute of Computing, University of Campinas, Av. Albert Einstein 1251, Campinas SP, Brazil, 13083-852; email: bit@ic.unicamp.br.

ACM 1556-4665/2016/09-ART19 $15.00
DOI: http://dx.doi.org/10.1145/2967499

day. Clearly, processing and understanding this enormous amount of data requires innovative approaches.

Complex Event Processing (CEP) and Stream Processing (SP) are technologies commonly applied in this new Big Data reality. Both CEP and SP are concerned with processing continuous data flows coming from distributed sources to obtain timely responses to queries [Cugola and Margara 2012]. These queries are defined by the user through a set of operators and aim to extract knowledge from incoming data. Because of their generality, the tasks to which CEP and SP are applied range from simple monitoring to highly complex financial applications such as fraud detection and automated trading [Grolinger et al. 2014].

Despite this recent surge of interest, the current CEP/SP research and development landscape is still young and particularly fragmented. A large variety of solutions exist, and they often use inconsistent terminology and different query definition languages. Consequently, most ongoing research is performed in the context of specific systems and languages.

Of particular interest for this research, algorithms and techniques aimed at query lifecycle management have often been developed in such a system-specific fashion. For instance, Aurora* can dynamically move processing load to neighbouring servers [Cherniack et al. 2003], and Nephele can dynamically resize the output buffers of query operators [Lohrmann et al. 2013]. Both these examples illustrate important query management techniques, in which the system self-adapts to changing conditions. However, they were developed in the context of their respective systems and therefore cannot be easily generalized. More recently, these limitations have become even more pronounced due to two recent trends in CEP/SP research:

—Many modern systems [Gulisano et al. 2012; Qian et al. 2013] use cloud infrastructures as runtime environments to leverage cloud performance and inherent elasticity. Management of such large deployments has led to more complex algorithms and reinforces the need to reuse them;
—Recent CEP/SP systems are being offered as services [Amazon 2015; Google 2015] and accept user-defined operators. In this context, the ability to integrate these operators in the query management loop becomes essential to these systems.

This research presents **A**ttributed **G**raph R**e**writing for **C**omplex **E**vent **P**rocessing Management (*AGeCEP*), a novel formalism that enables creation of generic and reusable procedures for CEP/SP query management. To achieve this goal, *AGeCEP* provides a language-agnostic extensible representation of queries and of reconfiguration actions that specify how queries can be transformed. Once an algorithm is expressed by this formalism, it can be integrated into diverse modern cloud-based CEP/SP systems and applied to an extensible set of query operators. In this way, *AGeCEP* overcomes the specificity of current CEP/SP research and enables creation of universal algorithms and strategies.

More specifically, this research develops the following contributions:

—The *query lifecycle management* (QLM) concept as a composition of four main steps: single-query optimization, multi-query optimization, operator placement, and runtime management;
—A novel classification of query operators focused on their reconfiguration capabilities;
—*AGeCEP*, the formalism used to represent CEP/SP queries and reconfiguration actions in a way that is both language-agnostic and appropriate for query management. This formalism is based on the classification just mentioned, attributed directed acyclic graphs, and graph rewriting rules;
—The design of an *autonomic manager* based on *AGeCEP* and the MAPE-K framework [IBM 2006] that can be used to control modern CEP systems.

The fitness of this proposal is then investigated with regard to the following:

—its feasibility as demonstrated by a general methodology that can be used to adapt operator placement procedures to *AGeCEP* and by a selected set of self-management policies defined with *AGeCEP* in the context of an autonomic manager;
—its viability as demonstrated through performance measurement experiments.

This article is structured as follows: Section 2 discusses existing work regarding CEP and related formal models. In Section 3, the query lifecycle management concept is presented and situated in the context of current CEP research. The objectives, basis, and requirements of the *AGeCEP* formalism are introduced in Section 4. In accordance with these requirements, the classification of CEP query operators is described in Section 5, and *AGeCEP* is further developed in Section 6. Section 7 presents the design of an autonomic manager based on *AGeCEP*. To prove its feasibility, placement is discussed in Section 8 and a selected set of self-management policies is examined in Section 9. Finally, a viability study is presented in Section 10, followed by conclusions in Section 11.

2. RELATED WORK

2.1. Complex Event Processing

The basis of CEP was established by the work of Luckham [2002]. At about the same time, the database community developed the first classical SP systems, such as Aurora [Abadi et al. 2003] and STREAM [Arasu et al. 2004]. CEP and SP technologies share related goals, as both are concerned with processing continuous data flows from distributed sources to obtain timely responses to queries [Cugola and Margara 2012].

Much of the early distributed CEP research was done in the context of publish/subscribe systems, such as the Publish/Subscribe Applied to Distributed REsource Scheduling (PADRES) system [Li and Jacobsen 2005]. Simultaneously, distributed SP research was under discussion, as in Aurora* [Cherniack et al. 2003] and Borealis [Abadi et al. 2005]. Current research has obviously been influenced by these works, but the recent emergence of *cloud computing* has been strongly shaping the CEP/SP landscape as well. For instance, StreamCloud [Gulisano et al. 2012] and TimeStream [Qian et al. 2013] are CEP/SP systems that use cloud infrastructures as their runtime environments.

This work uses a terminology based on the Event Processing Technical Society (EPTS) glossary [Luckham and Schulte 2011], which originated from the CEP literature. This terminology has been chosen because its terms are broadly defined and encompass most of the SP concepts. Hereafter, CEP is used as a superset of SP, as defined in the EPTS glossary.

2.2. Query Languages

In CEP systems, users create rules that specify how to process input event streams and derive "complex events." These rules have usually been defined by means of proprietary *rule languages* such as Continuous Query Language (CQL) [Arasu et al. 2005]. Similarly, the term "query" has been used by many systems to refer to user-defined event processing rules. The work described in this article uses the *query* terminology to avoid confusion with graph rewriting *rules*, which are introduced in Section 6.

Despite standardization efforts [Jain et al. 2008], a huge variety of languages are still in use today. Cugola and Margara [2012] classified existing languages into three groups as follows:

—**Declarative:** The expected results of the computation are declared, often using a language similar to Structured Query Language (SQL). The CQL [Arasu et al. 2005] is the most prominent representative of this category.

—**Imperative:** The computations to be performed are directly specified as a sequence of operators. The Aurora Stream Query Algebra (SQuAl) [Abadi et al. 2003] inspired most languages in this category.
—**Pattern-based:** Languages are used to define patterns of events using logical operators, causality relationships, and time constraints. The TESLA [Cugola and Margara 2010] language is an example of this category.

2.3. CEP Formal Models

Most previous research into CEP formal models was developed in the context of specific query languages [Arasu et al. 2005; Brenninkmeijer et al. 2008]. These models attach semantics to queries written using these languages, but they generally cannot be applied to other contexts without significant adaptation.

More recent research has targeted the development of language-independent formalisms for CEP [Krämer and Seeger 2009; Herbst et al. 2015]. These authors recognized the importance of a generic model to enable formal analysis of user-defined queries. Nevertheless, these models are still significantly different from the research described in this article because they focus on defining query semantics instead of reconfiguration actions.

Sharon and Etzion [2008] proposed the *event processing network* (EPN) formalism as a way to specify event-based applications independently of the underlying implementation. More recently, Rabinovich et al. [2010] and Weidlich et al. [2013] built on their research by implementing simulation, static, and dynamic analysis of EPNs. EPNs share similarities with *AGeCEP* because they are also language agnostic and use directed graphs as their basic representation. However, the main goal of EPNs is to represent applications that can be translated into system-specific queries, whereas the proposed *AGeCEP* aims to provide a generic representation of queries.

Cugola et al. [2015], on the other hand, proposed an approach and an accompanying tool called CAVE that can be used to prove generic properties about user-defined queries by solving a constraint satisfiability problem.

Finally, Hong et al. [2009] presented the work that most closely approximates the objectives of this research. Queries written in both declarative and pattern-based languages are converted to a graph-based query execution plan, and a set of transformation rules is applied to optimize them. Note, however, that the focus is solely on multi-query optimization, whereas this research targets the entire query lifecycle management process.

2.4. The Autonomic MAPE-K Loop

Self-adaptation through self-management is at the core of the autonomic computing paradigm [Kephart and Chess 2003]. In this paradigm, a system is provided with self-management capabilities by an autonomic manager. This manager usually implements an autonomic control loop conceptualised by the MAPE-K framework [IBM 2006], as depicted in Figure 1. This framework is named after the five functions composing it:

(1) Monitor: monitors events from the managed system to infer symptoms and sends them to analysis;
(2) Analyse: analyses symptoms and infers whether changes are required. If needed, sends request for changes to the plan function;
(3) Plan: selects the actions that must be performed based on the analysis results;
(4) Execute: executes the selected actions;
(5) Knowledge base: contains every required piece of information about the system, including actions that may be performed, their representations, and the inference rules used by the four other functions.

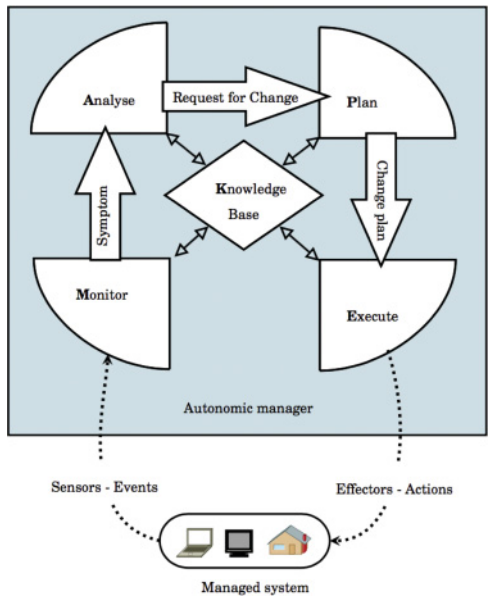


Fig. 1. The MAPE-K autonomic loop.

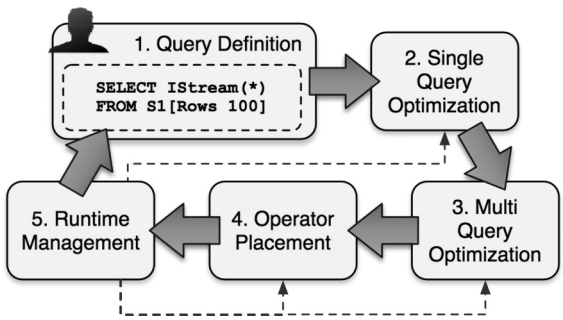


Fig. 2. Query lifecycle.

Note the MAPE-K functions might not exist as separated entities, but, logically, all of them are still present in an autonomic manager. For the sake of simplicity, this article assumes each function is implemented by its own software module.

3. QUERY LIFECYCLE MANAGEMENT

QLM can be defined as the set of tasks necessary to manage a query from the time of its definition by a user up to its execution and subsequent retirement.

In this research, the query lifecycle is defined by the five major steps illustrated in Figure 2. The cycle starts with the user creating queries using a CEP query definition language. Each query is submitted to the CEP execution engine, where it is first analysed and optimized in isolation (*Single-Query Optimization*) and, following, in conjunction with other running queries (*Multi-Query Optimization*).

In the *Operator Placement* step, the query operators are mapped to a subset of the available computing resources and starts executing. Following, during *Runtime Management*, the system maintains the query execution, responding to context changes such as hardware and software failures. In this step, the system may need to re-optimize

and re-place queries. This dependency is represented in Figure 2 by dashed arrows from box 5 to boxes 2, 3, and 4. Finally, based on the results obtained by the query, users can decide to refine it or to create one or more new queries, which originates a new cycle.

In the following subsections, each of the QLM steps is detailed.

3.1. Query Definition

Query definition is the step in which users define the CEP queries they want to execute. As mentioned in Section 2.2, each system usually has its own query language that is used for this purpose. In addition, the way that users interact with the system to define and submit queries differs enormously from system to system. For example, commercial CEP packages such as Oracle Stream Explorer [Oracle 2015] and Software AG Apama [Software AG 2015] have full-fledged interfaces that help to define queries and monitor their execution. On the other hand, many academic [Qian et al. 2013] and lower-level systems [Storm 2015] provide only programming language Application Programming Interfaces (APIs) that are mostly targeted to software developers.

Because the formalism presented in this research is independent of query language and, consequently, of how queries are defined, this step is not discussed further in the rest of this article.

3.2. Single-Query Optimization

Single-query optimization (SQO) is the action of modifying a query to improve its efficiency while keeping its functional properties unchanged. The "efficiency" of a query is usually measured with respect to some optimization criterion such as processing latency, CPU usage, or network consumption. This step is essential because it reduces the need for technical knowledge by users: Non-optimized queries are corrected before they are run, reducing their impact on the system.

SQO is executed right after a new query is created and registered. Consequently, this step assumes no *a priori* knowledge about available resources or about the state of the network and servers.

Among all the QLM steps, SQO has been the least studied in the literature. For example, both STREAM [Arasu et al. 2005] and TimeStream [Qian et al. 2013] rely mostly on runtime adaptation and use only a few *ad hoc* heuristics in SQO.

3.3. Multi-Query Optimization

Multi-query optimization (MQO) consists of finding overlaps (common partial results) between queries and merging them into a single query while maintaining their logical separation. This step usually optimizes the same criteria as the single-query optimization step. MQO can be executed as a separate step when a new query is created or periodically to take account of modifications in the underlying queries. In both cases, one of the greatest challenges is to decide which queries should be considered.

MQO has received more attention than SQO. Early research was largely based on MQO for relational databases and focused on sharing specific operators, such as in Madden et al. [2002] and Hammad et al. [2003]. Hong et al. [2009] was one of the first work that formalized MQO as an independent task that can be applied to continuous queries.

3.4. Operator Placement

Operator placement is the step in which a query execution is mapped into the set of available computing resources. In the context of distributed and cloud-based CEP systems, this usually translates into determining the number and types of servers required and how the queries should be split among multiple processors. More recently, placement has also been considered in multi-cloud environments [Borgetto et al. 2016].

This step is executed during initial system deployment, when a new query or a new operator is created and, in general, whenever a reconfiguration requires a placement decision. Because of this variety of scenarios, it is common to have different approaches to deal with incremental and global placement decisions, for example, placement of a new operator versus placement of all running queries.

Xing et al. [2005] and Pietzuch et al. [2006] presented classical operator placement approaches for distributed CEP systems. For more information about operator placement strategies, the survey by Lakshmanan et al. [2008] can be consulted.

3.5. Runtime Management

Runtime management refers to the self-managed evolution of a system at runtime. Here, queries are *reconfigured* in response to context changes such as hardware and software failures and sudden bursts of events. This step is the most commonly implemented of all the steps in query lifecycle management.

To support proper runtime management, CEP systems usually define and enforce a number of *self-management policies* aiming to improve or to maintain the quality of service for queries. The implementation of these policies requires two main capabilities: *detecting* when a reconfiguration is required and then *executing* reconfiguration actions.

The *detection* step frequently involves monitoring system metrics, such as CPU load and operator queue size, and comparing them with some threshold. The *execution* of reconfiguration actions, on the other hand, can have many different forms and implementations.

One possible classification of reconfiguration actions focuses on their *scope* and coarsely categorizes them as *behavioural* or *structural*. *Behavioural* actions change operator and system parameters but do not modify the query or the system structure. Common examples are load shedding [Abadi et al. 2003] and buffer resizing [Lohrmann et al. 2013]. Conversely, *structural* actions require adapting the structure of queries and their mapping into system resources. Splitting a query to distribute its execution [Cherniack et al. 2003] and migrating operators to underloaded servers [Heinze et al. 2014] are examples of *structural* actions. The *AGeCEP* formalism focuses on structural reconfigurations, as detailed in Section 6.

4. ATTRIBUTED GRAPH REWRITING FOR CEP MANAGEMENT

The *AGeCEP* formalism has been developed to enable specification of self-management policies that can be applied to query lifecycle management.

To achieve this goal, two main challenges have to be overcome: The first is to find a query representation that is language agnostic, yet can encode all information required by self-management policies. The second is to find a way to specify unambiguous reconfiguration actions that act on the represented queries. The following subsections discuss these challenges further.

4.1. Modelling Queries

AGeCEP represents CEP queries as Attributed Directed Acyclic Graphs (ADAGs). In an *AGeCEP* query graph, each vertex represents a query element, and each edge represents an event stream flowing from one element to another. Query elements, in turn, are further classified as:

—*event producers*: sources of events processed by the query;
—*event consumers*: consumers of query results;
—*query operators*: any processing logic that can be applied to one or more input streams and generates one or more output streams as a result.

Because the graphs used are attributed, it is also possible to represent properties that qualify the vertices and edges and enrich their representation. Here, the attributes considered should include all pieces of information required by self-management policies.

To identify these properties, a novel classification of query operators focusing on their reconfiguration capabilities was elaborated. Integrating a new operator into *AGeCEP*, therefore, is simply equivalent to classifying it properly. Details of this classification are presented in Section 5.

Use of ADAG as a language agnostic representation of CEP queries is a natural choice corroborated by many studies in the literature. For instance, most CEP systems based on imperative languages use (non-attributed) DAGs to represent user queries [Abadi et al. 2003; Storm 2015]. Systems that use declarative languages, on the other hand, transform user queries into query plans to make them "executable," which often leads to DAG-like structures (e.g., the STREAM system and the CQL language [Arasu et al. 2005]). *AGeCEP* generality is further discussed in Section 6.1.

4.2. Modelling Reconfiguration Actions

In CEP systems, self-management policies may act on different steps of the query lifecycle and have various goals. In a broad sense, however, they all follow a similar structure in which (i) potential problems are detected; (ii) appropriate reconfiguration actions are selected; and (iii) the selected actions are applied as a response.

In this structure, problem detection and action selection are mostly independent of the chosen query representation. On the other hand, the representation of reconfiguration actions is heavily influenced by this choice. *AGeCEP*, therefore, also focuses on the definition and representation of reconfiguration actions. These actions can be applied to transform queries and can be used by any procedure, including but not limited to self-management policies. More precisely, because this research focuses on structural reconfiguration of queries modelled by ADAGs, it is natural to represent the actions under consideration using a graph transformation formalism.

Such reconfigurations can be modelled formally, yet visually and intuitively by graph rewriting rules. Graph rewriting is a well-studied technique [Rozenberg 1997] with multiple applications, including self-management [Rodriguez et al. 2010; Eichler et al. 2013].

Note that a graph rewriting rule formally specifies both a reconfiguration (i.e., its effect) and the context in which it can be applied (i.e., its applicability), enabling the study and establishment of guarantees of reconfiguration correctness [Eichler et al. 2016].

4.3. Discussion

In the context of self-management policies and autonomic computing, *AGeCEP* queries and reconfiguration actions are part of the *knowledge base* (KB). Specifically, *AGeCEP* focuses on representing *"what can be done"* and not on the decision-making process that determines *"what should be done."* The MAPE-K modules are expected to use *AGeCEP* to implement their functions in conjunction with other information available in the KB such as monitored events, inference rules, and runtime environment models.

Note that by limiting *AGeCEP* scope to queries and reconfiguration actions, it is possible to integrate *AGeCEP* with existing models and techniques rather than forcing the adoption of particular ones. By doing so, *AGeCEP* can be applied to a broader range of scenarios.

Section 7 shows how existing representations and meta-models can be coupled with *AGeCEP* to cover the whole MAPE-K loop and thereby implement a complete autonomic manager.

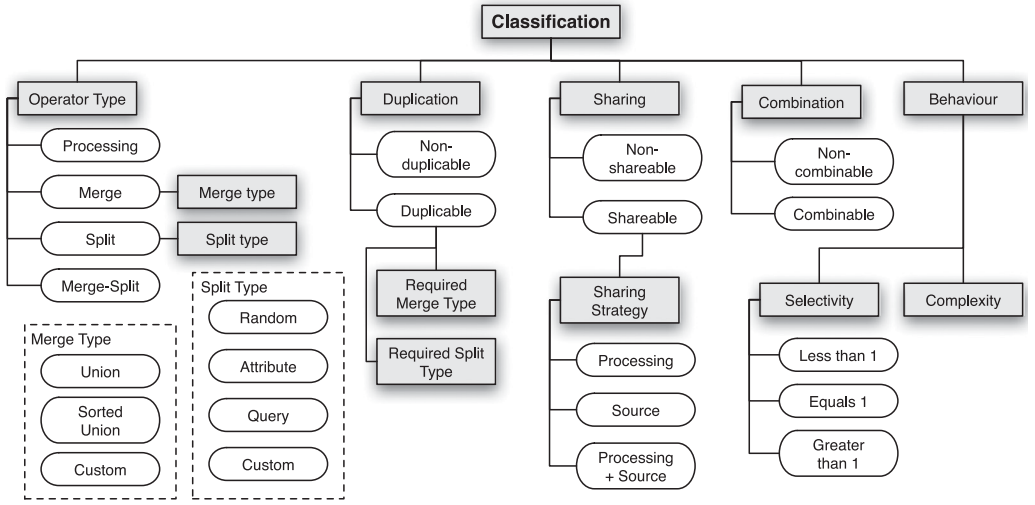
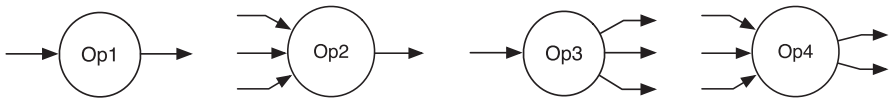


Fig. 3. Proposed classification.



(a) Processing. (b) Merge. (c) Split. (d) Merge-Split.

Fig. 4. Operator types—examples.

5. CLASSIFICATION OF CEP OPERATORS WITH RESPECT TO THEIR RECONFIGURATION CAPABILITIES

One underlying purpose of *AGeCEP* is to abstract queries and operators while expressing any information relevant to their self-management. To achieve this goal, this section identifies a set of criteria related to operator management and presents a novel classification of CEP query operators focused on their reconfiguration capabilities.

This classification is at the core of the proposed approach because generic query rewriting rules in *AGeCEP* are applicable to virtually any set of properly classified CEP operators. Figure 3 presents an overview of the criteria on which the operators are classified. Each criterion is detailed in one of the following subsections.

5.1. Operator Type

The *type* criterion classifies operators according to the number of input and output streams. There are four different categories in this criterion, illustrated in Figure 4:

—**Processing:** The operator has one input and one output stream only. These operators can filter events from the input stream, transform them, or both.
—**Merge:** The operator has two or more input streams, which are processed together and merged into one output stream.
—**Split:** The operator has one input stream, which is processed and split into two or more output streams.
—**Merge-Split:** The operator has more than one input stream and more than one output stream.

Merge operators are sub-classified according to the type of merge they execute:

—**Union:** Input events are output as they arrive, with no ordering guarantees.
—**Sorted union:** Input events are output sorted based on a specified set of attributes.
—**Custom:** A customized function defines how the input streams are merged.

Finally, split operators are also characterized based on the type of split they perform:

—**Random:** Input events are sent to a randomly selected output stream.
—**Attribute:** The output stream is selected based on the values of a specified set of attributes.
—**Query:** Input events are split according to the query from which they come.
—**Custom:** A customized function defines how the events are split.

5.2. Sharing

The *sharing* criterion refers to the ability of a single operator instance to be shared by two or more occurrences of the operator. This characteristic is especially important for multi-query optimization, in which the results of common query subgraphs are reused among queries.

This criterion is essentially determined by the operator implementation. An operator is *non-shareable* if one runtime instance must be created for each operator occurrence. On the other hand, an operator is *shareable* if a single instance implements more than one occurrence. In this case, three *sharing strategies* are identified:

—**Processing:** One operator instance is shared among occurrences that execute the exact same data processing but using different input streams as sources.
—**Source:** One operator instance is shared among occurrences that execute similar data processing using the same input streams as sources.
—**Processing+Source:** One operator instance is shared among occurrences that execute the same data processing on the exact same input streams.

Figure 5(a) informally illustrates an example of a *processing* shareable operator. In this case, a single instance can be used to process both input streams s_a and s_b, as represented in the right-hand part of the figure. This sharing is possible only because the same filter ($loc = 1$ or 2) is applied to both streams. This type of sharing is usually chosen when an operator instance consumes a lot of memory, and it is therefore important to create as few instances as possible.

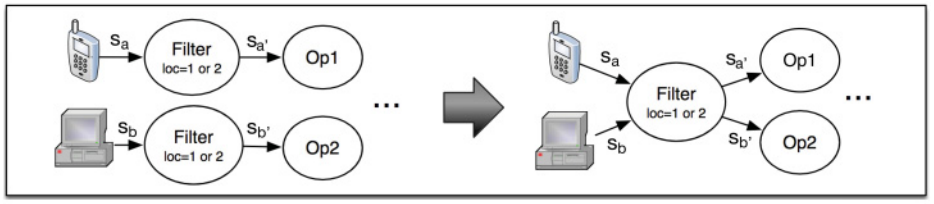
In the example from Figure 5(b), the filter operator is *source* shareable. In this case, both filter occurrences process the same input stream s_a and have predicates over the attribute *loc*. The resulting filter instance implements both predicates. This type of sharing is applied when it is more efficient to implement multiple processing logics as a single operation than it is to implement these logics independently.

Finally, the filter operator is assumed to be *processing+source* shareable in Figure 5(c). In this example, the exact same data processing is executed on the same input stream, and therefore only a single instance is necessary. This type of sharing enables savings in both memory and CPU consumption and is the most commonly used by CEP systems.

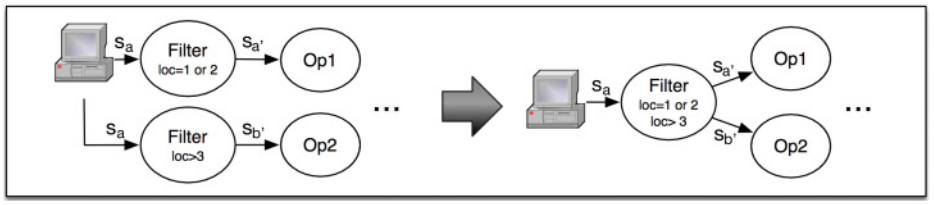
5.3. Duplication

A common strategy used to increase operator throughput is to create more than one instance of the operator, assign them to different servers (or cores), and split the input events among these instances. This strategy is illustrated in Figure 6.

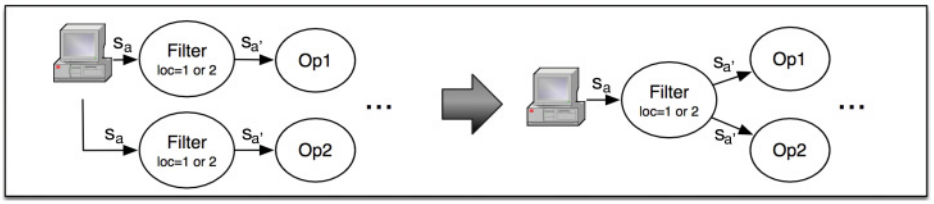
Because of the prevalence of this strategy, the proposed classification contains a *duplicable* criterion, which is true when the operator can be duplicated and the processing load distributed according to the described strategy. Moreover, when an operator is *duplicable*, two other aspects must be considered: the *required split type* and the *required*



(a) Processing sharing.



(b) Source sharing.



(c) Processing+Source sharing.

Fig. 5. Sharing strategies.

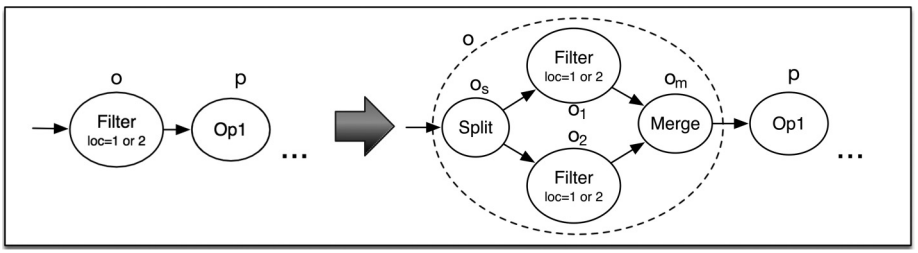


Fig. 6. Duplication strategy.

merge type. These two criteria determine the type of split (merge) operator that precedes (succeeds) the duplicated operator. The possible split (merge) types are the ones defined in Section 5.1.

The *required split* and *required merge* types are ultimately defined by the duplicated operator implementation. Generally speaking, stateless operators can be duplicated and accept *random* splits because each event is processed in isolation. Conversely, stateful operators usually require *attribute* splits because they process together events with similar characteristics (the same attribute values).

Finally, note that a *sorted union* merge type is needed in scenarios in which the output stream must be kept ordered after duplication. For example, in Figure 6, there is no guarantee that the events will reach the merge operator o_m in the same order

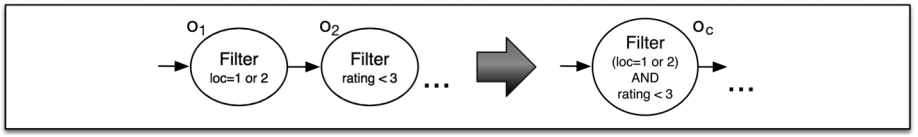


Fig. 7. Combination strategy.

they reached the split o_s. If order needs to be maintained, then the operator o_m must be replaced by a *sorted union*.

5.4. Combination

The *combinable* criterion is true when two or more consecutive occurrences of an operator o can be combined into a single operator o_c, which effect is equivalent to applying all the combined operators in any order (Figure 7).

In most cases, the combinable operators o_i and the equivalent combined one o_c have the same implementation. In other cases, o_c differs. For example, two binary joins may be combined as a multiple join operator, which usually has a different implementation than the binary operator. Hence, the implementer of a combinable operator o must provide:

—the implementation of operator o_c resulting from a combination of o instances.
—a function that, given the parameters of successive instances of o, returns the parameters of the equivalent combined operator o_c.

This criterion is especially useful for SQO, in which operators can be combined to reduce the number of operators in a query.

5.5. Behaviour

This category groups the characteristics of an operator related to its functional behaviour. More specifically, operators are classified according to two criteria:

—**Complexity:** refers to computational complexity as a function of the size of the input streams.
—**Selectivity:** refers to the relation between the numbers of output and input events. An operator selectivity less than 1 means that the number of output events is less than the number of input events, whereas a selectivity greater than 1 implies that the number of output events is larger than the number of input events.

5.6. Discussion

The classification presented in this section has been created based on an extensive literature review of query lifecycle management research. It focuses on intrinsic reconfiguration capabilities of query operators that are crucial to establish how they can be reconfigured. As demonstrated in Section 9, these properties enable the expression of a myriad of different procedures in the context of CEP systems.

Nevertheless, it is expected that not all properties required by current and future systems are expressed in this classification. For this reason, the classification can be easily extended with other criteria as needed. In addition, extrinsic operator properties, such as runtime information, are not part of the classification because they are too numerous and tightly coupled to the management procedures that use them. Section 6.1 discusses how new criteria and attributes are handled in *AGeCEP*.

6. REPRESENTATION OF QUERIES AND THEIR TRANSFORMATIONS

AGeCEP provides graph-based models to represent two fundamental aspects of dynamic CEP systems: the system state, which is primarily defined by the running queries, and possible transformations of this state, given by a set of reconfiguration actions. The following subsections detail both models.

6.1. Query Representation Using ADAGs

In *AGeCEP*, each user-defined query q is represented by an *attributed directed acyclic graph* G. Because the graph is attributed, the vertices and edges are augmented with a set of attributes that qualify them. Formally, such an attributed graph is specified by a triple (V, E, ATT), where:

—the vertices V represent the query elements,
—the edges E represent event streams flowing from one element to another,
—and ATT is a family of attribute sets indexed by $V \cup E$.

Formally, each set of this family is defined as a sequence of triples (N, L, T), where N, L, and T are the attribute name, value, and type (i.e., domain) respectively.

To represent the types of elements and interactions that may be involved in a CEP system, *AGeCEP* also defines stereotypes for the vertices and edges of a query graph. Each stereotype specifies a set of attributes that are common to elements of that specific stereotype.

6.1.1. Vertex Attributes. The vertices from a query graph $G = (V, E, ATT)$ can represent event producers, event consumers, or query operators, denoted as V_p, V_c, and V_o, respectively. V_p, V_c, and V_o specify a partition of V, that is, all sets are disjoint subsets of V, and their union is V.

Query operators all belong to the same stereotype and therefore share the same list of attributes depicted in Table I. The nature of these attributes is directly related to the properties considered relevant for defining self-management policies, which were identified in the classification presented in Section 5. As mentioned, note this classification is extensible and new criteria can be added as needed. In this case, the new criteria translate directly to new attributes, and the possible values for the criteria correspond to the attributes domain.

Event producers and *consumers* also define their own stereotypes, which contain the first five attributes of the *operator* stereotype: *id*, *impl*, *params*, *inDegree*, and *outDegree*. Event producers (consumers) necessarily have an *inDegree* (*outDegree*) equal to 0.

6.1.2. Edge Attributes. AGeCEP uses a single stereotype for edges. The attributes of this stereotype are described in Table II.

Note that except for *id*, all edge attributes can be inferred from the graph structure and vertex attributes. Similarly, the *inDegree* and *outDegree* of a vertex can also be inferred from the graph. Nevertheless, they are maintained as attributes to simplify the definition and implementation of reconfiguration rules.

Furthermore, it should be emphasized that neither vertex nor edge attributes are closed sets and can be extended whenever necessary. In particular, extrinsic properties such as operator placement and runtime information can also be modelled as vertex and edge attributes.

6.1.3. Example 1: Imperative Language. Figure 8 shows two queries q_1 and q_2 using the *AGeCEP* representation. To simplify the figure, some attributes have been omitted. The notation $\langle\langle op \rangle\rangle$ specifies that the vertex is of the operator stereotype, whereas $\langle\langle prod \rangle\rangle$

Table I. Attributes of the Vertex Stereotype "Operator"

Name	Type	Description
id	String	a unique identifier
impl	String	the operator implementation name
params	List of strings	the operator parameters
inDegree	$\mathbb{N}$	the number of incoming edges
outDegree	$\mathbb{N}$	the number of outgoing edges
type	{"processing", "merge", "split", "merge-split"}	operator type
mergeType	{"union", "sorted", "custom", "N/A"}	if *type* ="merge", the merge type
splitType	{"random", "attribute", "query", "custom", "N/A"}	if *type* ="split", the split type
shareable	Boolean	Is the operator shareable?
shStrategy	{"processing", "source", "proc+source", "N/A"}	if *shareable*, the sharing strategy
combinable	Boolean	Is the operator combinable?
combImpl	String	if *combinable*, the combined operator o_c's impl. name
combParam	fun: List of List of strings → List of strings	and o_c's parameters function
duplicable	Boolean	Is the operator duplicable?
reqMerge	{"union", "sorted", "custom", "N/A"}	if *duplicable*, the succeeding *mergeType*
reqSplit	{"random", "attribute", "query", "custom", "N/A"}	and the preceding *splitType*

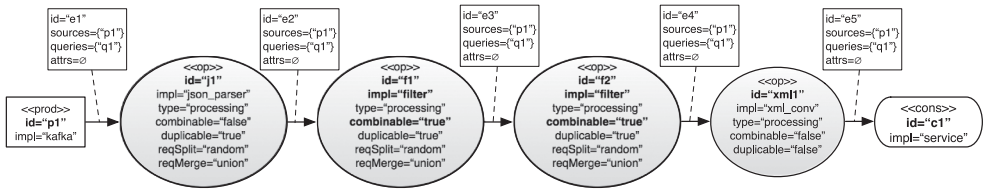
Table II. Edge Attributes

Name	Type	Description
id	String	a unique identifier
sources	the power set of V_p	producers of events flowing through the edge
queries	List of String	the set of queries that share the edge
attrs	List of String	name of attributes according to which the events in the edge are grouped

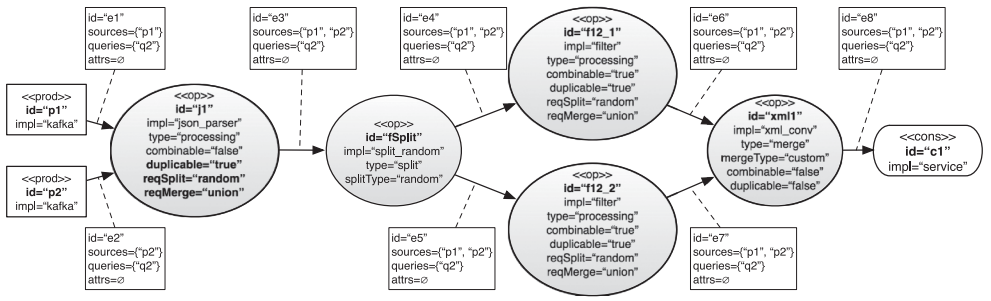
and $\langle\langle cons \rangle\rangle$ qualify an event producer or an event consumer. These queries have been extracted from the Powersmiths' WOW system [Powersmiths 2015], a sustainability management platform that uses live measurements of buildings to support energy management and education. In WOW, queries are implemented in Apache Storm [2015] and are used to process readings coming from building sensors managed by the platform. The conversion from Storm queries to *AGeCEP* is straightforward because Storm also represents queries using DAGs.

Query q_1 in Figure 8(a) is used to convert readings from JSON to the native WOW format (XML). The query is implemented as a sequence of four operators: first, operator j_1 converts the JSON reading into a Java object. Then filters f_1 and f_2 remove invalid readings from the event stream. Finally, operator xml_1 converts the reading to an XML document and forwards it to the appropriate service.

Query q_2 (Figure 8(b)) is used for the same purpose but has a different structure. The query has two producers and two instances of operator f_{12}, which executes a processing (filtering) logic equivalent to the sequential application of f_1 and f_2.



(a) Query q_1—Sequential version.



(b) Query q_2—Duplicated filtering.

Fig. 8. JSON to XML conversion—Storm queries.

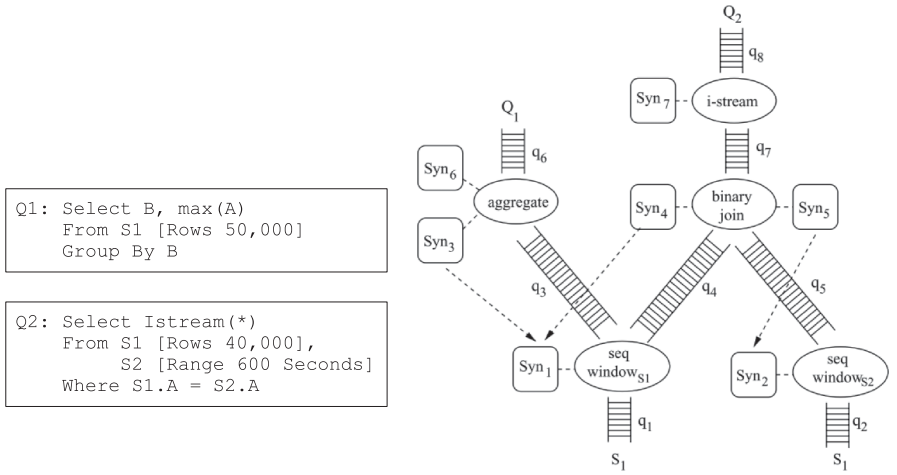
6.1.4. Example 2: Declarative Language. Figure 9 exemplifies how queries written in the CQL language [Arasu et al. 2005] can be converted to the ADAG format used in *AGeCEP*. The original queries q_1 and q_2 are shown in Figure 9(a). As it is common in declarative query languages, CQL queries are transformed into a graph-based execution plan before being actually run. Figure 9(b) depicts the resulting plan for q_1 and q_2. Both queries were processed together and transformed into a single plan.

From this plan, the conversion to the *AGeCEP* representation is direct: Operators and queues are mapped to vertices and edges, respectively. The resulting ADAG can be visualized in Figure 9(c). Note that the graph expresses most information presented in the query plan, including the fact that the seq_window operator can be shared among queries that process the same input sources.

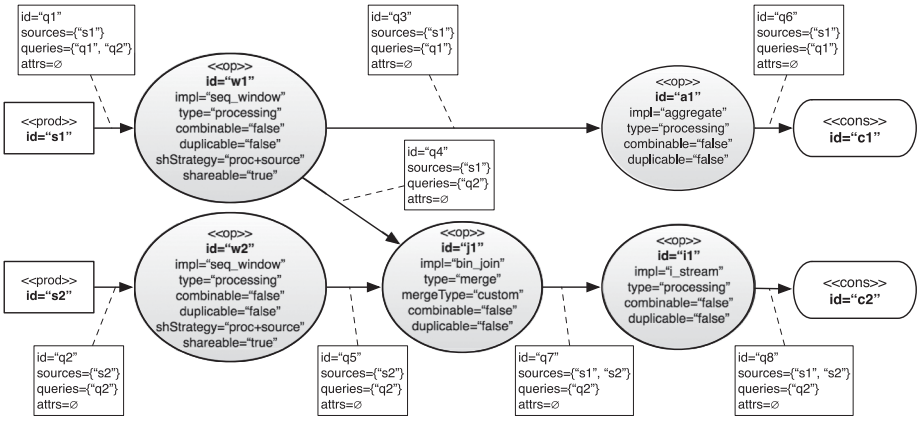
6.1.5. Example 3: Pattern-Based Language. Figure 10 shows the conversion from a Cayuga Event Language (CEL) [Demers et al. 2007] query to *AGeCEP*. CEL is a pattern-based language, even though it uses keywords that are similar to SQL. For instance, CEL uses the operators $NEXT$ and $FOLD$ to search for sequence of events that satisfy a stated condition, which is a construct characteristic of this language group.

In Cayuga, queries are transformed into a non-deterministic finite state automaton to be executed. Figure 10(a) shows a query q and its corresponding automaton. Even though this automaton can be represented as a graph, its semantics differs from *AGeCEP* queries. For instance, the automaton states (vertices) are associated with input streams, whereas in *AGeCEP* vertices represent operators.

Hong et al. [2009] presented a procedure to transform Cayuga automata to graph-based execution plans. Basically, they introduced two new query operators that implement the $NEXT$ and $FOLD$ logics and a procedure to convert edge transitions to a sequence of a filter followed by a projection. The execution plan for the example query is depicted in Figure 10(b). Once transformed to a graph-based execution plan, the conversion to *AGeCEP* is direct and results in the ADAG shown in Figure 10(c).



(a) CQL queries [Arasu et al. 2005]. (b) Generated execution plan [Arasu et al. 2005].

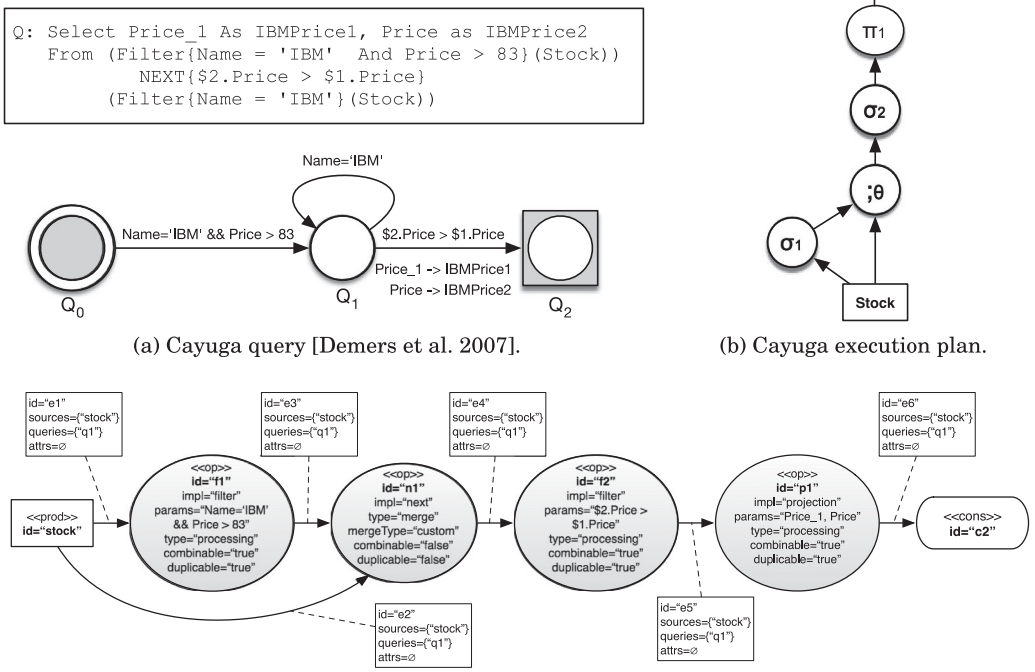


(c) *AGeCEP* representation.

Fig. 9. Conversion from a CQL query to *AGeCEP*.

6.1.6. Discussion. The *AGeCEP* query representation has been designed to be as generic as possible. Most queries written in imperative and declarative languages can be converted directly to an *AGeCEP* ADAG. Pattern-based languages, on the other hand, require additional procedures for conversion, such as the one presented by Hong et al. [2009] and demonstrated in the previous example. These additional procedures are needed because most pattern-based languages are executed as automata that do not follow *AGeCEP* graph-based model. In other words, there is a semantic mismatch between the models that must be solved before using *AGeCEP* to represent pattern-based queries.

Nevertheless, such mismatch should be solvable in most cases. For instance, Hong et al. [2009] mentioned that the Sase language [Wu et al. 2006] could be transformed to a graph-based execution plan using a procedure similar to that used to transform CEL queries. Similar procedures could also be applied to TESLA [Cugola and Margara 2010]. The development of such procedures, however, is outside the scope of this article and may need to be analysed case by case.



(a) Cayuga query [Demers et al. 2007].

(b) Cayuga execution plan.

(c) *AGeCEP* representation.

Fig. 10. Conversion from a Cayuga query to *AGeCEP*.

6.2. Query Transformation Using Graph Rewriting

In *AGeCEP*, query reconfigurations are formally expressed in a rule-oriented fashion using graph rewriting rules.

Various ways of specifying graph rewriting rules have been developed in the past [Rozenberg 1997]. This research uses the graphical representation and underlying formalism of the AGG[1] tool [Taentzer 2004], a well-established graph transformation environment [Segura et al. 2008]. AGG is based on the Single Push-Out (SPO) approach [Löwe 1993; Ehrig et al. 1997].

6.2.1. Graph Rewriting Rules. The SPO approach is an algebraic technique for graph rewriting based on the category theory [Awodey 2006], where a rule r is specified by $L \xrightarrow{m} R$, where:

—L and R are attributed graphs called the left-hand and right-hand sides of r.
—m is a partial morphism from L to R, that is, a morphism from a sub-graph L_m of L to R. This morphism is not necessarily injective.

A rule $r : L \xrightarrow{m} R$ is applicable to a graph G if G contains an image of L, that is, if there is a homomorphism h from L to G. Such homomorphism is denoted as $h : L \to G$. Also, the notation $h(G_s)$ is used to denote the image of some subgraph G_s of G by the morphism h. The application of r to G with regard to h consists of constructing the push-out [Awodey 2006] of m and h, as illustrated in Figure 11. The result of this application is the graph $m_h(G)$.

[1]AGG's homepage: http://user.cs.tu-berlin.de/~gragra/agg/index.html.

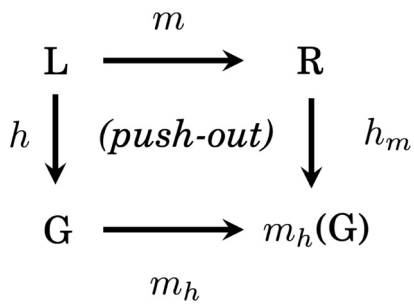
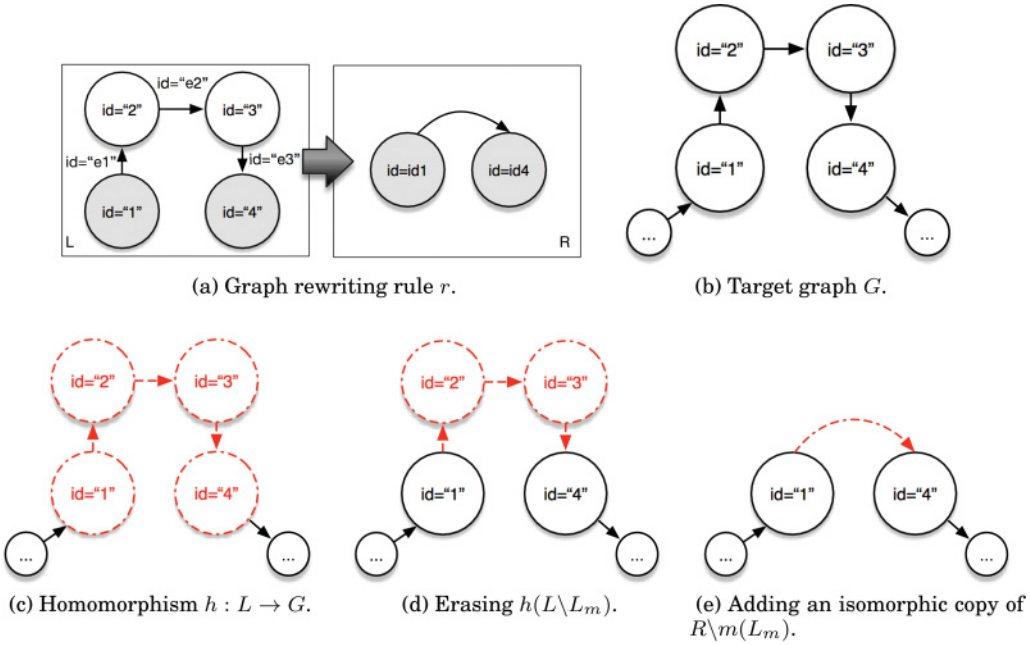


Fig. 11. Construction of a push-out: Application of a graph rewriting rule.



(a) Graph rewriting rule r. (b) Target graph G.

(c) Homomorphism $h : L \to G$. (d) Erasing $h(L \backslash L_m)$. (e) Adding an isomorphic copy of $R \backslash m(L_m)$.

Fig. 12. Illustration of a graph rewriting rule r and its application.

Informally, the application of r to G with regard to h consists of replacing the image of L in G by an image of R. This can be understood as a three-step process:

(1) erasing the image by h of the part of L that is not in m's domain, $h(L \backslash L_m)$.
(2) adding an isomorphic copy of the part of R that is not in the image of m (a copy of $R \backslash m(L_m)$).
(3) if m is not injective, that is, if some vertices v_i of L have the same image by m, then the images of these v_i are merged into $m_h(G)$.

For the rest of this article, morphisms m of the introduced rules may not be explicitly shown. Such morphisms are implicitly defined as the identity mapping between the largest common sub-graphs of L and R, where vertices are uniquely identified by their id.

The application of a rule r to a graph G is illustrated in Figure 12. The rule r and its corresponding left- and right-hand sides (L, R) are depicted in Figure 12(a). In this

rule, the morphism m from L to R is implicit and defined by the identity mapping. The highlighted nodes in L and R correspond to L_m and $m(L_m)$, respectively.

The target graph is presented in Figure 12(b), and the steps required to apply the rule are shown in Figures 12(c) to 12(e). First, a homomorphism $h : L \rightarrow G$ is found. Next, $h(L \backslash L_m)$ is removed from G, followed by the addition of an isomorphic copy of $R \backslash m(L_m)$. The rule suppresses the nodes with id equal to 2 and 3 and connects directly the nodes with id 1 and 4.

6.2.2. Rewriting Rules and Attributes in AGeCEP. Vertices and edges appearing on the left- and right-hand side of *AGeCEP* rules are analogous to those appearing in queries: operators, event producers or consumers, and event streams. Hence, they can also be classified according to the stereotypes described in Section 6.1.

One of the main differences is that attributes appearing in a rule may be defined as follows:

—*fixed value*. Fixing an attribute value in L means that the corresponding attribute in the image by h should have the same value. A fixed value is either a parameter of the rule or a constant written between quotes.
—*non-fixed value*. If an attribute value is not fixed in L, then the corresponding attribute in the image by h can have any value. Non-fixed valued attributes are omitted in the rule definition.
—*variable*. If an attribute is associated with a variable in L, then the variable is bound to the value of the corresponding attribute in the image by h. If the variable appears more than once in L, then all its occurrences must bind to the same value; otherwise, the rule is not applicable. A variable that appears in L can also appear in R. In this case, the variable in R is replaced with its bound value.
—*operations*. Attributes may be associated with simple operations in R (typically increment or decrement). These operations are applied along with the rule.

6.2.3. Mutators: Extending Rewriting Rules with Actions on the Real System. Mutators were first introduced as a lightweight method for handling attribute changes [Eichler et al. 2016]. As such, they are described as *arbitrary algorithms updating the value(s) of no, one, or some attributes*. Any rewriting rule may be enriched with a set of mutators executed at the end of its application phase. Later, a new kind of mutators has been introduced [Eichler 2015] to describe actions on real systems, typically through method or API calls. Such mutators are called external as opposed to internal mutators that act only on the model. In *AGeCEP*, graph rewriting rules are specified as a couple $(L \overset{m}{\rightarrow} R, ACTS)$, each rule being enriched with a set $ACTS$ of external mutators μ enforcing model changes on the real system through API calls.

6.2.4. On the Correctness of Rewriting Rules in AGeCEP. In *AGeCEP*, the correctness of a reconfiguration is linked to the reconfiguration capabilities of the operators it impacts: A rule describing a reconfiguration should be applied only to operators with the proper capabilities (e.g., duplication should be applied to a duplicable operator). This is guaranteed by fixing the value of the corresponding attributes on the left-hand side of a rule. Therefore, a properly classified operator can be safely transformed using the defined rules.

6.2.5. Examples. Figure 13 illustrates a graph-rewriting rule P_{comb} whose goal is to combine a sequence of two query operators into a single new operator.

The left-hand side of the rule encodes all necessary conditions that operators must satisfy to enable the combination:

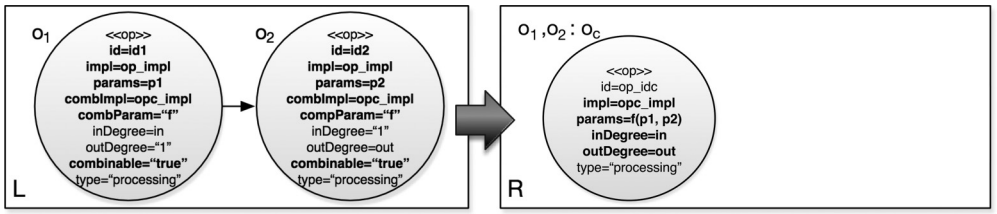


Fig. 13. Combination of two combinable successive operators P_{comb}.

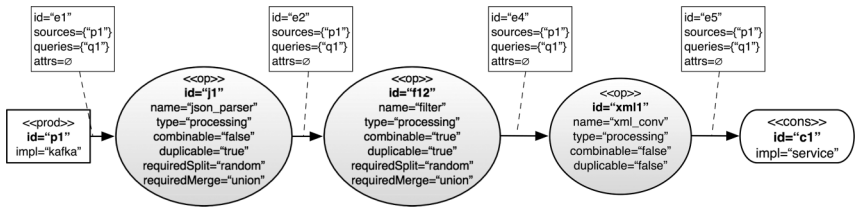


Fig. 14. Query q_1—Optimized version.

(1) the output of o_1 is exactly the input of o_2, that is:
   (a) they are directly connected, as represented by the edge (o_1, o_2), and
   (b) $o_1(outDegree) =$ "1" and $o_2(inDegree) =$ "1";
(2) they are combinable with each other, that is:
   (a) they are combinable, that is, $combinable =$ "*true*", and
   (b) they have the same implementation, as represented by the attribute *impl* being associated to the same variable in L.

The right-hand side of the rule describes the result of a combination. It consists of deploying a new operator whose *impl* is determined by the *combImpl* attribute of the combined operators and whose parameters are calculated using the function *combParam* applied to $o_1(params)$ and $o_2(params)$. The rule morphism is not injective and associates both o_1 and o_2 with o_c. As a result, the inputs of o_1 and outputs of o_2 are exactly those of o_c.

The result of applying this rule to query q_1 from Figure 8(a) is shown in Figure 14.

7. AGECEP-BASED AUTONOMIC MANAGER

This section discusses how existing approaches can be coupled with *AGeCEP* to tackle the whole MAPE-K loop and thus implement a complete autonomic manager. The focus is on design aspects that must be taken into consideration when using *AGeCEP*.

The presented design is primarily based on FRAMESELF [Alaya and Monteil 2015], an autonomic framework that aims to enable implementation of autonomic managers that rely on the MAPE-K loop. FRAMESELF provides meta-models and mechanisms for implementing inference rules and communication between modules.

7.1. Runtime Environment Representation

Modelling the runtime environment is an important aspect of an autonomic CEP manager and is mostly determined by the operator placement strategy used by the system. Previous research has traditionally represented queries and the runtime environment as graphs [Ahmad and Çetintemel 2004; Lakshmanan et al. 2008]. In this research, a similar approach has been used: *AGeCEP* queries are extended with attributes that are relevant for placement decisions, and the runtime environment is also modelled as an (undirected, potentially cyclic) attributed graph.

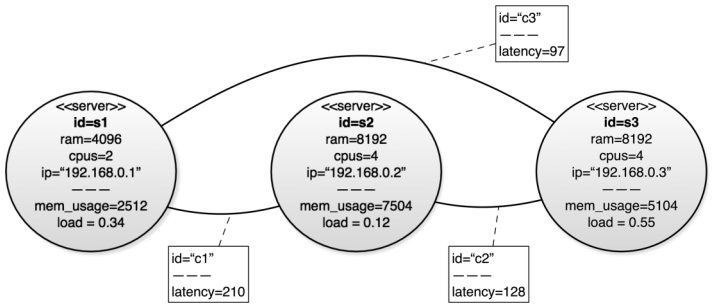


Fig. 15. Runtime environment representation.

Table III. Monitored Events

Type	Event	Description
Runtime	$QueueSize(o, n)$	Queue from operator o has size n
	$CPULoad(l, s)$	CPU load from server s has value l
User	$NewQuery(q)$	New query q was created
	$NewQueries(Q)$	Set of queries Q was created
Manager	$Duplicated(o, q)$	Operator o from query q was duplicated

Figure 15 shows an attributed graph that represents a runtime environment composed of three servers. In the graph, vertices and edges represent computational resources and logical connections between them, respectively. The vertices contain attributes that model server characteristics and also include runtime information.

By using this environment representation and the *AGeCEP* query model, the placement of an operator into a server can be represented using two approaches: as an operator attribute whose value contains a unique server identifier or as an edge connecting the operator to the server. For the remainder of this article, the first approach is assumed. Operator placement procedures are further discussed in Section 8.

7.2. Autonomic Manager: Handling MAPE

The following subsections discuss how the *AGeCEP* formalism can be used by each module of the MAPE-K loop in an autonomic manager based on FRAMESELF.

7.2.1. Monitor: Receiving Events. To implement the *monitor* module, it is assumed that the runtime environment and user queries are instrumented to publish monitoring events to the autonomic manager. As an alternative, a specialized monitoring module can poll the system for monitoring data and forward them to the manager on behalf of system components. Moreover, it is expected that events representing user interaction with the system, such as creation of new queries, will also be made available.

Once the manager receives monitoring events, it updates the query and environment models that are stored in the KB and continues to execute the MAPE-K loop.

Table III shows common monitored events used by CEP systems. Note that instead of trying to enumerate all possible events, the table only includes events used by the self-management policies from Section 9.

This decision is aligned with *AGeCEP* extensible and generic nature: The monitoring data that CEP systems must provide are tightly linked to the placement procedures and self-management policies implemented by the autonomic manager. Therefore, defining a fixed set of monitoring events would restrict the scope of policies that can be implemented. Instead, *AGeCEP* allows policies to define the events they need. Analogously to

how rule mutators establish a contract that CEP systems must implement, the events required by a policy define a contract of events CEP systems must provide.

7.2.2. Monitor, Analyse, and Plan: Inference Rules. Inference rules are central to the *monitor*, *analyse*, and *plan* modules of the MAPE-K loop. In general, these rules are used to infer new information based on the KB and on freshly received information.

In the *monitor* module, the events received by the autonomic manager are processed by inference rules to infer *symptoms*. The *analysis* module uses these symptoms along with the *AGeCEP* query and environment models that are stored in the KB to generate *Request for Changes* (RFC). Finally, in the *plan* module, the RFCs and the KB models are used by another set of rules to create *Change Plans* (CP).

In the FRAMESELF framework, inference rules are implemented by inference engines such as Jess [2016] and Drools [JBoss 2016]. This research used Drools.

7.2.3. Execute. The *execute* module is in charge of carrying out the CPs that it receives. In an *AGeCEP*-based autonomic manager, a CP is simply a sequence of reconfiguration actions modelled by graph rewriting rules that are associated with a set of side-effect mutators. The *execute* module enforces each reconfiguration of the CP in two steps involving the application of (1) a graph rewriting rule to update the model and (2) its associated mutators to update the real system.

In practice, a mutator is an API call that must be implemented by the CEP system being managed. In cases where the CEP system already exists, API calls can be handled by wrappers encapsulating the legacy components [Hagimont et al. 2009].

Accordingly, for the feasibility study presented in Section 9, a minimal API has been defined as follows:

—$startOperator(o, m)$: deploys and starts an operator o in server m;
—$stopOperator(o)$: stops and deletes an operator o;
—$connect(o_1, o_2)$: creates a connection between operators o_1 and o_2;
—$disconnect(o_1, o_2)$: deletes a connection between operators o_1 and o_2;
—$redirect(o_1, o_2)$: redirects all o_1 input (output) streams to input (output) o_2;
—$migrate(o, s_1, s_2)$: migrates an operator o from server s_1 to server s_2.

8. FEASIBILITY: OPERATOR PLACEMENT

This section discusses placement in an *AGeCEP*-based autonomic manager. As mentioned in Section 3.4, operator placement procedures can be used to determine the global initial placement of all queries, to decide on the placement of a new query or of new operators, or to adjust the current placement dynamically. In particular, placement can be used by management policies that may require placement decisions (e.g., whenever a new operator is deployed).

8.1. General Principle

Independently of the goal and of the algorithm used, operator placement procedures can usually be described by a general framework composed of three steps:

(1) Metrics from the operators and servers are *collected* to build a snapshot of the current system status. In the case of dynamic placement adjustments, these metrics are also used to *trigger* procedure execution. For example, in Heinze et al. [2014], dynamic adjustment is triggered when an overloaded server is detected.
(2) Using the collected data as input, an algorithm is executed to find the new placement. Because the general operator placement problem is NP-hard [Lakshmanan et al. 2008], these algorithms are usually heuristics that aim to maximize or minimize a utility function estimated from the collected metrics. For instance, Pietzuch

et al. [2006] aimed to minimize network usage, whereas Xing et al. [2005] tried to maximize load correlation among servers.
(3) The results of the algorithm are applied. If a placement has been calculated for new queries or operators, then they are created in the appropriate servers. Conversely, if a *dynamic adjustment* is being performed, then the operators that have changed allocation are *migrated* to their new servers.

It is argued here that most operator placement procedures can be expressed using the *AGeCEP* formalism and integrated within an *AGeCEP*-based autonomic manager by adapting them as follows:

(1) The query and runtime environment representations are augmented with attributes corresponding to the monitored metrics. The metrics are collected using the same mechanisms as before and are sent to the autonomic manager, which updates the corresponding attributes upon receipt. In the case of dynamic adjustment procedures, a monitoring inference rule can also be created to start the placement based on the collected metrics.
(2) When a placement decision is required, the input data are obtained from the autonomic manager KB, and execution takes place in the same way.
(3) Finally, with the newly calculated placement information, the operators are deployed or migrated through rewriting rules that update the KB and invoke the API calls *startOperator* or *migrate* accordingly.

8.2. Examples

This section presents how two different placement procedures can be expressed using *AGeCEP*.

Borealis. In their work, Xing et al. [2005] presented heuristics for global and dynamic adjustment of placements with the goal of minimizing the end-to-end latency of queries. The general idea of the presented heuristics is that, given an operator o that needs to be placed, a server with a current workload that is not correlated with o's workload must be found. To calculate load correlation, the heuristics build a time series of each operator's load based on monitored data. A server load, in its turn, is defined as the sum of all its operators' loads.

To adapt these heuristics to *AGeCEP*, the load time series can be maintained as an attribute of operator vertices. Calculations performed by the heuristics require only this data. As a result of the algorithms, migration rewriting rules are executed for each operator that has been selected to move.

FUGU. Heinze et al. [2014] presented a dynamic adjustment placement procedure for the FUGU system. The general idea is to detect overloaded servers and to move operators from them to underloaded servers. The operators to be moved are selected based on the latency spikes that their migration will cause; operators with small spikes are moved first.

A server is detected as overloaded when its CPU utilization exceeds a threshold for x consecutive measurements. This detection can be easily implemented as a monitoring inference rule. To decide which operators are moved, the latency spike estimation procedure uses metrics such as the the operator load and state size, which can be stored as attributes of the corresponding operator vertices.

After the operators to be moved are selected, their destination is determined based on a heuristic analogous to the bin-packing problem, in which the server's available CPU capacity constitutes the bins and the operators' loads are the items weight. Once again, these data are readily available in the KB. Finally, the resulting migrations are enforced with the aid of rewriting rules.

ALGORITHM 1: *CombineAll*(q) Action, Combination of all Combinable Operator Sequences in q.

while *exists a homomorphism* $h : L_{comb} \rightarrow q$ **do**
  apply P_{comb} rule to q w.r.t. h;
end

9. FEASIBILITY: DEFINITIONS OF SELF-MANAGEMENT POLICIES

This section introduces a selection of self-management policies defined using *AGeCEP* and the models and techniques presented in Section 7. Additional policies are introduced in Appendix A. For the sake of readability, algorithms and inference rules are presented as informal descriptions or pseudocode. Appendix B contains the corresponding inference rules defined in Drools Rule Language [JBoss 2016].

9.1. Operator Combination

9.1.1. Description. The *operator combination* (*Comb*) policy is directly related to the "combinable" criterion of the classification. This policy is used to combine any sequences of n combinable operators $o_1, \ldots, o_n$ into a single operator o_c that has the same effect on the event stream as the combined sequence. Figure 7 shows an example of such a sequence and the result of the policy application.

This policy is mostly applied in the single-query optimization step and reduces the number of operators constituting the query, which brings savings in memory consumption and can also improve query latency and throughput.

9.1.2. Realization using the MAPE-K Loop.

Monitor. A new query q is submitted by the user, which is signalled by a $NewQuery(q)$ event. The event is simply forwarded as a $NewQuery(q)$ symptom to the analysis module.

Analysis. When a $NewQuery(q)$ symptom is received, the analysis module checks whether at least one pair of successive operators (o_1, o_2) are combinable. This is equivalent to checking whether there is a homomorphism $h : L_{comb} \rightarrow q$, where L_{comb} is the left-hand side of the graph rewriting rule shown in Figure 13. If such a homomorphism exists, then a $Combine(q)$ request for change (RFC) is sent to the plan module (Algorithm B.1).

Plan. Upon receipt of a $Combine(q)$ RFC, the $CombineAll(q)$ action is inserted into the change plan.

Execute. The execution of the $CombineAll(q)$ action is described by Algorithm 1. The P_{comb} rule is specified in Figure 13; its applicability and effect have been described in Section 6.2.5. In the single query optimization step, this rule operates at the model level only and therefore has no associated mutator.

9.2. Operator Duplication

9.2.1. Description. Operator duplication (*Dupl*) is a policy used to parallelize an operator execution by creating multiple instances of the operator and splitting input events among these instances. It can be used to achieve load balancing by distributing operator instances over several servers to improve query throughput or both. Generally, query throughput can be improved when the following conditions are satisfied:

(1) the operator processing rate is lower than the incoming event rate;
(2) additional resources exist to which extra instances of the operator can be allocated.

However, this policy may also lead to an increase in resource consumption due to the deployment of supplementary operators.

9.2.2. Realization Using the MAPE-K Loop.

Monitor. The autonomic manager receives periodic events containing operators performance metrics. Based on these data, a monitoring rule might identify an operator as a bottleneck if it is not outputting events as fast as it is receiving them. Whenever a bottleneck is pinpointed, a *Bottleneck*(o) symptom is sent to the analysis module (Algorithm B.2).

Analysis. Whenever a *Bottleneck*(o) symptom is received, the analysis module checks whether o is duplicable. If this condition is satisfied, then a *Duplicate*(q, o) RFC is sent to the plan module (Algorithm B.3).

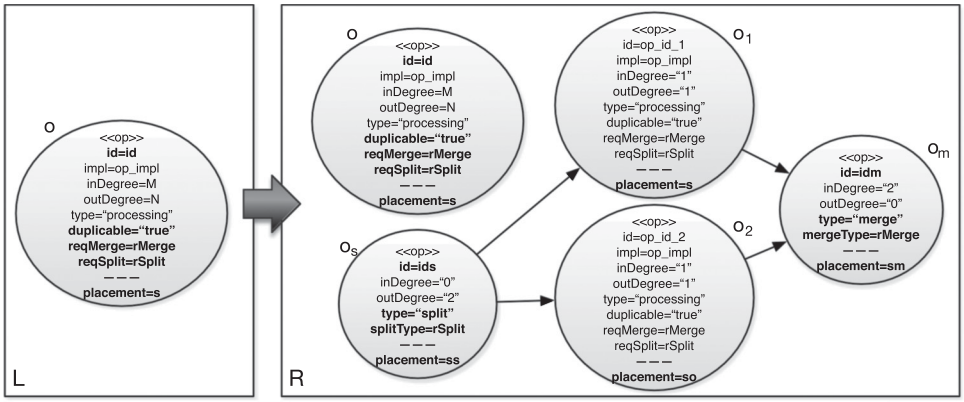
Plan. When planning a *Duplicate*(q, o) operation, two scenarios must be considered (Algorithm B.4):

—o has not yet been parallelized, which implies that duplication requires deployment of a new instance of o and of the appropriate split and merge operators. In this case, the rule invokes the placement procedure to determine the place of these new operators and inserts an *InitialDuplication*(q, o, s_s, s_o, s_m) action into the change plan, where s_s, s_o, and s_m are the placements determined for the split, the new instance of o, and the merge operator.
—o has already been parallelized, or, more precisely, adequate split and merge operators have already been deployed. In this case, duplication consists only of adding a new operator instance. The rule determines the placement s_o of this new operator and adds the *AdditionalDuplication*(q, o, s_o) action to the change plan.

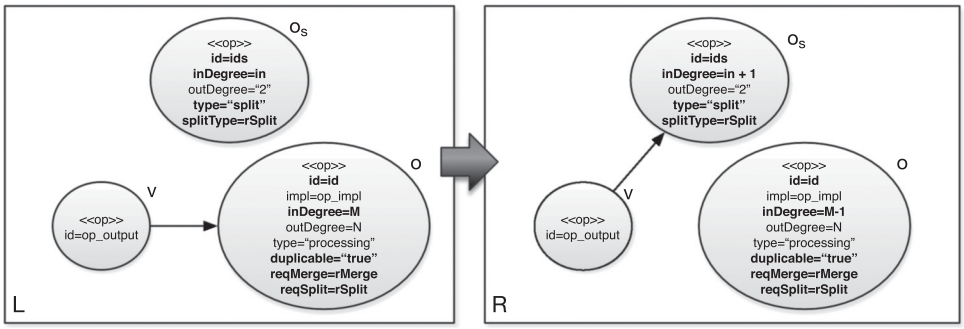
Execute. Depending on the change plan received, two different actions may be executed: an *Initial Duplication*(q, o, s_s, s_o, s_m) or an *AdditionaDuplication*(q, o, s_o) action.

The *InitialDuplication*(q, o, s_s, s_o, s_m) action is described by Algorithm 2. Rules $P^{init_1}_{dupl}(id, s_s, s_o, s_m)$, $P^{init_2}_{dupl}(id, id_s)$, and $P^{init_3}_{dupl}(id, id_m)$ used by the algorithm are depicted in Figure 16. Some attributes of o, o_1, and o_2 are not shown because of space constraints, but the copies o_1 and o_2 have the same values as o for all attributes described in Table I, except for *id*, *inDegree*, and *outDegree*. In addition, note that the morphism characterizing $P^{init_3}_{dupl}(id, id_m)$ is not trivial and not injective. The algorithm consists of three steps:

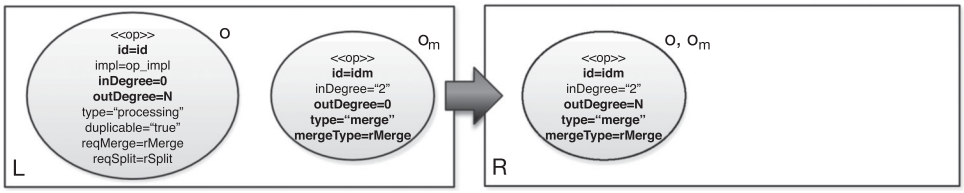
(1) $P^{init_1}_{dupl}(id, s_s, s_o, s_m)$ is applied to q with respect to the unique possible homomorphism. Hereafter, when a single morphism is acceptable, it is omitted. Application of this rule creates the respective split o_s and merge o_m and two instances of operator o. Note that operators o_s, o_2, and o_m are created in servers s_s, s_o, and s_m as indicated by the "placement" attribute value. In addition, operator o_1 is created on the same server as the original operator o. The mutators executed for this rule are API calls to *startOperator*(o_s, s_s), *startOperator*(o_1, s), *startOperator*(o_2, s_o), and *startOperator*(o_m, s_m).
(2) $P^{init_2}_{dupl}(id, id_s)$ is repeatedly applied as long as possible to redirect all input edges previously connected to o towards o_s. This rule is associated with the mutators *disconnect*(v, o) and *connect*(v, o_s).
(3) The $P^{init_3}_{dupl}(id, id_m)$ rule merges o and o_m. As a result, all output edges previously connected to o are redirected to o_m, and the original operator o is deleted. These changes are performed on the system by the mutators *redirect*(o, o_m) and *stopOperator*(o).



(a) Step 1: $P^{init_1}_{dupl}(id, s_s, s_o, s_m)$, operator duplication.



(b) Step 2: $P^{init_2}_{dupl}(id, id_s)$, redirect input edge.



(c) Step 3: $P^{init_3}_{dupl}(id, id_m)$, redirect output edges.

Fig. 16. $P^{init}_{dupl}(id)$: Initial operator duplication.

ALGORITHM 2: *InitialDuplication*(q, o, s_s, s_o, s_m) Action, Execution of an Initial Duplication.

$id \leftarrow o(id)$;
apply $P^{init_1}_{dupl}(id, s_s, s_o, s_m)$ to q;
while *exists a homomorphism* $h : L^{init_2}_{dupl}(id, id_s) \rightarrow q$ **do**
  apply $P^{init_2}_{dupl}(id, id_s)$ to q w.r.t. h;
end
apply $P^{init_3}_{dupl}(id, id_m)$ to q;

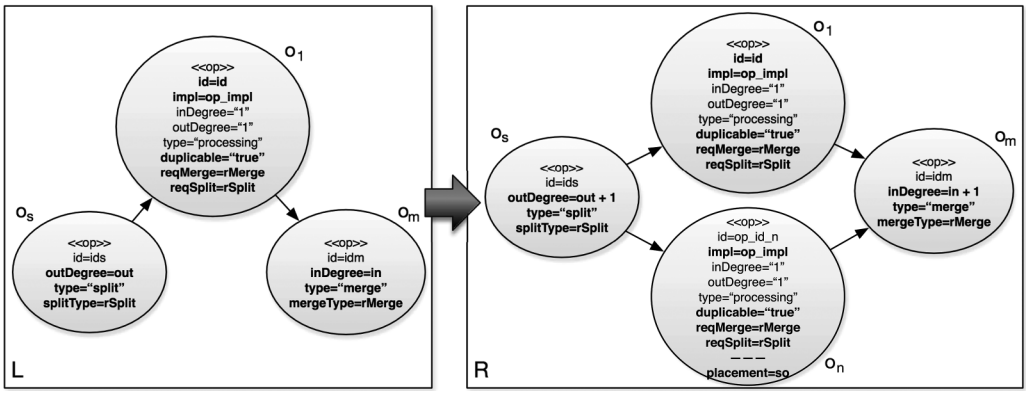


Fig. 17. $P^{add}_{dupl}(id, s_o)$: Addition of an operator instance.

The $AdditionalDuplication(q, o, s_o)$ action, on the other hand, is accomplished by applying the $P^{add}_{dupl}(id, s_o)$ rule shown in Figure 17. It consists of the simple addition of a new instance of o connected to the already existing split o_s and merge o_m. This rule is associated with the mutators $startOperator(o_n, s_o)$, $connect(o_s, o_n)$, and $connect(o_n, o_m)$.

9.3. Removal of an Unnecessary Merge/Split

9.3.1. Description. The *removal of an unnecessary merge/split* (*RemMS*) policy describes the removal of a particular pattern of a merge operator followed by a split whose impact on the event streams is null. Such a pattern has null impact if:

—the merge does not modify the streams that it processes. According to the *AGeCEP* classification, union is the only merge type satisfying this condition.
—the split operator does not strengthen the stream specificities, or, in other words, the output streams of the split have the same or fewer constraints than the input streams of the merge.

The following discussion considers only the case where the number of input streams in the merge is equal to the number of output streams in the split.

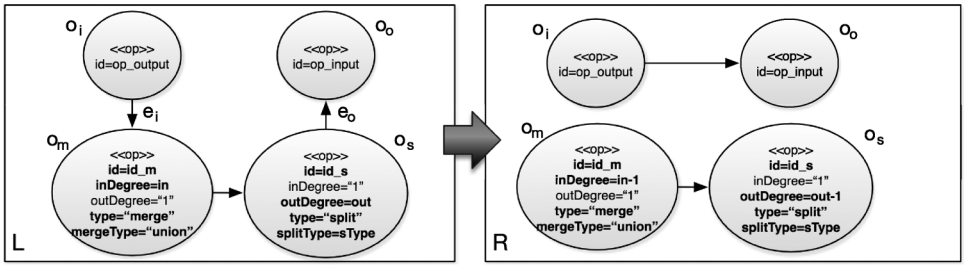
The impact of this policy is positive on both system performance and on resource consumption because unnecessary operators are suppressed. Hence, this policy is used whenever possible.

9.3.2. Realization using the MAPE-K Loop.

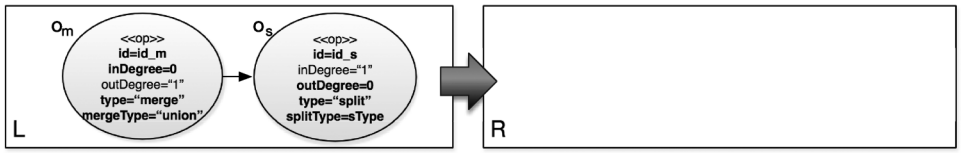
Monitor. A new query q is submitted by the user, resulting in the $NewQuery(q)$ symptom being sent to the analysis module. In addition, whenever an operator o from query q is duplicated by the *Dupl* policy, the $Duplicated(q, o)$ symptom is also sent to analysis.

Analysis. Whenever a $NewQuery(q)$ or a $Duplicated(q, o)$ symptom is received, the analysis module checks for the existence of an unnecessary merge/split sequence as follows:

(1) There is a homomorphism $h : L^{byp}_{rem}(id_m, id_s, e_i, e_o) \rightarrow q$, where $L^{byp}_{rem}(id_m, id_s, e_i, e_o)$ is the left-hand side of the $P^{byp}_{rem}(id_m, id_s, e_i, e_o)$ rule depicted in Figure 18(a);
(2) The split operator does not strengthen the stream specificities. This condition cannot be checked for *"custom"* splits. For the other cases, let $o_m(pred)$ be the set of all incoming edges of o_m and $o_s(succ)$ be the set of all outgoing edges of o_s. This



(a) Step 1: Bypassing the merge/split $P^{byp}_{rem}(id_m, id_s, e_i, e_o)$.



(b) Step 2: Suppressing the merge/split $P^{sup}_{rem}(id_m, id_s)$.

Fig. 18. $P_{rem}(id_m, id_s)$: removal of a unnecessary merge/split.

ALGORITHM 3: *RemoveMergeSplit*(q, o_m, o_s, f) Action, Execution of a Removal.

$id_m \leftarrow o_m(id)$;
$id_s \leftarrow o_s(id)$;
forall the *edge* e_i *whose tail is* o_m **do**
  apply $P^{byp}_{rem}(id_m, id_s, e_i, f(e_i))$ to q;
end
apply $P^{sup}_{rem}(id_m, id_s)$ to q;

condition is met if there is a bijective function $f : o_m(pred) \rightarrow o_s(succ)$ such that for all $e_i \in o_m(pred)$, with $e_o = f(e_i)$, one of the following conditions is satisfied:
—$o_s(splitType) =$ "*query*" and $e_i(queries) = e_o(queries)$;
—$o_s(splitType) =$ "*attribute*" and $e_i(attrs) = e_o(attrs)$;
—$o_s(splitType) =$ "*random*."

For each pair (o_m, o_s), a *RemoveMergeSplit*(q, o_m, o_s, f) RFC is created using an arbitrarily selected function f that satisfies condition 2 (Algorithm B.5).

Plan. On receiving a *RemoveMergeSplit*(q, o_m, o_s, f) RFC, the action *RemoveMergeSplit*(q, o_m, o_s, f) is inserted into the change plan.

Execute. Algorithm 3 details how to remove an unnecessary merge/split pattern. The algorithm is executed in two steps. First, the unnecessary merge and split are bypassed using the $P^{byp}_{rem}(id_m, id_s, e_i, e_o)$ rewriting rule, as defined in Figure 18(a). This rule is repeated for all pairs of edges (e_i, e_o) returned by the function f found in the analysis step. In the second step, the bypassed merge/split is removed using the $P^{sup}_{rem}(id_m, id_s)$ rule (Figure 18(b)).

Note this policy may be executed in a running query. In such a case, each application of rule $P^{byp}_{rem}(id_m, id_s, e_i, e_o)$ triggers the mutators $disconnect(o_i, o_m)$, $disconnect(o_s, o_o)$, and $connect(o_i, o_o)$, whereas the rule $P^{sup}_{rem}(id_m, id_s)$ triggers $stopOperator(o_m)$ and $stopOperator(o_s)$.

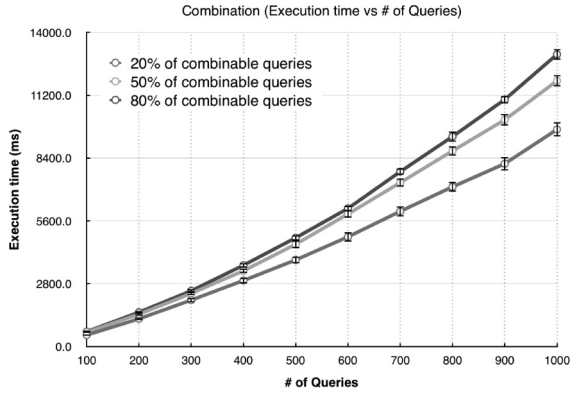


Fig. 19. Combination policy execution time.

10. VIABILITY: PERFORMANCE EVALUATION

This section discusses the viability of *AGeCEP* as a formal foundation for developing generic CEP applications and algorithms. The analysis focuses on the time required to transform CEP queries using both simple and complex graph rewriting rules.

In the following, the AGG tool [Taentzer 2004] was used to define and apply the graph rewriting rules. The experiments were conducted on a server with two six-core processors (Intel Xeon E5-2630, 2.6GHz) and 96GB of RAM. The server was running Ubuntu Linux 14.04 and Java 1.7.0_75.

10.1. Simple Policy

The first experiment verified the execution time and scalability of the actions executed by the *Comb* policy (Section 9.1). This is a simple reconfiguration that consists of a single rewriting rule in which only two nodes are matched.

The total number of queries to which the rule was applied varied from 100 to 1,000, and, for each number, three query compositions were tested. In the first composition, 20% of the queries were clones of query q_1 (Figure 8(a)), and 80% were clones of q_2 (Figure 8(b)). In the second and third compositions, query q_1 represented 50% and 80% of the total queries, respectively. Note that only query q_1 has a sequence of combinable filters f_1 and f_2.

The graph in Figure 19 shows the average execution time of 30 runs along with the 99% confidence interval. The growth in execution time is almost linear. For all three compositions, 100 queries were processed in less than 1s and 1,000 queries in less than 14s. For the 80% composition, this is equivalent to rewriting 800 queries according to the operator combination policy.

10.2. Complex Policy

This experiment verified the performance and scalability of complex sequences of reconfiguration actions. To perform this experiment, the analysis was divided into two parts. First, the execution time for applying the *Dupl* policy (Section 9.2) was assessed. Then, the execution time for applying *Dupl* followed by *RemMS* (Section 9.3) was analysed.

Both parts were executed using the same numbers of queries and the same query compositions as in the previous experiment. In this case, however, the duplication was applied only to the operator j_1 belonging to query q_2 clones (Figure 8(b)). Note that after j_1 duplication, the newly created merge forms a void sequence with the $fSplit$ operator. Applying *RemMS* therefore causes this sequence to be removed, resulting in the query depicted in Figure 20.

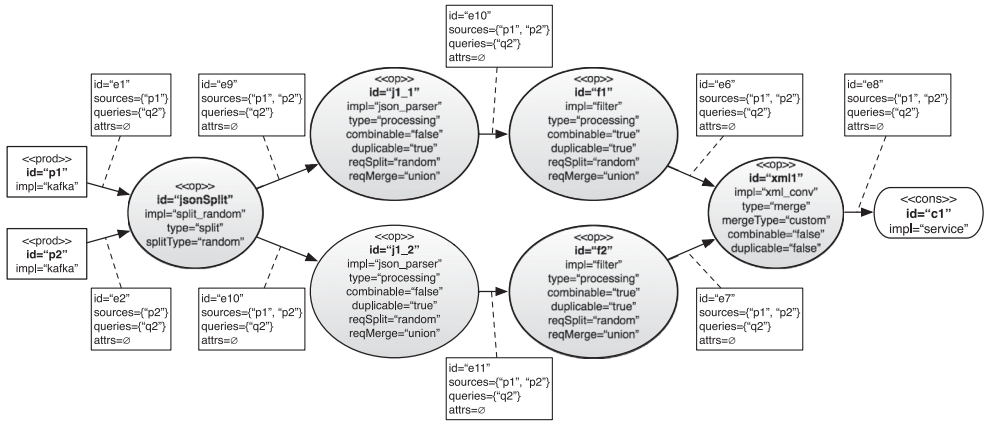
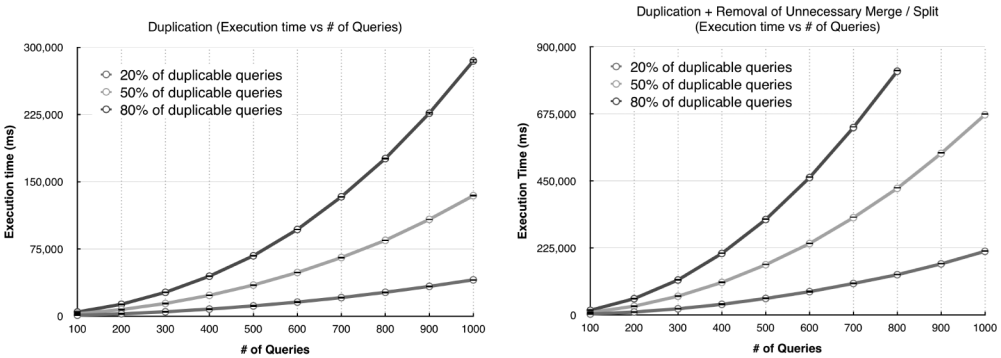


Fig. 20. Query q_2—optimized version.



(a) $Dupl$ execution time (b) $Dupl$ followed by $RemMS$ execution time

Fig. 21. *Dupl* and *RemMS* execution times.

Figure 21(a) depicts the execution time of *Dupl* as a function of the number of queries for all three compositions. For each duplication, four reconfiguration rules were applied: $P_{dupl}^{init_1}$ once to create the two instances of j_1 connected to a new split and merge, $P_{dupl}^{init_2}$ twice to redirect the j_1 inputs (p_1 and p_2) to the new split, and $P_{dupl}^{init_3}$ to connect the new merge to the j_1 successor ($fSplit$). For the 20% composition, 1,000 queries were processed in less than 40s, which still is within reasonable time bounds.

The execution time to apply *Dupl* followed by *RemMS* is shown in Figure 21(b). To execute the *RemMS* policy, three more rewriting rules were applied: P_{rem}^{byp} twice to connect each instance of j_1 to an instance of f_{12} and P_{rem}^{sup} once to remove the redundant merge and split. Therefore, *Dupl* followed by *RemMS* requires the application of seven rewriting rules in total. The graph clearly shows an exponential growth that is especially pronounced in the 50% and 80% scenarios. In these scenarios, rewriting all queries may take minutes. Indeed, for the 80% scenario, there are no data points for 900 and 1,000 queries because the execution time exceeded the established timeout of 15min.

It is important to discuss these results under proper assumptions about how these rules will be applied in practice. Finding homomorphisms in graphs is a well-known NP-complete problem [Garey and Johnson 1979]. Nevertheless, most of the time, these

rules will be applied to a much smaller number of queries. For example, SQO policies are executed in response to new queries, and, therefore, only they need to be analysed right away. Similarly, most runtime management policies are applied only to the small subset of all running queries that need to be rewritten. For instance, as described in Section 9.2, duplication is performed only after a bottleneck has been pinpointed. The extreme cases described in this section were investigated for theoretical purposes and for completeness of analysis.

11. CONCLUSIONS

This work has introduced and investigated the feasibility and viability of the *AGeCEP* formalism. This formalism was developed to overcome the fragmentation of current CEP/SP solutions. *AGeCEP* proposes a language-agnostic abstraction of CEP queries and a formalism to manipulate them, enabling definition of self-management policies that can be integrated into potentially any CEP system.

AGeCEP represents CEP queries using *attributed directed acyclic graphs* (ADAG) whose vertices and edges have a standardized set of attributes that encode information relevant to self-management. These standard attributes are based on a novel classification of CEP operators that focus on their reconfiguration capabilities and also constitutes a major contribution of this research.

Self-management policies may ultimately trigger the execution of query reconfigurations. *AGeCEP* formalizes structural reconfigurations of queries using graph rewriting rules. Moreover, *AGeCEP* graph rewriting rules are enriched with mutators, which associate API calls with the effects of a rule and guarantees that model changes are also applied in the real system.

To demonstrate the feasibility of *AGeCEP* for specification and enforcement of self-management policies, this research introduced the design of an *autonomic manager* based on *AGeCEP* and a selection of policies built on this design. Furthermore, it also presented a generic procedure to adapt operator placement procedures to *AGeCEP*. Finally, this research investigated the viability of *AGeCEP* by executing performance measurements of query reconfigurations. By considering both expressiveness and performance, these results suggest that *AGeCEP* can be effectively used to develop algorithms for application and integration into diverse modern CEP systems.

It is important to emphasize that preliminary versions of *AGeCEP* have already been used in various studies. For instance, *CEPSim* [Higashino et al. 2016] is a simulator of cloud-based CEP systems that uses an *AGeCEP*-based formalism to represent queries. Furthermore, an autonomic manager for CEP based on *AGeCEP* ideas has been introduced by Higashino et al. [2014].

As future work, more self-management policies will be developed, especially focusing on the operator placement and runtime management steps of the query lifecycle. Moreover, these policies will be integrated into and tested in a real CEP system.

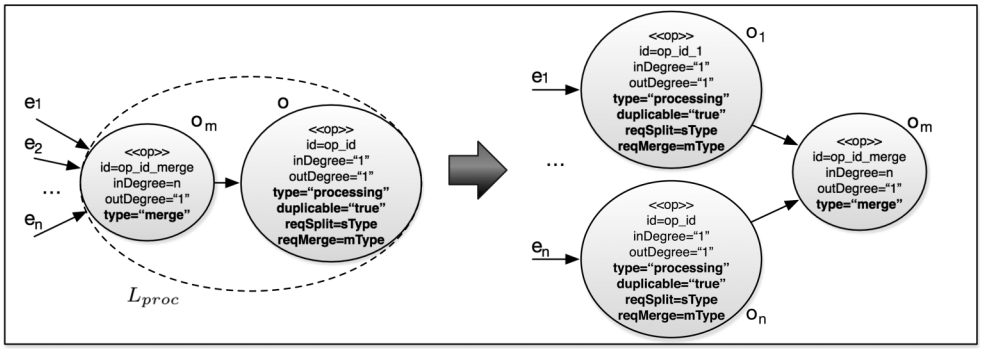
APPENDIXES

A. ADDITIONAL SELF-MANAGEMENT POLICIES

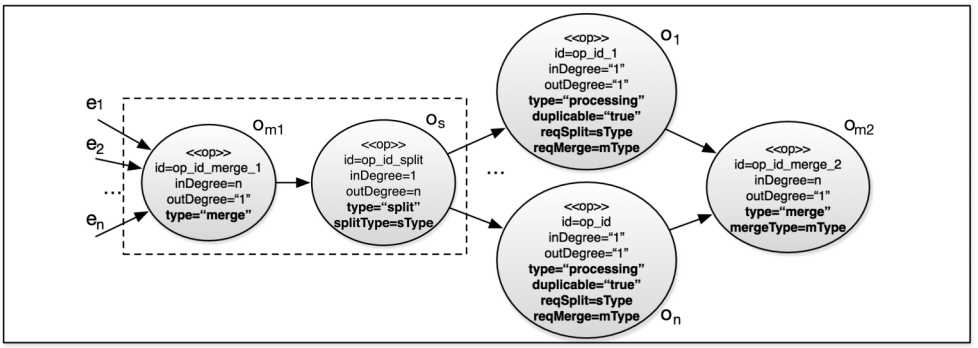
This appendix presents two additional self-management policies defined using *AGeCEP* and the models and techniques presented in Section 7.

A.1. Processing Sub-Streams (ProcSubS)

A.1.1. Description. The *processing sub-streams* (*ProcSubS*) policy transposes to CEP the strategy of dividing a problem into the solution of several sub-problems. The policy considers an operator o processing the result of a merge o_m, as illustrated on the left side of Figure 22(a). Ideally, this operation should be parallelized and conducted on



(a) Initial and final situation.



(b) Intermediate step of the processing sub-streams policy.

Fig. 22. Processing sub-streams policy.

each of the merged streams. In rough terms, o and o_m should be "swapped," as shown on the right side of Figure 22(a).

This transformation is equivalent to multiple duplications of o followed by removal of the initial merge o_m and the new split introduced by the duplication. Figure 22(b) illustrates the query after n duplication steps, where n is the number of o_m input streams. The merge and split sequence highlighted in the figure can be removed by the *RemMS* policy, resulting in the desired final situation. The policy realization described in the next section leverages this fact and reuses the *Dupl* and *RemMS* policies described in Sections 9.2 and 9.3.

This policy can be applied under various circumstances:

—If there are enough resources to process o instances in parallel, then this policy can be used to improve query throughput and latency. This effect is even more pronounced when $o(selectivity) < 1$. In this case, the policy can be applied in the SQO step because the number of events processed by o_m may be significantly reduced.
—If o processes groups of events and $o(complexity)$ is greater than linear, then this policy reduces query CPU consumption;
—In general, the policy can be used to split the processing load of o with other servers and cores.

A.1.2. Realization using the MAPE-K Loop.

Monitor. This policy can be triggered at runtime whenever a bottleneck of operator o is detected, which results in the *Bottleneck*(o) symptom being sent to analysis.

Analysis. When a *Bottleneck*(o) is received, the analysis rule first checks if the policy can indeed improve query throughput. For instance, it can verify whether the selectivity of operator o is less than 1. If this is true, then the rule searches for a sequence formed by a merge o_m and a duplicable operator o by checking for a homomorphism $h : L_{proc} \rightarrow q$, where the graph L_{proc} is indicated in Figure 22(a). Finally, the rule verifies whether the found sequence o_m and o satisfies the following conditions:

(1) The merge has more than one input stream,

$$o_m(inDegree) > 1.$$

(2) The merge o_m followed by the split introduced by the duplication of o produces a removable pattern, which translates to:
   (a) The merge type of o_m is "union",

$$o_m(mergeType) = \text{“}union\text{”}.$$

   (b) The split o_s introduced during duplication of o does not strengthen stream specificities. In the *RemMS* policy, it was shown that a "random" split is always valid, whereas a "custom" split cannot be considered. In the other cases:
   —if $o(reqSplit) = \text{“}query,\text{”}$ then for each stream e entering o_m, $|e(queries)| = 1$;
   —if $o(reqSplit) = \text{“}attribute,\text{”}$ then for each stream e entering o_m, $e(attrs) = o_s(param)$, meaning the streams are already grouped with respect to the same attributes that the split discriminates.

If these conditions are satisfied, then the processing sub-stream policy can be applied. To achieve this, the rule inserts n requests for duplication of operator o (Algorithm B.6). Because the *RemMS* policy is already executed after each duplication, there is no need to request it explicitly. In addition, note that even though *RemMS* is triggered n times, only the last time succeeds because the others cannot find the mapping function f required by the policy.

Plan. There is no specific plan for this policy.

Execute. There is no specific execution for this policy.

A.2. Predicate Indexing

A.2.1. Description. This policy (*PredIndex*) implements the *predicate indexing* MQO technique introduced by Madden et al. [2002]. The technique detects when two or more filters process the same input stream and have predicates over the same attributes and replaces both occurrences with a single one. In this case, the resulting filter has special data structures that enable it to evaluate multiple (range) predicates more efficiently than evaluating each predicate independently. This is an example of "*source*" sharing, as explained in Section 5.2.

A.2.2. Realization Using the MAPE-K Loop.

Monitor. A set of queries Q is submitted by the user, resulting in a *NewQueries*(Q) symptom being set to the analysis module.

Analysis. The analysis module checks whether a pair of filters can be shared by searching for a homomorphism $h : L_{pred} \rightarrow Q$, where L_{pred} is the left-hand side of the graph rewriting rule in Figure 23. If a homomorphism exists, then the module also checks whether the predicates range over the same attributes. If so, then a *PredicateIndex*(Q) RFC is sent to the plan module (Algorithm B.7).

Plan. On receipt of the *PredicateIndex*(Q) RFC, the *PredicateIndexAll*(Q) action is inserted into the change plan.

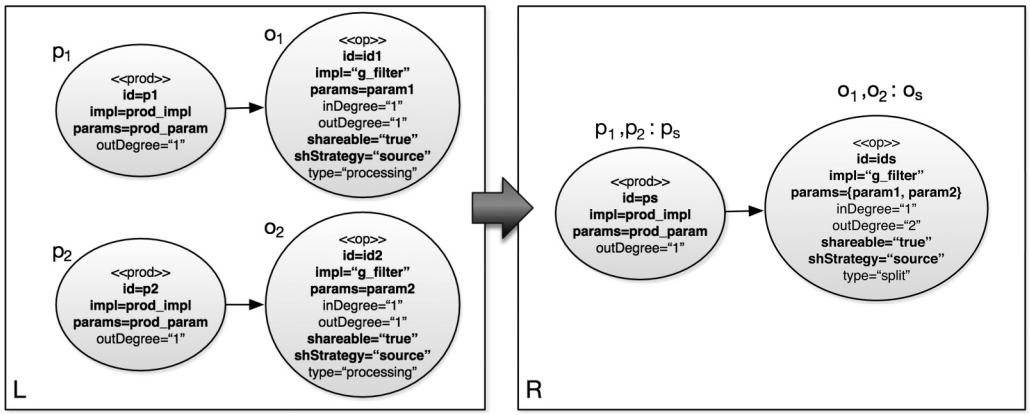


Fig. 23. P_{pred}: Predicate indexing rewriting rule.

Execute. Execution of this policy is equivalent to repeatedly applying rule P_{pred}, described in Figure 23. Note that the two producer vertices p_1 and p_2 shown in L actually represent the same input source, as they are associated with the same implementation and parameters. The resulting grouped filter is logically equivalent to the execution of both predicates. This policy is applied only at the query model level, and therefore there is no associated mutator.

B. INFERENCE RULES

This appendix presents the inference rules used in the MAPE loop by the self-management policies from Section 9 and Appendix A. The rules are written in Drools Rule Language, a declarative language based on the event-condition-action paradigm.

ALGORITHM B.1: *Operator Combination—Analysis Inference Rule*.

```
rule "comb-analysis"
    when
        $s: Symptom(category == "NewQuery", $query: query)
        $rule: Rule(name == "P_comb", $lhs : left)
        eval (Util.homomorphism($lhs, $query).value)
    then
        insert(new Rfc("Combine", $query))
end
```

ALGORITHM B.2: *Operator Duplication—Monitoring Inference Rule*. A Bottleneck Is Detected if an Operator Queue Size Is Trending Up Considering the Last Five Monitoring Events.

```
rule "dupl-monitor"
    when
        $op: Operator($id: id)
        Trend(this == Trend.UP) from accumulate(
            Event(source == $id, category == "queueSize", $value: value) over window:length(5),
            trend( $value ))
    then
        insert(new Symptom("Bottleneck", $op))
end
```

ALGORITHM B.3: *Operator Duplication—Analysis Inference Rule*.

```
rule "dupl-analysis"
    when
        $s:  Symptom(category == "Bottleneck")
        $op: Operator(duplicable == true) from $s.operator
    then
        insert(new Rfc("Duplicate", $op.query, $op))
        delete($s)
end
```

ALGORITHM B.4: *Operator Duplication—Plan Inference Rule*. The Left-Hand Side of Rule $P^{add}_{dupl}(id, s_o)$ (Figure 17) Is Used to Verify if the Appropriate Merge and Split Operators Are Already in Place.

```
rule "dupl-plan"
    when
        $rfc: Rfc(category == "Duplicate", $op: operator, $query: operator.query)
        Rule(name == "P_add_dupl", $lhs: left)
    then
        params = ["id": $op.id];
        h      = Util.homomorphism($lhs, $query, params);

        if (!h.getValue()) {
            p1 = Util.placementForInitDupl($query, $op);
            insert(new Action("InitialDuplication", $query, $op, p1["split"],
                p1["op"], p1["merge"] ));
        } else {
            p2 = Util.placementForAddDupl($query, $op);
            insert(new Action("AdditionalDuplication", $query, $op, p2["op"]));
        }
        delete($rfc);
end
```

ALGORITHM B.5: *Removal of an Unnecessary Merge/Split—Analysis Inference Rule*. The Method *hasMapping* Searches for the Bijective Function f Defined in Section 9.3.

```
rule "unnec-ms-analysis"
    when
         $s:  Symptom(category == "NewQuery" || category == "Duplicated", $query: query)
         Rule(name == "P_rem_byp", $lhs: left)
    then
        Homomorphism h = Util.homomorphism($lhs, $query)
        if (h.value) {
            foreach (Graph g : h.results) {
                merge = g.byType("merge")[0];
                split = g.byType("split")[0];
                Function f     = Util.hasMapping(merge, split);
                if (f != null) {
                    insert(new Rfc("RemoveMergeSplit", $query, merge, split, f))
                }
            }
        }
end
```

ALGORITHM B.6: *Processing Sub-Streams—Analysis Inference Rules*. It Finds a Sequence of a Merge and a Duplicable Operator by Searching for an Homomorphism $L_{proc} \rightarrow q$. The Method *checkSubStream* Verifies the Sub-Stream Conditions Defined in Appendix A.1.

```
declare ProcSubS
    query: Query
    merge: Operator
    op: Operator
end

rule "proc-subs-bottleneck-analysis"
    when
        $s:  Symptom(category == "Bottleneck", $query: query)
        $op: Operator(selectivity < 1.0) from $s.operator
        $rule: Rule(name == "L_proc", $lhs: left)
    then
        params = ["id": $op.id ]
        h = Util.homomorphism($lhs, $query, params)
        for (Graph g : h.results) {
            insert(new ProcSubS($query, g.byType("merge")[0], g.byId($op.id)))
        }
end

rule "proc-subs-conditions-analysis"
    when
        $ps: ProcSubS()
        $query: Query() from $ps.query
        $op: Operator(duplicable == true) from $ps.op
        $merge: Operator(mergeType == "union", inDegree > 1) from $ps.merge
        eval(Util.checkProcSubStream($merge, $op))
    then
        for (int i = 0; i < $merge.inDegree; i++) {
            insert(new Rfc("Duplicate", $query, $op))
        }
end
```

ALGORITHM B.7: *Predicate Indexing—Analysis Inference Rules*.

```
rule "mqo-analysis-init"
    when
        Rule(name == "L_pred", $lhs: left)
        $s: NewQueries($queries: queries)
    then
        Query q = $queries[0]
        for (int i = 1; i < $queries.size(); i++) {
            q = q.add((Query) $queries[i])
        }
        if (Util.homomorphism($lhs, q).value) {
            insert(new Rfc("PredicateIndex", q))
        }
        delete($s)
end
```

REFERENCES

Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik.

2005. The design of the Borealis stream processing engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*. 277–289.

Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: A new model and architecture for data stream management. *Int. J. Very Large Data Bases* 12, 2 (Aug. 2003), 120–139. DOI:http://dx.doi.org/10.1007/s00778-003-0095-z

Yanif Ahmad and Ugur Çetintemel. 2004. Network-aware query processing for stream-based applications. In *Proceedings of the 13th International Conference on Very Large Data Bases–Volume 30 (VLDB'04)*. VLDB Endowment, 456–467.

Mahdi Ben Alaya and Thierry Monteil. 2015. FRAMESELF: An ontology-based framework for the self-management of machine-to-machine systems. *Concurr. Comput.: Pract. Exper.* 27, 6 (2015), 1412–1426. DOI:http://dx.doi.org/10.1002/cpe.3168

Amazon. 2015. Amazon Kinesis. Retrieved from http://aws.amazon.com/kinesis.

Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. 2004. *STREAM: The Stanford Data Stream Management System*. Technical Report 2004-20. Stanford InfoLab.

Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2005. The CQL continuous query language: Semantic foundations and query execution. *VLDB J.* 15, 2 (July 2005), 121–142. DOI:http://dx.doi.org/10.1007/s00778-004-0147-z

Steve Awodey. 2006. *Category Theory*. Oxford Logic Guides, Vol. 49. Oxford University Press.

Damien Borgetto, Rodrigue Chakode, Benjamin Depardon, Cédric Eichler, Jean-Marie Garcia, Hassen Hbaieb, Thierry Monteil, Elie Pelorce, Anouar Rachdi, A. Al Sheikh, and Patricia Stolf. 2016. Hybrid approach for energy aware management of multi-cloud architecture integrating user machines. *J. Grid Comput.* 14, 1 (March 2016), 91–108.

Christian Y. A. Brenninkmeijer, Ixent Galpin, Alvaro A. A. Fernandes, and Norman W. Paton. 2008. A semantics for a query language over sensors, streams and relations. In *Sharing Data, Information and Knowledge SE-9*, Alex Gray, Keith Jeffery, and Jianhua Shao (Eds.). Lecture Notes in Computer Science, Vol. 5071. Springer Berlin Heidelberg, 87–99. DOI:http://dx.doi.org/10.1007/978-3-540-70504-8_9

Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Çetintemel, Ying Xing, and Stan Zdonik. 2003. Scalable distributed stream processing. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR'03)*. 257–268.

Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: A formally defined event specification language. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS'10)*. ACM, New York, NY, 50. DOI:http://dx.doi.org/10.1145/1827418.1827427

Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *Comput. Surv.* 44, 3 (June 2012), 1–62. DOI:http://dx.doi.org/10.1145/2187671.2187677

Gianpaolo Cugola, Alessandro Margara, Mauro Pezzè, and Matteo Pradella. 2015. Efficient analysis of event processing applications. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS'15)*. ACM, New York, NY, 10–21. DOI:http://dx.doi.org/10.1145/2675743.2771834

Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. 2007. Cayuga: A general purpose event monitoring system. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR'07)*. 412–422.

H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. 1997. In *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, Grzegorz Rozenberg (Ed.). World Scientific, Chapter Algebraic Approaches to Graph Transformation. Part II: Single Pushout Approach and Comparison with Double Pushout Approach, 247–312.

Cédric Eichler. 2015. *Modélisation Formelle de Systèmes Dynamiques Autonomes: Graphe, Réécriture et Grammaire*. Ph.D. Dissertation. Université Toulouse III.

Cédric Eichler, Ghada Gharbi, Nawal Guermouche, Thierry Monteil, and Patricia Stolf. 2013. Graph-based formalism for machine-to-machine self-managed communications. In *Proceedings of the 2013 IEEE 22nd International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*. 74–79.

Cédric Eichler, Thierry Monteil, Patricia Stolf, Luigi Alfredo Grieco, and Khalil Drira. 2016. Enhanced graph rewriting systems for complex software domains. *Softw. Syst. Model.* 15, 3 (July 2016), 685–705. DOI:http://dx.doi.org/10.1007/s10270-014-0433-1

M. R. Garey and D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

Google. 2015. Google Cloud Dataflow. Retrieved from http://cloud.google.com/dataflow/.

Katarina Grolinger, Michael Hayes, Wilson A. Higashino, Alexandra L'Heureux, David S. Allison, and Miriam A. M. Capretz. 2014. Challenges for MapReduce in big data. In *Proceedings of the IEEE 10th 2014 World Congress on Services (SERVICES'14)*.

Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. 2012. StreamCloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.* 23, 12 (Dec. 2012), 2351–2365. DOI: http://dx.doi.org/10.1109/TPDS.2012.24

Daniel Hagimont, Patricia Stolf, Laurent Broto, and Noel Palma. 2009. *Autonomic Computing and Networking*. Springer US, Boston, MA, Chapter Component-Based Autonomic Management for Legacy Software, 83–104. DOI: http://dx.doi.org/10.1007/978-0-387-89828-5_4

Moustafa A. Hammad, Michael J. Franklin, Walid G. Aref, and Ahmed K. Elmagarmid. 2003. Scheduling for shared window joins over data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases-Volume 29*. VLDB Endowment, 297–308.

Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2014. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS'14)*. ACM, New York, NY, 13–22. DOI: http://dx.doi.org/10.1145/2611286.2611294

Sebastian Herbst, Niko Pollner, Johannes Tenschert, Frank Lauterwald, Gregor Endler, and Klaus Meyer-Wegener. 2015. An algebra for pattern matching, time-aware aggregates and partitions on relational data streams. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS'15)*. ACM, New York, NY, 140–149. DOI: http://dx.doi.org/10.1145/2675743.2771830

Wilson A. Higashino, Miriam A. M. Capretz, and Luiz F. Bittencourt. 2016. CEPSim: Modelling and simulation of complex event processing systems in cloud environments. *Fut. Gen. Comput. Syst.* 65 (Dec. 2016), 122–139. DOI: http://dx.doi.org/10.1016/j.future.2015.10.023

Wilson A. Higashino, Cédric Eichler, Miriam A. M. Capretz, Thierry Monteil, Maria B. F. De Toledo, and Patricia Stolf. 2014. Query analyzer and manager for complex event processing as a service. In *Proceedings of the 2014 IEEE 23rd International WETICE Conference*. 107–109. DOI: http://dx.doi.org/ 10.1109/WETICE.2014.53

Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan Demers. 2009. Rule-based multi-query optimization. In *Proceedings of the 12th International Conference on Extending Database Technology Advances in Database Technology (EDBT'09)*. ACM, New York, NY, 120. DOI: http://dx.doi.org/10.1145/1516360.1516376

IBM. 2006. *An Architectural Blueprint for Autonomic Computing*. Technical Report. IBM.

Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. 2008. Towards a streaming SQL standard. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1379–1390.

JBoss. 2016. Drools. Retrieved April 13, 2016 from http://www.drools.org.

Jess. 2016. Jess, the Rule Engine for the Java Platform. (2016). Retrieved April 13rd, 2016 from http://www.jessrules.com/.

J. O. Kephart and D. M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (Jan 2003), 41–50.

Jürgen Krämer and Bernhard Seeger. 2009. Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst.* 34, 1 (2009), 1–49. DOI: http://dx.doi.org/10.1145/1508857.1508861

Geetika T. Lakshmanan, Ying Li, and Rob Strom. 2008. Placement strategies for internet-scale data stream systems. *IEEE Internet Comput.* 12, 6 (Nov. 2008), 50–60. DOI: http://dx.doi.org/10.1109/MIC.2008.129

Guoli Li and Hans-Arno Jacobsen. 2005. Composite subscriptions in content-based publish/subscribe systems. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware (Middleware'05)*. Springer-Verlag, New York, NY, 249–269.

Björn Lohrmann, Daniel Warneke, and Odej Kao. 2013. Nephele streaming: Stream processing under QoS constraints at scale. *Cluster Comput.* 17, 1 (July 2013), 61–78. DOI: http://dx.doi.org/10.1007/s10586-013-0281-8

Michael Löwe. 1993. Algebraic approach to single-pushout graph transformation. *Theor. Comput. Sci.* 109, 12 (1993), 181–224. DOI: http://dx.doi.org/10.1016/0304-3975(93)90068-5

David Luckham. 2002. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems* (1st ed.). Addison-Wesley Professional.

David Luckham and Roy Schulte. 2011. *Event Processing Glossary—Version 2.0*. Technical Report July. Event Processing Technical Society. 1–19 pages. Retrieved from http://www.complexevents.com/2011/08/23/event-processing-glossary-version-2/.

Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. 2002. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*. Vol. 13. ACM, New York, NY, 49. DOI:http://dx.doi.org/10.1145/564691.564698

Oracle. 2015. Oracle Stream Explorer. Retrieved October 31, 2015 from http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html.

Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. 2006. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 49.

Powersmiths. 2015. Powersmiths WOW - Build a more sustainable future. Retrieved October 28, 2015 from http://www.powersmithswow.com/.

Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. TimeStream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM Press, New York, NY, 1. DOI:http://dx.doi.org/10.1145/2465351.2465353

Ella Rabinovich, Opher Etzion, Sitvanit Ruah, and Sarit Archushin. 2010. Analyzing the behavior of event processing applications. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS'10)*. 223–234. DOI:http://dx.doi.org/10.1145/1827418.1827465

Ismael B. Rodriguez, Khalil Drira, Christophe Chassot, Karim Guennoun, and Mohamed Jmaiel. 2010. A rule-driven approach for architectural self adaptation in collaborative activities using graph grammars. *Int. J. Auton. Comput.* 1, 3 (2010), 226–245.

Grzegorz Rozenberg (Ed.). 1997. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific.

Sergio Segura, David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. 2008. Automated merging of feature models using graph transformations. In *Generative and Transformational Techniques in Software Engineering II*, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.). Lecture Notes in Computer Science, Vol. 5235. Springer, Berlin, 489–505.

Guy Sharon and Opher Etzion. 2008. Event-processing network model and implementation. *IBM Syst. J.* 47, 2 (2008), 321–334. DOI:http://dx.doi.org/10.1147/sj.472.0321

Software AG. 2015. APAMA Streaming Analytics. Retrieved October 31, 2015 from http://www.softwareag.com/corporate/products/apama_webmethods/analytics/overview/.

Storm. 2015. Storm: distributed and fault-tolerant realtime computation. Retrieved October 1, 2015 from http://storm-project.net/.

Gabriele Taentzer. 2004. AGG: A graph transformation environment for modeling and validation of software. In *Applications of Graph Transformations with Industrial Relevance*, John L. Pfaltz, Manfred Nagl, and Boris Bhlen (Eds.). Lecture Notes in Computer Science, Vol. 3062. Springer, Berlin, 446–453.

Matthias Weidlich, Jan Mendling, and Avigdor Gal. 2013. Net-based analysis of event processing networks the fast flower delivery case. In *Application and Theory of Petri Nets and Concurrency SE-15*, José-Manuel Colom and Jörg Desel (Eds.). Lecture Notes in Computer Science, Vol. 7927. Springer, Berlin, 270–290. DOI:http://dx.doi.org/10.1007/978-3-642-38697-8_15

Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data—SIGMOD'06*, Vol. 10. ACM Press, New York, NY, 407. DOI:http://dx.doi.org/10.1145/1142473.1142520

Ying Xing, Stan Zdonik, and Jeong-hyon Hwang. 2005. Dynamic load distribution in the Borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*. IEEE, 791–802. DOI:http://dx.doi.org/10.1109/ICDE.2005.53

Received January 2016; revised June 2016; accepted June 2016