Mining abstract XML data-types


Published in:
ACM Transactions on the Web

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

Publisher rights
Copyright 2018 ACM. This work is made available online in accordance with the publisher’s policies. Please refer to any applicable terms of use of the publisher.

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.
Schema integration has been a long-standing challenge for the data-engineering community that has received steady attention over the past three decades. General-purpose integration approaches construct unified schemas that encompass all schema elements. Schema integration has been revisited the last decade in service-oriented computing, since the input/output data-types of service interfaces are heterogeneous XML schemas. However, service integration differs from the traditional integration problem, since it should generalize schemas (mining abstract data-types), instead of unifying all schema elements. To mine well-formed abstract data-types, the fundamental Liskov Substitution Principle (LSP), which generally holds between abstract data-types and its subtypes, should be followed. However, due to the heterogeneity of service data-types, the strict employment of LSP on them is not usually feasible. On top of that, XML offers a rich type system, based on which data-types are defined via combining type patterns (e.g. composition, aggregation). The existing integration approaches have not dealt with the challenges of defining subtyping relation between XML type patterns. To address these challenges, we propose a relaxed version of LSP between XML type patterns and an automated generalization process for mining abstract XML data-types. We evaluate the effectiveness and the efficiency of our process on the schemas of two datasets against two representative state-of-the-art approaches. The evaluation results show that our process has higher effectiveness and efficiency than those of the state-of-the-art approaches.

CCS Concepts:
• Applied computing → Extensible Markup Language (XML);
• Information systems → Web services;

Additional Key Words and Phrases: XML schema, Web services, type pattern, subtyping relation, embedded subtree, pruning.

1. INTRODUCTION

Schema integration has been a long-standing challenge for the data-engineering community [Doan and Halevy 2005]. Schema integration is the process of merging schemas into a unified schema that encompasses all schema elements [Batini et al. 1986]. In particular, the integration process first establishes semantic correspondences between schema elements and then merges them [Pottinger and Bernstein 2003].

Schema integration has been revisited the last decade in the field of service-oriented computing [Erl 2005] to reconcile heterogeneity in service interfaces. In particular, famous software vendors (e.g. Google1, Amazon2) currently make available their resources over the Web as services, exposing the programmable service interfaces3. Service interfaces are generally defined as sets of operation signatures, whose input/output data-types are XML schemas4.

1 http://developers.google.com/maps/web-services/overview
2 http://aws.amazon.com
3 http://www.w3.org/TR/ws-arch
4 http://www.w3.org/XML
However, service interfaces are usually characterized by high heterogeneity. To reconcile the heterogeneity in service interfaces, some approaches have been proposed in the literature (e.g. [Athanasopoulos et al. 2011; Liu and Liu 2012]). These approaches automatically mine abstract services out of a set of alternative services. The data-types of abstract services should generalize the common/similar elements of input/output data-types. In this way, service integration actually deals with the problem of schema generalization, instead of the general-purpose schema-integration problem.

To mine well-formed abstract data-types, the fundamental Liskov Substitution Principle (LSP), which generally holds between abstract data-types and its subtypes, should be followed [Liskov and Wing 1994]. Specifically, LSP imposes requirements on type signatures and behaviours. Based on the signature requirements, an input (resp. output) type is subtype (resp. super-type) of another type if the former (resp. latter) is declared subtype\(^5\) of the latter (resp. former). Based on the behavioural requirements, objects of a subtype should behave only as allowed by objects of its super-type.

Due to the heterogeneity of service data-types, the strict employment of LSP in service interfaces is not usually feasible. For instance, data-types defined by a service provider are not declared subtypes of the data-types defined by another service provider. Thus, the challenge is to define a version of LSP for independently defined service data-types. On top of that, XML offers a rich type-system, based on which service data-types are defined via combining type patterns (e.g. composition, aggregation) [Rahm et al. 2004]. However, the existing approaches that integrate XML schemas ignore type patterns. In this way, the challenge is to define subtyping relation between type patterns. The last (time and space efficiency) challenge is related to the fact that schemas of service data-types usually include a high number of type patterns [Rahm et al. 2004]. This high number leads to the enumeration of a very high number of pattern combinations for examining subtyping relations between type patterns. For instance, the Amazon Web service EC2\(^6\) includes type patterns that are repeated more than 1000 times.

To address the first challenge, we propose a relaxed version of the LSP subtyping relation. In particular, we define LSP-based subtyping based on (i) the number, the structure, and the syntax of schema elements (syntactic subtyping) and (ii) the generalization/specialization relationships of element labels (semantic subtyping). To address the second challenge, we propose an automated generalization process that includes three sequentially executed mechanisms (Fig. 1). The first mechanism enumerates type patterns that have syntactic subtyping relation. The second mechanism matches and filters out the enumerated type patterns that have semantic subtyping relation. The third mechanism constructs abstract data-types from the filtered type patterns. To address the third challenge, we enhance the above mechanisms with pruning and greedy techniques. These techniques achieve the efficient enumeration of patterns, avoiding pattern redundancy. To do so, the efficiency techniques enumerate the best possible patterns by using two suites of metrics. The first suite assesses the syntactic confidence of type patterns (a.k.a pattern confidence) via calculating the pattern density. The second suite of metrics assesses the semantic confidence of type patterns (a.k.a. matching confidence).

We evaluate the effectiveness and the (time and space) efficiency of our process on two datasets. The first dataset includes the schema pairs provided by the (publicly available) benchmark XBenchMatch\(^7\) of XML schemas. The second dataset includes the data-type schemas of Amazon Web services. We evaluate the generalization process against two representative state-of-the-art approaches.

---

\(^5\) In Java, a type \(S\) is declared subtype of a type \(T\) if \(S\) implements or extends \(T\).

\(^6\) http://aws.amazon.com/releasenotes/Amazon-EC2

\(^7\) http://liris.cnrs.fr/~fduchate/research/tools/xbenchmatch
other approach is an LSP-based generalization approach [Athanasopoulos et al. 2011]. To evaluate the effectiveness of mined abstract data-types, we further propose an effectiveness metric that compares mined data-types against expert data-types. Moreover, we evaluate the impact of the pruning and the greedy techniques on both the efficiency and the effectiveness of our process. The evaluation results show (i) high effectiveness of our process for both datasets in identifying type patterns and mining abstract data-types; (ii) the pruning and greedy techniques significantly increase the efficiency of our process, without reducing its effectiveness.

We summarize our contribution as follows:

i. survey of state-of-the-art integration approaches
ii. definition of the notions of type pattern, subtyping relation, and abstract data-type
iii. definition of pattern and matching confidence metrics
iv. specification of the mechanisms of our process
v. description of a methodology for tuning the thresholds used by our process
vi. definition of the effectiveness metric for abstract data-types;
vii. evaluation of the effectiveness and the efficiency of our process.

The rest of the paper is structured as follows. Section 2 categorizes and compares the related state-of-the-art approaches. Section 3 specifies the basic notions used by the generalization process. Section 4 provides an overview of the generalization process. Sections 5, 6, and 7 detail the modus operandi of the mechanisms and define the confidence metrics. Section 8 evaluates the efficiency and the effectiveness of our process and describes the tuning methodology. Finally, Section 9 summarizes our contribution and proposes future research directions.

2. RELATED WORK

Our approach belongs to the research field of schema integration. Schema integration is the process of firstly establishing schema matchings and following, merging schemas into a unified schema [Pottinger and Bernstein 2003]. This process is also known as view integration [Batini et al. 1986]. Apart from unified schemas, mediated schemas can be further constructed to provide uniform query scripts to users [Halevy et al. 2006]. The construction of mediated schemas is the outcome of an extended process, known as data integration [Pottinger and Bernstein 2008]. Since we generalize schemas providing an abstract view of schemas (instead of constructing querying scripts), we focus in the remainder of the paper on view integration approaches.

In the decade of 80’s, [Batini et al. 1986] specified a view integration process that includes the pre-integration, schema comparison, schema conformance, and merging phases. In the first phase, the integration strategy is chosen depending on the number of input schemas (e.g. binary vs. n-ary strategies). In the second phase, schemas are compared to identify correspondences between elements and detect possible conflicts. The third phase deals with the conflicts resolution to make the schema merging possible. In the last phase, schemas are integrated in a unified schema.

In the decade of 90’s, [Parent and Spaccapietra 1998] evolved the process into one that includes three phases (schema transformation, correspondence investigation, and schema integration). The purpose of the first phase is to make source schemas syntactically and semantically homogeneous. In the second phase, schemas are compared to identify correspondences between elements. In the last phase, schemas are merged guided by the identified correspondences.

In the last two decades, the integration process has been further evolved into a three-phase matching-centred process (schema matching, matching-driven integration, mapping formulation) due to the popularity and plethora of schema-matching approaches [Li 2012]. In this vein, we categorize the existing approaches in terms of the followed matching techniques as follows.

i. Conflict-driven approaches. The approaches of this category (e.g. [Baqasah et al. 2014; Ma et al. 2005; Pottinger and Bernstein 2003; Parent and Spaccapietra 1998; Kashyap and Sheth 1996]) follow the process described in [Batini et al. 1986] and aim at resolving conflicts. The most used categories of conflicts concern element names, built-in types, element cardinalities,
and schema structures. Some approaches deal with generalization conflicts [Batini et al. 1986], but these conflicts concern the element labels or built-in types and not type patterns.

ii. **Constraint-driven approaches.** The approaches of this category (e.g. [Li and Quix 2011; Arenas et al. 2010; Pottinger and Bernstein 2008; Bernstein et al. 2004]) accept logical constraints as input and perform reasoning on the schema merging. For instance, [Bernstein et al. 2004] applies inter- and intra-schema integrity constraints in unified schemas. [Pottinger and Bernstein 2008] generates unified schemas based on conjunctive queries that specify where input schemas overlap. However, these approaches do not consider constraints on type patterns.

iii. **Matching-driven approaches.** The approaches of this category (e.g. [Liu and Liu 2012; Athanasopoulos et al. 2011; Radwan et al. 2009; Sarma et al. 2008; Saleem et al. 2008; Melnik et al. 2003]) identify correspondences between schema elements as hints for the merging phase. For instance, [Saleem et al. 2008] and [Melnik et al. 2003] apply structure-preserving subtree matching techniques. [Athanasopoulos et al. 2011] and [Liu and Liu 2012] match the leaf elements of schema trees.

**Moving beyond the existing integration approaches.** The above approaches are compared in Table I in terms of the type of input schemas (e.g. XML, relational), integration techniques (e.g. rule-based, clustering, mining), properties of the unified schemas (e.g. generalization, completeness), and efficiency techniques (e.g. greedy matching, pruning). Based on the above comparison, our approach is closely related to the matching-driven approaches of [Saleem et al. 2008] and [Athanasopoulos et al. 2011]. These are the two approaches that we use for evaluating our approach in Section 8.

[Saleem et al. 2008] is a general-purpose integration approach, but it is not a schema generalization approach. In particular, it is the only approach that applies XML subtree mining as an integration technique, without though considering the integration of type patterns. Concerning the properties of the unified schemas, [Saleem et al. 2008] achieves schema completeness\(^8\) and minimality\(^9\), but not schema generalization.

[Athanasopoulos et al. 2011] is a generalization approach that adopts an LSP-based subtyping relation between the leaf elements of XML schemas, without though examining this relation between the type patterns formed by the internal schema elements. In this way, the approach considers a simplified version of subtyping relation between schemas.

Concerning our approach, it extends pattern mining and matching techniques for generalizing type patterns. To clarify how our techniques extend the existing ones, we further describe the most representative pattern mining and schema matching techniques in Sections 2.1 and 2.2.

### 2.1. Pattern mining

Pattern mining techniques aim at extracting subtrees in schema trees. The core types of subtrees are bottom-up, induced, and embedded [Jiménez et al. 2010]. A bottom-up subtree is rooted at any node of the tree and involves all the descendants of the root node. An induced subtree is obtained by a bottom-up subtree by repeatedly removing leaf nodes. An embedded subtree consists of nodes that do not break the ancestor-descendant relations. Embedded subtree mining techniques extract more patterns than the other subtree mining techniques, since embedded subtrees do not necessarily preserve parent-child node relations [Zaki 2005a]. To mine embedded subtrees, a suitable algorithmic technique can be pumped from the subtree mining domain [Chi et al. 2005]. The techniques of this domain can be divided into two broad categories: candidate-generation-and-test and pattern-growth. In the first category, the existing techniques (e.g. [Zaki 2005a; Zaki 2005b]) a-priori form all the possible subtrees and iteratively extend them if they appear in schemas. On the other hand, the techniques of the second category (e.g. [Pei et al. 2004; Yan et al. 2003; Wang et al. 2004; Zou...
Table I. Summary and comparison of schema integration approaches.

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Categories</th>
<th>Input</th>
<th>Integration Techniques</th>
<th>Unified-Schema Properties</th>
<th>Efficiency Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Kashyap and Sheth 1996]</td>
<td></td>
<td>relational</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Bagasah et al. 2014]</td>
<td></td>
<td>XML</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Parent and Spaccapietra 1998]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Pottinger and Bernstein 2003]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Arenas et al. 2010]</td>
<td>constraint-driven</td>
<td>meta-model</td>
<td>correspondences</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Li and Quix 2013]</td>
<td></td>
<td>mapping-based</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Pottinger and Bernstein 2008]</td>
<td></td>
<td>query-based</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Bernstein et al. 2004]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Athanasopoulos et al. 2011]</td>
<td></td>
<td></td>
<td>leaf subtyping</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Radwan et al. 2009]</td>
<td></td>
<td></td>
<td>generalization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Sarma et al. 2008]</td>
<td></td>
<td>meta-model</td>
<td>context-based</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Saleem et al. 2008]</td>
<td></td>
<td>relational</td>
<td>element clustering</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Melnik et al. 2004]</td>
<td></td>
<td>XML</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Melnik et al. 2003]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our approach</td>
<td>matching-driven</td>
<td>XML subtree mining</td>
<td>generalization (LSP-based subtyping)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Due to the high efficiency of the techniques of the second category, we extend in our problem a pattern-growth technique. In particular, we extend the embedded-subtree pattern-growth technique of [Zou et al. 2006] as follows.

1. Mining of valid embedded type patterns that have syntactic subtyping relation. While the recent subtree mining technique of [Chowdhury and Nayak 2014] checks the pattern validity based on the canonical form of a tree, our technique checks the pattern validity based on the XML syntax.

2. Mining of type patterns of two layers. The first layer concerns the mining of type patterns in an abstract form. The second layer concerns the enumeration of alternative instantiations of a pattern in schemas.

3. Employment of a pruning technique. Since the number of patterns and instantiations may be very high, our technique avoids pattern redundancy via applying a confidence-driven pruning-technique, in contrast to general purpose pattern pruning techniques (e.g. [Zou et al. 2006; Chowdhury and Nayak 2014]). In particular, our pruning technique pre-calculates upper-bounds of the final confidence values of enumerated patterns.

4. Employment of an indexing technique. We further increase the efficiency of our technique by constructing an indexing data-structure that accelerates the traversal of schema trees.

2.2. Schema Matching

Matching approaches have been applied in hierarchical (e.g. XML), relational (e.g. SQL [Beaulieu 2009]), and ontology schemas (e.g. OWL). We compare matching approaches based on two criteria: (i) whether an approach matches the schema structure; (ii) whether an approach accounts for the efficiency of matching algorithm. The outcome of the comparison is summarized in Table II.

Concerning the first criterion, we hereafter call structure-based the approaches that match the schema structure. We organize the structure-based approaches into two categories. The first category includes approaches that match the schema structure in a relaxed manner, i.e. they match the context of schema elements (e.g. element ancestors, descendants). We call the approaches of the first category context-based. The second category includes approaches that match the schema struc-

---

12 The interested reader can also refer to the excellent surveys of schema [Rahm and Bernstein 2001; Bellahsene et al. 2011] and ontology [Shvaiko and Euzenat 2013] matching approaches.

13 http://www.w3.org/TR/owl-features
Table II. Summary and comparison of (schema and ontology) structure-based matching approaches.

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Categories</th>
<th>Input</th>
<th>Type Patterns</th>
<th>Schema Structure Parts</th>
<th>Efficiency Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Hamdaqa and Tahvildari 2014]</td>
<td>XML-based</td>
<td>Graph-structured element neighbourhood</td>
<td>X</td>
<td>B-tree structure</td>
<td>Partition-based matching</td>
</tr>
<tr>
<td>[Duchateau et al. 2007b]</td>
<td>XML</td>
<td>Element path, children, leaves</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Duchateau et al. 2007c]</td>
<td>OWL, RDFS</td>
<td>Element children, leaves</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Duchateau et al. 2007a]</td>
<td>XML</td>
<td>Element ancestors, children, leaves</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Hu et al. 2008]</td>
<td>XML, OWL, SQL</td>
<td>Tree leaves</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Do and Rahm 2007]</td>
<td>XML</td>
<td>Element ancestor path, children, leaves</td>
<td>Prüfer sequences</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Madhavan et al. 2001]</td>
<td>XML</td>
<td>Element children, leaves</td>
<td></td>
<td></td>
<td>Greedy graph matching</td>
</tr>
<tr>
<td>[Lee et al. 2002]</td>
<td>XML, OWL</td>
<td>Element ancestors, children, leaves</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Nayak and Iryadi 2007]</td>
<td>XML</td>
<td>Induced subtree</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>[Cruz et al. 2009]</td>
<td>XML, OWL, RDFS</td>
<td>Embedded subtree</td>
<td></td>
<td></td>
<td>Pruned subtree mining indexing structure greedy pattern matching</td>
</tr>
<tr>
<td>[Algergawy et al. 2010]</td>
<td>XML</td>
<td>Element path, children, leaves</td>
<td>Prüfer sequences</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Kim et al. 2011]</td>
<td>XML</td>
<td>Element ancestors, children, leaves</td>
<td>X</td>
<td></td>
<td>Greedy subtree mining</td>
</tr>
<tr>
<td>[Meijer 2008]</td>
<td>XML</td>
<td>Root path</td>
<td></td>
<td></td>
<td>Greedy graph matching</td>
</tr>
<tr>
<td>[Melnik et al. 2003]</td>
<td>XML, UML, SQL, RDF</td>
<td>Embedded subtree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Voigt 2011]</td>
<td>XML</td>
<td>Embedded subtree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our approach</td>
<td>Syntax-conforming</td>
<td>Embedded subtree (type-pattern subtyping)</td>
<td>Pruned subtree mining indexing structure greedy pattern matching</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Context-based approaches. We organize the approaches into two groups based on how the former define the element context. The approaches of the first group models a schema by a graph [Hamdaqa and Tahvildari 2014; Duchateau et al. 2007b; Duchateau et al. 2007c; Duchateau et al. 2007a; Meo et al. 2006]. In this case, the context of an element is captured by the graph-structured neighbourhood of an element, i.e. the nodes that can be reached by traversing graph edges. The approaches of the second group models a schema by a tree [Do and Rahm 2002; Aumueller et al. 2005; Do and Rahm 2007; Madhavan et al. 2001; Algergawy et al. 2009; Lee et al. 2002; Giunchiglia et al. 2004; Nayak and Iryadi 2007; Cruz et al. 2009; Algergawy et al. 2010; Kim et al. 2011; Jean-Mary et al. 2009; Lambrix and Tan 2006]. In this case, the context of an element includes the element descendants, ancestors, children, leaves, or a combination of them. The second group also includes approaches (e.g. [Meijer 2008]) that match the schema structure via adopting tree edit-distance techniques [Bille 2005]. Independently of the considered definition of the element context, context-based approaches are usually less effective than structure-preserving ones. For instance, as evaluated in [Saleem et al.
Table III. The mapping between type patterns and XML elements.

<table>
<thead>
<tr>
<th>Categories</th>
<th>Basic Type-Patterns</th>
<th>Sub-categories</th>
<th>XML Compositors/Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>has-as-attributes</td>
<td>has-ordered-attributes</td>
<td>sequence</td>
<td></td>
</tr>
<tr>
<td></td>
<td>has-one-attribute</td>
<td>choice</td>
<td></td>
</tr>
<tr>
<td></td>
<td>has-unordered-attributes</td>
<td>all</td>
<td></td>
</tr>
<tr>
<td>group-of-attributes</td>
<td>group-of-ordered-attributes</td>
<td>sequence</td>
<td></td>
</tr>
<tr>
<td></td>
<td>group-of-one-attribute</td>
<td>choice</td>
<td></td>
</tr>
<tr>
<td></td>
<td>group-of-unordered-attributes</td>
<td>all</td>
<td></td>
</tr>
<tr>
<td>specialization</td>
<td>complexType extension</td>
<td>complexContent &amp; extension</td>
<td></td>
</tr>
<tr>
<td></td>
<td>complexType restriction</td>
<td>complexContent &amp; restriction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>simpleType extension</td>
<td>simpleContent &amp; extension</td>
<td></td>
</tr>
<tr>
<td></td>
<td>simpleType restriction</td>
<td>simpleContent &amp; restriction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>built-in type restriction</td>
<td>simpleType &amp; restriction</td>
<td></td>
</tr>
<tr>
<td>is-a</td>
<td>complexType inheritance</td>
<td>element</td>
<td></td>
</tr>
<tr>
<td>list</td>
<td>built-in type list</td>
<td>simpleType &amp; list</td>
<td></td>
</tr>
<tr>
<td>union</td>
<td>simpleType/built-in type union</td>
<td>simpleType &amp; union</td>
<td></td>
</tr>
</tbody>
</table>

2008], a structure-preserving matching technique is more effective than the context-based matching technique of [Do and Rahm 2007].

2.2.2. Structure-preserving matching approaches. Starting with the approach of [Saleem et al. 2008], it produces structure-preserving matchings by using the subtree-mining technique of [Zaki 2005a]. The main difference with our approach is that [Saleem et al. 2008] matches and integrates schemas aiming at the completeness and minimality of unified schemas, without considering type patterns. Continuing to the approach of [Melnik et al. 2002], it produces structure-preserving matchings via enumerating all possible subtrees of XML schemas. Particularly, it mines induced subtrees, which are less general than embedded subtrees. Moreover, it does not consider type patterns and their subtyping relation. The approach of [Voigt 2011] mines embedded subtrees from each input schema independently of the other. In this way, it may miss subtrees that co-exist in schemas. Moreover, the approach does not consider type patterns and their subtyping relation. Finally, [Voigt 2011] mines the complete set of patterns, having time and space efficiency issues.

3. BASIC NOTIONS

We define the following notions that are used by our process: schema representation (Section 3.1), type pattern (Section 3.2), pattern-instantiation matching (Section 3.3), abstract data-type (Section 3.4), subtyping relation (Section 3.5), and abstract-to-source matching (Section 3.6).

3.1. Schema Representation

XML schemas contain two categories of elements: components and compositors. A component is used for defining a concept. A compositor relates components. Compositors and components are combined to form type patterns. A type pattern may include multiple successive compositors. The XML type patterns are provided in Table III.

Independently of the formed type patterns, we model a schema by an ordered labelled tree $S$ (Table IV (Eq. 1)), which is characterized by its id (i.e. a URI that uniquely recognizes a schema over the Web) and its root element $e$. An element $E$ (Table IV (Eq. 2)) is characterized by (i) its field $oe$ (Table IV (Eq. 3)) that includes the kind and label of the element; (ii) its scope; (iii) the set of its children. Regarding the (possibly empty) element label, it consists of pairs of attributes and values (Table IV (Eq. 4)). The value of an attribute may be a reference to another element. Element referencing permits the reuse of components, making schemas graph-structured. Since a tree-structured model is conceptually closer to the hierarchical structure of a data-type, we transform any schema into its corresponding tree-structured schema by removing element references that form cycles. We call a leaf node in a formed tree as component declaration (a leaf always corresponds to a component). We further call a non-leaf node as element definition (a non-leaf node may correspond to a component or a compositor).
Table IV. The definition of the schema representation model (using the convention instance:Type).

\[
[\text{Schema}]: \quad S := (id: \text{anyURI}^{15}, e : E) \quad (1)
\]

\[
[\text{Element}]: \quad E := (oe : OE, \text{scope} : \text{SCOPE}, \text{children}) \mid \text{children} = \{e_i : E\} \quad \land \quad e_i.\text{oe.kind} \leq \quad e_{i+1}.\text{oe.kind} \quad (2)
\]

\[
[\text{Original element}]: \quad OE := (\text{kind} : \text{String}, \text{label} : \text{LABEL}) \quad (3)
\]

\[
[\text{Element label}]: \quad \text{LABEL} := \{(a : \text{String}, v : \text{anyType}^{15})\} \quad (4)
\]

\[
[\text{Element scope}]: \quad \text{SCOPE} := [\text{left} \in \mathbb{N}, \text{right} \in \mathbb{N}] \quad (5)
\]

We use in the above definition the term element kind to refer to its XML kind (e.g. complexType, simpleType). We consider the element kind as the syntactic field of an element, since the element kind determines the type pattern in which an element participates. On the contrary, we consider the element label as the semantic content of an element, since a label is related to the semantic information needed for specifying a concept.

Concerning the element scope, it corresponds to the position of an element in a schema tree. The scope is defined as an interval of two integers (Table IV (Eq. 5)). The left endpoint corresponds to the numbering (starting from zero) of the element based on the pre-order traversal of the schema tree. The right endpoint corresponds to the numbering of the rightmost descendant of the element. Using element scopes, our mechanisms can check at constant time tree-node properties (e.g. whether a tree node is root, leaf, descendant of another node) [Saleem et al. 2008].

Finally, the set of the children of an element includes its in-line (a.k.a locally declared) and/or referenced elements. To recognize different paths from the root node to a reused element, our model maintains links to (instead of copies of) reused elements [Rahm et al. 2004]. To this end, our model uses the field OE\(^{14}\) (Table IV (Eq. 3)) to keep the content (i.e. kind and label) of a reused element only once. In this way, the size \(|s|\) of a schema \(s\) equals to the number of distinct elements, increased by the number of the links to reused elements. In our model, the schema size is found at constant time by retrieving the right endpoint of the scope of the rightmost element of a schema tree.

Finally, XML schemas are generally semi-ordered, since some compositors (e.g. sequence) define an order among their children. Without lose of generality, our schema parser automatically transforms a semi-ordered tree in its ordered representation [Zaki 2005a] via defining a linear (i.e. alphabetical) order on its elements with respect to their kinds (Table IV (Eq. 2)). In this way, we decrease the complexity of the mechanisms of the generalization process, since they examine less combinations of ordered elements.

Taking an example, Fig. 2 depicts three XML schemas that define three alternative data-types (StudiesInfo1, StudiesInfo2, and StudiesInfo3). Fig. 3 represents the schemas by using our model and gives some examples of scopes, schema sizes, and type patterns.

3.2. Type Pattern

We distinguish two categories of type patterns: basic and composite patterns. A basic pattern includes one or multiple successive compositors, followed by one or multiple components. The permissible syntactic combinations of components and compositors that form basic type-patterns are enumerated in Table III. Fig. 3 (a) depicts three basic patterns, outlined by rectangles. Multiple basic type-patterns can be combined in a syntactically valid way to form a composite type-pattern.

Type pattern. To put it formally, a basic/composite type-pattern \(P\) (Table V (Eq. 1)) is characterized by its tree structure \(\text{PAE}\) and instantiations \(\text{FIs}\). An instantiation is an occurrence of a pattern

\(^{14}\)The acronym OE stands for the term Original Element, i.e. element that is not copied.

\(^{15}\)www.w3.org/TR/xmlschema-2

ACM Transactions on the Web, Vol. 00, No. 00, Article 00. Publication date: 00.
in a schema, forming a syntactically-valid embedded-subtree of the schema tree. The instantiations of a pattern are indexed by their schema id (Table V (Eq. 2)).

Pattern structure. The pattern structure is an abstract representation of pattern instantiations. We model it as an ordered labelled tree (Table V (Eq. 3)). Each node of the pattern structure is characterized by the kind of the corresponding element. Our process constructs the nodes of the pattern structure without including labels inside them. Thus, we hereafter call these nodes as abstract elements. In particular, an abstract element \( \text{PAE} \) (Table V (Eq. 3)) is characterized by its kind, scope, and set of children. Abstract elements with a parent-child relation correspond to valid combinations of XML elements (verified by the function \( \text{isContent} \) (Table V (Eq. 4))).

Pattern instantiation. We also model a pattern instantiation \( \text{PI} \) (Table V (Eq. 5)) as an ordered labelled tree, whose nodes are concrete elements (a.k.a. they have labels). A concrete element \( \text{PCE} \) (Table V (Eq. 6)) is actually a schema element \( e \) and is further characterized by its scope and set of children. By construction, pattern instantiations preserve the ancestor-descendant relations of schema trees, forming embedded subtrees of the latter. Moreover, instantiations of the same pattern are isomorphic\(^{17}\) to each other, since they follow the same pattern structure.

Returning to our running example, Fig. 4 (a) and (b) depicts an instantiation and the structure of the composite pattern mined from the schemas of Fig. 3 (b) and (c). We observe in Fig. 4 (a) that the concrete elements of the pattern instantiation (outlined by rectangles) preserve the ancestor-descendant relations, forming an embedded subtree of the corresponding schema-tree. Moreover, the tree structure of the pattern instantiation is isomorphic to the pattern structure. We also observe in Fig. 4 (b) that the abstract elements of the pattern structure include the element kind, but they do not include labels. Finally, the pattern structure and instantiation are syntactically valid (i.e. they have permissible combinations of components and compositors).

\(^{16}\)Our process defines a linear (esp. alphabetical) order on the kinds of the children of an abstract element (Table V (Eq. 3)).  
\(^{17}\)Two trees are isomorphic if there is a mapping between the nodes and edges of the trees [Valiente 2002].
Fig. 3. The representation of the schemas of the running example using our model.
3.3. Pattern-Instantiation Matching

We use the term pattern-instantiation matching to refer to the set of the correspondences that have been assigned between the concrete elements of pattern instantiations across different schemas. Two concrete elements are linked if their labels have semantic subtyping relation. More formally, a matching $\mathcal{M}$ (Table VI (Eq. 1)) between two instantiations of the same pattern is defined as a set of 1–1 correspondences. A correspondence $c_i$ (Table VI (Eq. 2)) is assigned between concrete elements that are of the same kind, have labels with semantic subtyping-relation (if the elements include...
Table V. The definition of the notion of (basic/composite) type-pattern.

| Pattern: | \( P := (\text{pae} : \text{PAE}, \text{pis} : \text{PIs}) \) (1) |
| Pattern instantiations: | \( \text{PIs} := \{ (\text{s.id} : \text{URI}, \{ \text{pi} : \text{PI} \}) \} \) (2) |
| Abstract element: | \( \text{PAE} := (\text{kind} : \text{String}, \text{scope} : \text{SCOPE}, \text{children} = \{ \text{pae} : \text{PAE} \} \) \( \text{isContent} (\text{kind}, \text{pae}.\text{kind}) \land \text{pae}.\text{kind} \leq \text{pae}_{i+1}.\text{kind} \) (3) |
| Pattern instantiation: | \( \text{PI} := \text{pce} : \text{PCE} \) (5) |
| Concrete element: | \( \text{PCE} := (\epsilon : \text{E}, \text{scope} : \text{SCOPE}, \text{children} = \{ \text{pce} : \text{PCE} \} \) (6) |

labels), and have the same left-endpoint in their scopes (if the elements are internal tree-elements – see relation \( \text{inC} \) in Table VI (Eq. 3)). If both elements are leaves, then they do not need to have equal left-endpoints (different left-endpoints do not break the tree structure and syntax – see relation \( \text{outC} \) in Table VI (Eq. 4)). Concerning the assessment of the semantic relation of labels (a.k.a. semantic confidence), our process uses the metric \( \text{conL} \) (Table VIII (Eq. 6)) that we propose in Section 6.3.

Returning to our running example, Fig. 5 depicts the matching between the pattern instantiations that exist in the schemas StudiesInfo2 and StudiesInfo3. The element correspondences are depicted by fine dashed inter-schema lines. We observe that the correspondences have been assigned between internal concrete-elements that are of the same kind, have labels with semantic relation (if the elements include labels) and equal left-endpoints in their scopes. In the case of leaf elements, we also observe that the elements are of the same kind and have labels with semantic relation, without though having necessarily equal left-endpoints in their scopes.

3.4. Abstract Data-Type

An abstract data-type \( \text{AT} \) (Table VII (Eq. 1)) is characterized by its schema \( \text{as} \), pattern \( \text{p} \), and set of abstract-to-source matchings. The latter refers to correspondences between the elements of an abstract data-type and those of source schemas (Section 3.6). Concerning the schema of an abstract data-type, it follows the definition of Table IV. Our process constructs an abstract data-type in such a way that the structure and syntax of its schema are identical to those of the corresponding pattern (see the recursive function \( \text{isIdentical} \) in Table VII (Eq. 2)).

\[^{18}\text{We use the symbol} \oplus \text{to denote the XOR operation.}\]
Returning to our running example, Fig. 6 depicts the schema of the abstract data-type mined from the schemas StudiesInfo2 and StudiesInfo3 of Fig. 3 (b) and (c). We observe that the structure and syntax of the abstract data-type are identical to those of the pattern structure of Fig. 4.

3.5. Subtyping Relation

LSP imposes requirements on the signatures and behaviours of types [Liskov and Wing 1994]. Concerning signatures requirements, LSP imposes the contra-variance rule for input data-types and the covariance rule for output data-types. According to the contra-variance (resp. covariance) rule, a well-formed subtype has more and more specific (resp. less and less specific) constituent input (resp.
A correspondence (Table VII (Eq. 4)) is assigned between the elements of an abstract and a source. We observe that the abstract-to-source matching is defined as follows:

\[
M := \{ e_i \in C_A \mid isIdentical \left( ae.oe.kind = pce.e.oe.kind \land \right. \\
\left. ae.e.pae.children \supseteq \bigcup_{i \in \{1, \ldots, |ae.children|\}} pae_i \right\}
\]

(1)

We define the notions of abstract data-type and abstract-to-source matching. Overall, we define that an XML schema is subtype (<\text{syn}> of another schema (Table VIII (Eq. 2)) if the former contains at least the same kinds and numbers of basic type-patterns with those of the latter and if the formed composite-patterns in both schemas are syntactically valid. In particular, a schema is syntactic subtype of another schema in parallel (Table VIII (Eq. 5)). The syntactic-subtyping relation is checked by traversing the tree structures of \(s\) and \(p\) in parallel (Table VIII (Eq. 5)).

Syntactic subtyping. We define that an XML schema is syntactic subtype (<\text{syn}> of another schema (Table VIII (Eq. 2)) if the former contains at least the same kinds and numbers of basic type-patterns with those of the latter and if the formed composite-patterns in both schemas are syntactically valid. In particular, a schema is syntactic subtype of an abstract data-type (mined from pattern \(p\)) if each instantiation \(p_i\) of \(p\) in \(s\) contains at least the same kinds and numbers of components and compositors (Table VIII (Eq. 2)).

Semantic subtyping. We define that an XML schema is semantic subtype (<\text{sem}> of another schema (Table VIII (Eq. 5)) if the labels of the elements of the basic type-patterns of the former are more specific than those of the latter. In particular, a schema is a semantic subtype of schema \(s\) as of an abstract data-type (mined from pattern \(p\)) if the element labels of each instantiation \(p_i\) of \(p\) in \(s\) are more specific than those of the abstract data-type (Table VIII (Eq. 5)).

Subtyping relation. Overall, we define that an XML schema is subtype (<\text{R}> of another schema (Table VIII (Eq. 1)) if the former is both syntactic and semantic subtype of the latter.

Returning to our running example, Fig. 6 depicts the abstract data-type StudiesInfo and its matchings to StudiesInfo3. StudiesInfo has been mined from the schemas of Fig. 3 (b) and (c). We observe that StudiesInfo3 is syntactic subtype of StudiesInfo, since the former contains at least the same kinds and numbers of basic type-patterns with those of the latter. We also observe that StudiesInfo3 is semantic subtype of StudiesInfo, since the labels of the elements of the former are more specific compared to those of the latter.

3.6. Abstract-to-Source Matching

We define an abstract-to-source matching \(M_A\) (Table VII (Eq. 3)) as a set of 1–1 correspondences. A correspondence (Table VII (Eq. 4)) is assigned between the elements of an abstract and a source.
Fig. 6. The abstract-to-source matching between a mined abstract data-type and the source schema of Fig. 3 (c).

schema only if the elements are of the same kind. Moreover, the elements have equal left-endpoints in their scopes if the elements are not leaves (Table VI (Eq. 5)).

Returning to our running example, Fig. 6 depicts the matching between an abstract data-type and its pattern instantiation in StudiesInfo3. The element correspondences are depicted by dashed inter-schema lines. We observe that the correspondences have been assigned between internal elements of the same kinds. Moreover, the labels of the abstract elements are more generic (if the elements include labels) than those of the concrete elements. Finally, the non-leaf abstract elements have equal left-endpoints to those of the concrete elements.
Table VIII. The definition of the notion of subtyping relation.

<table>
<thead>
<tr>
<th>Subtyping relation</th>
<th>Syntactic subtype</th>
<th>Semantic subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ S &lt;: s as : S ] := s &lt;:_syn as \land s &lt;:_sem as \mid \text{as} \in \text{at} \land \text{at} \in \text{AT} ] (1)</td>
<td>[ e : E &lt;:_syn ae : E := \text{pi}, \text{pce}, \text{e} &lt;:_syn as, \text{as}, \forall(s, \text{pi}) \in \text{at} \land \text{pis} \land \text{at} \in \text{AT} ] (2)</td>
<td>[ e : E &lt;:_sem ae : E := \text{pi}, \text{pce}, \text{e} &lt;:_sem as, \text{as}, \forall(s, \text{pi}) \in \text{at} \land \text{pis} \land \text{at} \in \text{AT} ] (5)</td>
</tr>
<tr>
<td>[ \text{isContent}(e.oe.kind, e_i, oe.kind) \land \forall e_i \in [1,</td>
<td>e.children</td>
<td>] \land e_j \in [1,</td>
</tr>
</tbody>
</table>

4. OVERVIEW OF THE GENERALIZATION PROCESS

We provide below an overview of the mechanisms of the generalization process (Fig. 1).

**Enumerating type patterns.** The mechanism accepts as input two schemas and enumerates type patterns that have syntactic subtyping-relation. To avoid the enumeration of a potentially high number of embedded type patterns, the mechanism applies a pruning technique. The mechanism further increases its efficiency by using an indexing structure that accelerates the schema traversal.

**Matching type patterns.** The mechanism accepts as input a set of patterns and determines the pairs of pattern instantiations that have semantic subtyping-relation. To avoid the examination of a prohibitively high number of element combinations, the mechanism employs a greedy technique. The mechanism finally returns the top-k matchings between pattern instantiations.

**Constructing abstract data-types.** The mechanism accepts as input matchings between pattern instantiations and constructs the top-k abstract data-types.

5. ENUMERATING TYPE PATTERNS

We specify the underlying algorithm (Section 5.1), the pruning technique (Section 5.2), the pattern confidence metric (Section 5.3), and the theoretical complexity of the algorithm (Section 5.4).

5.1. Algorithm for Enumerating Type Patterns

Algorithm 1 accepts as input two schemas and the thresholds, $\text{con}_{\text{PI min}}$, $\text{con}_{\text{\delta PI min}}$, and $\delta$ (used by the pruning technique). Eventually, the algorithm enumerates a non-redundant (i.e., generated at most once) set of patterns, passing through the following two phases (Sections 5.1.1 and 5.1.2).

5.1.1. Initialization phase. The algorithm constructs an (in-memory) indexing data-structure $\text{IAE}s$ (Table IX (Eq. 1)) via traversing the input schemas (Alg. 1 (2)) and organizing their elements (Alg. 1 (3)). Each group is indexed by an abstract element $\text{IAE}$ (Alg. 1 (5)), characterized by its element kind and concrete elements (Table IX (Eq. 2)). The concrete elements are indexed by their scopes, forming interval trees $\text{ICE}$ (Table IX (Eq. 4)). Interval trees offer element searching in logarithmic time with respect to the tree size [Cormen et al. 2001]. Following, the algorithm constructs a singleton pattern (i.e., includes one abstract element) (Alg. 1 (6)) and inserts the pattern into the list $\text{AP}$ (Accepted Patterns) that keeps all enumerated patterns (Alg. 1 (8)). The algorithm further inserts the pattern into the list $\text{UP}$ (Unexamined-for-growth Patterns) that keeps the unexamined patterns (Alg. 1 (9)).

5.1.2. Pattern-enumeration phase. The current phase includes two activities, in which the algorithm grows patterns with elements that preserve the schema structure and syntax, a.k.a. syntax-
growth elements (Table X (Eq. 1-2)). We also use the term growth (Table X (Eq. 3)) to refer to elements that preserve only the schema structure [Zaki 2005a].

Determining syntax-growth elements. The algorithm identifies for each pattern in $UP$ (Alg. 1 (10)) candidate elements that can extend the pattern structure and instantiations, without violating the schema structure and syntax (Alg. 1 (14)). To this end, the algorithm determines the nodes of the rightmost path of a pattern (Alg. 1 (12)). The rightmost expansion further guarantees the enumeration of non-redundant sets of patterns [Asai et al. 2002; Zaki 2002].

Pattern growth. The algorithm extends each node of the rightmost path of a pattern with the syntax-growth elements of the node, creating new patterns (Alg. 1 (16-19)). Following, the algorithm repeats the step for the pattern instantiations (Alg. 1 (20)).

5.2. Pruning Technique

The pruning technique copes with the case that input schemas have a high number of common type patterns. To reduce this number, the pruning technique does not grow patterns that are not dense. To illustrate the pattern density, Fig. 7 (a) and (b) depict the instantiation in StudiesInfo3 (outlined by elements in rectangles) and the structure of the pattern mined from the schemas of Fig. 3 (a) and

```plaintext
ALGORITHM 1: Enumerating Type Patterns

Input: List<S> I, double conP_{min}, double conP_{max}, double δ
Output: List<P> AP

// Phase 1: Initialization
1. IAEs iaes ← new IAEs()
2. for s_i ∈ I do
3.   for c_j ∈ s_i do
4.      iae_c ← iaes.identify(c_j, oc_kind)
5.      iae_c.insert(c_j)
6.     p ← new P()
7.     p.pae.kind ← “root”
8.     AP.insert(p)
9.     UP.insert(p)

// Phase 2: Pattern enumeration
10. for p_i ∈ UP do
11.   UP.remove(p_i)
12.   List<P>EA rightmostPath ← p_i.determineRightmostPath()
13.   List<IAE> qaes ← p_i.formQueryingAbstractElements(rightmostPath)
14. for qae_j ∈ qaes do
15.   IAEs gae ← iaes.lookUpGAEs(qae_j, kind)
16. for iae_k ∈ gae do
17.   GCE gce ← iaes.lookUpGCEs(qae_j, iae, iae_k)
18.   P_i^{new} ← new P(p_i)
19.   P_i^{new}.insert(iae_k)
20.   P_i^{new}.insert(gce)
21. if conP_{i}(P_i^{new}.pis) ≥ conP_{min} then
22.   AP.insert(P_i^{new})
23.   UP.insert(P_i^{new})
24. else
25. if conP_{i}(P_i^{new}.pis) ≥ conP_{min} then UP.insert(P_i^{new});
26.   for s_e ∈ P_i^{new}.pis do
27.     for pce_m ∈ P_i^{new}.pis.pce do
28.       if denPCE(pce_m) > maxDen then maxDen ← denPCE(pce_m);
29.     for pce_m ∈ P_i^{new}.pis.pce do
30.       if maxDen − denPCE(pce_m) > δ then P_i^{new}.pis.pce.remove(pce_m);
```
Table IX. The definition of the indexing data-structure.

[Indexing abstract-elements]: \( \text{IAEs} := \left\{ \text{iae}_i : \text{IAE} \right\} \) \hspace{1cm} (1)

[Indexing abstract-element]: \( \text{IAE} := \left( \text{kind} : \text{String}, \text{ices} : \text{ICEs} \right) \) \hspace{1cm} (2)

[Indexing concrete-elements]: \( \text{ICEs} := \left\{ \left( \text{s.id} : \text{URI}, \text{ice} : \text{ICE} \right) \right\} \) \hspace{1cm} (3)

[Indexing concrete-element]: \( \text{ICE} := \left( \text{e} : \text{E}, \text{children} \right) \mid \text{children} = \left\{ \text{ice}_i : \text{ICE} \right\} \) \hspace{1cm} (4)

(c). We observe in Fig. 7 (a) that five non-participating elements intervene between the nodes of the instantiation, making the instantiation to have low density. To measure the pattern density, we specify a confidence metric in Section 5.3.

Prior to defining the confidence metric, we firstly describe the modus operandi of the pruning technique. The pruning technique accepts to grow patterns only if their confidence values are high. We consider that a pattern is accepted if its confidence value is higher than a threshold \( \text{con}_{\text{PI}} \text{min} \) (Alg. 1 (21)). In this case, the pruning technique inserts the pattern in the lists \( \text{AP} \) (Accepted Patterns) and \( \text{UP} \) (Unexamined-for-growth Patterns) (Alg. 1 (22-23)). \( \text{UP} \) keeps the patterns whose growth may lead to accepted patterns. If the pattern confidence is lower than the threshold, then the pruning technique does not reject the pattern. On the contrary, the pruning technique further examines if the pattern is promising, i.e. if the pattern growth leads to production of accepted patterns. To decide that, the pruning technique compares an upper-bound of pattern confidence to another threshold \( \text{con}_{\text{PI}} \text{min} \) (Alg. 1 (25)). If the upper-bound is greater than \( \text{con}_{\text{PI}} \text{min} \), then the pruning technique inserts the pattern in \( \text{UP} \) (Alg. 1 (25)).

Apart from the pattern structure, the pruning technique does not grow pattern instantiations of low density. To decide that, the pruning technique assesses the closeness of the density of a pattern instantiation to the maximum density achieved by all pattern instantiations in a schema. If the difference between the two density values is lower than the value of another threshold \( \delta \), then the pruning technique does not grow the pattern instantiation (Alg. 1 (26-30)).

5.3. Metric of Pattern Confidence

We propose the metric \( \text{con}_{\text{PI}} \) (Table XI (Eq. 1)) that calculates the product of the densities of two schemas. We assume that the pattern density in a schema should reflect the best density achieved by all pattern instantiations in a schema. Based on this assumption, we propose the metric \( \text{den}_{\text{PI}} \) (Table XI (Eq. 2)) that calculates the maximum density of pattern instantiations in a schema. We further propose the metric \( \text{den}_{\text{PCE}} \) (Table XI (Eq. 3)) that calculates the density of a pattern instantiation by the percentage of the concrete elements that participate in a pattern instantiation over all elements of a pattern instantiation. We call the latter elements \textit{internal} and their number \( \text{numOfIEs} \) (Table XI (Eq. 4)) is calculated in constant time via subtracting the left endpoint of the root of a pattern-instantiation tree from the left endpoint of the rightmost element of the tree (Table XI (Eq. 6)).

To calculate an upper-bound of pattern confidence, we assume that its pattern instantiations shall grow with all possible syntax-growth concrete-elements. In this case, the size of a pattern instantiation will be increased by the number \( \text{numOfGEs} \) (Table XI (Eq. 5)) of these elements. Analogously to the pattern-confidence metrics, we calculate their upper-bounds by using the metrics \( \text{con}_{\text{PI}} \) (Table XI (Eq. 7)), \( \text{den}_{\text{P}1} \) (Table XI (Eq. 8)), and \( \text{den}_{\text{PCE}} \) (Table XI (Eq. 9)) that we propose.

5.4. Time and Space Complexity of the Algorithm for Enumerating Type Patterns

Time complexity. The most time consuming part of the algorithm is the pattern-enumeration phase. Concerning the first activity of this phase, the time complexity scales quadratically with the numbers...
of pattern instantiations and syntax-growth elements. In the worst case, each number equals to all schema elements. Thus, the time complexity of the first activity scales with the square of the schema size, multiplied by its logarithm (the latter is the complexity for searching for syntax-growth elements [Cormen et al. 2001]). Regarding the second activity, the time complexity scales up with the number of the pattern instantiations and the complexity of the density metric (the latter also scales up with the number of the pattern instantiations). Finally, the time complexity of the algorithm $TC_1$ (Table XIII (Eq. 1)) further depends on the number $|EP|$ of the enumerated patterns.

Space complexity. The algorithm instantiates in its initialization phase the indexing data-structure. Since the latter uses references to schema elements (instead of multiple copies), the initialization phase is in-place with respect to the schema size (i.e. it does not consume extra memory) [Cormen et al. 2001]. Regarding the pattern-enumeration phase, the space complexity scales up with the memory footprints of the abstract and concrete elements. Given that the number of the concrete elements is bounded by the schema size, the memory footprint of the former scales up with the total memory footprint of all schema elements. Finally, the space complexity of the algorithm $SC_1$ (Table XIII (Eq. 2)) depends on the number $|EP|$ of the enumerated patterns.

6. MATCHING TYPE PATTERNS

We specify the underlying algorithm (Section 6.1), the greedy technique (Section 6.2), the matching confidence metric (Section 6.3), and the theoretical complexity of the algorithm (Section 6.4).

6.1. Algorithm for Matching Type Patterns

Algorithm 2 takes as input a set of accepted patterns $AP$ and the thresholds $k^{20}$ and $con_{C,min}$. The algorithm identifies for each accepted pattern semantic matchings between the pattern instantiations via calculating their Cartesian product (Alg. 2 (4)). For each tuple of the Cartesian product (Alg. 2 (5)), the algorithm traverses the tree structure of the compared pattern-instantiations and checks if correspondences between their concrete elements can be assigned. To decide whether such a correspondence exists, the algorithm calculates the confidence $con_C$ of a correspondence (Alg. 2 (13)). The set of the assigned correspondences forms a matching $m$ (Alg. 2 (8)) that relates a pair of pattern instantiations. Regarding the assignment of correspondences between leaf elements of pattern instantiations, the algorithm further calculates the Cartesian product of sibling leaf-elements. Among all possible combinations of leaf-element correspondences, the algorithm accepts only those that link disjoint sets of elements. However, the calculation of the Cartesian product of sibling leaf-elements may be computationally expensive if their number is high. To reduce this complexity, the algorithm adopts the greedy technique specified in Section 6.2 (Alg. 2 (19)). Finally, the algorithm calculates the confidence $con_M(m, p_i)$ (Alg. 2 (25)) of a matching. If the confidence value belongs to the top-k values, then the algorithm stores that matching (Alg. 2 (25)).

6.2. Greedy Technique

The greedy technique calculates for each leaf element of a pattern instantiation the value of its correspondence confidence with the leaf elements of the other pattern instantiation and selects

---

20The threshold $k$ corresponds to the maximum number of returned matchings.
(a) The pattern instantiation that exists in StudiesInfo3.

(b) The pattern structure

Fig. 7. An instantiation and the structure of the pattern mined from the schemas of Fig. 3 (a) and (c).

the element pair with the maximum confidence value. A combination of correspondences is kept only if the confidence value of at least one of the participating correspondences is greater than the threshold $\text{con}_{C_{\text{min}}}$ (Alg. 2 (21-24)).

6.3. Metric of Matching Confidence

The metric assesses the extent to which generic labels can be constructed from the linked concrete-elements. We define the metric $\text{con}_M$ (Table XII (Eq. 1-2)) that calculates the average of the confidence values of all element-to-element correspondences of a matching. To calculate the semantic confidence of a correspondence, we define the metric $\text{con}_C$ (Table XII (Eq. 3)), which calculates the semantic confidence of element labels $\text{con}_L$. The metric $\text{con}_L$ (Table XII (Eq. 4)) calculates the product of the semantic-confidence values of the label attributes that play dominant role in the matching confidence. According to [Algergawy et al. 2010], the dominant attributes are the element name, type, and value.
ALGORITHM 2: Matching Type Patterns

Input: List<P> AP, int k, double \(c_{\text{min}}\)
Output: List<M> \(M_s\)

1 \(M_s \leftarrow \emptyset\)
2 for \(p_i \in AP\) do
3 \(AP_.\text{remove}(p_i)\)
4 \(\text{List}\langle\text{Tuple}\langle PI\rangle\rangle \text{cartesianProduct} \leftarrow p_i \cdot \text{pis} \cdot p_i \times p_i \cdot \text{pis} \cdot p_i\)
5 for \((pce_i, pce_j) \in \text{cartesianProduct}\) do
6 \(\text{Queue}\langle\text{Tuple}\langle P I\rangle\rangle Q \leftarrow \emptyset\)
7 \(Q_.\text{insert}(pce_i, pce_j)\)
8 \(M_m \leftarrow \emptyset\)
9 boolean \(f \leftarrow \text{true}\)
10 while \(|Q| > 0 \land f = \text{true}\) do
11 \(\text{Tuple}\langle PI\rangle (pce_i, pce_j) \leftarrow Q_.\text{dequeue}()\)
12 \(C_o \leftarrow \text{formCorrespondence}(pce_i, e, pce_j, e)\)
13 if \(\text{conc}(c_o) > \text{conc}_{\text{min}}\) then \(M_m_.\text{insert}(m)\);
14 else if \(pce_i_.\text{children} \neq \emptyset \land pce_j_.\text{children} \neq \emptyset\) then \(f \leftarrow \text{false}\);
15 if \(pce_i_.\text{children} \neq \emptyset\) then \(f \leftarrow \text{false}\);
16 for \((pce_k \in pce_i_.\text{children}, pce_j \in pce_j_.\text{children})\) do
17 \(\text{if } pce_k_.\text{scope.left} = pce_j_.\text{scope.left} \text{ then } Q_.\text{enqueue}(pce_k, pce_j)\);
18 else
19 \(\text{Tuple}\langle C \rangle \text{ correspondences} \leftarrow \text{selectGreedy}(pce_i_.\text{children}, pce_j_.\text{children})\)
20 \(\text{counter} \leftarrow 0\)
21 for \(c_i \in \text{correspondences}\) do
22 \(\text{if } \text{conc}(c_i) > \text{conc}_{\text{min}}\) then \(M_m_.\text{insert}(m)\);
23 else \(\text{counter} \leftarrow \text{counter} + 1\);
24 \(\text{if } \text{counter} = \text{correspondences}\) then \(f \leftarrow \text{false}\);
25 if \(f = \text{true} \land \text{cond}(m, p_i, \text{conc}_{\text{min}}) > \text{cond}(m_k, p_k, \text{conc}_{\text{min}})\) then \(M_s_.\text{insert}(m)\);

Concerning the confidence of pairs of type attributes, we consider the case of built-in datatypes via using the metric \(\text{cond}_T\) (Table XII (Eq. 4)). That confidence is calculated in state-of-the-art approaches based on statically defined similarity tables [Madhavan et al. 2001; Stroulia and Wang 2005; Plebani and Pernici 2009]. We do not specify a formula for \(\text{cond}_T\), since its values are directly retrieved by the similarity table in [Plebani and Pernici 2009].

Regarding the confidence of pairs of name attributes, we define the metric \(\text{cond}_N\) (Table XII (Eq. 7)) that calculates the relatedness of WordNet concepts [Miller 1995]. \(\text{cond}_N\) uses Lin’s metric, which is the most effective WordNet-based metric [Pedersen et al. 2004]. If names consist of a concatenation of tokens, then \(\text{cond}_N\) automatically segments them into tokens. The tokenization is based on two widely used naming conventions, the Java-capitalized and the Pascal-underscore22. \(\text{cond}_N\) calculates the average confidence of the most related pairs of tokens (Table XII (Eq. 5)). To automatically identify the latter, the metric solves the assignment problem (Table XII (Eq. 6)) of the maximum-weighted matching in a bipartite graph [Burbard et al. 2009].

6.4. Time and Space Complexity of the Algorithm for Matching Type Patterns

**Time complexity.** The time complexity \(T_{C2}\) (Table XIII (Eq. 3)) of the algorithm scales up with the cardinality of the product of the instantiations \(\prod |\text{pis} \cdot |\text{pis} \cdot |pce|\) of accepted patterns \(AP\). The complexity further scales up with the number \(|\text{pis} \cdot |\text{pis} \cdot |pce|\) of the concrete elements of pattern instantiations (Table XIII (Eq. 3)). Moreover, the complexity scales up with the number of the combinations of

---

21 [www.oracle.com/technetwork/java/codeconventions-135099.html](http://www.oracle.com/technetwork/java/codeconventions-135099.html)
22 [cdn.embarcadero.com/article/10280](http://cdn.embarcadero.com/article/10280)
pattern instantiations (and (iv) and the variable isInput as follows. The algorithm calls the recursive function (Alg. 3 (2)), along with their matchings (Alg. 3 (3)), and constructs an abstract data-type as follows.

Algorithm 3 accepts as input a pattern p, a list of pairs of pattern instantiations pisPairs (along with their matchings), and a boolean variable isInput. The algorithm returns a list of abstract data-types. In detail, the algorithm examines each pair of pattern instantiations (pi1, pi2) of pisPairs (Alg. 3 (2)), along with their matchings (Alg. 3 (3)), and constructs an abstract data-type at (Alg. 3 (5)) from each matching. The algorithm calls the recursive function construct to build an abstract data-type (Alg. 3 (4)) as follows.

The function construct (Alg. 3 (7-25)) accepts as input (i) the root concrete-elements of two pattern instantiations (pce1 and pce2), (ii) a matching m between them, (iii) the pattern structure pae, and (iv) the variable isInput. The function returns the root abstract-element e of a constructed

\[
\text{con}_{\text{PCE}}(\text{pis : PIs}) := \prod_i \text{den}_{\text{PCE}}(\text{pis.pi}_i) 
\]

\[
\text{den}_{\text{PCE}}(\text{pis.pi}) := \max \text{den}_{\text{PCE}}(\text{pi.pi}) 
\]

\[
\text{den}_{\text{PCE}}(\text{pis.pi}) := \frac{|\text{pce}|}{\text{numOfIEs}(\text{pce})} 
\]

\[
\text{numOfIEs}(\text{pce : PCE}) := \text{rightmost}((\text{pce}.e).scope.left) - (\text{pce}.e).scope.left + 1 
\]

\[
\text{rightmost}(\text{pce : PCE}) := \begin{cases} 
\text{pce}, & \text{if pce.children = } \emptyset \\
\text{rightmost}((\text{pce}.e).scope.left) - 1, & \text{otherwise}
\end{cases}
\]

\[
\text{numOfGEs}(\text{s} : \text{S}, \text{pce : PCE}) := |\text{s}| - \text{rightmost}((\text{pce}.e).scope.left) - 1 
\]

\[
\text{den}_{\text{PCE}}(\text{s} : \text{S}, \text{pce : PCE}) := \frac{|\text{pce}| + \text{numOfGEs}(\text{s}, \text{pce})}{\text{numOfIEs}(\text{pce}) + \text{numOfGEs}(\text{s}, \text{pce})} 
\]

7. CONSTRUCTING ABSTRACT DATA-TYPES

We specify the underlying algorithm (Section 7.1) and its theoretical complexity (Section 7.2).

7.1. Algorithm for Constructing Abstract Data-Types

Algorithm 3 accepts as input a pattern p, a list of pairs of pattern instantiations pisPairs (along with their matchings), and a boolean variable isInput. The algorithm returns a list of abstract data-types. In detail, the algorithm examines each pair of pattern instantiations (pi1, pi2) of pisPairs (Alg. 3 (2)), along with their matchings (Alg. 3 (3)), and constructs an abstract data-type at (Alg. 3 (5)) from each matching. The algorithm calls the recursive function construct to build an abstract data-type (Alg. 3 (4)) as follows.

The function construct (Alg. 3 (7-25)) accepts as input (i) the root concrete-elements of two pattern instantiations (pce1 and pce2), (ii) a matching m between them, (iii) the pattern structure pae, and (iv) the variable isInput. The function returns the root abstract-element e of a constructed

---

23 The number of the leaves of a tree (independently of its type, e.g. binary, triadic, etc.) generally scales up with the sum of the degrees (i.e. number of children) of the tree nodes, \( \sum \text{degree}(v) \) [Aho et al. 1983].

24 It indicates whether the patten instantiations have been mined from the schemas of input or output data-types.
elements are siblings to each other nations of sibling leaf-elements of pattern instantiations. Assuming in the worst case that all leaf

tern p. Since the algorithm examines all of the pairs of the instantiations of a pattern p, the time complexity TC\(_3\) (Table XIII (Eq. 5)) scales up with the number of those pairs, \(|p.pis|*(|p.pis|-1)/2\). Furthermore, the complexity scales up with the number of the combinations of sibling leaf-elements of pattern instantiations. Assuming in the worst case that all leaf elements are siblings to each other\(^{25}\), the above number is captured by the coefficient \((|leaves|)^2\).

\(^{25}\)www.w3.org/TR/xmlschema-2

---

**7.2. Time and Space Complexity of the Algorithm for Constructing Abstract Data-Types**

**Time complexity.** Since the algorithm examines all of the pairs of the instantiations of a pattern p, the time complexity \(TC_3\) (Table XIII (Eq. 5)) scales up with the number of those pairs, \(|p.pis| * (|p.pis| - 1)/2\). Furthermore, the complexity scales up with the number of the combinations of sibling leaf-elements of pattern instantiations. Assuming in the worst case that all leaf elements are siblings to each other\(^{25}\), the above number is captured by the coefficient \((|leaves|)^2\).

\(^{25}\)www.w3.org/TR/xmlschema-2
Table XIII. The total complexity of the generalization process (without considering the efficiency techniques).

\[
\begin{align*}
TC_1 & := O\left(|EP| \times \sum_{i=1}^{2} |s_i|^2 \times \log |s_i|\right) \\
SC_1 & := O\left(|EP| \times \left(p_{pae} \times \text{space}(p_{pae}) \times \sum_{i=1}^{2} |s_i| \times \text{space}(p_{pis}.pi.pce)\right) \right) \quad |p \in EP \\
TC_2 & := O\left(|AP| \times \sum_{s=1}^{2} |p_{pis}.pi.\ast| \times \left(|p_{pis}.pi.pce| + \left(\frac{|\text{leaves}|^2}{2}\right)\right) \right) \quad |p \in AP \land |\text{leaves}| = 0 \\
SC_2 & := O\left(|AP| \times \left(\sum_{s=1}^{2} |p_{pis}.pi.\ast| \times \sum_{i=1}^{2} |s_i| \times \text{space}(p_{pis}.pi.pce) + \sum_{j=1}^{M} \text{space}(m_j)\right) \right) \quad |p \in AP \\
TC_3 & := O\left(|p_{pis}| \times \frac{|(p_{pis} - 1)}{2} \times \left(|p_{pis}.pi.pce| + \left(\frac{|\text{leaves}|^2}{2}\right)\right) \right) \quad |p \in AP \land |\text{leaves}| = 0 \\
SC_3 & := O\left(|p_{pis}| \times \frac{|(p_{pis} - 1)}{2} \times \left(|p_{pis}.pi.pce| + \left(\frac{|\text{leaves}|^2}{2}\right)\right) \times |s| \times \text{space}(e) \right) \quad |p \in AP
\end{align*}
\]

Space complexity. The space complexity scales up with the memory footprint of a constructed abstract schema (\(|s| \times \text{space}(e)\)). Furthermore, given that the algorithm constructs so many abstract schemas as many the different matchings of pattern-instantiation pairs are, the space complexity \(SC_3\) (Table XIII (Eq. 6)) further scales up with the number of the matchings.

8. EXPERIMENTAL EVALUATION

We implemented in Java the research-prototype Hector of our process. We evaluate the effectiveness and the efficiency of Hector on two datasets against the two representative state-of-the-art approaches described in Section 2. Concerning the effectiveness evaluation, we compare the mined abstract data-types against data-types defined by experts (Section 8.2). Regarding the efficiency evaluation, we examine whether the execution time of Hector is low enough to be practically applicable (Section 8.3). We also evaluate the impact of the pruning and greedy techniques on both efficiency and effectiveness of Hector (Section 8.4). Finally, we describe how end-users can tune the thresholds of Hector based on the evaluation results (Section 8.4.3).

8.1. Experimental Setup

8.1.1. Datasets. The first dataset includes the schema pairs of the (publicly available) benchmark XBenchMatch\(^{26}\). XBenchMatch covers a wide range of schema sizes and structural & semantic heterogeneity. XBenchMatch has been used for the evaluation of top-rated schema integration and matching approaches. The second dataset corresponds to the schemas of Amazon Web services\(^{27}\) that include a high number of basic patterns. We assign the identifiers \(s_1\)\(\sim s_{18}\) and \(s_{19}\)\(\sim s_{30}\) to the schemas of XBenchMatch and Amazon services, respectively (Table XIV). According to the literature [Duchateau et al. 2008], the size of large-sized schemas is greater than 1000 elements and the size of small-sized schemas is lower than 100 elements.

8.1.2. Effectiveness Metrics. To compare a mined schema against a schema defined by experts, three complementary metrics have been proposed in the literature [Duchateau and Bellahsene 2010].

\(^{26}\)http://liris.cnrs.fr/~fduchate/research/tools/xbenchmatch
\(^{27}\)http://aws.amazon.com
The completeness metric assesses the percentage of the common elements between two schemas. The minimality metric assesses the percentage of the extra elements between two schemas. The structurality metric checks if the elements of a mined schema have the same ancestors with those in an expert schema. However, these metrics do not assess the quality of schemas with respect to type patterns. Thus, we propose the metric $GEN$ for evaluating the effectiveness of enumerated patterns and the metric $EFFE$ for evaluating the effectiveness of pattern matchings.

$GEN_{at}$ (Table XV (Eq. 2)) calculates the percentage of the common basic type-patterns $btp$ of a mined abstract data-type $at$ over the basic type-patterns $btp_{ex}$ of an expert abstract data-type $at_{ex}$ (first fraction of Table XV (Eq. 2)). The fraction denominator is the max number of basic type-patterns to cover the case that either a mined or an expert data-type has more type patterns. $GEN_{btp}$ calculates the percentage of the common compositors $tor$, components $nent$, and links in a mined type-pattern against those in an expert type-pattern (Table XV (Eq. 3–4)). $GEN_{at}$ further calculates the percentage of the common direct-links between basic type-patterns (second fraction of Table XV (Eq. 2)). A direct link exists between the components of one type-pattern and the composers of another type-pattern. $GEN_{at}$ calculates the product of the two fractions, assuming that the effectiveness of an abstract data-type is high when both percentages of common basic type-patterns and direct links are high. Given that Hector returns the top-k abstract data-types, we further propose the metric $GEN_{ats}$ (Table XV (Eq. 1)) that calculates the effectiveness of an abstract data-type based on its position in the list of the top-k data-types $ats$.

$EFFE$ (Table XVI (Eq. 2)) assesses the F-measure (precision and recall) [Baeza-Yates and Ribeiro-Neto 1999] of a mined matching against an expert matching. The precision metric (Ta-
Table XIV. The schemas of the datasets used for the evaluation of Hector.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Schema Name</th>
<th>Domain</th>
<th>Schema Size</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>XBenchMatch</td>
<td>person1, person2</td>
<td>e-commerce</td>
<td>14</td>
<td>s1</td>
</tr>
<tr>
<td></td>
<td>univ-dept1, univ-dept2</td>
<td>education</td>
<td>15</td>
<td>s3</td>
</tr>
<tr>
<td></td>
<td>univ-courses1</td>
<td></td>
<td>16</td>
<td>s4</td>
</tr>
<tr>
<td></td>
<td>univ-courses2</td>
<td></td>
<td>27</td>
<td>s5</td>
</tr>
<tr>
<td></td>
<td>biology1, biology2</td>
<td>biology</td>
<td>23</td>
<td>s6</td>
</tr>
<tr>
<td></td>
<td>stock1, stock2</td>
<td>finance</td>
<td>38</td>
<td>s9</td>
</tr>
<tr>
<td></td>
<td>currency1, currency2</td>
<td></td>
<td>15</td>
<td>s11</td>
</tr>
<tr>
<td></td>
<td>betting1, betting2</td>
<td></td>
<td>23</td>
<td>s13</td>
</tr>
<tr>
<td></td>
<td>sms1, sms2</td>
<td>communication</td>
<td>70</td>
<td>s15</td>
</tr>
<tr>
<td></td>
<td>Travel1, travel2</td>
<td>entertainment</td>
<td>11</td>
<td>s17</td>
</tr>
<tr>
<td></td>
<td>SQS1, SQS2, SQS3, SQS4</td>
<td>queue service</td>
<td>290</td>
<td>s19</td>
</tr>
<tr>
<td></td>
<td>EC21, EC22, EC23, EC24</td>
<td>elastic compute cloud</td>
<td>3312</td>
<td>s23</td>
</tr>
<tr>
<td></td>
<td>VPC1, VPC2, VPC3, VPC4</td>
<td>virtual private cloud</td>
<td>1923</td>
<td>s27</td>
</tr>
</tbody>
</table>

8.1.3. Efficiency metrics. To evaluate the efficiency of Hector, we measure the execution times of the three mechanisms of Hector, their main-memory consumptions, and the values of their main complexity coefficients (the numbers of patterns and their instantiations) (Table XIII).

8.2. Effectiveness Evaluation

We evaluate the effectiveness of Hector on both datasets against the two representative state-of-the-art approaches, RW1 [Saleem et al. 2008] and RW2 [Athanasopoulos et al. 2011]. We set the threshold values for Hector by using our tuning methodology that is described in Section 8.4.3.

XBenchMatch schemas. The GEN and EFFE values are depicted in Fig. 8. We observe that the GEN values for Hector are higher than those of RW1 and RW2 in all schema pairs. It happens since RW1 and RW2 do not match compositors and links. Moreover, the EFFE values for Hector

We characterize low the execution time of a mining/matching technique for large-sized schemas if the time is < 10 min.
Table XV. The definition of the suite of effectiveness metrics for abstract data-types (on the absolute scale [0, 1]).

[Abstract data-type in top-k list]: \[ GEN_{ats} := (a, p, ats) := \begin{cases} 0, & \text{if } at \not\in ats \\ \frac{|ats| - p + 1}{|ats|}, & \text{otherwise} \end{cases} \] (1)

[Abstract data-type]: \[ GEN_{at} := \frac{\sum_{k=1}^{n} |btp|}{MAX\{\{|btp|, |btp_{czl}\|\}} \times \sum_{i=2}^{n} \frac{\sum_{j=i-1}^{MAX\{\{|ln_{btp_{i-1}|}, |ln_{btp_{i-1}_{czl}}|\}}}{\sum_{i=2}^{n} \frac{\sum_{j=i-1}^{MAX\{\{|ln_{btp_{i-1}|}, |ln_{btp_{i-1}_{czl}}|\}}}{}} (2) \]

[Basic type-pattern]: \[ GEN_{btp} := \frac{\{|tor \cap tor_{czl}\| + \{|nent \cap nent_{czl}\| + MAX\{|ln_{c}, |ln_{czl}\|\}}}{MAX\{|tor|, |tor_{czl}|\} + MAX\{|nent|, |nent_{czl}|\} + MAX\{|ln_{c}|, |ln_{czl}|\}} \] (3)

[Compositors and components links]: \[ ln_{c} := \sum_{i=1}^{n} ln_{c_{i}}, \text{ where pairs is the set of the pairs of compositors and components that are connected with directed links} ln_{c_{i}} \] (4)

Table XVI. The definition of the suite of effectiveness metrics for pattern matchings (on the absolute scale [0, 1]).

[F-measure of pattern-instantiation in top-k list]: \[ EFFE(m, p, M) := \begin{cases} 0, & \text{if } m \not\in M \\ \frac{|M| - p + 1}{|M|}, & \text{otherwise} \end{cases} \] (1)

[F-measure of pattern-instantiation]: \[ F := 2 \times \frac{pr \times r}{pr + r} \] (2)

[Precision]: \[ pr := \frac{tp}{tp + fp} \] (3)

[Recall]: \[ r := \frac{tp}{tp + fn} \] (4)

[True positives]: \[ tp := |m.C \cap m_{czl}.C| \] (5)

[False positives]: \[ fp := |m.C - m_{czl}.C| \] (6)

[False negatives]: \[ fn := |m_{czl}.C - m.C| \] (7)

are higher than or at the same levels with those of RW_1 and RW_2. In the cases of close EFFE values, we manually inspected the results and we observed that RW_1 and RW_2 identify correct element-correspondences. However, such cases appear in small-sized schemas, in which few alternative matching options exist. In medium- and large-sized schemas (e.g. s_7 and s_9), Hector achieves higher EFFE values than those of RW_1 and RW_2.

Amazon schemas. The GEN and EFFE values are depicted in Fig. 9. We observe that both GEN and EFFE values for Hector are higher than those of RW_1 and RW_2 in all schema pairs. To explain that, we inspected the results and we observed that the results of RW_1 and RW_2 include false positives related to components that do not have subtyping relation.

8.3. Efficiency Evaluation

We executed Hector using the pruning and greedy techniques. The execution times\(^{29}\) of all of the approaches for the XBenchMatch and Amazon schemas are depicted in Fig. 10. We observe that

\(^{29}\)The execution time of Hector equals to the sum of the execution times of all of its mechanisms.
the execution times are very low\textsuperscript{28} in all schema pairs. We also observe that the execution times of \textit{Hector} are close to or slightly higher than those of \textit{RW}$_1$ and \textit{RW}$_2$.

8.4. Evaluation of the Pruning and Greedy Techniques

We evaluate the impact of the pruning and greedy techniques on the efficiency (execution time and memory consumption) and the effectiveness of \textit{Hector} (Sections 8.4.1 and 8.4.2).

8.4.1. Impact of the pruning and greedy techniques on the efficiency of \textit{Hector}. The pruning technique is used by the mechanism \textit{Enumerating Type Patterns} (we denote it by $M_1$) of \textit{Hector} and the greedy technique is used by the mechanism \textit{Matching Type Patterns} ($M_2$).

\textit{Execution time}. We indicatively present in Table XVII the execution times of \textit{Hector} for small- and large-sized schemas from both \textit{XBenchMatch} and \textit{Amazon} datasets. We observe from the re-
Fig. 9. The effectiveness results on the Amazon schemas using the metrics GEN and EFFE.

results that the execution times of both $M_1$ and $M_2$ are low for any schema pair when the efficiency techniques are applied. However, when the efficiency techniques have not been applied, the execution times of $M_1$ and $M_2$ for large-sized schemas are high.

Memory consumption. We measured the amount of the consumed main-memory via using a JVM profiler\(^{30}\). When the efficiency techniques have been applied, the memory consumption of Hector for the pair ($s_7$, $s_8$) was on average 1.5GB. When the efficiency techniques have not been applied, the average memory-consumption of both $M_1$ and $M_2$ was 3.5GB.

\(^{30}\)http://www.jvmmonitor.org
8.4.2. Impact of the pruning and greedy techniques on the effectiveness of Hector. The techniques depend on the thresholds $con\Pi_{min}$, $con\Pi_{prom}$, and $\delta$. $con\Pi_{min}$ and $con\Pi_{prom}$ are used for characterizing patterns as accepted and promising, respectively. $\delta$ is used for pruning pattern instantiations.

Impact of $con\Pi_{min}$. We measured the numbers of accepted patterns ($con\Pi_{min}$ affects their numbers)\(^\text{31}\). We indicatively plot in Fig. 11 the results for the first four schema pairs of our datasets that include small-, medium-, and large-sized schemas. We observe from the results that for low values of $con\Pi_{min}$, the EFFE value is maximized. Hence, the effectiveness of Hector is kept high even

\(^{31}\)We had set the value of $con\Pi_{prom}$ to zero, so that all unaccepted patterns to be promising.
Table XVII. Execution times of Hector in minutes (we have set in bold the values that are not low (> 10 min)).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Schema pair</th>
<th>Applying efficiency techniques</th>
<th>Without applying efficiency techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$M_1$</td>
<td>$M_2$</td>
</tr>
<tr>
<td>XBenchMatch</td>
<td>$(s_1, s_2)$</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td></td>
<td>$(s_3, s_4)$</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td></td>
<td>$(s_5, s_6)$</td>
<td>0.3700</td>
<td>3.1900</td>
</tr>
<tr>
<td></td>
<td>$(s_7, s_8)$</td>
<td>6.3500</td>
<td>2.2000</td>
</tr>
</tbody>
</table>

if the number of accepted patterns is drastically decreased. For high values of $con_{PI_{min}}$ ($\geq 0.81$), $M_1$ does not produce any pattern and the EFFE value equals to zero.

Impact of $con_{PI_{min}}^u$. We measured the numbers of promising patterns ($con_{PI_{min}}^u$ affects their numbers). From the results (Fig. 12), we observe that for low values of $con_{PI_{min}}^u$, the number of promising patterns increases. From a value of $con_{PI_{min}}^u$, and then, the number of promising patterns remains constant and the EFFE value is maximized. This happens since all enumerated patterns are promising and the same set of patterns is enumerated.

Impact of $\delta$. We measured the numbers of pattern instantiations ($\delta$ affects their numbers)\(^{32}\). From the results (Fig. 13), we observe that for high values of $\delta$, the average number of the instantiations of accepted patterns is drastically decreased and the EFFE value remains high.

8.4.3. Tuning the thresholds. To tune $con_{PI_{min}}$, the end-user should keep constant at one the values of $con_{PI_{min}}^u$ and $\delta$ to avoid their interference in the execution of Hector. If the input schemas are large-sized, the initial value of $con_{PI_{min}}$ should be greater than 0.2 (e.g. [0.2, 0.4]). The end-user should execute Hector for these threshold values (starting with the higher values) to check if many accepted patterns are produced. If that happens, the end-user should re-execute Hector for lower values of $con_{PI_{min}}$. Then, the end-user should continue with $con_{PI_{min}}^u$. To tune $con_{PI_{min}}^u$, the initial value of $con_{PI_{min}}^u$ should be greater than 0.8. Then, the end-user should execute Hector

\(^{32}\)We had set the values of other two thresholds at zero to avoid their interference in the experiment.
for these threshold values (starting with the higher values) to check if many promising patterns are produced. If that happens, the end-user should re-execute Hector for lower values of $\text{con}_{PI_{min}}^u$. Finally, the end-user should tune $\delta$. The initial value of $\delta$ should be medium (e.g. [0.4, 0.6]). The end-user should execute Hector for these threshold values (starting with the higher values) to check
if many pattern instantiations are produced. If that happens, the end-user should re-execute Hector for lower values of $\delta$.

9. CONCLUSIONS AND FUTURE WORK

We proposed an automated generalization process that accepts as input two schemas and returns as output the top-k abstract data-types. To this end, the process enumerates and matches type patterns that have syntactic and syntactic subtyping-relation. The latter is a relaxed version of the LSP subtyping relation. We further enhance the process mechanisms with a pruning and greedy technique for facing the case of large-size schemas that contain many type patterns. We evaluated the effectiveness and the efficiency of the process on two datasets against two representative state-of-the-art approaches. The evaluation results showed (i) high effectiveness of our process in identifying type patterns and mining abstract data-types; (ii) the pruning and greedy techniques significantly increase the efficiency of our process, without reducing its effectiveness.

A future direction of our work is the definition of the notion of type pattern and the underlying algorithms to be independent of the schema language (e.g. JSON\textsuperscript{33}). Another future direction is the (semi-)automated tuning of the thresholds using machine-learning techniques [Zhang and Tsai 2007]. A final direction is the generalization of the underlying algorithms for multiple schemas.

REFERENCES


\textsuperscript{33}http://www.w3schools.com/json

ACM Transactions on the Web, Vol. 00, No. 00, Article 00, Publication date: 00.


Mining Abstract XML Data-Types

Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. 2004. Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach. IEEE Transactions on Knowledge and Data Engineering 16, 11 (2004), 1424–1440.
Xifeng Yan, Jiawei Han, and Ramin Afshar. 2003. CloSpan: Mining Closed Sequential Patterns in Large Databases. In SIAM International Conference on Data Mining.
Du Zhang and Jeffrey J. P. Tsai. 2007. Advances in Machine Learning Applications in Software Engineering. IGI Global, Hershey, PA, USA.

ACM Transactions on the Web, Vol. 00, No. 00, Article 00, Publication date: 00.