Mining Abstract XML Data-Types

DIONYSIS ATHANASOPOULOS, Victoria University of Wellington
APOSTOLOS ZARRAS, University of Ioannina

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.

ACM Reference Format:
Dionysis Athanasopoulos, Apostolos Zarras, 2018. Mining Abstract XML Data-Types. *ACM Transactions on the Web* 00, 00, Article 00 (00), 35 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

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 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. Google[1], Amazon[2]) currently make available their resources over the Web as services, exposing the programmable service interfaces[3]. Service interfaces are generally defined as sets of operation signatures, whose input/output data-types are XML schemas[4].

[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

Author's addresses: D. Athanasopoulos, School of Engineering & Computer Science, Victoria University of Wellington, Wellington, New Zealand; A. Zarras, Computer Science & Engineering Department, University of Ioannina, Ioannina, Greece.

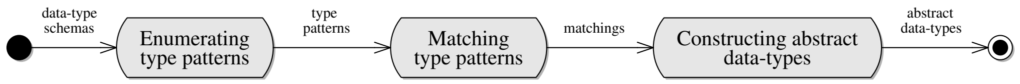


Fig. 1. The proposed generalization process that mines abstract XML data-types.

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. I 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. The one approach is a general-purpose integration approach [Saleem et al. 2008] and the

[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

[8] The schema completeness is the percentage of the source elements that are found in a unified schema [Batini et al. 1986].
[9] The schema minimality is achieved if no redundant concept appears in a unified schema [Batini et al. 1986].
[10] The term HERM stands for the Higher-order Entity-Relationship Model, which is an extension of the flat ER model.
[11] A data model capable of describing various data models, e.g. entity-relationship, relational, hierarchical, etc.

Table I. Summary and comparison of schema integration approaches.

Approaches	Categories	Input	Integration Techniques	Unified-Schema Properties	Efficiency Techniques
[Ma et al. 2005]	conflict-driven	HERM[10]	rule-based	minimality & completeness	x
[Kashyap and Sheth 1996]	conflict-driven	relational	rule-based	minimality & completeness	x
[Baqasah et al. 2014]	conflict-driven	XML	rule-based	minimality & completeness	x
[Parent and Spaccapietra 1998]	conflict-driven	meta-model[11]	rule-based	minimality & completeness	x
[Pottinger and Bernstein 2003]	conflict-driven	meta-model[11]	correspondences	minimality & completeness	x
[Arenas et al. 2010]	constraint-driven	meta-model[11]	mapping-based	minimality & completeness	x
[Li and Quix 2011]	constraint-driven	relational	mapping-based	minimality & completeness	x
[Pottinger and Bernstein 2008]	constraint-driven	relational	query-based	minimality & completeness	x
[Bernstein et al. 2004]	constraint-driven	relational	schema similarity	completeness	x
[Liu and Liu 2012]	matching-driven	WSDL & XML	leaf matching	–	x
[Athanasopoulos et al. 2011]	matching-driven	WSDL & XML	leaf subtyping	generalization	x
[Radwan et al. 2009]	matching-driven	meta-model[11]	context-based	minimality & completeness	x
[Sarma et al. 2008]	matching-driven	relational	element clustering	minimality & completeness	x
[Saleem et al. 2008]	matching-driven	XML	subtree mining	minimality & completeness	greedy mining
[Melnik et al. 2003]	matching-driven	meta-model[11]	subtree mining	minimality & completeness	greedy mining
Our approach	*matching-driven*	*XML*	*subtree mining & type-pattern subtyping*	*generalization (LSP-based subtyping)*	*pruned mining indexing structure greedy matching*

et al. 2006]) are more efficient than those of the first category, since the former directly enumerate subtrees that appear in schemas.

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.

i. *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.
ii. *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.
iii. *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.
iv. *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[12] have been applied in hierarchical (e.g. XML), relational (e.g. SQL [Beaulieu 2009]), and ontology schemas (e.g. OWL[13]). 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.

Approaches	Categories	Input	Type Patterns	Schema Structure Parts	Efficiency Techniques
[Hamdaqa and Tahvildari 2014]	Context-based	XML-based	x	Graph-structured element neighbourhood	x
[Duchateau et al. 2007b]		XML			
[Duchateau et al. 2007c]					
[Meo et al. 2006]					
[Duchateau et al. 2007a]					B-tree structure
[Hu et al. 2008]		OWL, RDFS			Partition-based matching
[Do and Rahm 2002]		XML		Element path, children, leaves	x
[Do and Rahm 2007]		XML, OWL SQL			Partition-based matching
[Madhavan et al. 2001]		XML		Tree leaves	x
[Algergawy et al. 2009]				Element ancestor path, children, leaves	Prüfer sequences
[Lee et al. 2002]				Element children, leaves	x
[Giunchiglia et al. 2004]		XML, OWL		Element descendants	
[Nayak and Iryadi 2007]		XML		Element ancestor	
[Algergawy et al. 2010]				Element ancestor, children, leaves	
[Kim et al. 2011]				Root path	Greedy graph matching
[Meijer 2008]				Element path	x
[Cruz et al. 2009]		XML, OWL RDFS		Element descendants, siblings	
[Jean-Mary et al. 2009]		OWL		Element structural relations	
[Lambrix and Tan 2006]					
[Saleem et al. 2008]	Structure-preserving	XML		Embedded subtree	Greedy subtree mining
[Melnik et al. 2003]		XML, UML SQL, RDF		Induced subtree	x
[Voigt 2011]				Embedded subtree	
Our approach	*Syntax-conforming*	*XML*	✓	*Embedded subtree (type-pattern subtyping)*	*Pruned subtree mining indexing structure greedy pattern matching*

ture in a *strict manner*, i.e. they match the tree/graph schema structure. We call the approaches of the second category *structure-preserving*. Context-based approaches are not necessarily structure-preserving (e.g. ancestor-descendant node relations may be violated). Moreover, context-based approaches are generally less effective than structure-preserving ones [Saleem et al. 2008]. Certain structure-preserving approaches mine subtree patterns and consequently, are more computationally demanding than the rest approaches of the second category.

Regarding the second criterion, the (time and space) complexity of an algorithm is acceptable if it scales polynomially with the size of the algorithm input (in our case, the number of schemas) [Papadimitriou 1994]. The algorithm efficiency in structure-preserving (esp. pattern-based) approaches is important, since they are computationally demanding.

2.2.1. Context-based matching 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.

Basic Type-Patterns		XML Compositors/Components
Categories	**Sub-categories**	
has-as-attributes	has-ordered-attributes	sequence
	has-one-attribute	choice
	has-unordered-attributes	all
group-of-attributes	group-of-ordered-attributes	sequence
	group-of-one-attribute	choice
	group-of-unordered-attributes	all
specialization	complexType extension	complexContent & extension
	complexType restriction	complexContent & restriction
	simpleType extension	simpleContent & extension
	simpleType restriction	simpleContent & restriction
	built-in type restriction	simpleType & restriction
is-a	complexType inheritance	element
list	built-in type list	simpleType & list
union	simpleType/built-in type union	simpleType & union

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 represention model (using the convention `instance:Type`).

[Schema]:	$S := \Big(id : \texttt{anyURI}^{15}, e : E\Big)$	(1)
[Element]:	$E := \Big(oe : OE, scope : SCOPE, children\Big) \,\vert\, children = \{e_i : E\} \wedge e_i.oe.kind \leq e_{i+1}.oe.kind$	(2)
[Original element]:	$OE := \Big(kind : \texttt{String}, label : LABEL\Big)$	(3)
[Element label]:	$LABEL := \Big\{\Big(a : \texttt{String}, v : \texttt{anyType}^{15}\Big)\Big\}$	(4)
[Element scope]:	$SCOPE := \Big[left \in \mathbb{N}, right \in \mathbb{N}\Big]$	(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 `PAE` and instantiations `PIs`. 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

```
<xsd:complexType name="StudiesInfo1">
  <xsd:sequence>
    <xsd:element name="advisor" type="xsd:string"/>
    <xsd:element name="id" type="xsd:integer"/>
    <xsd:element name="Grades">
      <xsd:simpleType>
        <xsd:list itemType="xsd:integer"/>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

(a) `StudiesInfo1`

```
<xsd:complexType name="Student">
  <xsd:sequence>
    <xsd:element name="id" type="xsd:integer"/>
    <xsd:element name="advisor" type="xsd:string"/>
    <xsd:element name="myEvaluation" type="Evaluation"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:simpleType name="Evaluation">
  <xsd:list itemType="xsd:integer"/>
</xsd:simpleType>
<xsd:complexType name="StudiesInfo2">
  <xsd:complexContent>
    <xsd:extension base="Student"/>
  </xsd:complexContent>
</xsd:complexType>
```

(b) `StudiesInfo2`

```
<xsd:complexType name="Student">
  <xsd:sequence>
    <xsd:element name="id" type="xsd:integer"/>
    <xsd:element name="supervisor" type="xsd:string"/>
    <xsd:element name="myGrades" type="Grades"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:simpleType name="Grades">
  <xsd:list itemType="xsd:integer"/>
</xsd:simpleType>
<xsd:complexType name="Committee">
  <xsd:sequence>
    <xsd:element name="member" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="PersonalInfo">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="Student"/>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
<xsd:complexType name="StudiesInfo3">
  <xsd:sequence>
    <xsd:element ref="PersonalInfo"/>
    <xsd:element name="myCommittee" type="Committee"/>
  </xsd:sequence>
</xsd:complexType>
```

(c) `StudiesInfo3`

Fig. 2. The XML specification of the three schemas of our running example.

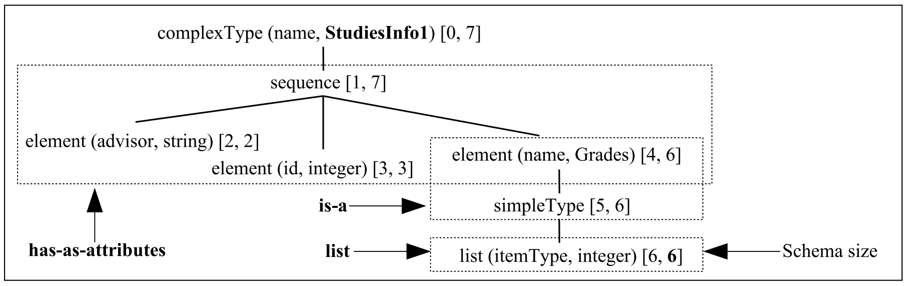
in a schema, forming a syntactically-valid embedded-subtree of the schema tree. The instantiations of a pattern are indexed by their schema `id` (Table V (Eq. 2)).

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 `PAE` (Table V (Eq. 3)) is characterized by its `kind`[16], `scope`, and set of `children`. Abstract elements with a parent-child relation correspond to valid combinations of XML elements (verified by the function `isContent` (Table V (Eq. 4))).

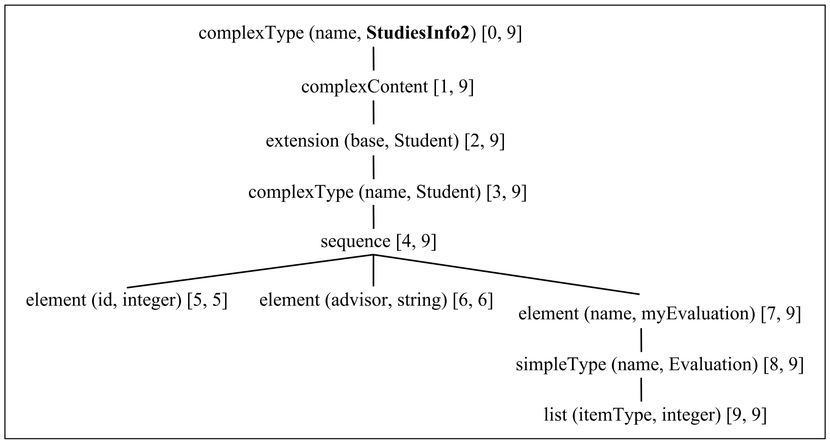
Pattern instantiation. We also model a pattern instantiation `PI` (Table V (Eq. 5)) as an ordered labelled tree, whose nodes are concrete elements (a.k.a. they have labels). A concrete element `PCE` (Table V (Eq. 6)) is actually a schema element `e` and is further characterized by its `scope` and set 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).

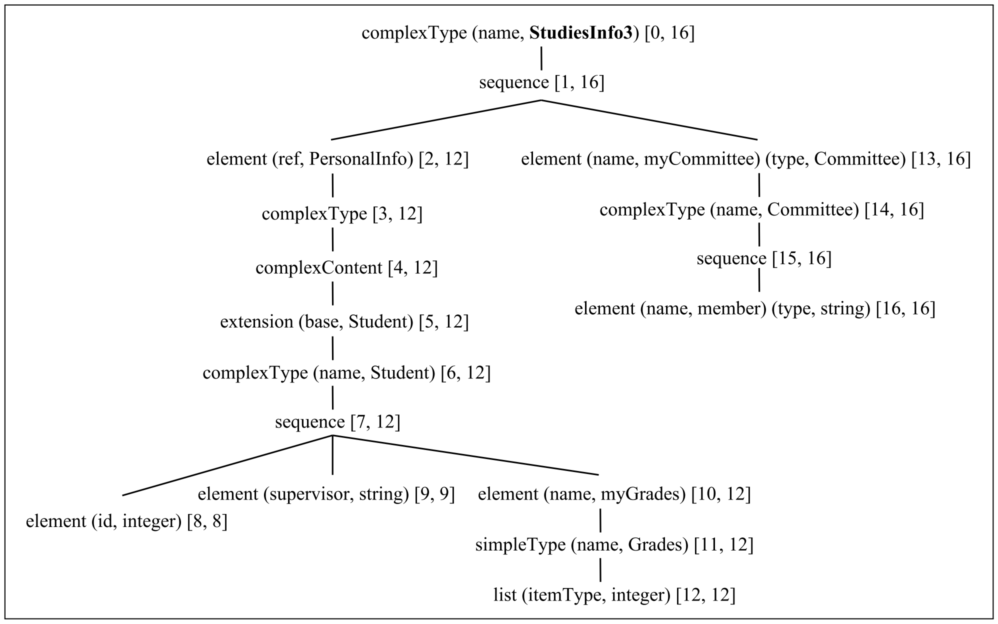
[16] Our process defines a linear (esp. alphabetical) order on the kinds of the children of an abstract element (Table V (Eq. 3)).
[17] Two trees are isomorphic if there is a mapping between the nodes and edges of the trees [Valiente 2002].


complexType (name, StudiesInfo1) [0, 7]
sequence [1, 7]
element (advisor, string) [2, 2]
element (id, integer) [3, 3]
element (name, Grades) [4, 6]
is-a
simpleType [5, 6]
has-as-attributes
list
list (itemType, integer) [6, 6]
Schema size

(a) StudiesInfo1

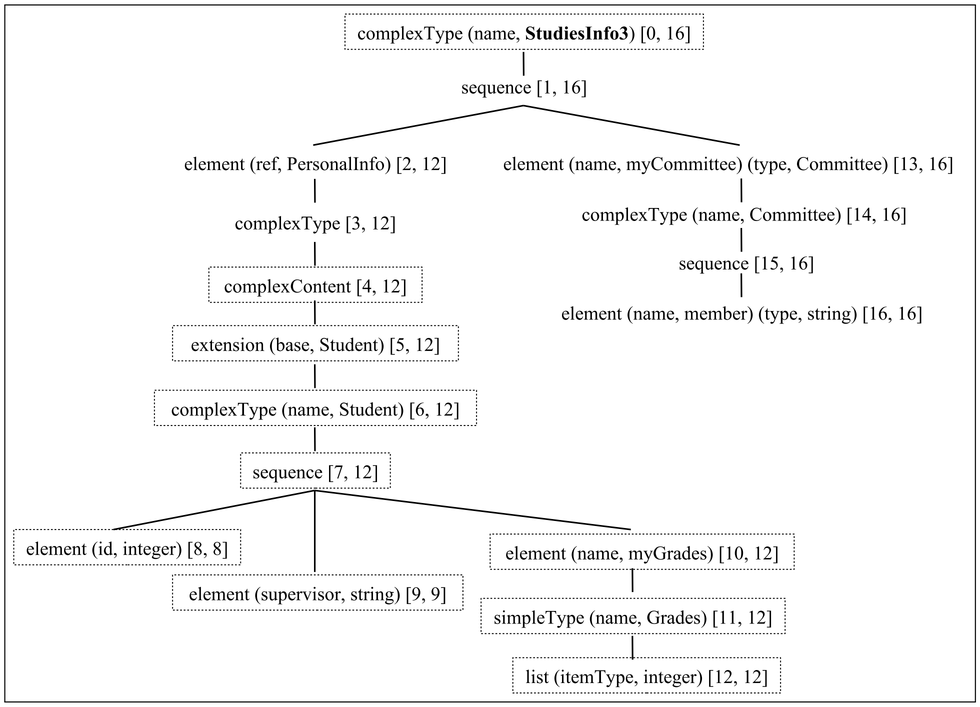

complexType (name, StudiesInfo2) [0, 9]
complexContent [1, 9]
extension (base, Student) [2, 9]
complexType (name, Student) [3, 9]
sequence [4, 9]
element (id, integer) [5, 5]
element (advisor, string) [6, 6]
element (name, myEvaluation) [7, 9]
simpleType (name, Evaluation) [8, 9]
list (itemType, integer) [9, 9]

(b) StudiesInfo2

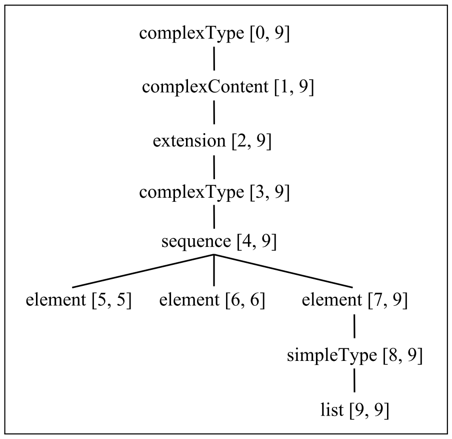

complexType (name, StudiesInfo3) [0, 16]
sequence [1, 16]
element (ref, PersonalInfo) [2, 12]
element (name, myCommittee) (type, Committee) [13, 16]
complexType [3, 12]
complexType (name, Committee) [14, 16]
complexContent [4, 12]
sequence [15, 16]
element (name, member) (type, string) [16, 16]
extension (base, Student) [5, 12]
complexType (name, Student) [6, 12]
sequence [7, 12]
element (supervisor, string) [9, 9]
element (name, myGrades) [10, 12]
element (id, integer) [8, 8]
simpleType (name, Grades) [11, 12]
list (itemType, integer) [12, 12]

(c) StudiesInfo3

Fig. 3. The representation of the schemas of the running example using our model.


complexType (name, **StudiesInfo3**) [0, 16]
sequence [1, 16]
element (ref, PersonalInfo) [2, 12]
element (name, myCommittee) (type, Committee) [13, 16]
complexType [3, 12]
complexType (name, Committee) [14, 16]
sequence [15, 16]
complexContent [4, 12]
element (name, member) (type, string) [16, 16]
extension (base, Student) [5, 12]
complexType (name, Student) [6, 12]
sequence [7, 12]
element (id, integer) [8, 8]
element (name, myGrades) [10, 12]
element (supervisor, string) [9, 9]
simpleType (name, Grades) [11, 12]
list (itemType, integer) [12, 12]

(a) The pattern instantiation that is encountered inside `StudiesInfo3`.



(b) The pattern structure

Fig. 4. An instantiation and the structure of the pattern mined from the schemas of Fig. 3 (b) and (c).

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 `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]: $$P := \Big(pae : PAE, pis : PIs\Big) \quad (1)$$

[Pattern instantiations]: $$PIs := \Big\{\Big(s.id : \texttt{URI}, \{pi : PI\}\Big)\Big\} \quad (2)$$

[Abstract element]: $$PAE := \Big(kind : \texttt{String}, scope : SCOPE, children\Big) \Big| \, children = \Big\{pae_i : PAE \mid isContent(kind, pae_i.kind) \wedge pae_i.kind \leq pae_{i+1}.kind\Big\} \quad (3)$$

$$isContent\Big(kind_1 : \texttt{String}, kind_2 : \texttt{String}\Big) := \begin{cases} \texttt{true} & \text{if } \big(kind_1, kind_2\big) \text{ is a valid combination} \\ \texttt{false} & \text{otherwise} \end{cases} \quad (4)$$

[Pattern instantiation]: $$PI := pce : PCE \quad (5)$$

[Concrete element]: $$PCE := \Big(e : E, scope : SCOPE, children\Big) \Big| \, children = \Big\{pce_i : PCE\Big\} \quad (6)$$

Table VI. The definition of the notion of pattern-instantiation matching.

[Pattern-instantiation matching]: $$M := \{c_i : C\} \quad (1)$$

[Element correspondence]: $$C := \Big(pce_1 : PCE, pce_2 : PCE\Big) \Big| \, pce_1.e.oe.kind = pce_2.e.oe.kind \wedge sim_L\Big(pce_1.e.oe.label,\ pce_2.e.oe.label\Big) > 0 \wedge \Big(inC\big(pce_1, pce_2\big) \oplus outC\big(pce_1, pce_2\big)\Big)^{18} \quad (2)$$

[Internal-element correspondence]: $$inC\Big(pce_1 : PCE, pce_2 : PCE\Big) := pce_1.children \neq \emptyset \wedge pce_2.children \neq \emptyset \wedge pce_1.scope.left = pce_2.scope.left \quad (3)$$

[Leaf-element correspondence]: $$outC\Big(pce_1 : PCE, pce_2 : PCE\Big) := pce_1.children = \emptyset \wedge pce_2.children = \emptyset \quad (4)$$

labels), and have the same left-endpoint in their scopes (if the elements are internal tree-elements – see relation `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 `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 con_L (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 `AT` (Table VII (Eq. 1)) is characterized by its schema `as`, pattern `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 `isIdentical` in Table VII (Eq. 2)).

[18] We use the symbol $\oplus$ to denote the XOR operarion.

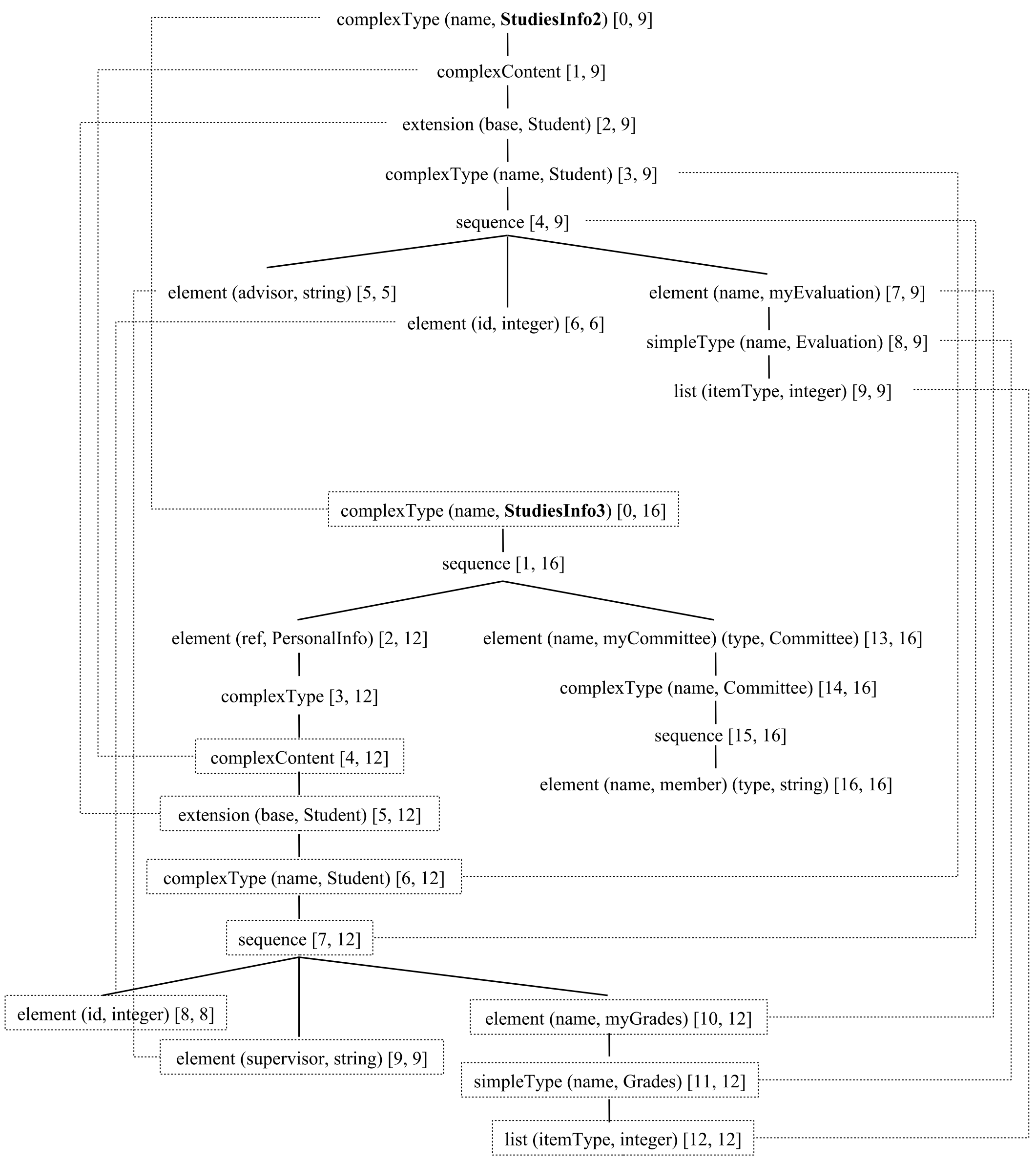

complexType (name, **StudiesInfo2**) [0, 9]
complexContent [1, 9]
extension (base, Student) [2, 9]
complexType (name, Student) [3, 9]
sequence [4, 9]
element (advisor, string) [5, 5]
element (id, integer) [6, 6]
element (name, myEvaluation) [7, 9]
simpleType (name, Evaluation) [8, 9]
list (itemType, integer) [9, 9]
complexType (name, **StudiesInfo3**) [0, 16]
sequence [1, 16]
element (ref, PersonalInfo) [2, 12]
element (name, myCommittee) (type, Committee) [13, 16]
complexType [3, 12]
complexType (name, Committee) [14, 16]
sequence [15, 16]
complexContent [4, 12]
element (name, member) (type, string) [16, 16]
extension (base, Student) [5, 12]
complexType (name, Student) [6, 12]
sequence [7, 12]
element (id, integer) [8, 8]
element (name, myGrades) [10, 12]
element (supervisor, string) [9, 9]
simpleType (name, Grades) [11, 12]
list (itemType, integer) [12, 12]

Fig. 5. The matching between the pattern instantiations that exist in the schemas of Fig. 3 (b) and (c).

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.

Table VII. The definition of the notions of abstract data-type and abstract-to-source matching.

[Abstract data-type]: $$AT := \Big(as : S,\ p : P,\ \{m_i : M_A\}\Big)\ \Big|\ isIdentical\Big(as.e, p.pae\Big) \quad (1)$$

$$isIdentical\big(ae : E, pae : PAE\big) := ae.oe.kind = pae.kind\ \wedge\ ae.scope = pae.scope\ \wedge$$
$$isIdentical\big(e_i, pae_i\big),\ \forall e_i \in \Big[1, |ae.children|\Big]\ \wedge\ pae_i \in \Big[1, |pae.children|\Big] \quad (2)$$

[Abstract-to-source matching]: $$M_A := \{c_i : C_A\} \quad (3)$$

[Abstract-to-source element correspondence]: $$C_A := \Big(ae : E, pce : PCE\Big)\ \Big|\ ae.oe.kind = pce.e.oe.kind\ \wedge$$
$$sim_L\Big(ae.oe.label,\ pce.e.oe.label\Big) > 0\ \wedge\ \Big(inC\big(ae, pce\big)\ \oplus\ outC\big(ae, pce\big)\Big)^{18} \quad (4)$$

[Internal-element correspondence]: $$inC\Big(ae : E, pce : PCE\Big) := ae.children \neq \emptyset\ \wedge$$
$$pce.children \neq \emptyset\ \wedge\ ae.scope.left = pce.scope.left \quad (5)$$

[Leaf-element correspondence]: $$outC\Big(ae : E, pce : PCE\Big) := ae.children = \emptyset\ \wedge$$
$$pce.children = \emptyset \quad (6)$$

output) data-types, compared to those of its super-type. Behavioural requirements are related to the pre/post-conditions, invariants, and history constraints of types. Since behavioural requirements are not usually documented in service descriptions, we consider only signature requirements. Based on signature requirements, an LSP-based subtyping relation is related to the number, structure, and syntax of constituent data-types (a.k.a syntactic subtyping), along with the generalization/specialization relation of the labels of the data-type elements (a.k.a. semantic subtyping).

Syntactic subtyping. We define that an XML schema is syntactic subtype ($<:_{syn}$) 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 `s` is syntactic subtype of schema `as` of an abstract data-type (mined from pattern `p`), if each instantiation `pi` of `p` in `s` contains at least the same kinds and numbers of components and compositors (Table VIII (Eq. 2)). The syntactic-subtyping relation is checked by traversing the tree structures of `pi` and `p` in parallel (Table VIII (Eq. 2-3)).

Semantic subtyping. We define that an XML schema is semantic subtype ($<:_{sem}$) 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 `s` is a semantic subtype of schema `as` of an abstract data-type (mined from pattern `p`), if the element labels of each instantiation `pi` of `p` in `s` are more specific than those of the abstract data-type (Table VIII (Eq. 5)). The semantic-subtyping relation is checked by traversing the tree structures of `s` and `as` in parallel (Table VIII (Eq. 5-6)).

Subtyping relation. Overall, we define that an XML schema is subtype ($<:$) 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

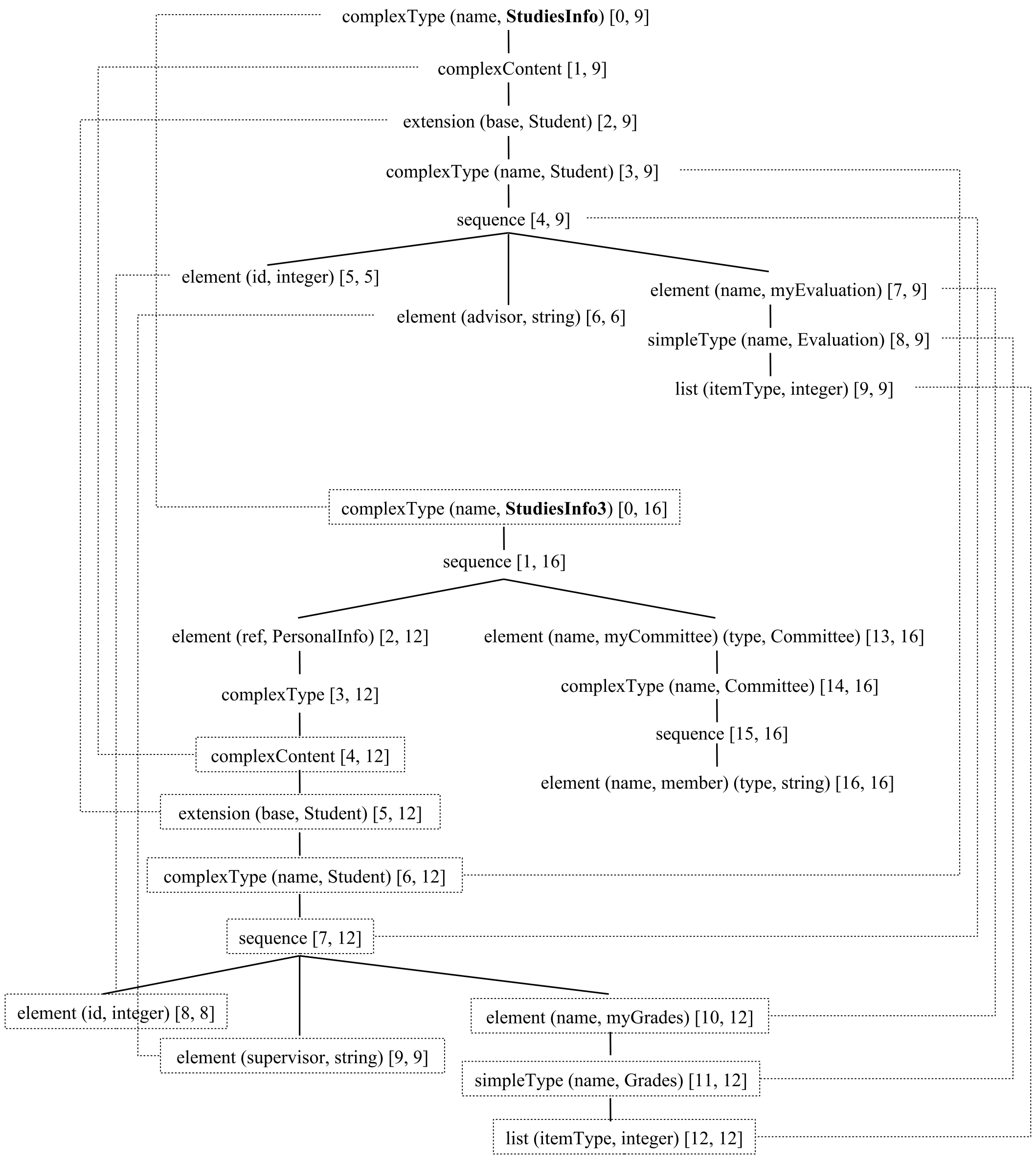

complexType (name, **StudiesInfo**) [0, 9]
complexContent [1, 9]
extension (base, Student) [2, 9]
complexType (name, Student) [3, 9]
sequence [4, 9]
element (id, integer) [5, 5]
element (advisor, string) [6, 6]
element (name, myEvaluation) [7, 9]
simpleType (name, Evaluation) [8, 9]
list (itemType, integer) [9, 9]
complexType (name, **StudiesInfo3**) [0, 16]
sequence [1, 16]
element (ref, PersonalInfo) [2, 12]
element (name, myCommittee) (type, Committee) [13, 16]
complexType [3, 12]
complexType (name, Committee) [14, 16]
sequence [15, 16]
element (name, member) (type, string) [16, 16]
complexContent [4, 12]
extension (base, Student) [5, 12]
complexType (name, Student) [6, 12]
sequence [7, 12]
element (id, integer) [8, 8]
element (supervisor, string) [9, 9]
element (name, myGrades) [10, 12]
simpleType (name, Grades) [11, 12]
list (itemType, integer) [12, 12]

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.

[Subtyping relation]: $s : S \ <: as : S \ := \ s <:_{syn} as \ \wedge \ s <:_{sem} as \,\big|\, as \in at \ \wedge \ at \in AT$ (1)

[Syntactic subtype]: $s : S \ <:_{syn} \ as : S \ := \ pi.pce.e \ <:_{syn} \ as.e, \ \forall (s, pi) \in at.p.pis \ \wedge \ at \in AT$ (2)

$$e : E \ <:_{syn} \ ae : E \ := \ e.kind = ae.kind \ \wedge \ e_i \ <:_{syn} \ e_j \ \wedge \ isContent\big(e.oe.kind, e_i.oe.kind\big) \ \wedge \ isContent\big(ae.oe.kind, e_j.oe.kind\big), \forall e_i \in \Big[1, |e.children|\Big] \ \wedge \ e_j \in \Big[1, |ae.children|\Big] \ \wedge \ i = j \tag{3}$$

$$isContent\Big(kind_1 : \texttt{String}, kind_2 : \texttt{String}\Big) := \begin{cases} \texttt{true} & \text{if } \big(kind_1, kind_2\big) \text{ is a valid combination} \\ \texttt{false} & \text{otherwise} \end{cases} \tag{4}$$

[Semantic subtype]: $s : S \ <:_{sem} \ as : AS \ := \ pi.pce.e \ <:_{sem} \ as.e, \ \forall (s, pi) \in at.p.pis \ \wedge \ at \in AT$ (5)

$$e : E \ <:_{sem} \ ae : E \ := \ sim_L\Big(e.oe.label, \ ae.oe.label\Big) > 0 \ \wedge \ e_i \ <:_{sem} \ e_j, \ \forall \, e_i \in \Big[1, |e.children|\Big] \ \wedge \ e_j \in \Big[1, |ae.children|\Big] \ \wedge \ \big(e_i, e_j\big) \in at.M_A.C_A \ \wedge \ at \in AT \tag{6}$$

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, $con_{PI_{min}}$, $con^{u}_{PI_{min}}$, and δ (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 `IAEs` (Table IX (Eq. 1)) via traversing the input schemas (Alg. 1 (2)) and organizing their elements (Alg. 1 (3)) into groups of the same kind (Alg. 1 (4)). Each group is indexed by an abstract element `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 `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 `AP` (Accepted Patterns) that keeps all enumerated patterns (Alg. 1 (8)). The algorithm further inserts the pattern into the list `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-*

ALGORITHM 1: Enumerating Type Patterns

```
Input: List<$S$> $l$, double $con_{PI_{min}}$, double $con^{u}_{PI_{min}}$, double $\delta$
Output: List<$P$> $AP$

//Phase 1: Initialization
1   $IAEs\ iaes \leftarrow$ new $IAEs()$
2   for $s_i \in l$ do
3       for $e_j \in s_i$ do
4           $iae_x \leftarrow iaes$.identify$(e_j.oe.kind)$
5           $iae_x$.insert$(e_j)$
6   $p \leftarrow new\ P()$
7   $p.pae.kind \leftarrow$ "$root$"
8   $AP$.insert$(p)$
9   $UP$.insert$(p)$
//Phase 2: Pattern enumeration
10  for $p_i \in UP$ do
11      $UP$.remove$(p_i)$
12      List<$PAE$> $rightmostPath \leftarrow p_i$.determineRightmostPath()
13      List<$IAE$> $qaes \leftarrow p_i$.formQueryingAbstractElements$(rightmostPath)$
14      for $qae_j \in qaes$ do
15          $IAEs\ gae \leftarrow iaes$.lookUpGAEs$(qae_j.kind)$
16          for $iae_k \in gae$ do
17              $GCE\ gce \leftarrow iaes$.lookUpGCEs$(qae_j.iae, iae_k)$
18              $P\ p_i^{new} \leftarrow$new $P(p_i)$
19              $p_i^{new}$.insert$(iae_k)$
20              $p_i^{new}$.insert$(gce)$
21              if $con_{PIs}(p_i^{new}.pis) \geq con_{PI_{min}}$ then
22                  $AP$.insert$(p_i^{new})$
23                  $UP$.insert$(p_i^{new})$
24              else
25                  if $con^{u}_{PIs}(p_i^{new}.pis) \geq con^{u}_{PI_{min}}$ then $UP$.insert$(p_i^{new})$;
26              for $s_v \in p_i^{new}.pis$ do
27                  for $pce_m \in p_i^{new}.pis.pi_v$ do
28                      if $den_{PCE}(pce_m) > maxDen$ then $maxDen \leftarrow den_{PCE}(pce_m)$;
29                  for $pce_m \in p_i^{new}.pis.pi_v$ do
30                      if $maxDen - den_{PCE}(pce_m) > \delta$ then $p_i^{new}.pis.pi_v$.remove$(pce_m)$;
```

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

Table IX. The definition of the indexing data-structure.

[**Indexing abstract-elements**]:	$IAEs := \Big\{iae_i : IAE\Big\}$	(1)
[**Indexing abstract-element**]:	$IAE := \Big(kind : \texttt{String}, ices : ICEs\Big)$	(2)
[**Indexing concrete-elements**]:	$ICEs := \Big\{\Big(s.id : \texttt{URI}, ice : ICE\Big)\Big\}$	(3)
[**Indexing concrete-element**]:	$ICE := \Big(e : E, children\Big) \Big\vert\, children = \{ice_i : ICE\}$	(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 $con_{PI_{min}}$ (Alg. 1 (21)). In this case, the pruning technique inserts the pattern in the lists `AP` (`A`ccepted `P`atterns) and `UP` (`U`nexamined-for-growth `P`atterns) (Alg. 1 (22-23)). `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 $con^{u}_{PI_{min}}$ (Alg. 1 (25)). If the upper-bound is greater than $con^{u}_{PI_{min}}$, then the pruning technique inserts the pattern in `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 δ, then the pruning technique does not grow the pattern instantiation (Alg. 1 (26-30)).

5.3. Metric of Pattern Confidence

We propose the metric con_{PIs} (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 den_{PI} (Table XI (Eq. 2)) that calculates the maximum density of pattern instantiations in a schema. We further propose the metric den_{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 *internal* and their number *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 *numOfGEs*[19] (Table XI (Eq. 5)) of these elements. Analogously to the pattern-confidence metrics, we calculate their upper-bounds by using the metrics con^{u}_{PIs} (Table XI (Eq. 7)), den^{u}_{PI} (Table XI (Eq. 8)), and den^{u}_{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

[19]The value of *numOfGEs* is calculated in constant time via subtracting the left endpoint of the scope of the rightmost element of a pattern-instantiation tree (Table XI (Eq. 6)) from the size of the schema to which the instantiation belongs.

Table X. The definition of the syntax-growth relation.

$$isSyntaxGrowth\Big(e_i : E, e_g : E\Big) := isContent\big(e_i.oe.kind, e_g.oe.kind\big) \;\wedge\; isGrowth(e_i, e_g) \quad (1)$$

$$isContent\Big(kind_1 : \texttt{String}, kind_2 : \texttt{String}\Big) := \begin{cases} \texttt{true} & \text{if } \big(kind_1, kind_2\big) \text{ is a valid combination} \\ \texttt{false} & \text{otherwise} \end{cases} \quad (2)$$

$$isGrowth\Big(e_i : E, e_g : E\Big) := \begin{cases} \texttt{true} & \text{if } e_g.scope.left \;\in\; e_i.scope \\ \texttt{false} & \text{otherwise} \end{cases} \quad (3)$$

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).

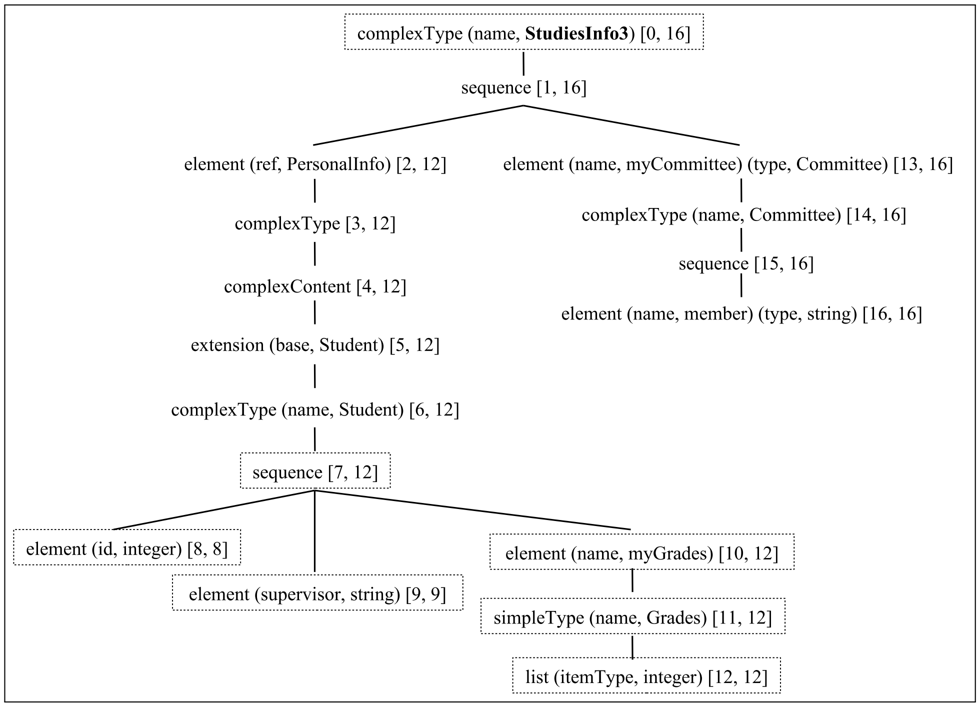
6.1. Algorithm for Matching Type Patterns

Algorithm 2 takes as input a set of accepted patterns $\texttt{AP}$ and the thresholds $\texttt{k}$[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 $\texttt{m}$ (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)).

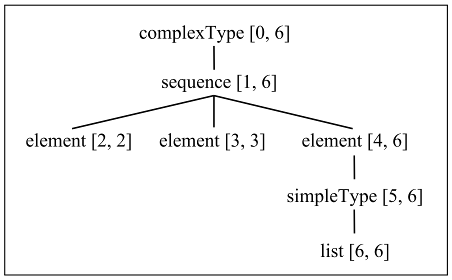
6.2. Greedy Technique

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

[20]The threshold $\texttt{k}$ corresponds to the maximum number of returned matchings.


complexType (name, **StudiesInfo3**) [0, 16]
sequence [1, 16]
element (ref, PersonalInfo) [2, 12]
complexType [3, 12]
complexContent [4, 12]
extension (base, Student) [5, 12]
complexType (name, Student) [6, 12]
sequence [7, 12]
element (id, integer) [8, 8]
element (supervisor, string) [9, 9]
element (name, myGrades) [10, 12]
simpleType (name, Grades) [11, 12]
list (itemType, integer) [12, 12]
element (name, myCommittee) (type, Committee) [13, 16]
complexType (name, Committee) [14, 16]
sequence [15, 16]
element (name, member) (type, string) [16, 16]

(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 $con_{C_{min}}$ (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 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 con_C (Table XII (Eq. 3)), which calculates the semantic confidence of element labels con_L. The metric 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 $con_{C_{min}}$
Output: List<$M$> $Ms$

$Ms \leftarrow \emptyset$
for $p_i \in AP$ do
    $AP$.remove$(p_i)$
    List<Tuple<$PI$>> $cartesianProduct \leftarrow p_i.pis.pi_1$ x $p_i.pis.pi_2$
    for $(pce_i,\ pce_j) \in cartesianProduct$ do
        Queue<Tuple<$PI$>> $Q \leftarrow \emptyset$
        $Q$.insert$(pce_i,\ pce_j)$
        $M\ m \leftarrow \emptyset$
        boolean $f \leftarrow$ true
        while $|Q| > 0\ \wedge\ f = true$ do
            Tuple<$PI$> $(pce_i,\ pce_j) \leftarrow Q$.dequeue()
            $C\ c \leftarrow$formCorrespondence$(pce_i.e,\ pce_j.e)$
            if $con_C(c) > con_{C_{min}}$ then $m$.insert$(mc)$;
            else if $pce_i.children \neq \emptyset\ \wedge\ pce_j.children \neq \emptyset$ then $f \leftarrow$ false;
            if $pce_i.children \neq \emptyset\ \wedge\ pce_j.children \neq \emptyset$ then
                for $(pce_h \in pce_i.children,\ pce_t \in pce_j.children)$ do
                    if $pce_h.scope.left = pce_t.scope.left$ then $Q$.enqueue$(pce_h,\ pce_t)$;
            else
                Tuple<$C$> $correspondences \leftarrow$ selectGreedily$(pce_i.children, pce_j.children)$
                $counter \leftarrow 0$
                for $c_i \in correspondences$ do
                    if $con_C(c_i) > con_{C_{min}}$ then $m$.insert$(c_i)$;
                    else $counter \leftarrow counter + 1$;
                if $counter = |correspondences|$ then $f \leftarrow$ false;
        if $f$ = true $\wedge\ con_M(m, p_i, con_{C_{min}}) > con_M(m_k, p_k, con_{C_{min}})$ then $Ms$.insert$(m)$;
```

Concerning the confidence of pairs of `type` attributes, we consider the case of built-in data-types via using the metric con_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 con_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 con_N (Table XII (Eq. 7)) that calculates the relatedness of `WordNet` concepts [Miller 1995]. con_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 con_N automatically segments them into tokens. The tokenization is based on two widely used naming conventions, the Java-capitalized[21] and the Pascal-underscore[22]. con_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 [Burkard et al. 2009].

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

Time complexity. The time complexity TC_2 (Table XIII (Eq. 3)) of the algorithm scales up with the cardinality of the product of the instantiations $\prod |p.pis.pi_s|$ of accepted patterns AP. The complexity further scales up with the number $|p.pis.pi.pce|$ 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
[22] edn.embarcadero.com/article/10280

Table XI. The definition of the suite of the metrics for pattern confidence (on the absolute scale $[0, 1]$).

[Pattern confidence]: $$con_{PIs}\Big(pis : PIs\Big) := \prod_i den_{PI}\big(pis.pi_i\big) \tag{1}$$

[Density of pattern]: $$den_{PI}\Big(pi : PI\Big) := \max_i den_{PCE}\big(pi.pce_i\big) \tag{2}$$

[Density of pattern instantiation]: $$den_{PCE}\Big(pce : PCE\Big) := \frac{|pce|}{numOfIEs\big(pce\big)} \tag{3}$$

[Internal elements]: $$numOfIEs\Big(pce : PCE\Big) := rightmost\big(pce\big).e.scope.left - pce.e.scope.left + 1 \tag{4}$$

[Growth elements]: $$numOfGEs\Big(s : S, pce : PCE\Big) := |s| - rightmost\big(pce\big).e.scope.left - 1 \tag{5}$$

$$rightmost\Big(pce : PCE\Big) := \begin{cases} pce, & \text{if } pce.children = \emptyset \\ rightmost\big(pce_r\big) \mid \forall pce_i \in pce.children, isGrowth\big(\\ \qquad pce_i.e, pce_r.e\big) = \mathtt{true}, & \text{otherwise} \end{cases} \tag{6}$$

[Upper-bound of pattern confidence]: $$con^u_{PIs}\Big(pis : PIs\Big) := \prod_i den^u_{PI}\big(pis.pi_i\big) \tag{7}$$

[Upper-bound of pattern density]: $$den^u_{PI}\Big(pi : PI\Big) := \max_i den^u_{PCE}\big(pi.pce_i\big) \tag{8}$$

$$den^u_{PCE}\Big(s : S, pce : PCE\Big) := \frac{|pce| + numOfGEs\big(s, pce\big)}{numOfIEs\big(pce\big) + numOfGEs\big(s, pce\big)} \tag{9}$$

sibling leaf-elements. Assuming in the worst case that all leaf elements are siblings to each other[23], the above number is captured by the coefficient $\binom{|leaves|^2}{2}$ (Table XIII (Eq. 3)). Finally, the algorithm complexity further scales up with the complexity of the matching-confidence metric, which depends on the number of the concrete elements of pattern instantiations.

Space complexity. The space complexity SC_2 (Table XIII (Eq. 4)) scales up with the cardinality of the product $\prod |p.pis.pi_s|$ of the instantiations of accepted patterns AP. Additionally, the space complexity scales up with the memory requirements of each pattern-instantiation tuple $\sum |s_i| * space(p.pis.pi.pce)$ and with the memory footprint of the produced matchings $\sum space\big(m_j\big)$. Finally, the space complexity further scales up with the sum of the memory footprint of the produced matchings, $\sum space\big(m_j\big)$.

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`[24]. The algorithm returns a list `ats` of abstract data-types. In detail, the algorithm examines each pair of pattern instantiations (pi_1, pi_2) 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 (pce_1 and pce_2), (ii) a matching `m` between them, (iii) the pattern structure `pae`, and (iv) and 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, $\sum degree(v)$ [Aho et al. 1983].

[24] It indicates whether the patten instantiations have been mined from the schemas of input or output data-types.

Table XII. The definition of the suite of the metrics for matching confidence (on the absolute scale $[0, 1]$).

[Matching]: $$con_M\Big(m : M, p : P,\ con_{C_{min}} : \texttt{double}\Big) := con_{Cs}\big(m,\ con_{C_{min}}\big) \quad (1)$$

[Correspondences]: $$con_{Cs}\Big(m : M,\ con_{C_{min}} : \texttt{double}\Big) := \frac{\sum_{i=1}^{|m|} con_C\big(m.c_i.pce_1.e,\ m.c_i.pce_2.e,\ con_{C_{min}}\big)}{|m|} \quad (2)$$

$$con_C\Big(e_1 : E, e_2 : E, con_{C_{min}} : \texttt{double}\Big) := \begin{cases} con_L\big(e_1.oe.label, e_2.oe.label\big), & \text{if } con_L \geq con_{C_{min}} \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

$$con_L\Big(l_1, l_2\Big) := \begin{cases} 0, & \text{if } \texttt{name}, \texttt{type}, \texttt{value} \notin l_1, l_2 \\ con_N\big(l_1.\texttt{name}, l_2.\texttt{name}\big), & \text{if } \texttt{name} \in l_1, l_2 \ \wedge\ \texttt{type} \notin l_1, l_2 \\ con_T\big(l_1.\texttt{type}, l_2.\texttt{type}\big), & \text{if } \texttt{type} \in l_1, l_2 \ \wedge\ \texttt{name} \notin l_1, l_2 \\ con_N\big(l_1.\texttt{name}, l_2.\texttt{name}\big) * con_T\big(l_1.\texttt{type}, l_2.\texttt{type}\big), & \text{if } \texttt{name}, \texttt{type} \in l_1, l_2 \\ \begin{cases} 1, & \text{if } l_1.\texttt{value} = l_2.\texttt{value} \\ 0, & \text{otherwise} \end{cases}, & \text{if } \texttt{value} \in l_1, l_2 \end{cases} \quad (4)$$

$$con_N\Big(n_1 : \texttt{String}, n_2 : \texttt{String}\Big) := \frac{\sum_{k=1}^{|n_1|} con_N\Big(t_k, f\big(t_k\big)\Big)}{|n_1|} \ \Bigg|\ n_1 = \big\{t_k\big\} \ \wedge\ n_2 = \big\{t_m\big\} \ \wedge\ f\big(t_k\big) \in n_2 \ \wedge\ |n_1| \leq |n_2| \quad (5)$$

[Assignment problem]: $$f : t_k \rightarrow t_m \ \Bigg|\ \sum_{k=1}^{|n_1|} con_N\Big(t_k, f\big(t_k\big)\Big) \text{ is maximized} \quad (6)$$

[Name attributes]: $$con_N\Big(t_k : \texttt{String}, t_m : \texttt{String}\Big) := \begin{cases} Lin\big(t_k, t_m\big), & \text{if } t_k, t_m \in \text{WordNet} \\ 0, & \text{otherwise} \end{cases} \quad (7)$$

abstract data-type, and the set of the abstract-to-source matchings m_a. The function constructs an abstract element (Alg. 3 (13-22)) by mining (i) the common *kind* of the concrete elements (Alg. 3 (13)), a *label* from the labels of the concrete elements (Alg. 3 (20)), a *scope* that has as left endpoint the min of those of the concrete elements (Alg. 3 (21)).

To mine a more generic (resp. specific) label than those of the concrete elements, the function mines a generic (resp. specific) value for each common attribute of the elements (e.g. `name`, built-in `type`). To this end, the function uses the functions `constructAN` and `constructAT`. `constructAN` extracts a `name` as the common hypernym (resp. hyponym) of the compared names using the `WordNet`. `constructAT` considers that all possible built-in `types` are organized in the following type groups: `Integer`, `Real`, `String`, `Date`, and `Boolean`. Having determined a group, `constructAT` extracts as new `type` the input (resp. output) `type` of the elements that has the lowest (resp. highest) depth in the standard XML type hierarchy[25].

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, $|\texttt{p.pis}|*(|\texttt{p.pis}|-1)/2$. 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[23], the above number is captured by the coefficient $\binom{|leaves|^2}{2}$.

[25] www.w3.org/TR/xmlschema-2

Table XIII. The total complexity of the generalization process (without considering the efficiency techniques).

$$TC_1 := \mathcal{O}\Big(|EP| * \sum_{i=1}^{2} |s_i|^2 * \log |s_i|\Big) \tag{1}$$

$$SC_1 := \mathcal{O}\Big(|EP| * \Big(|p.pae| * space(p.pae) * \sum_{i=1}^{2} |s_i| * space(p.pis.pi.pce)\Big)\Big) \Big| \; p \in EP \tag{2}$$

$$TC_2 := \mathcal{O}\Big(|AP| * \prod_{s=1}^{2} |p.pis.pi_s| * \Big(|p.pis.pi.pce| + \binom{|leaves|^2}{2}\Big)\Big) \Big| \; p \in AP \; \wedge \; |leaves| = \mathcal{O}\Big(\sum_{v \in pce} degree(v)\Big) \tag{3}$$

$$SC_2 := \mathcal{O}\Big(|AP| * \Big(\prod_{s=1}^{2} |p.pis.pi_s|\Big) * \sum_{i=1}^{2} |s_i| * space(p.pis.pi.pce) + \sum_{j=1}^{|Ms|} space(m_j)\Big) \Big| \; p \in AP \tag{4}$$

$$TC_3 := \mathcal{O}\Big(\frac{|p.pis| * (|p.pis| - 1)}{2} * \Big(|p.pis.pi.pce| + \binom{|leaves|^2}{2}\Big)\Big) \Big| \; p \in AP \; \wedge \; |leaves| = \mathcal{O}\Big(\sum_{v \in pce} degree(v)\Big) \tag{5}$$

$$SC_3 := \mathcal{O}\Big(\frac{|p.pis| * (|p.pis| - 1)}{2} * \Big(|p.pis.pi.pce| + \binom{|leaves|^2}{2}\Big) * |s| * space(e)\Big) \Big| \; p \in AP \tag{6}$$

Space complexity. The space complexity scales up with the memory footprint of a constructed abstract schema ($|s| * space(e)$). 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 out 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–s_{18} and s_{19}–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

ALGORITHM 3: Constructing Abstract Data-types

```
Input: List<($PI$, $PI$, List<$M$>)> $pisPairs$, $P$ $p$, boolean $isInput$
Output: List<$AT$> $ats$

$ats \leftarrow \emptyset$
for ($pi_1$, $pi_2$, List<$M$>) $\in$ $pisPairs$ do
    for $m_i \in (pi_1, pi_2,$ List<$M$>$)$ do
        $(E\ e, M_A\ m_a) \leftarrow$ construct$(pi_1.pce, pi_2.pce, m_i, p.pae)$
        $AT\ at \leftarrow$new$\Big($new S$\big($new URI(), e$\big)\Big), p, m_a\Big)$
        $ats$.add$(at)$

function construct($PCE\ pce_1$, $PCE\ pce_2$, $M\ m$, $PAE\ pae$, boolean $isInput$): $(E, M_A)$
    List<$E$> $ch \leftarrow \emptyset$
    for ($pce_i$, $pce_j$) $\in$ ($pce_1.children$, $pce_2.children$) do
        if ($pce_i$, $pce_j$) $\in m.C$ then
            $E\ e \leftarrow$ construct$(pce_i, pce_j)$
            $ch$.add$(e)$
    String $kind \leftarrow pce_1.e.kind$
    String $name \leftarrow$ constructAN$(pce_1.e.oe.label.name, pce_2.e.oe.label.name, isInput)$
    String $t \leftarrow$ null
    if $pce_1.e.oe.label.type$ is built-in then
        $t \leftarrow$ constructAT$(pce_1.e.oe.label.type, pce_2.e.oe.label.type, isInput)$
    else
        $t \leftarrow$ constructAN$(pce_1.e.oe.label.type, pce_2.e.oe.label.type, isInput)$
    $LABEL\ label \leftarrow (name, builtinType)$
    $SCOPE\ scope \leftarrow \Big[$min$\big(pce_1.scope.left, pce_2.scope.left\big), pce_1.scope.right\Big]$
    $E\ e \leftarrow (kind, label, scope, children)$
    $m_a$.add$($new $C_A(e, pce_1))$
    $m_a$.add$($new $C_A(e, pce_2))$
    return$(e, m_a)$
```

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 compositors 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.

Dataset	Schema Name	Domain	Schema Size	ID
XBenchMatch	$person_1$	e-commerce	14	s_1
	$person_2$		12	s_2
	$univ\text{-}dept_1$	education	15	s_3
	$univ\text{-}dept_2$		16	s_4
	$univ\text{-}courses_1$		27	s_5
	$univ\text{-}courses_2$		23	s_6
	$biology_1$	biology	1283	s_7
	$biology_2$		533	s_8
	$stock_1$	finance	38	s_9
	$stock_2$		23	s_{10}
	$currency_1$		15	s_{11}
	$currency_2$		62	s_{12}
	$betting_1$		23	s_{13}
	$betting_2$		24	s_{14}
	sms_1	communication	70	s_{15}
	sms_2		104	s_{16}
	$travel_1$	entertainment	11	s_{17}
	$travel_2$		20	s_{18}
Amazon	SQS_1	queue service	290	s_{19}
	SQS_2		145	s_{20}
	SQS_3		187	s_{21}
	SQS_4		259	s_{22}
	$EC2_1$	elastic compute cloud	3312	s_{23}
	$EC2_2$		1589	s_{24}
	$EC2_3$		1697	s_{25}
	$EC2_4$		1798	s_{26}
	VPC_1	virtual private cloud	1923	s_{27}
	VPC_2		1191	s_{28}
	VPC_3		1426	s_{29}
	VPC_4		1697	s_{30}

ble XVI (Eq. 3, 5–6)) calculates the percentage of the true positives over all elements of a mined matching. Elements are considered true (resp. false) positives if they are (resp. are not) present in an expert matching. The recall metric (Table XVI (Eq. 4, 7)) calculates the percentage of the true positives over all elements of an expert matching. Given that `Hector` returns a list of top-k matchings, F-measure is calculated based on the position of a matching in the list (Table XVI (Eq. 1)).

If the GEN value is high, then the $EFFE$ value is expected to be high, since the majority of components, compositors, and links has been correctly matched. However, if the $EFFE$ value is high, then the GEN value may be low, since the numbers of correctly matched compositors and links are unknown. We meet these cases in the results of the effectiveness evaluation (Section 8.2).

8.1.3. Efficiency metrics. To evaluate the efficiency of `Hector`, we measure the execution times[28] 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, RW_1 [Saleem et al. 2008] and RW_2 [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 RW_1 and RW_2 in all schema pairs. It happens since RW_1 and RW_2 do not match compositors and links. Moreover, the `EFFE` values for `Hector`

[28] 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]: $$GEN_{ats} := (at, p, ats) := \begin{cases} 0, & \text{if } at \notin ats \\ GEN_{at} * \frac{|ats|-p+1}{|ats|}, & \text{otherwise} \end{cases} \quad (1)$$

[Abstract data-type]: $$GEN_{at} := \frac{\sum_{i=1}^{|btp|} GEN_{btp_i}}{MAX\Big(|btp|, |btp_{ex}|\Big)} * \frac{\sum_{i=2}^{|btp|} MIN\Big(|ln_{btp_{i-1}}|, |ln_{btp_{i-1_{ex}}}|\Big)}{\sum_{i=2}^{|btp|} MAX\Big(|ln_{btp_{i-1}}|, |ln_{btp_{i-1_{ex}}}|\Big)} \quad (2)$$

[Basic type-pattern]: $$GEN_{btp} := \frac{\big|tor \cap tor_{ex}\big| + \big|nent \cap nent_{ex}\big| + MIN\Big(|ln_e|, |ln_{e_{ex}}|\Big)}{MAX\Big(|tor|, |tor_{ex}|\Big) + MAX\Big(|nent|, |nent_{ex}|\Big) + MAX\Big(|ln_e|, |ln_{e_{ex}}|\Big)} \quad (3)$$

[Compositors and components links]: $ln_e := \sum_{i=1}^{|pairs|} ln_{e_i}$, where $pairs$ is the set of the pairs of compositors and components that are connected with directed links ln_{e_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]: $$EFFE(m, p, M) := \begin{cases} 0, & \text{if } m \notin M \\ F(m) * \frac{|M|-p+1}{|M|}, & \text{otherwise} \end{cases} \quad (1)$$

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

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

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

[True positives]: $$tp := \big|m.C \cap m_{ex}.C\big| \quad (5)$$

[False positives]: $$fp := \big|m.C - m_{ex}.C\big| \quad (6)$$

[False negatives]: $$fn := \big|m_{ex}.C - m.C\big| \quad (7)$$

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_8), 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.

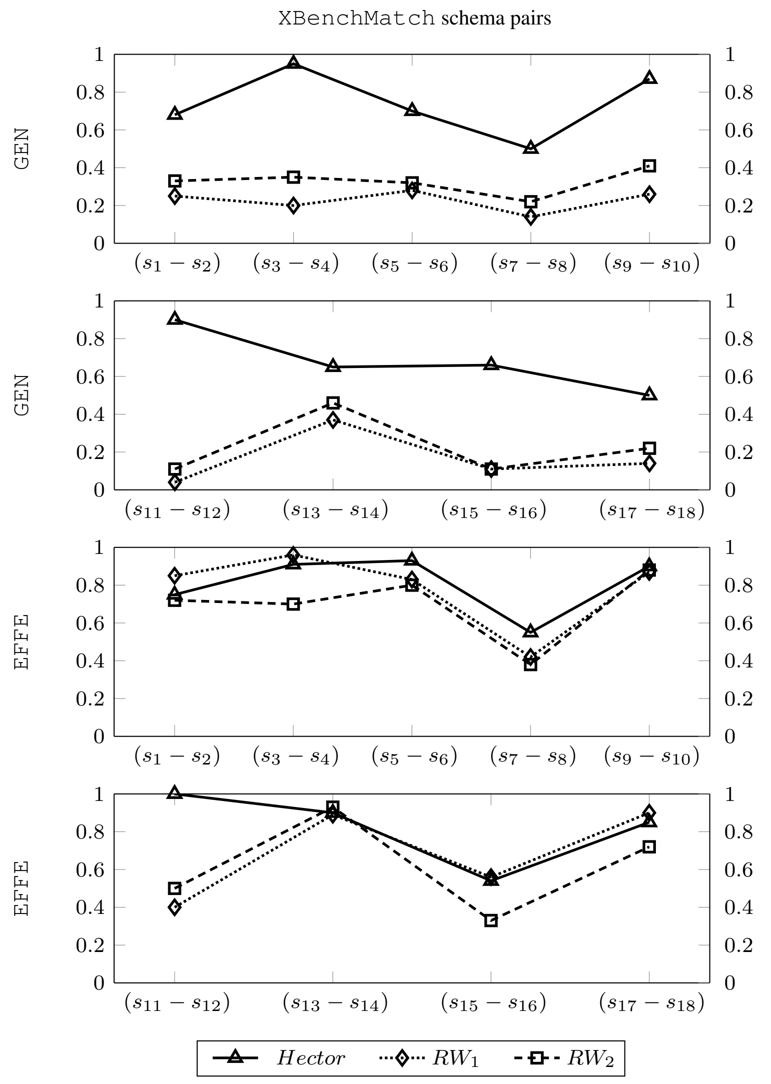


Fig. 8. The effectiveness results on the XBenchMatch schemas using the metrics GEN and EFFE.

the execution times are very low[28] in all schema pairs. We also observe that the execution times of Hector are close to or slightly higher than those of RW_1 and 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 Hector (Sections 8.4.1 and 8.4.2).

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

Execution time. We indicatively present in Table XVII the execution times of Hector for small- and large-sized schemas from both XBenchMatch amd Amazon datasets. We observe from the re-

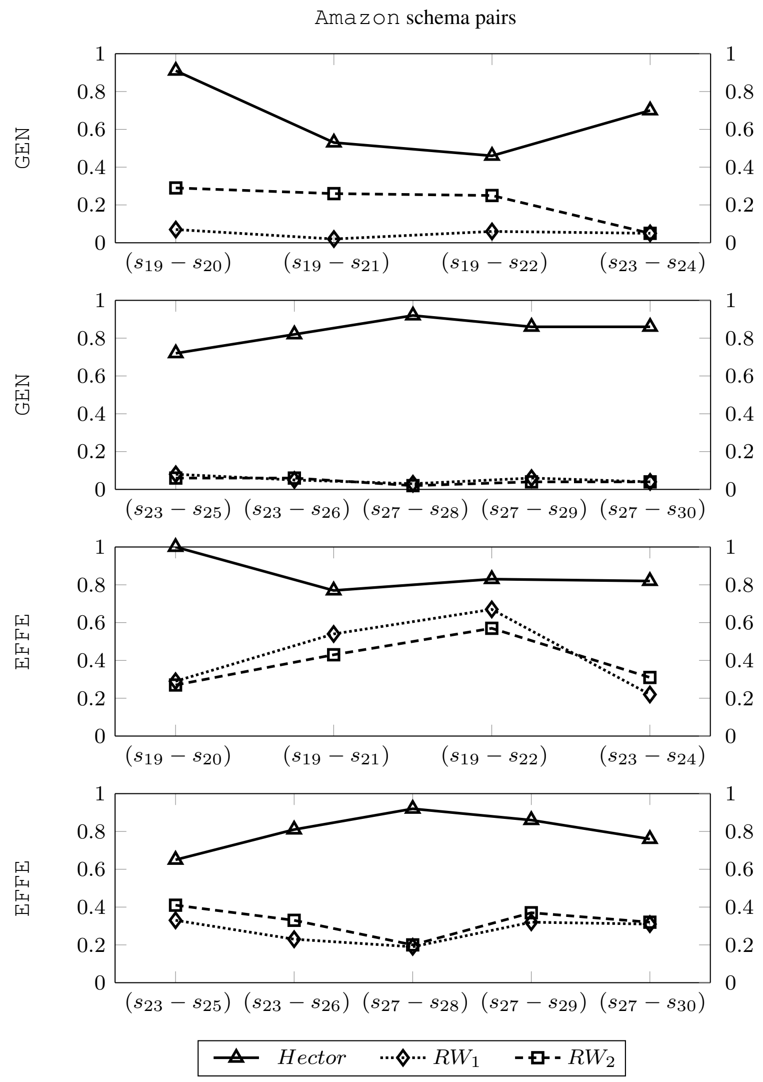


Fig. 9. The effectiveness results on the Amazon schemas using the metrics GEN and EFFE.

sults 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

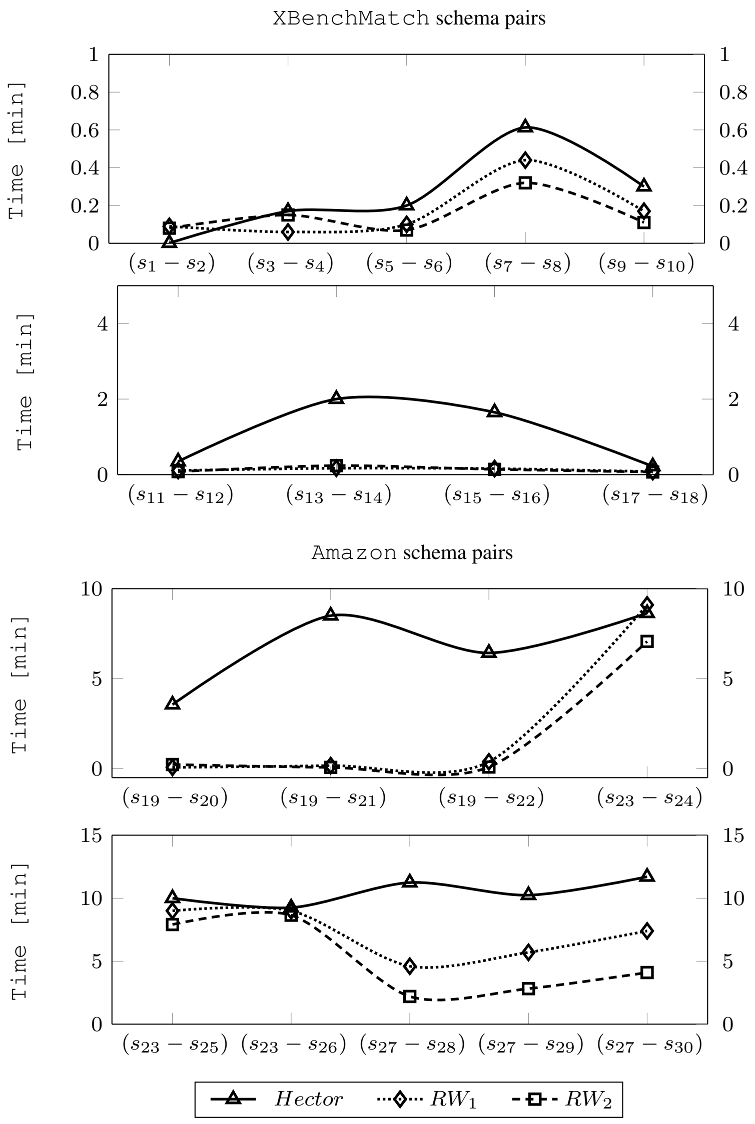


Fig. 10. The execution times of all of the approaches for the XBenchMatch and Amazon schemas.

8.4.2. Impact of the pruning and greedy techniques on the effectiveness of Hector. The techniques depend on the thresholds $con_{PI_{min}}$, $con^u_{PI_{min}}$, and δ. $con_{PI_{min}}$ and $con^u_{PI_{min}}$ are used for characterizing patterns as accepted and promising, respectively. δ is used for pruning pattern instantiations.

Impact of $con_{PI_{min}}$. We measured the numbers of accepted patterns ($con_{PI_{min}}$ affects their numbers)[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 $con^u_{PI_{min}}$ to zero, so that all unaccepted patterns to be promising.

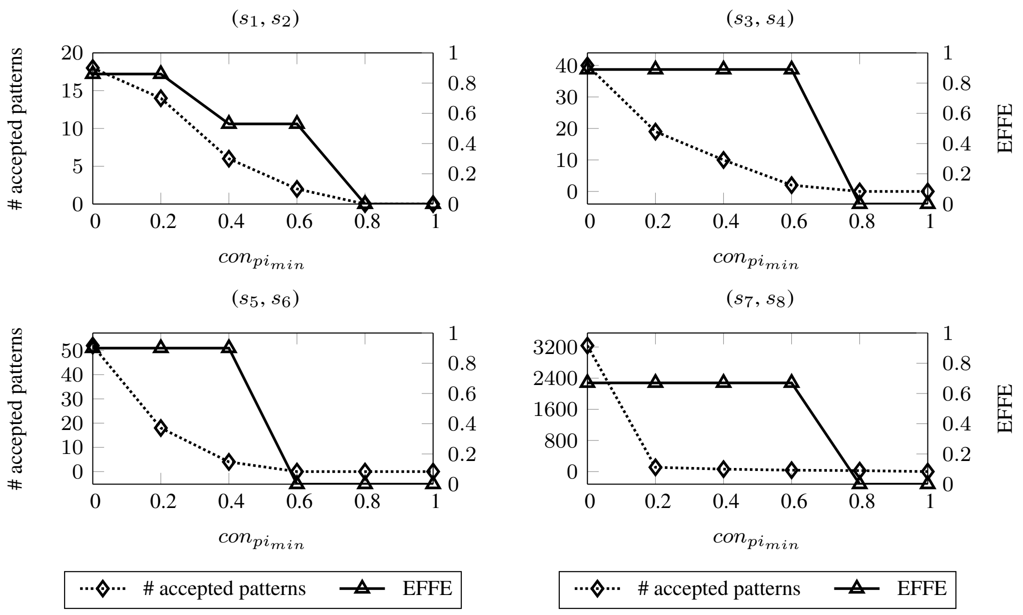


Fig. 11. The impact of $con_{PI_{min}}$ on the effectiveness of Hector for the first four schema pairs of the datasets.

Table XVII. Execution times of Hector in minutes (we have set in bold the values that are not low ($>$ 10 min)).

Dataset	Schema pair	Applying efficiency techniques		Without applying efficiency techniques	
		M_1	M_2	M_1	M_2
XBenchMatch	(s_1, s_2)	0.0006	0.0007	0.0006	0.0007
	(s_7, s_8)	0.3290	0.2840	**46.0043**	**81.3384**
Amazon	(s_{19}, s_{20})	0.3700	3.1900	8.4488	**11.0552**
	(s_{23}, s_{24})	6.3500	2.2000	**67.6166**	**64.4333**

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

Impact of $con^u_{PI_{min}}$. We measured the numbers of promising patterns ($con^u_{PI_{min}}$ affects their numbers). From the results (Fig. 12), we observe that for low values of $con^u_{PI_{min}}$, the number of promising patterns increases. From a value of $con^u_{PI_{min}}$ 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 δ. We measured the numbers of pattern instantiations (δ affects their numbers)[32]. From the results (Fig. 13), we observe that for high values of δ, 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 $con^u_{PI_{min}}$ and δ 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 $con^u_{PI_{min}}$. To tune $con^u_{PI_{min}}$, the initial value of $con^u_{PI_{min}}$ 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.

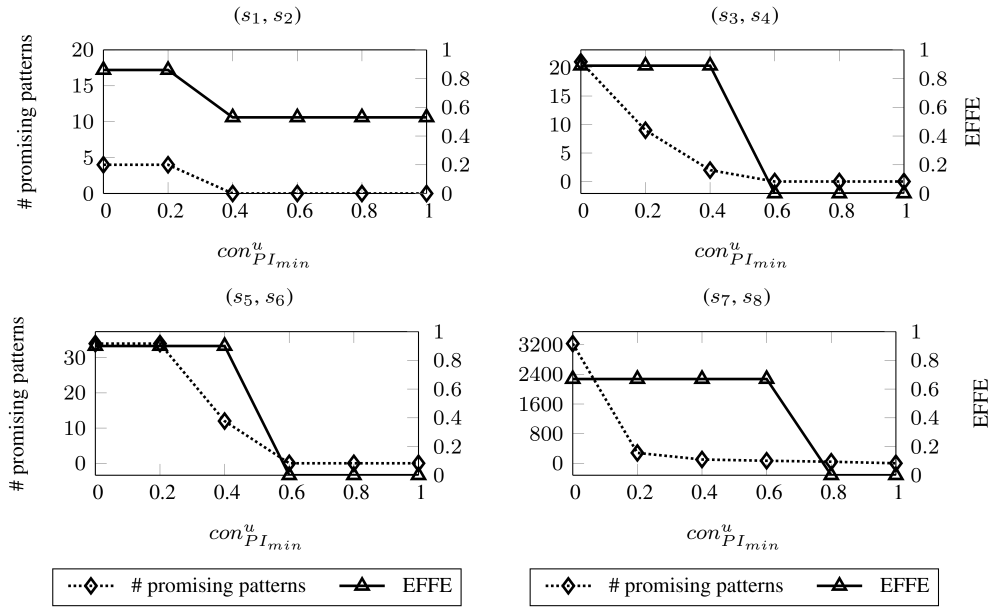


Fig. 12. The impact of $con^u_{PI_{min}}$ on the effectiveness of Hector for the first four schema pairs of the datasets.

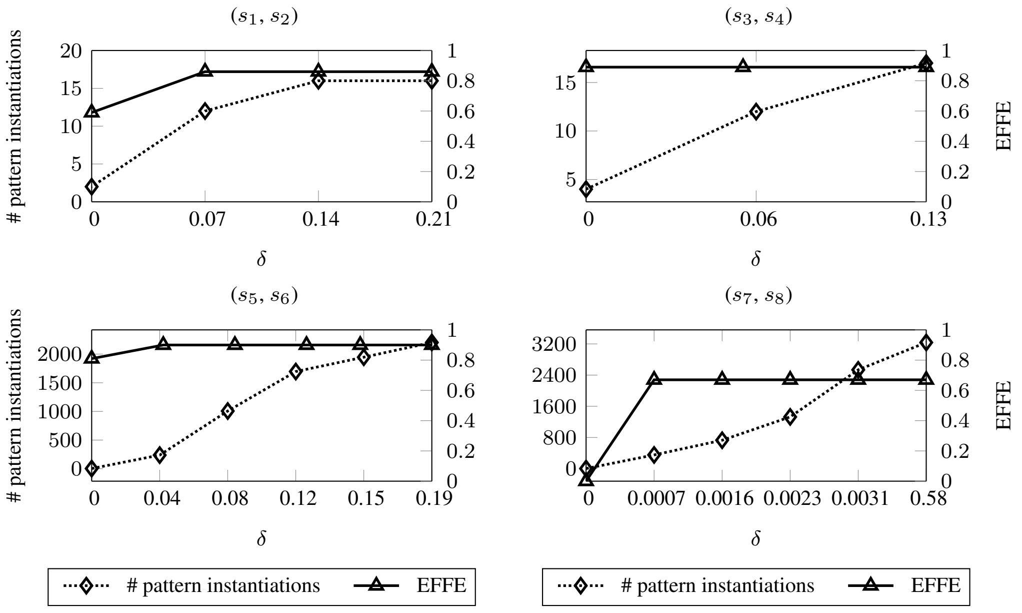


Fig. 13. The impact of δ on the effectiveness of Hector for the first four schema pairs of the datasets.

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 $con_{PI_{min}}$. Finally, the end-user should tune δ. The initial value of δ 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 δ.

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[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

A. V. Aho, J. E. Hopcroft, and J. Ullman. 1983. *Data Structures and Algorithms*. Addison-Wesley.

Alsayed Algergawy, Richi Nayak, and Gunter Saake. 2010. Element Similarity Measures in XML Schema Matching. *Information Sciences* 180, 24 (2010), 4975–4998.

Alsayed Algergawy, Eike Schallehn, and Gunter Saake. 2009. Improving XML Schema Matching Performance Using Prüfer Sequences. *Data and Knowledge Engineering* 68, 8 (2009), 728–747.

M. Arenas, J. Pérez, J. L. Reutter, and C. Riveros. 2010. Foundations of Schema Mapping Management. In *ACM Symposium on Principles of Database Systems*. 227–238.

Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroki Arimura, Hiroshi Sakamoto, and Setsuo Arikawa. 2002. Efficient Substructure Discovery from Large Semi-Structured Data. In *SIAM International Conference on Data Mining*.

D. Athanasopoulos, A. Zarras, P. Vassiliadis, and V. Issarny. 2011. Mining Service Abstractions. In *International Conference on Software Engineering*. 944–947.

D. Aumueller, H. H. Do, S. Massmann, and E. Rahm. 2005. Schema and Ontology Matching with COMA++. In *ACM SIGMOD International Conference on Management of Data*. 906–908.

R. A. Baeza-Yates and B. A. Ribeiro-Neto. 1999. *Modern Information Retrieval*. ACM Press/Addison-Wesley.

A. Baqasah, E. Pardede, and J. W. Rahayu. 2014. A New Approach for Meaningful XML Schema Merging. In *International Conference on Information Integration and Web-based Applications & Services*. 430–439.

Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. 1986. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computings Surveys* 18, 4 (1986), 323–364.

Alan Beaulieu. 2009. *Learning SQL – Master SQL Fundamentals*. O'Reilly.

Zohra Bellahsene, Angela Bonifati, and Erhard Rahm (Eds.). 2011. *Schema Matching and Mapping*. Springer.

P. A. Bernstein, S. Melnik, M. Petropoulos, and C. Quix. 2004. Industrial-Strength Schema Matching. *ACM SIGMOD Record* 33, 4 (2004), 38–43.

P. Bille. 2005. A Survey on Tree Edit Distance and Related Problems. *Theoretical Computer Science* 337, 1-3 (2005), 217–239.

Rainer Burkard, Mauro Dell'Amico, and Silvano Martello. 2009. *Assignment Problems*. Society for Industrial and Applied Mathematics, USA.

Yun Chi, Richard R. Muntz, Siegfried Nijssen, and Joost N. Kok. 2005. Frequent Subtree Mining - An Overview. *Fundamenta Informaticae* 66, 1-2 (2005), 161–198.

J. I. Chowdhury and R. Nayak. 2014. *BEST: An Efficient Algorithm for Mining Frequent Unordered Embedded Subtrees*.

T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. 2001. *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education.

Isabel F. Cruz, Flavio Palandri Antonelli, and Cosmin Stroe. 2009. AgreementMaker: Efficient Matching for Large Real-World Schemas and Ontologies. *VLDB Endowment* 2, 2 (2009), 1586–1589.

[33] http://www.w3schools.com/json

Hong Hai Do and Erhard Rahm. 2002. COMA - A System for Flexible Combination of Schema Matching Approaches. In *International Conference on Very Large Data Bases*. 610–621.

Hong Hai Do and Erhard Rahm. 2007. Matching Large Schemas: Approaches and Evaluation. *Information Systems* 32, 6 (2007), 857–885.

A. Doan and A. Y. Halevy. 2005. Semantic Integration Research in the Database Community: A Brief Survey. *AI Magazine* 26, 1 (2005), 83–94.

F. Duchateau and Z. Bellahsene. 2010. *Measuring the Quality of an Integrated Schema*.

Fabien Duchateau, Zohra Bellahsene, Mark Roantree, and Mathieu Roche. 2007c. Poster Session: An Indexing Structure for Automatic Schema Matching. In *IEEE International Conference on Data Engineering Workshop*. 485–491.

Fabien Duchateau, Zohra Bellahsene, and Mathieu Roche. 2007a. BMatch: a Semantically Context-based Tool Enhanced by an Indexing Structure to Accelerate Schema Matching. In *Journées Bases de Données Avancées*.

Fabien Duchateau, Zohra Bellahsene, and Mathieu Roche. 2007b. A Context-based Measure for Discovering Approximate Semantic Matching between Schema Elements. In *International Conference on Research Challenges in Information Science*. 9–20.

Fabien Duchateau, Mathieu Roche, and Zohra Bellahsene. 2008. Improving Quality and Performance of Schema Matching in Large Scale. *Ingénierie des Systèmes d'Information* 1 (Nov. 2008), 59–82. http://hal-lirmm.ccsd.cnrs.fr/lirmm-00343491

T. Erl. 2005. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall.

Fausto Giunchiglia, Pavel Shvaiko, and Mikalai Yatskevich. 2004. S-Match: an Algorithm and an Implementation of Semantic Matching. In *European Semantic Web Symposium*. 61–75.

A. Y. Halevy, A. Rajaraman, and J. J. Ordille. 2006. Data Integration: The Teenage Years. In *International Conference on Very Large Data Bases*. 9–16.

M. Hamdaqa and L. Tahvildari. 2014. Prison Break: A Generic Schema Matching Solution to the Cloud Vendor Lock-in Problem. In *International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*. 37–46.

Wei Hu, Yuzhong Qu, and Gong Cheng. 2008. Matching Large Ontologies: A Divide-and-Conquer Approach. *Data & Knowledge Engineering* 67, 1 (2008), 140 – 160.

Yves R. Jean-Mary, E. Patrick Shironoshita, and Mansur R. Kabuka. 2009. Ontology Matching with Semantic Verification. *Web Semantics: Science, Services and Agents on the World Wide Web* 7, 3 (2009), 235 – 251.

Aída Jiménez, Fernando Berzal, and Juan Carlos Cubero Talavera. 2010. Frequent Tree Pattern Mining: A Survey. *Intelligent Data Analysis* 14, 6 (2010), 603–622.

V. Kashyap and A. P. Sheth. 1996. Semantic and Schematic Similarities between Database Objects: A Context-Based Approach. *The VLDB Journal* 5, 4 (1996), 276–304.

Jaewook Kim, Yun Peng, Nenad Ivezik, and Junho Shin. 2011. An Optimization Approach for Semantic-based XML Schema Matching. *International Journal of Trade, Economics, and Finance* 2, 1 (2011), 78–86.

Patrick Lambrix and He Tan. 2006. SAMBO – A System for Aligning and Merging Biomedical Ontologies. *Web Semantics: Science, Services and Agents on the World Wide Web* 4, 3 (2006), 196 – 206.

Mong-Li Lee, Liang Huai Yang, Wynne Hsu, and Xia Yang. 2002. XClust: Clustering XML Schemas for Effective Integration. In *ACM International Conference on Information and Knowledge Management*. 292–299.

Xiang Li. 2012. *Constraint-driven Schema Merging*. Ph.D. Dissertation. RWTH Aachen University.

X. Li and C. Quix. 2011. Merging Relational Views: A Minimization Approach. In *International Conference on Conceptual Modeling*. 379–392.

B. Liskov and J. M. Wing. 1994. A Behavioural Notion of Subtyping. *ACM Transactions on Programming Languages and Systems* 16, 6 (1994), 1811–1841.

X. Liu and H. Liu. 2012. Automatic Abstract Service Generation from Web Service Communities. In *International Conference on Web Services*. 154–161.

H. Ma, K.-D. Schewe, B. Thalheim, and J. Zhao. 2005. View Integration and Cooperation in Databases, Data Warehouses and Web Information Systems. (2005), 213–249.

Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. 2001. Generic Schema Matching with CUPID. In *International Conference on Very Large Data Bases*. 49–58.

M.M. Meijer. 2008. On A Method For XML Schema Matching. In *Twente Student Conference on Information Technology*.

Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. 2002. Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching. In *International Conference on Data Engineering*. 117–128.

Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. 2003. Rondo: A Programming Platform for Generic Model Management. In *ACM SIGMOD Conference*. 193–204.

Pasquale De Meo, Giovanni Quattrone, Giorgio Terracina, and Domenico Ursino. 2006. Integration of XML Schemas at Various "Severity" Levels. *Information Systems* 31, 6 (2006), 397–434.

George A. Miller. 1995. WordNet: a Lexical Database for English. *ACM Communications* 38, 11 (1995), 39–41.

Richi Nayak and Wina Iryadi. 2007. XML Schema Clustering with Semantic and Hierarchical Similarity Measures. *Knowledge-Based Systems* 20, 4 (2007), 336–349.

Christos H. Papadimitriou. 1994. *Computational Complexity*. Addison-Wesley.

C. Parent and S. Spaccapietra. 1998. Issues and Approaches of Database Integration. *Commun. ACM* 41, 5 (1998), 166–178.

Ted Pedersen, Siddharth Patwardhan, and Jason Michelizzi. 2004. WordNet: : Similarity – Measuring the Relatedness of Concepts. In *National Conference on Innovative Applications of Artificial Intelligence*. 1024–1025.

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.

P. Plebani and B. Pernici. 2009. URBE: Web Service Retrieval Based on Similarity Evaluation. *IEEE Transactions on Knowledge and Data Engineering* 21, 11 (2009).

R. Pottinger and P. A. Bernstein. 2003. Merging Models Based on Given Correspondences. In *International Conference on Very Large Data Bases*. 826–873.

R. Pottinger and P. A. Bernstein. 2008. Schema Merging and Mapping Creation for Relational Sources. In *International Conference on Extending Database Technology: Advances in Database Technology*.

A. Radwan, L. Popa, I. R. Stanoi, and A. Younis. 2009. Top-k Generation of Integrated Schemas Based on Directed and Weighted Correspondences. In *ACM SIGMOD International Conference on Management of Data*. 641–654.

Erhard Rahm and Philip A. Bernstein. 2001. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal* 10, 4 (2001), 334–350.

Erhard Rahm, Hong Hai Do, and Sabine Massmann. 2004. Matching Large XML Schemas. *SIGMOD Record* 33, 4 (2004), 26–31.

K. Saleem, Z. Bellahsene, and E. Hunt. 2008. PORSCHE: Performance ORiented SCHEma mediation. *Information Systems* 33, 7–8 (2008), 637–657.

A. D. Sarma, X. Dong, and A. Halevy. 2008. Bootstrapping Pay-as-you-go Data Integration Systems. In *ACM SIGMOD International Conference on Management of Data*. 861–874.

Pavel Shvaiko and Jérôme Euzenat. 2013. Ontology Matching: State of the Art and Future Challenges. *IEEE Transactions on Knowledge and Data Engineering* 25, 1 (2013), 158–176.

Eleni Stroulia and Yiqiao Wang. 2005. Structural and Semantic Matching for Assessing Web-Service Similarity. *International Journal of Cooperative Information Systems* (2005), 407–438.

Gabriel Valiente. 2002. *Algorithms on Trees and Graphs*. Springer.

Konrad Voigt. 2011. *Structural Graph-based Metamodel Matching*. Ph.D. Dissertation. Technical University of Dresden, Department of Computer Science.

Chen Wang, Mingsheng Hong, Jian Pei, Haofeng Zhou, Wei Wang, and Baile Shi. 2004. Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining. In *Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*. 441–451.

Xifeng Yan, Jiawei Han, and Ramin Afshar. 2003. CloSpan: Mining Closed Sequential Patterns in Large Databases. In *SIAM International Conference on Data Mining*.

Mohammed Javeed Zaki. 2002. Efficiently Mining Frequent Trees in a Forest. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 71–80.

Mohammed Javeed Zaki. 2005a. Efficiently Mining Frequent Embedded Unordered Trees. *Fundamenta Informaticae* 66, 1–2 (2005), 33–52.

Mohammed Javeed Zaki. 2005b. Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications. *IEEE Transactions on Knowledge and Data Engineering* 17, 8 (2005), 1021–1035.

Du Zhang and Jeffrey J. P. Tsai. 2007. *Advances in Machine Learning Applications in Software Engineering*. IGI Global, Hershey, PA, USA.

L. Zou, Y. Lu, H. Zhang, and R. Hu. 2006. PrefixTreeESpan: A Pattern Growth Algorithm for Mining Embedded Subtrees. In *International Conference on Web Information Systems Engineering*. 499–505.