Dissertation zur Erlangung des akademischen Grades
Dr. rer. soc. oec.
im Doktoratsstudium der Sozial- und Wirtschaftswissenschaften

Keys and Foreign Keys for XML

Design and Reasoning

Angefertigt am

Institut für Wirtschaftsinformatik
Data & Knowledge Engineering
Johannes Kepler Universität Linz

Eingereicht von

Mag. Michael Karlinger

Betreut von

o. Univ.-Prof. Dipl.-Ing. Dr. Michael Schrefl
Prof. M.Eng.Sc. Ph.D. Millist Vincent

Linz, Februar 2010

Abstract

Integrity constraints are the primary means to ensure that data is an accurate representation of reality, which is vital to organizational success in today's economy. The most fundamental types of integrity constraints are keys and foreign keys, irrespective of the data model used. Keys and foreign keys establish meaningful connections between real world entities and their representations in data (data entities) on the basis of entity properties.

With the adoption of the eXtensible Markup Language (XML) as the standard for data exchange over the internet and its increasing usage as format for the permanent storage of data, the importance of studying keys and foreign keys in the XML data model has increased in recent years. The design of XML integrity constraints is challenging because of the hierarchical and semi-structured nature of XML data which allows data entities to have multiple or absent values for an entity property. In previous proposals to XML integrity constraints, multiple or absent property values lead to counter-intuitive results in checking the satisfaction of a constraint in an XML document. In contrast, XML keys (XKeys) and foreign keys (XFKeys) as proposed in this thesis handle multiple or absent property values in a way intuitively expected by the application developer.

It is shown that XKeys and XFKeys preserve the semantics of relational keys and foreign keys when relational data is mapped to XML, as frequently required in data exchange scenarios. Moreover, the consistency and implication problems related to XKeys and XFKeys are discussed in the context of 'complete' XML documents, which generalize complete relations. It is shown that every set of XKeys or XFKeys is consistent, and that there are sound and complete sets of inference rules for both XKey and XFKey implication.

Contents

Chapter 1

Introduction

Contents

This chapter gives a general introduction to the topics of this thesis, starting with the motivation in Section 1.1. Section 1.2 describes its main challenges and Section 1.3 reviews related work. Section 1.4 introduces the objectives pursued in this thesis together with the proposed approach and the main contributions. Finally, Section 1.5 outlines the overall structure of this thesis.

1.1 The Demand for XML Integrity Constraints

Organizations heavily rely on corporate data in operational and also decision making processes, which makes corporate data a key strategic asset to organizations in today's information age [1, 2]. For correct organizational decisions to be made, corporate data is required to be an unbiased representation of reality. The quality of data in this regard is vital to organizational success [1]. This has been pinpointed for instance in the report on data quality by the Data Warehousing Institute [3]. Based on interviews with industry experts, leading edge customers, and survey data from 647 respondents, the Data Warehousing Institute estimates that data quality problems cost United States businesses 600 billion dollar annually.

Database systems ensure the quality of data by requiring the data to be consistent with semantic assertions called *integrity constraints* [1]. In addition to ensuring the quality of data, database systems utilize integrity constraints also in order to accomplish essential tasks like automatic schema design and query optimization [4]. The study of integrity constraints has a long tradition in database theory starting approximately in the mid 1970's. The ongoing efforts in the development of integrity constraints yield a plethora of different types of integrity constraints for all major data models, including the relational, nested relational, and object-oriented data model. It has been estimated for instance that about 100 different types of integrity constraints have been developed for the relational data model alone.

With the adoption of the eXtensible Markup Language (XML) [5] as the industry standard for data interchange over the internet [6], and its increasing usage as format for the permanent storage of data in database systems [7], a new data model became more and more popular in recent years: the *XML data model*. In order to ensure the quality of XML data and to facilitate essential tasks of XML database systems, it is necessary to develop and study XML integrity constraints. Besides the traditional database tasks of automatic schema design and query optimization, XML integrity constraints are also useful in several new areas such as data exchange [8] and data integration [9] that have no parallels in the relational setting. For this reason, it has been argued that the development and study of integrity constraints is even more important for the XML data model than for the relational data model [10].

The most fundamental types of integrity constraints, irrespective of the data model used, are *key* and *foreign key* constraints. The reason for this is that keys and foreign keys establish *meaningful* connections between real world entities and their relationships on the one side, and data entities[1] and references between data entities on the other side. In general, keys facilitate the *identification* of data entities based on distinguished combinations of *values*. The rationale behind a key is that if some properties identify an entity in reality, then the values of these properties should identify the corresponding data entity in a database. A key makes this connection between real world entities and their representations in data explicit. In particular, the semantic assertion of a key is that no two distinct data entities have the same combination of values with respect to those properties that identify the represented real world entities.

The purpose of a foreign key is to represent relationships between real world entities by means of *references* between data entities. The reference to a data entity is thereby

[1]We use the term 'data entity' to denote the representation of a real world entity in a database.

established by referencing the combination of values that identify this data entity. In particular, the semantic assertion of a foreign key is that if certain combinations of values in a designated set of data entities are references to data entities in some other designated set of data entities, then each of these combinations of values identifies one data entity in the set of referenced data entities. A foreign key hence essentially demands a subset relationship between distinguished combinations of values of two designated sets of data entities.

The purpose of this thesis is to investigate the notions of value-based identification of data entities and value-based references between data entities for the XML data model. The aim of this thesis is therefore to develop and study XML keys and XML foreign keys. Developing XML keys and XML foreign keys poses a couple of challenges which we now illustrate.

1.2 Challenges in Developing Keys and Foreign Keys for XML

In developing XML keys and foreign keys one has to bear in mind two issues. First of all, XML keys and foreign keys must accommodate characteristics of XML data such that the notions of value-based identification of data entities and value-based references between data entities are adequately realized. We detail on this issue in Section 1.2.1. The second issue is that XML is the defacto standard for data exchange over the internet. XML data is therefore often generated from data that originally resides in a data model other than XML. In such a scenario, XML keys and foreign keys must allow for expressing original data semantics. We detail on this issue in Section 1.2.2.

1.2.1 Characteristics of XML Data

XML data is available in form of documents written in the eXtensible Markup Language, which has been developed by the World Wide Web Consortium [5]. XML data is hence basically text that contains markup. The tags in an XML document are required to form pairs of a start tag and a matching end tag. Also, the pairs of tags, called *XML elements*, are demanded to be properly nested. An XML element may have associated a set of attribute/value pairs, which we call *XML attributes* in the following. The XML elements in an XML document form a tree structure because of the requirement on XML elements to be properly nested. In terms of a logical data model, an XML document is therefore represented as a *tree* of *labeled nodes*, and so the XML data model is a *hierarchical* data model. Three kinds of nodes are distinguished in the XML data model. In particular, *element nodes* and *attribute nodes* represent XML elements and XML attributes, respectively. The label of an element node corresponds to the start tag of the represented XML element, and the label of an attribute node corresponds to the name of the represented XML attribute. Text that is enclosed by XML elements is represented by *text nodes* in an XML tree. Individual labels for text nodes cannot be derived from the text in an XML document, and so text nodes all have the same distinguished label. A sequence of node labels is said to be a *path*.

Example 1.1 (XML document and XML tree) *Suppose that a phone company stores information about customers and their cell phones using XML. A sample customer record representing customer* `Jones` *and his/her cell phone* `0660/1010` *is depicted in Figure 1.1a in form of an XML document. The XML elements in this document are* `<Customer>`*,* `<Phone>` *and* `<Status>`*, where matching end tags are indicated by the symbol '*`/`*'. XML element* `<Customer>` *has associated XML attribute* `name` *with value* `Jones`*. Likewise, XML element* `<Phone>` *has associated XML attributes* `code` *and* `no`*, representing the code* `0660` *and number* `1010` *of the phone of customer* `Jones`*. Whereas XML element* `<Status>` *does not have associated any XML attribute, it encloses the text '*`premium`*', meaning that customer* `Jones` *is a premium customer.*

Figure 1.1b depicts the tree representation of the XML document depicted in Figure 1.1a. The element nodes and attribute nodes as well as their arrangement in the XML tree directly correspond to the XML elements and attributes in the XML document. The text '`premium`*' is however represented by a separate node which has the distinguished label* `S` *(String) assigned and is a child node of the element node* `Status`*. A path in the XML tree in Figure 1.1b is for example the sequence of labels* `Customer.name`*. This path leads to the attribute node representing the name of customer* `Jones`*.*

(a)

```
<Customer name="Jones">
  <Phone code="0660" no="1010"/>
  <Status>premium</Status>
</Customer>
```

(b)

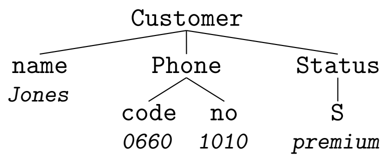


Figure 1.1: A customer record represented as XML document and as XML tree.

Apart from the organization of data in form of a tree of nodes, the structure of data is not further constrained in the XML data model since XML documents do not necessarily have to conform to a-priori schemas like an XML Schema (XSD) [11–13] or a Document Type Definition (DTD) [5]. This is in sharp contrast to the relational data model, where data always conforms to a rigid schema. The flexible structure of data is what makes the XML data model a *semi-structured* data model, and is probably also the primary reason for the adoption of XML as the standard format for data exchange over the internet where heterogenous data naturally occurs.

The hierarchical and semi-structured nature of XML data poses a couple of challenges for the design of XML keys and foreign keys. In order to illustrate these challenges, we first render the notions of keys and foreign keys in the XML data model more precisely. Given that the primary data structure in the XML data model is a tree of nodes, real world entities as well as properties of real world entities are represented by nodes in XML trees. In order to distinguish nodes that represent real world entities from nodes that represent properties of real world entities, we call the former *entity nodes* and the latter *property nodes.* The purpose of an XML key is then to identify entity nodes on the basis of values of distinguished property nodes. Analogously, the purpose of an XML foreign key is to establish references to entity nodes on the basis of values of property nodes that identify these entity nodes.

Example 1.2 (entity nodes and property nodes) *Suppose that no two customers of the phone company own the same cell phone, and so that customers are identified by their cell phones. With respect to the layout of customer records illustrated in Example 1.1 this means in terms of an XML key that* `Customer` *nodes are identified by the values in related* `code` *and* `no` *nodes. In this example* `Customer` *nodes are the targeted entity nodes, and* `code` *and* `no` *nodes are the property nodes.*

From a conceptual point of view, determining if an XML tree satisfies an XML key or XML foreign key can be done by comparing entity nodes on the basis of values of distinguished property nodes. The only difference between checking if an XML key is satisfied and checking if an XML foreign key is satisfied lies in the particular type of comparison used. Whereas an XML key demands for certain sets of entity nodes that the values of distinguished combinations of property nodes are *different*, an XML foreign key requires the existence of certain pairs of entity nodes with the *same* values for distinguished combinations of property nodes. Thus an XML key is a uniqueness constraint, whereas an XML foreign key is a matching constraint.

Because of the hierarchical nature of XML data, it is in general possible that multiple property nodes, which represent the same property, are related to a single entity node. In such a situation it is necessary to carefully choose the combinations of property nodes that are used for the comparison of entity nodes in order to achieve desired semantics of XML keys and foreign keys. We now illustrate this point by an example.

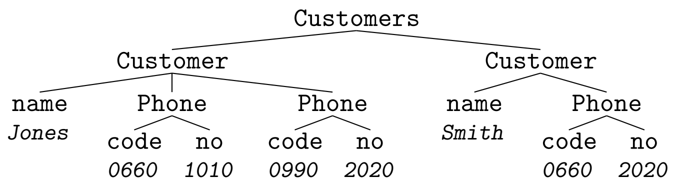


Figure 1.2: XML document representing customers and their phones. The phones are unique and identify the customers.

Example 1.3 (multiple property nodes) *Consider again the XML key from Example 1.2 which asserts that* `Customer` *nodes are identified by the values in related combinations of* `code` *nodes and* `no` *nodes. If this XML key is checked in the XML tree depicted in Figure 1.2, then there are multiple* `code` *nodes and* `no` *nodes related to the* `Customer` *node representing customer* `Jones` *who owns two phones. Hence, it is necessary to decide which combinations of the four possible combinations* `0660/1010`*,* `0660/2020`*,* `0990/2020` *and* `0990/1010` *to use for comparing customers* `Smith` *and* `Jones`*. Intuitively, only the combinations* `0660/1010` *and* `0990/2020` *should be used, since combinations* `0660/2020` *and* `0990/1010` *obviously represent the code and number of different phones and are therefore semantically incorrect. Note that* *if in contrast also the combination* `0660/2020` *is used for comparing the two* `Customer` *nodes, then the XML key is violated, which is clearly counter-intuitive.*

The first challenge posed by the characteristics of XML data to the design of XML keys and foreign keys is therefore the question of

> *How can one ensure that only semantically correct combinations of property nodes are used for the comparison of entity nodes?*

The occurrence of multiple property nodes also results in another design issue, one that is specific to keys in XML and that does not exist in the relational data model. The concept of a key, irrespective of the data model used, requires two fundamental properties - *uniqueness* and *identification*. Uniqueness means that the combinations of values for specified entity properties are unique within a distinguished set of data entities. Identification means that for any combination of values for specified entity properties there is at most one corresponding data entity. Now in the relational model, since duplicate tuples (data entities) are not allowed, the uniqueness and identification properties of a key are equivalent. That is, key identification implies that there is at most one tuple with the corresponding key values, and if there is at most one tuple with specific key values then the key values are unique within the relation. However in XML the situation is different and while key uniqueness still implies key identification, key identification no longer implies key uniqueness and so duplicate combinations of key values can occur. This situation is undesirable, since it can lead to data inconsistencies in the data if one duplicate combination of key values is updated but not the other. We now illustrate this point by an example.

Example 1.4 (uniqueness versus identification) *Consider once more the XML key on* `Customer` *nodes from Example 1.2, and the XML document depicted in Figure 1.3, which represents information about customers* `Henry` *and* `Smith` *and their phones. The identification of customers* `Henry` *and* `Smith` *is achieved by their phones* `0500/3030` *and* `0600/2020`*. However, the phone* `0500/3030` *is redundantly represented for customer* `Henry` *and so intuitively one would expect that the XML key on* `Customer` *nodes is violated even though it uniquely identifies customers.*

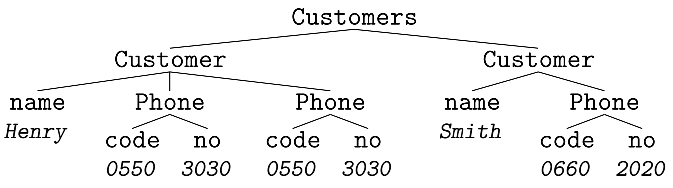


Figure 1.3: XML document representing customers and their phones. The phone of one customers is represented twice.

In addition to allowing an entity node to have multiple property nodes that represent the same entity property, the XML data model also allows for the property nodes to be absent.

The second challenge posed by the characteristics of XML data to the design of XML keys and foreign keys is therefore the question of

How to compare entity nodes in case that some property nodes are absent?

It is worth being mentioned that the absence of a property node does not necessarily mean that there is some information missing. Entity nodes may also lack certain property nodes because of heterogeneity in represented real world entities, and so the absent property nodes may simply not be applicable to the specific entity node. XML keys and foreign keys hence must accommodate the situation where some property nodes are absent even in case of complete information. We now illustrate this observation by an example.

Example 1.5 (absent property nodes) *Suppose that the phone company stores the address of a customer within his/her record. Two customer records including the customer's address are depicted in Figure 1.4 for the purpose of illustration. Note that the address of customer* `Smith` *states the number of his/her apartment, whereas the address of customer* `Henry` *does not have this property. Now, suppose that there is an XML key defined for the customer records asserting that the address of a customer is identified by the combination of city, street (*`strt`*), house (*`hno`*), and apartment number (*`ano`*). Intuitively, this XML key is expected to hold for the two addresses in the XML tree depicted in Figure 1.4, since the address* `NY, 5th Avenue, 50` *of customer* `Henry` *is obviously different from the address* `NY, Park Avenue, 30, apartment 2B` *of customer* `Smith`*. The absence of an apartment number in the address of customer* `Smith` *does not imply that there is some information missing. Instead this merely reflects the fact that address data is heterogeneous.*

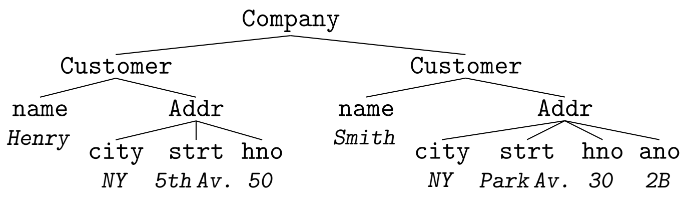


Figure 1.4: XML document representing customers and their addresses. The addresses are complete yet heterogenous.

To deal with multiple and absent property nodes are clearly not the only challenges posed by the characteristics of XML data to the design of XML keys and foreign keys. The hierarchical and semi-structured nature of XML data moreover demands for highly expressive integrity constraints. However, even more important than the expressiveness of the syntax of XML keys and foreign keys is that their semantics yields intuitive results in all situations. In particular, situations where there are multiple or absent property nodes are ones, where existing approaches to XML keys and foreign keys frequently result in incorrect semantics. Thus, the focus of this thesis is not on proposing more expressive XML keys and foreign keys but rather on ensuring that our definitions capture the correct semantics in the case of multiple or absent property nodes.

1.2.2 Usage of XML as Format for Data Exchange

When exchanging data over the internet in form of XML documents, it is often the case that the data originally resides in a format other than XML [10]. The original data therefore needs to be transformed to XML before it is exchanged. Irrespective of the data model used, the semantics of data, expressed in the form of integrity constraints, is essential to the effective use of the data. The usability of XML data is therefore diminished if only the original data but not the original data semantics is transformed. For instance, if XML integrity constraints are not available, updates issued on generated XML data have to be verified against the original data, which is an intricate task. It is therefore desirable to transform both the original data and also the original data semantics as specified by integrity constraints. The predominant data model used in practise is still the relational data model, and so the primary challenge in this regard is the question of

> *How to preserve the semantics of relational keys and foreign keys when XML data is generated from relational data?*

The answer to this question obviously depends on the specific procedure used for transforming relational data to XML data. In general, there are many possible transformation procedures. The essential requirements on a transformation procedure, and the design of XML integrity constraints, are (i) that the XML integrity constraints can be derived from the relational integrity constraints, and (ii) that the generated XML data satisfies the derived XML integrity constraints if the relational data satisfies the original integrity constraints.

Relational data is often restructured during the transformation to XML in realistic application scenarios like XML publishing for example [10]. It is therefore desirable that a transformation procedure facilitates the restructuring of information. The next example illustrates the restructuring of information when XML data is generated from relational data, and it also illustrates the notions of derived XML keys and foreign keys.

Example 1.6 (transforming relational data and semantics to XML) *Suppose that the phone company stores information about phones and invoices for phone charges within a relational database, which contains for this purpose relations* `Invoice` *and* `Phone` *as depicted in Figures 1.5a and 1.5b. Relation* `Phone` *states the codes* (`code`) *and numbers* (`no`) *of phones, and relates to each phone the customer* (`cno`) *who owns the phone. The information about invoices is represented in relation* `Invoice`*, which states for this purpose the addressed customer* (`cno`)*, the cell phone* (`code`, `no`) *together with the period* (`prd`) *for which phone charges have to be paid, and the invoiced amount* (`amt`).

Suppose now that relations `Invoice` *and* `Phone` *are mapped to XML, and that, unlike for the information about phones, the information about invoices is restructured in that invoices are grouped according to the addressed customer and the invoice period. Figure 1.5c depicts the resulting XML tree. Because the information about phones has not been restructured, there is a one-to-one correspondence between the* `Phone` *nodes in the XML tree and the tuples in relation* `Phone` *in that each phone is directly represented by one* `Phone` *node. In contrast, the information about invoices has been restructured, and the tuples in relation* `Invoice` *are therefore represented by only two* `Invoice` *nodes in the generated XML tree. This XML tree nevertheless represents the same information as relations* `Invoice` *and* `Phone`.

Lets turn to the semantics of data in relations `Phone` and `Invoice`. Suppose that (`code`, `no`, `cno`) is the key in relation `Phone`, and that (`code`, `no`, `prd`) is the key in relation `Invoice`. The semantics asserted by the key in relation `Invoice` is thereby that at most one phone charge is invoiced for a certain phone in a certain period. The XML keys to be derived from these relational keys assert that `Phone` nodes are identified by the combinations of `code`, `no`, and `cno` nodes, and analogously, that `Invoice` nodes are identified by the combinations of `code`, `no`, and `prd` nodes. Note that although the two invoices in the generated XML tree agree on the invoice period, they do not agree on any of the phones, and so the two `Invoice` nodes are indeed identified by the combinations of `code`, `no` and `prd` nodes.

Suppose further, that (`cno`, `code`, `no`) is a foreign key from relation `Invoice` to relation `Phone`, which asserts that a customer gets an invoice only for those phones that he or she owns. The XML foreign key to be derived from this relational foreign key asserts that if there exists an `Invoice` node together with a combination of `cno`, `code`, and `no` nodes, then there also exists a `Phone` node and a value equal combination of `cno`, `code`, and `no` nodes.

(a)

Invoice				
cno	code	no	prd	amt
C1	*0660*	*1010*	*01/09*	*$10*
C1	*0990*	*2020*	*01/09*	*$20*
C2	*0660*	*2020*	*01/09*	*$30*

(b)

Phone		
cno	code	no
C1	*0660*	*1010*
C1	*0990*	*2020*
C2	*0660*	*2020*

(c)

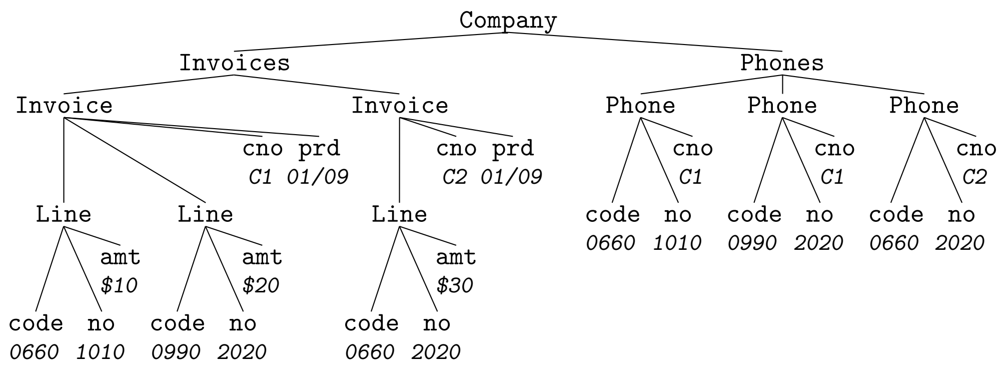


Figure 1.5: Relations representing phones and invoices, and a generated XML document.

It is also important to note that data is not only transformed from the relational data model to the XML data model in data exchange scenarios, but also in the opposite direction. Hence there is also need for preserving the semantics of XML keys and foreign keys when XML data is transformed to relational data. This is however clearly not an issue for the design of XML keys and foreign keys, but belongs instead to the design of (new) relational integrity constraints as well as to the design of procedures for transforming XML data to relational data. The issue of preserving the semantics of XML keys and foreign keys when XML data is transformed to relational data is therefore out of scope of this thesis.

1.3 Previous Approaches to XML Integrity Constraints

We now give an overview of previous approaches to XML integrity constraints. The focus is thereby on concepts that address the challenges which we have identified in the previous section. In particular, Section 1.3.1 illustrates concepts that address the challenges posed by characteristics of XML data. Section 1.3.2 outlines concepts that address the preservation of original data semantics when relational data is transformed to XML data.

1.3.1 Handling of Multiple or Absent Property Nodes

The challenges posed by the hierarchical and semi-structured nature of XML data are, in summary, to handle multiple and absent property nodes. A number of approaches to XML integrity constraints sidestep the problems caused by multiple or absent property nodes. For instance, the XML keys and foreign keys proposed in XML Schema put additional restrictions on the structure of XML data, and the approaches presented in [14–16] restrict the syntax of XML integrity constraints and so avoid situations in which multiple or absent property nodes occur. There are however several approaches to XML integrity constraints that permit multiple or absent property nodes. We now illustrate how these approaches handle multiple property nodes.

Apply cross-product semantics: A naive manner for handling multiple property nodes is to apply cross-product semantics when choosing combinations of property nodes. For instance the approaches to XML integrity constraints presented in [17–19] follow this idea. A consequence of applying cross-product semantics when choosing combinations of property nodes is that semantically incorrect combinations of property nodes are used for the comparison of entity nodes, resulting in an XML integrity constraint being violated when the semantics require that the constraint should be satisfied. We now illustrate this point.

Example 1.7 *Consider the XML key from Example 1.2 which asserts that customers are identified by the combinations of the code and number of each of their phones. When applying cross-product semantics in forming combinations of property nodes, this XML key is violated in the XML tree depicted in Figure 1.2. The reason is that `0660/2020` is a combination of a* `code` *node and a* `no` *node for both customers `Jones` and `Smith`. Regarding customer `Jones`, `0660/2020` is however semantically incorrect since `0660` and `2020` do not belong to the same phone. In fact one intuitively expects that this XML key on* `Customer` *nodes is satisfied in the XML tree in Figure 1.2.*

Exploit structural relationships: The approaches to XML integrity constraints presented in [20–23] take into account structural relationships between property nodes in checking XML integrity constraints. That is, a possible combination of property nodes is disregarded if the nodes do not possess certain structural relationships. In general there is a tight connection between the arrangement of nodes in an XML document and the coherence in the information represented by the nodes. Semantically incorrect combinations of property nodes can therefore be effectively avoided by exploiting structural relationships between property nodes. The particular structural relationships demanded for property nodes however differ between the approaches in [20–23], and so semantically incorrect combinations of property nodes are avoided in varying degree, that is some approaches always prevent semantically incorrect combinations of property nodes, whereas other approaches do not.

Example 1.8 *Referring to the XML tree depicted in Figure 1.2, the arrangement of* `code` *and* `no` *nodes related to the customer node representing customer* `Jones` *is not accidental. Instead the deliberate arrangement of these nodes reflects that* `0660/2020` *and* `0990/1010` *are indeed phones of customer* `Jones`*, whereas* `0660/1010` *and* `0990/2020` *are not. When taking into account the structural relationships between* `code` *and* `no` *nodes in checking the XML key which asserts that customers are identified by the code and number of each of their phones, the semantically incorrect combinations* `0660/1010` *and* `0990/2020` *are disregarded and so the XML key is satisfied in the XML tree in Figure 1.2.*

Concerning the manner in which absent property nodes are handled, the following ideas are to be found in previous approaches to XML integrity constraints:

Disregard incomplete combinations of property nodes: A naive manner for handling absent property nodes is to simply disregard incomplete combinations of property nodes in checking whether an XML tree satisfies an XML key or foreign key. This idea is followed in the approaches presented in [17, 24]. A consequence of disregarding incomplete combinations of property nodes is that an XML key or foreign key is automatically satisfied with respect to entity nodes that do not have any complete combination of property nodes. This is clearly not desirable as the next example illustrates.

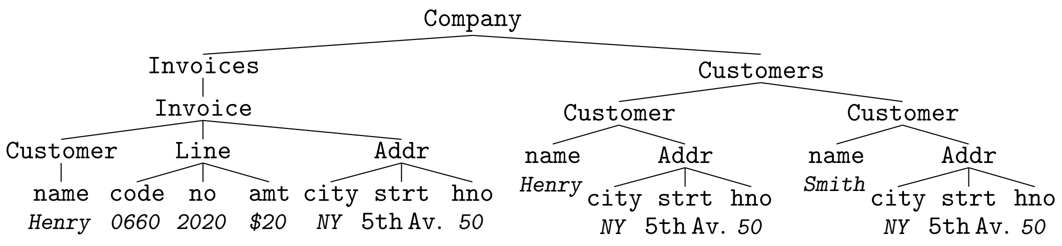


Figure 1.6: XML document representing an invoice and two customer records.

Example 1.9 *Consider the XML tree depicted in Figure 1.6 and the XML key introduced in Example 1.5 which asserts that the address of a customer is identified by the combination of city, street, house, and apartment number. The addresses of both customers* `Henry` *and* `Smith` *are incomplete since they lack of an apartment number. Since incomplete combinations of property nodes are simply disregarded, these two* `Customer` *nodes are not compared at all,* *and so the XML key on addresses of costumers is automatically satisfied in the approach of [17, 24]. This is however counter-intuitive since the addresses of customers* `Henry` *and* `Smith` *are obviously the same.*

Apply strong or weak satisfaction semantics: In order to deal with missing information, the notions of weak and strong satisfaction of integrity constraints have been translated from the relational setting to XML in the approaches presented in [17, 23] and [25], respectively. Intuitively, an XML integrity constraint is weakly satisfied if there exists *at least one completion* of the XML document under consideration that satisfies the constraint. If instead strong satisfaction semantics is applied, then *every possible completion* of the XML document must satisfy the XML integrity constraint.

Applying either strong or weak satisfaction semantics is not adequate for handling absent property nodes for two reasons. First, the notions of strong and weak satisfaction both rely on a (virtual) completion of the XML document. It is therefore implicitly assumed that absent nodes indicate information that actually exists but is currently *unknown* for some reason. This is however not in accordance with the *no information* interpretation of missing information which is recommended for XML data in the XML specification by the W3C. According to the *no information* interpretation, the absence of a node merely indicates that some information is not present for whatever reason. If the reason for the absence of some information is not known, it is clearly inadequate to act on the assumption that any completion of an XML document exists.

The second reason for which the notions of strong or weak satisfaction are not expedient for handing absent property nodes is that the absence of a node does not necessarily indicate some missing information at all. The absence of a node may also reflect some degree of heterogeneity within real world entities. In this case, the absence of a node merely reflects the fact that some information does not exist. The next example illustrates that applying either weak or strong satisfaction semantics to XML integrity constraints produces counter-intuitive results.

Example 1.10 *Consider again the XML tree depicted in Figure 1.6 and the XML key on addresses of customers introduced in Example 1.5. When applying weak satisfaction semantics this XML key is satisfied in the XML tree in Figure 1.6. The reason is that one possible completion of the XML tree contains the addresses* `NY`, `5th Avenue`, `50`, `10A` *for customer* `Henry` *as well as* `NY`, `5th Avenue`, `50`, `20b` *for customer* `Smith`. *In this completion, the XML key on customer addresses is satisfied, and so this XML key is satisfied in the XML tree in Figure 1.6 when applying weak satisfaction semantics. This is counter-intuitive, similar to Example 1.9 where incomplete combinations of property nodes are disregarded.*

Suppose now that an invoice states the invoice address, which is illustrated for example by the single invoice in the left of the XML tree in Figure 1.6. Suppose further that there is an XML foreign key expressing that invoice addresses reference customer addresses. When neglecting the record of customer `Smith` *in the XML tree in Figure 1.6, then this XML foreign key is expected to be satisfied, since the single invoice address is also the address of customer* `Henry`. *However, when applying strong satisfaction semantics this XML foreign key is violated in the XML tree in Figure 1.6 even if the record of customer* `Smith` *is not present. The reason for this is that one possible completion of the invoice address is* `NY`, *`5th Avenue`, `50`, `10b`, and a completion of the address of customer `Henry` is for example `NY`, `5th Avenue`, `50`, `20b`, in which case the XML foreign key from invoice addresses to customer addresses is violated.*

In addition to the methods for handling either multiple or absent property nodes illustrated above, some approaches to XML integrity constraints use another technique which we now discuss.

Use element nodes as property nodes: The approaches to XML integrity constraints presented in [12, 17, 18], permit the comparison of entity nodes not only on the basis of values of attribute nodes or text nodes, but also on the basis of 'values' of element nodes. Roughly speaking, the 'value' of an element node is the subtree rooted at the element node, and the 'values' of two element nodes are equal if the subtrees are isomorphic. In checking an XML key or foreign key the problems arising from multiple or absent property nodes can be avoided if entity nodes are compared on basis of the 'values' of element nodes that enclose the actual property nodes instead of comparing entity nodes on basis of the property nodes themselves. This is indeed a valid technique for the restricted case where all nodes that are nested within an element node are property nodes which are relevant for the comparison of entity nodes with respect to the XML key or foreign key under consideration. If this is not the case, this technique produces counter-intuitive semantics which is illustrated in the next example.

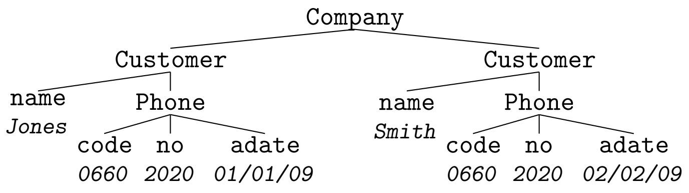


Figure 1.7: XML document representing two customers who bought the same phone.

Example 1.11 *Consider again the XML key from Example 1.2 which asserts that* `Customer` *nodes are identified by the combinations of* `code` *and* `no` *nodes. If this XML key is checked in the XML tree depicted in Figure 1.2 multiple* `code` *and* `no` *nodes are related to the* `Customer` *node representing customer* `Jones`*. The semantically correct combinations of* `code` *and* `no` *nodes must be chosen in order to achieve the desired semantics, which has been illustrated in Example 1.3. If the comparison of entity nodes on basis of the 'values' of element nodes is permitted, then this XML key can be defined such that* `Customer` *nodes are identified by the 'values' of* `Phone` *nodes which encapsulate the actual property nodes, i.e. the* `code` *and* `no` *nodes. Since the subtrees rooted at* `Phone` *nodes in the XML tree in Figure 1.2 are not isomorphic to each other, the XML key is satisfied as desired. Note that all nodes nested within* `Phone` *nodes are property nodes which are relevant for the comparison of* `Customer` *nodes. Therefore using element nodes as property nodes is semantically correct in this example.*

The XML key asserting the identification of `Customer` *nodes by* `Phone` *nodes is also satisfied in the XML tree depicted in Figure 1.7. The reason is that the subtrees rooted at* *the* Phone *nodes are not isomorphic because the phones of customers* Jones *and* Smith *have different acquisition dates* (adate)*. It is however clearly not desirable that the XML key on* Customer *nodes is satisfied in the XML tree in Figure 1.7 because both phones have the same code and number. Since* adate *nodes nested within* Phone *nodes are irrelevant for the identification of* Customer *nodes, the use of element nodes as property nodes is semantically incorrect in this case.*

We proceed with illustrating approaches that address the preservation of relational data semantics in data exchange scenarios.

1.3.2 Preserving Relational Data Semantics

There is a substantial body of work on the transformation of relational data to XML. Early approaches like XPERANTO [26–28] or SilkRoute [29] primarily focused on bridging the gap between the flat structure of relational data and the hierarchical structure of XML data. The semantics of data is kept buried within relational data in these approaches. In order to overcome this limitation, the following approaches have been proposed:

Direct transformation: In the approaches presented in [30, 31], relational data is transformed to XML data in a rigid and direct manner. In particular, each tuple in a relation is mapped to exactly one element node, and each value in a tuple is represented by a child node of this element node. Hence there is a 1-1 correspondence between the generated XML data and the original flat data. As a consequence, multiple property nodes do not occur when checking XML keys or foreign keys that express the semantics of relational keys or foreign keys. Standard XML keys and foreign keys offered by XML schema are therefore sufficient for expressing original data semantics in the approaches presented in [30, 31].

Example 1.12 *Referring to Example 1.6 and Figure 1.5, the transformation of relation* Phone *is an instance of the direct transformation procedure applied in the approaches presented in [30, 31]. Each tuple in relation* Phone *is represented by exactly one element node in the XML tree in Figure 1.5c, where* code*,* no*, and* cno *nodes represent the values in the corresponding attributes in a tuple. In checking the XML key derived from the key* (code*,* no*,* cno) *in relation* Phone*, multiple property nodes do not occur since each* Phone *node has related exactly one* code*,* no*, and* cno *node in correspondence to the single* code*,* no*, and* cno *values in tuples of relation* Phone*.*

Transformation on the basis of interconnected tuples: The approaches presented in [32–37] aim at generating more compact XML documents than those resulting from the direct transformation of relational data. For this purpose, the hierarchical nature of XML data is exploited in that interconnected tuples are mapped to nested element nodes. As opposed to the direct transformation of relational data, information is restructured during the transformation on the basis of interconnected tuples. The manner in which information is restructured however rigidly adheres to the connections between tuples, and so application developers are not permitted to govern the restructuring of information.

Compared to the direct transformation of relation data, the total number of nodes in generated XML data is substantially smaller when transforming relational data on the basis of interconnected tuples. Generated XML documents are therefore more compact even though not all types of connections between tuples can be represented by nested element nodes. Problematic types of connections are, for instance, many-to-many or circular references.

As for the direct transformation of relational data, the flat structure of individual tuples is kept during the transformation on the basis of interconnected tuples. Consequently, multiple property nodes do not occur when checking XML keys or foreign keys that express the semantics of relational keys or foreign keys. Hence, standard XML keys and foreign keys offered by XML Schema are again sufficient in order to preserve original data semantics.

Example 1.13 *Referring again to Example 1.6, there is a one-to-many relationship between phones and invoices in that each phone is related to many invoices. This relationship is expressed by the foreign key on attributes* (`cno`, `code`, `no`) *from relation* `Invoice` *to relation* `Phone`*. The tuples in relations* `Invoice` *and* `Phone` *are thus interconnected. Because of this interconnection,* `Invoice` *nodes representing the tuples in relation* `Invoice` *are nested within* `Phone` *nodes representing the tuples in relation* `Phone` *when transforming relations* `Invoice` *and* `Phone` *to XML on the basis of interconnected tuples. As for the direct transformation procedure illustrated in Example 1.12, the generated* `Invoice` *and* `Phone` *nodes adhere to the flat structure of tuples in relations* `Invoice` *and* `Phone`*. Therefore, multiple property nodes again do not occur when checking either the XML key derived from the key* (`code`, `no`, `cno`) *in relation* `Phones`*, or the XML key derived from the key* (`code`, `no`, `prd`) *in relation* `Invoice`*.*

Transformation on the basis of nested tuples: A two-step transformation procedure is applied in the approaches presented in [25, 38, 39]. In the first step, nesting operations are performed on the original relations. The resulting nested relations are then directly mapped to XML in the second step. It is worth being mentioned that the transformation on basis of nested tuples and the transformation on basis of interconnected tuples are complementary approaches. In fact, the transformation procedure presented in [38, 39] first performs nesting operations on the original relations and then maps the nested tuples to XML such that interconnected tuples are represented by nested elements. The primary aim of this transformation procedure is to yield XML documents that are even more compact than those that result from transforming relations solely on basis of interconnected tuples. The nesting operations to be performed are thereby automatically computed such that the number of nodes in the resulting XML tree is as small as possible. Because nesting operations are automatically computed, application developers are not permitted to govern the restructuring of information. The transformation procedure presented in [38, 39] is moreover limited in that only the nesting on a single attribute is permitted for each relation.

The transformation procedure presented in [25] in contrast allows to perform a sequence of arbitrary nesting operations on a relation. Application developers thereby individually specify the nesting operations to be performed, and are therefore able to govern the restructuring of information. The transformation procedure presented in [25] is however limited in that only the transformation of a single relation is permitted.

Compared to the approaches where relational data is either directly transformed or on the basis of interconnected tuples, it is more complicated to preserve original data semantics when relations are nested prior to the mapping. The reason for this is that because of the nesting of relations, multiple property nodes may occur when checking an XML integrity constraint that expresses original data semantics. Approaches to XML integrity constraints which do not allow for multiple property nodes at all, or which are subject to semantically incorrect combinations of property nodes are clearly inappropriate for preserving relational data semantics when it is transformed on basis of nested tuples. For this reason, approaches to XML integrity constraints have been proposed in [40] and [20] in accordance with the transformation procedures presented in [38, 39] and [25], respectively. Even though the approaches to XML integrity constraints proposed in [40] and [20] have different semantics in general, both approaches take into account structural relationships betweens property nodes when forming combinations of property nodes for the purpose of checking an XML integrity constraint. Semantically incorrect combinations of property nodes in generated XML documents are therefore disregarded in both approaches, and hence the ability to preserve original data semantics is achieved.

Example 1.14 *Referring again to Example 1.6 and Figure 1.5, the transformation of relation* `Invoice` *is an instance of the transformation procedure presented in [20]. In particular, relation* `Invoice` *is first nested on attributes* `code`*,* `no`*, and* `amt`*, which results in a nested relation where, conceptually, invoices are grouped according to the addressed customer* (`cno`) *and invoice period* (`prd`)*. This nested relation is then directly mapped to XML, which yields the XML tree rooted at the* `Invoices` *node in the left of Figure 1.5c. Because of the nesting of relation* `Invoice` *prior to the mapping, multiple property nodes occur when checking for instance the XML key expressing the semantics of the original key* (`code`*,* `no`*,* `cno`) *in relation* `Invoice`*. Note that each line of an invoice in the XML tree in Figure 1.5 states the phone for which phone charges are invoiced, and so there are multiple* `code` *and* `no` *nodes related to the* `Invoice` *node representing the invoice for customer* `C1` *in January 2009.*

1.4 The Enhanced Closest node Approach

In light of the challenges identified in Section 1.2, the following objectives are pursued in this thesis in order to provide keys and foreign keys for the XML data model:

- *Approach to XML integrity constraints*: To develop an approach to XML integrity constraints where the syntax allows application developers to specify XML keys and foreign keys in an intuitive manner, and the semantics accommodates the hierarchical and semi-structured nature of XML data. Especially to allow for the semantically correct handling of XML integrity constraints when property nodes are absent, an issue that has not been addressed in previous work.

- *Transformation Procedure*: To develop a procedure for the transformation of a set of relations to an XML document which allows application developers to govern the restructuring of information in individual relations. The interplay of the transformation procedure and the proposed approach to XML integrity constraints must allow to automatically derive XML keys and foreign keys that preserve original data semantics.

- *Consistency and implication problems*: To develop decision procedures for the implication and consistency problems related to the integrity constraints in the proposed approach.

The consistency problem is the question of whether there exists at least one XML document that satisfies a given set of XML integrity constraints. The practical use of a set of XML integrity constraints is obviously diminished if not even one XML document may satisfy the constraints. A decision procedure for the consistency problem is therefore a useful tool in developing XML applications. The implication problem is the question of whether an XML integrity constraint must necessarily hold in every XML document that satisfies a given set of XML integrity constraints. To have a decision procedure that answers this question is the fundamental prerequisite for facilitating essential database tasks like automatic schema design and semantic query optimization.

We now summarize the contributions of this thesis. Preliminary results from the thesis have already been published in [41, 42]. Details will be given in the subsequent chapters.

Keys and foreign keys for XML: A foreign key is in general the combination of a key and an integrity constraint commonly known as inclusion dependency. The key asserts that certain values identify referenced data entities, and the inclusion dependency asserts a subset relationship between distinguished values of referencing data entities and the identifying values of referenced data entities. For this reason *XML keys* and *XML inclusion dependencies* are developed in this thesis in order to provide keys and foreign keys for XML.

We call the approach to XML integrity constraints presented in this thesis the *enhanced 'closest node' approach*. This approach adopts the syntactic framework for XML integrity constraints proposed by XML Schema, which we call the *selector/field* framework subsequently. In the selector/field framework integrity constraints are specified in form of pairs of a *selector* and a set of *fields*. The selector is used for selecting entity nodes in an XML document and the fields are used to relate property nodes to the selected entity nodes. We adopt the selector/field framework so that the syntax of XML keys and inclusion dependencies directly corresponds to the intended purposes of XML keys and inclusion dependencies, i.e. to identify entity nodes on the basis of values of distinguished property nodes, and to assert subset relationships between values of distinguished property nodes related to two sets of entity nodes. Hence, application developers are provided with an intuitive manner to specify XML keys and XML inclusion dependencies.

Example 1.15 (selector/field framework) *Referring to Example 1.6, the XML key asserting that invoices are identified by the invoice period together with the code and number of each phone stated in an invoice line is specified by the statement* (`Company.Invoices.Invoice`, (`prd`, `Line.code`, `Line.no`)) *in the selector/field framework. The path* `Company.Invoices.Invoice` *is thereby the selector, and paths* `prd`*,* `Line.code`*, and* `Line.no` *are the fields which relate combinations of a* `prd` *node, a* `code` *node, and* `no` *node to each selected* `Invoice` *node.*

Analogously, the XML inclusion dependency that expresses the reference part of the XML foreign key from `Invoices` *to* `Phones` *is specified by the statement* (`Company.Invoices.Invoice`, (`cno`, `Line.code`, `Line.no`)) $\subseteq$ (`Company.Phones.Phone`, (`cno`, `code`, `no`)) *in the selector/field framework. This XML inclusion dependency consists of two pairs of a selector and a set of fields. In particular, paths* `Company.Invoices.Invoice` *and* `Company.Phones.Phone` *are the left hand side (LHS) and right hand side (RHS) selectors, which point to the referencing* `Invoice` *nodes and the referenced* `Phone` *nodes, respectively.*

The LHS fields `cno`*,* `Line.code`*,* `Line.no` *and the RHS fields* `cno`*,* `code`*,* `no` *then assert the desired subset relationship between the values of any combination of a* `cno` *node, a* `code` *node, and a* `no` *node nested within a selected* `Invoice` *node, and the values of at least one combination of a* `cno` *node, a* `code` *node, and a* `no` *node nested within a selected* `Phone` *node.*

Whereas XML integrity constraints in the enhanced 'closest node' approach have the same syntax as XML Schema constraints, they have different semantics in order to accommodate the requirements identified in Section 1.2.

The rationale for handling multiple property nodes in the enhanced 'closest node' approach relies on the assumption that the degree of structural coherence in a set of nodes is directly proportional to the degree of coherence in the represented information. That is, the closer some nodes are arranged in an XML document, the stronger is the coherence in the represented information. This assumption is not unusual, and is also postulated in approaches to XML keyword search for example [43, 44]. The rationale for handling multiple property nodes is then to only use those combinations of property nodes for the purpose of value-based comparison of entity nodes, where the degree of structural coherence is maximal. As a consequence, also the coherence in information represented by a combination of property nodes is maximal, and so semantically incorrect combinations of property nodes are disregarded in checking XML keys and inclusion dependencies. To determine whether the structural cohesion in a combination of property nodes is maximal, we use the *closest* property of nodes originally presented by Vincent et al. in defining an XML functional dependency [20]. Intuitively, a pair of nodes in an XML document satisfy the *closest* property if the nodes cannot be arranged more closely when taking into account the paths that lead to the nodes. Technically, a pair of nodes satisfy the *closest* property, if the nodes have a common ancestor node reachable over the intersection of the paths leading to the nodes.

Example 1.16 (*closest* property of nodes) *Consider the* `code` *nodes and* `no` *nodes in the XML tree depicted in Figure 1.5 which are nested within the* `Invoice` *node that represents the invoice for customer* `C1` *in January 2009. The paths leading to* `code` *nodes and* `no` *nodes from the root of the XML tree are* `Company.Invoices.Invoice.Line.code` *and* `Company.Invoices.Invoice.Line.no`*, respectively. With respect to these paths, the structural cohesion between a* `code` *node and a* `no` *node is maximal, if the nodes have a common ancestor node at path* `Company.Invoices.Invoice.Line`*. In fact, the pairs of a* `code` *node and a* `no` *node which satisfy this requirement are precisely the semantically correct combinations* `0660/1010` *and* `0990/2020`*. In contrast the semantically incorrect combinations* `0660/1010` *and* `0990/2020` *do not satisfy the* closest *property, since neither one of these pairs of nodes have a common ancestor at the intersection path* `Company.Invoices.Invoice.Line`*.*

It is worth being mentioned that two nodes satisfying the closest property is not identical with the property of two nodes to have the same parent node. For instance the combination of the `code` *node* `0660` *and the* `prd` *node* `01/09` *nested within the* `Invoice` *node representing the invoice for customer* `C1` *in January 2009 do not have the same parent node but satisfy the* closest *property. The reason is that these two nodes have a common ancestor node at the intersection path* `Company.Invoices.Invoice`*.*

The manner in which absent property nodes are handled in the enhanced 'closest node' approach addresses XML documents where

(i) information is complete and absent property nodes therefore reflect some degree of heterogeneity in real world entities, or

(ii) information is incomplete and absent property nodes are interpreted in terms of the *no information* interpretation as recommended in the XML specification by the W3C.

In both situations, checking for constraint satisfaction using a completion of the XML document is incorrect, as has been illustrated in Example 1.10. So, since applying weak or strong satisfaction semantics is incorrect in the context of absent property nodes, the conclusion we draw is that XML keys and inclusion dependencies must be checked solely on the basis of values in those combinations of property nodes that are present in an XML document. In order to enable the comparison of entity nodes by the values in incomplete combinations of property nodes, we introduce the novel concept of *maximum combinations of property nodes*. Instead of comparing entity nodes only on the basis of values in complete combinations of property nodes, entity nodes are compared in the enhanced 'closest node' approach on the basis of values in all semantically correct combinations of property nodes that are maximal with respect to the number of nodes. Two maximum combinations of property nodes are thereby value equal if (i) they contain the same number of nodes, and (ii) the nodes represent the same properties of real world entities, and (iii) the nodes have the same value assigned.

Example 1.17 (maximum combination of property nodes) *Referring to Example 1.5, in checking the XML key that asserts the identification of addresses of customers by the combinations of city, street, house number and apartment number in the XML tree depicted in Figure 1.4, the maximum combinations of property nodes are* `NY`*,* `5th Avenue`*,* `50` *and* `NY`*,* `Park Avenue`*,* `30`*,* `2B` *with respect to the addresses of customers* `Henry` *and* `Smith`*, respectively. These two maximum combinations of property nodes are not value equal, since they do not contain the same number of nodes, and so the XML key is satisfied when comparing entity nodes on basis of maximum combinations of property nodes.*

If instead this XML key is checked in the XML tree depicted in Figure 1.6, the maximum combination of property nodes is `NY`*,* `5th Avenue`*,* `50` *for the addresses of both customers* `Henry` *and* `Smith`*. These two maximum combinations of property nodes are value equal, since (i) they both contain three nodes, (ii) the nodes in both combinations represent the city, street and house number of an address, and (iii) the nodes in the two combinations are value equal. The XML key is therefore violated as desired when comparing entity nodes on basis of maximum combinations of property nodes.*

In summary, the first contribution of this thesis is the formal definition of XML keys (XKeys) and XML inclusion dependencies (XINDs) which adopt the intuitive selector/field syntax, as well as the 'closest node' approach in order to adequately handle multiple property nodes. The enhancement to the 'closest node' approach is to use the novel concept of maximum combinations of property nodes in order to adequately deal with absent property nodes.

Transformation Procedure: The procedure for transforming relational data to XML data presented in this thesis adopts the transformation procedure presented in [25], which allows for transforming a single relation on the basis of nested tuples. Application developers thereby govern the restructuring of information by specifying the sequence of nesting operations to be applied on the initial relation prior to the mapping. We extend this transformation procedure to the general case of transforming a set of relations as follows. Each relation is first transformed to an individual XML tree by applying the original transformation procedure. The final XML tree is then composed by adding the individual XML trees as principal subtrees to the final XML tree.

Example 1.18 (transformation procedure) *The transformation of relations* `Invoice` *and* `Phone` *illustrated in Example 1.6 is an instance of the transformation procedure presented in this thesis. The relations* `Invoice` *and* `Phone` *are first transformed to the individual XML trees rooted at the* `Invoices` *node and the* `Phones` *node depicted in Figure 1.5c, respectively. Relation* `Invoice` *is thereby restructured in that it is nested on attributes* `code`*,* `no`*,* `amt` *prior to the mapping. The final XML tree depicted in Figure 1.5c is then composed by adding these individual subtrees as principal subtrees.*

Further, a procedure for automatically deriving XKeys and XINDs from relational keys and inclusion dependencies is developed in correspondence to the transformation procedure. The second contribution of this thesis is to present the transformation procedure and the procedure for deriving XKeys and XINDs in terms of precise algorithms. It is shown moreover that derived XKeys and XINDs preserve relational data semantics. In particular the result is established that if a set of complete flat relations satisfy a set of keys and inclusion dependencies, then the XML tree obtained from these relations satisfies the XKeys and XINDs derived from the relational constraints.

Thus, the approach presented in this thesis satisfies all the requirements presented in Section 1.2.

Consistency and Implication Problems: The third contribution of this thesis is to solve the consistency and implication problems related to enhanced 'closest node' XML keys and inclusion dependencies in the context of a class of XML trees originally proposed by Vincent et al. [20], called *complete* XML trees. Intuitively, a complete XML tree is one that contains no missing data, and is intended to extend the notion of a complete relation to XML by requiring that every path in the XML tree, rather then every tuple as in the relational case, contains the maximal amount of information. A complete XML tree is however a more general notion than a complete relation since it includes XML trees that cannot be mapped to complete relations, such as those that contain duplicate nodes or subtrees, and XML trees that contain element leaf nodes rather than only text or attribute leaf nodes. The motivation for considering complete XML trees is that while XML explicitly caters for irregularly structured data, it is also widely used in more traditional business applications involving regularly structured data, often referred to as data-centric XML [7]. For example, a recent survey of several hundred large companies in the United States found that around 70% were now using XML enabled databases, or native XML databases, for their core data processing functions [45]. In this setting, complete XML trees are a natural and important subclass, just as complete relations are in relational databases.

In the context of complete XML trees we show in particular that every set of XML keys or XML inclusion dependencies is consistent. Concerning the implication of XML keys and

XML inclusion dependencies, sound and complete sets of inference rules as well as decision procedures for both types of constraints are developed.

1.5 Outline

The remainder of this thesis is organized into two parts devoted to the design and study of XML keys and XML inclusion dependencies.

Chapter 2 - *Related Work* gives an overview on previous approaches to XML integrity constraints. The focus is thereby on how the challenges identified in the introductory chapter are being dealt with.

Chapter 3 - *The Enhanced Closest node Approach* presents the formal definitions of XKeys and XINDs and illustrates that XKeys and XINDs adequately handle multiple property nodes and also absent property nodes.

Chapter 4 - *Preserving Relational Semantics* presents the algorithms for transforming a set of relations to a single XML document on the basis of nested tuples and also the algorithms for deriving XKeys and XINDs from relational keys and inclusion dependencies. It is shown that XKeys and XINDs preserve original data semantics.

Chapter 5 - *The Context for Reasoning about XKeys and XINDs* introduces the notion of complete XML trees and revisits the satisfaction of XKeys and XINDs in the context of complete XML trees.

Chapter 6 - *Reasoning about XKeys* investigates the consistency and implication problems related to XKeys in the context of complete XML trees. A sound and complete set of inference rules and as well as a decision procedure for the implication of XKeys in complete XML trees are presented.

Chapter 7 - *Reasoning about XINDs* investigates the consistency and implication problems related to XINDs in the context of complete XML trees. A sound and complete set of inference rules as well as a decision procedure for the implication of XINDs in complete XML trees are presented.

Chapter 8 - *Conclusion* summarizes the thesis and gives an outlook on possible future work.

Part I

Design

Chapter 2

Related Work

Contents

This chapter relates previous approaches to XML integrity constraints to the enhanced 'closest node' approach proposed in this thesis. The focus is thereby on how previous approaches deal with the challenges identified in Chapter 1. Also, major theoretical results related to previous approaches are summarized, where results on the implication and consistency problems are emphasized. Section 2.1 outlines the context of our survey. It presents a categorization of XML integrity constraints and a benchmark test for the class of value-based XML integrity constraints where enhanced 'closest node' XML integrity constraints fall into. Sections 2.2 - 2.6 then illustrate and evaluate previous approaches, and Section 2.7 finally summarizes their strengths and weaknesses.

2.1 Context of the Survey

A plethora of different types of XML integrity constraints is to be found in literature. These XML integrity constraints form the following categories with respect to their intended purpose: *schema constraints*, *path constraints*, *complex constraints* and *value-based constraints*. Section 2.1.1 gives a brief overview on these categories of XML integrity constraints and also on major theoretical results related to the XML integrity constraints in the individual categories.

2.1.1 A Categorization of XML Integrity Constraints

With respect to their intended purpose, XML integrity constraints are grouped into the following disjoint categories:

Schema Constraints: XML schema languages typically impose structural and domain constraints, which are subsumed under the umbrella term of schema constraints. Structural constraints thereby specify for example the labels, nesting and cardinality of nodes in an XML document. Domain constraints in contrast specify the set of permitted values for nodes carrying text. Numerous different XML schema languages have been proposed so far like, for example, XML Schema, RelaxNG [46], RegXPath [47] and XCSL [48]. Whereas structural and domain constraints are specified in form of type definitions in XML Schema and RelaxNG, these types of constraints are specified in form of rules in RegXPath and XCSL. In general, the expressivity between XML schema languages differs largely. The interested reader is referred to the excellent surveys in [49, 50] for a more detailed overview on XML schema languages.

Results on the consistency of structural constraints, i.e. the question of whether there exists at least one XML document that satisfies a given schema, have been established recently. For instance, the consistency of a Document Type Definition (DTD), which is the predecessor of XML Schema, is efficiently decidable [51]. In contrast, the consistency of a set of RegXPath constraints, which basically have the same expressivity as structural constraints in XML Schema, cannot be efficiently computed [47].

Path Constraints: In general, identity-based data models, like the object-oriented data model for example, permit references between objects by object identifiers, as opposed to value-based references in terms of foreign keys. Path constraints allow to specify dependencies between sequences of objects which are connected by object references, i.e. paths. The probably most prominent types of path constraints are *path inclusion constraints* and *path inverse constraints*. Roughly speaking, a path inclusion constraint requires that if an object o is reachable from an object o' by following some path, then o must also be reachable from o' by following some other path. A path inverse constraint requires that if an object o is reachable from an object o', then also o' must be reachable from o.

Path inclusion and path inverse constraints for XML data have been proposed by Abiteboul et al. [52] and Buneman et al. [53], where nodes and edges take on the roles of objects and object references, respectively. XML documents are thereby modeled as rooted graphs of nodes rather than as rooted trees, which allows to represent identity-based references in XML documents specified by advanced mechanisms such as XPointer [54] and XLink [55].

The implication problem for path constraints, i.e. the question of whether a certain path constraint must hold in every XML document which satisfies a given set of path constraints, has been studied both in the absence of DTDs [56, 57] and in the presence of DTDs [58], with the main finding that there are practical classes of path constraints in both settings for which the implication problem is efficiently decidable.

Complex Constraints: Constraint languages like INCOX [59] or SCHEMATRON [60] have been developed in order to specify complex, semantic integrity constraints on XML data. Such constraints result from general business rules like, for instance, *if a customer has at least three cell phones, he/she gets 10 percent discount on phone charges.* Complex constraint languages are similar to assertions in SQL, and allow the specification of constraints in a rule based manner, where powerful XML query languages like XPath [61] or even XQuery [62] are permitted in order to select those parts of an XML document against which the constraint is checked.

Theoretical results for complex XML constraints are missing so far. The reason for this is probably that the powerful XML query languages used for specifying complex XML constraints make reasoning very hard.

Value-based Constraints: The commonality of value-based XML integrity constraints is that they impose dependencies between the data in an XML document, i.e. the values of nodes. Apart from XML keys and XML foreign keys, prominent types of value-based XML integrity constraints are for example XML functional dependencies, XML inclusion dependencies and XML multivalued dependencies. These types of XML integrity constraints correspond to the respective types of relational integrity constraints. Value-based XML integrity constraints are surveyed in detail in the remainder of this chapter.

The approach to XML integrity constraints proposed in this thesis obviously belongs to the category of value-based XML integrity constraints. Therefore, a direct comparison between enhanced 'closest node' XML integrity constraints and value-based XML integrity constraints is possible, whereas a comparison with XML schema constraints, XML path constraints, and complex XML constraints is not.

2.1.2 A Benchmark Test for value-based XML Integrity Constraints

In our survey of previous approaches to value-based XML integrity constraints we pay special attention on how these approaches deal with the challenges identified in Section 1.2. For this purpose the XML integrity constraints listed in Table 2.1 are used as benchmark test, and we now explain how these benchmark constraints are related to the challenges identified in Section 1.2. These challenges are in summary (a) to handle multiple property nodes, with the specific challenge for XML keys to guarantee the uniqueness of property nodes, (b) to handle absent property nodes, and (c) to preserve original data semantics, when relational data is transformed to XML data.

Testing the ability of an approach to handle multiple property nodes: For this purpose, XML tree $\mathcal{T}_1$ depicted in Figure 2.1 is validated against XML key $\mathcal{K}_1$ and XML inclusion dependency $\mathcal{I}_1$ introduced in Example 1.6. XML key $\mathcal{K}_1$ asserts that invoices (`Invoice`) are identified by the invoice period (`prd`) in combination with the code (`code`)

#	*Description*	*Type*
$\mathcal{K}_1$	Invoices are identified by the invoice period together with the code and number of each phone for which phone charges are invoiced.	Key
$\mathcal{F}_1$	The id of an invoice is determined by the invoice period together with the code and number of each phone for which phone charges are invoiced.	FD
$\mathcal{I}_1$	The combinations of phone codes and numbers and the customer number in an invoice are subsets of the combinations of codes and numbers of existing phones and the numbers of customers owning the phones.	IND
$\mathcal{K}_2$	Addresses of customers are identified by the combinations of city, street, house number and apartment number.	Key
$\mathcal{F}_2$	The id of a customer's address is determined by the combination of city, street, house number and apartment number.	FD
$\mathcal{I}_2$	Invoice addresses are a subset of customer addresses.	IND

Table 2.1: Benchmark XML integrity constraints.

and number (`no`) of each phone for which phone charges are invoiced. This XML key should be satisfied in XML tree T_1, since the semantically correct combinations {*01/09*, *0660*, *1010*}, {*01/09*, *0990*, *2020*}, and {*01/09*, *0660*, *2020*} uniquely identify the `Invoice` nodes in XML tree $\mathcal{T}_1$. Note that the invoice for customer *C1* in XML tree $\mathcal{T}_1$ lists phone charges for two phones. Multiple `code` nodes and `no` nodes therefore must be handled in validating XML tree $\mathcal{T}_1$ against XML key $\mathcal{K}_1$.

For some approaches to value-based XML integrity constraints a key constraint has not been proposed so far, which prevents the evaluation of these approaches using benchmark constraint $\mathcal{K}_1$. To capture this situation in the benchmark test, we introduce the `id` of an invoice, and use XML functional dependency $\mathcal{F}_1$ instead of XML key $\mathcal{K}_1$. XML functional dependency $\mathcal{F}_1$ is inspired by XML key $\mathcal{K}_1$ and requires that `prd`, `code` and `no` determine the `id` of an invoice. Note that multiple `code` nodes and `no` nodes must also be handled when XML functional dependency $\mathcal{F}_1$ is checked in XML tree $\mathcal{T}_1$. XML functional dependency $\mathcal{F}_1$ should be satisfied in XML tree $\mathcal{T}_1$ for reasons similar to which XML key $\mathcal{K}_1$ should be satisfied in this XML tree.

XML inclusion dependency $\mathcal{I}_1$ asserts that a customer receives an invoice only for phones he/she owns, and therefore requires that the combinations of `cno`, `code`, and `no` nodes nested within `Invoice` nodes are a subset of the combinations of `cno`, `code`, and `no` nodes nested within `Phone` nodes. Again, since the invoice for customer *C1* in XML tree $\mathcal{T}_1$ lists phone charges for two phones, multiple property nodes occur when checking the satisfaction of XML inclusion dependency $\mathcal{I}_1$ in XML tree $\mathcal{T}_1$. XML inclusion dependency $\mathcal{I}_1$ should be satisfied in XML tree $\mathcal{T}_1$ since the semantically correct combinations {*C1*, *0660*, *1010*}, {*C1*, *0990*, *2020*}, and {*C2*, *0660*, *2020*} are indeed also combinations of `cno`, `code`, and `no` nodes nested within `Phone` nodes.

In order to evaluate whether a proposal for an XML key ensures uniqueness of property nodes, XML tree $\mathcal{T}_2$ depicted in Figure 2.2 is validated against XML key $\mathcal{K}_1$. Note that the

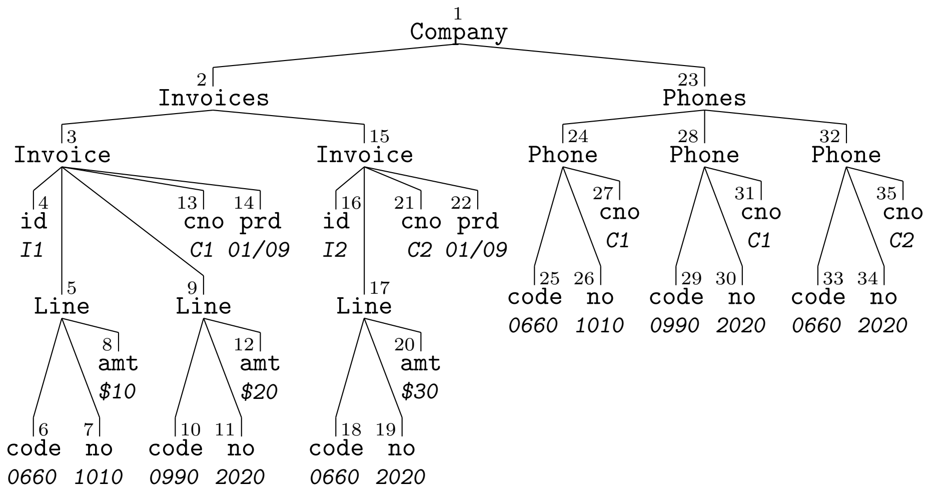


Figure 2.1: XML tree $\mathcal{T}_1$ used for testing the ability of value-based XML integrity constraints to handle multiple property nodes.

invoice for customer *C1* lists charges for the phone *0990/2020* in period *01/09* twice, which is clearly not desirable. Consequently, the combination {*01/09*, *0990*, *2020*} is redundantly represented in XML tree $\mathcal{T}_2$, and thus XML key $\mathcal{K}_1$ should be violated.

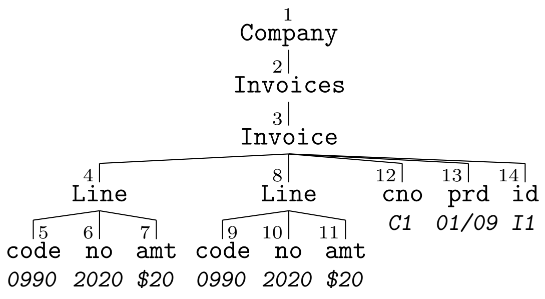


Figure 2.2: XML tree $\mathcal{T}_2$ used for testing the ability of XML keys to ensure uniqueness of property nodes.

Testing the ability of an approach to handle absent property nodes: The rationale for this test is that an XML integrity constraint must not be satisfied (or violated) solely because of the absence of some property nodes. For this purpose, the satisfaction of XML Key $\mathcal{K}_2$ and XML inclusion dependency $\mathcal{I}_2$ introduced in Examples 1.5 and 1.10 is checked in the XML trees depicted in Figures 2.3 and 2.4, respectively.

XML key $\mathcal{K}_2$ asserts that a customer's address is identified by the combination of city (`city`), street (`strt`), house number (`hno`), and apartment (`ano`). XML key $\mathcal{K}_2$ should

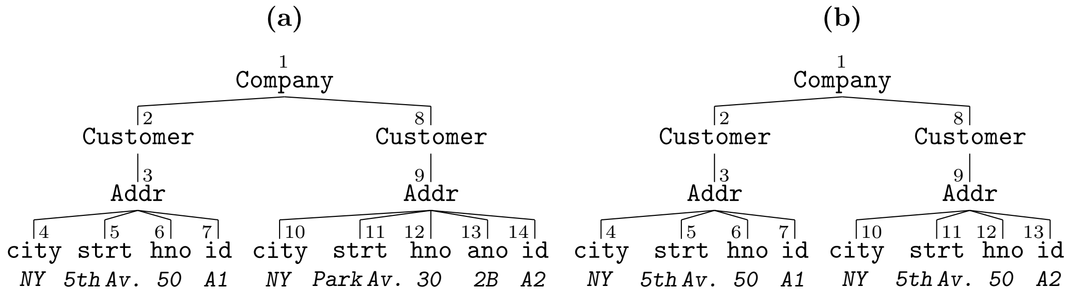


Figure 2.3: XML trees $\mathcal{T}_3$ and $\mathcal{T}_4$ used for testing the ability of XML keys and XML functional dependencies to handle absent property nodes.

be satisfied in XML tree $\mathcal{T}_3$, which is depicted in Figure 2.3a, for the obvious reason that {*NY*, *5th Avenue*, *50*}, and {*NY*, *Park Avenue*, *30*, *2B*} are different addresses. In contrast, XML key $\mathcal{K}_2$ should be violated in XML tree $\mathcal{T}_4$ depicted in Figure 2.3b. Both XML trees represent addresses which do not have an apartment number, and so absent `ano` nodes must be handled in checking the satisfaction of XML key $\mathcal{K}_2$.

In order to evaluate approaches to value-based XML integrity constraints for which a key constraint has not been defined so far, we introduce the `id` of an address and use XML functional dependency $\mathcal{F}_2$ instead of XML key $\mathcal{K}_2$. XML functional dependency $\mathcal{F}_2$ is inspired by XML key $\mathcal{K}_2$ and requires that the `id` of an address is determined by the combination of `city`, `strt`, `hno`, and `ano`. For reasons similar to which XML key $\mathcal{K}_2$ should be satisfied in XML tree $\mathcal{T}_3$ but violated in XML tree $\mathcal{T}_4$, also XML functional dependency $\mathcal{F}_2$ should be satisfied in XML tree $\mathcal{T}_3$ but violated in XML tree $\mathcal{T}_4$.

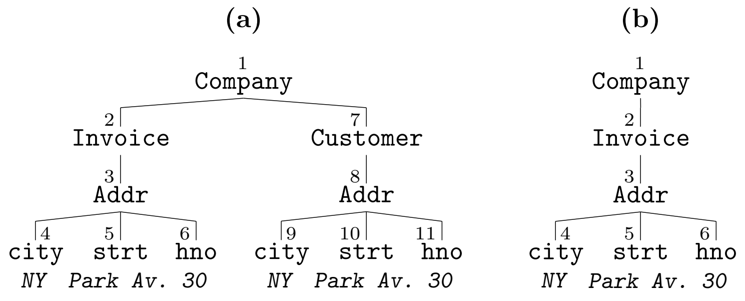


Figure 2.4: XML trees $\mathcal{T}_5$ and $\mathcal{T}_6$ used for testing the ability of XML inclusion dependencies to handle absent property nodes.

XML inclusion dependency $\mathcal{I}_2$ asserts that customer addresses are a subset of invoice addresses, and therefore requires that the combinations of `city`, `strt`, `hno`, and `ano` nodes nested within `invoice` nodes are a subset of the combinations of `city`, `strt`, `hno`, and `ano` nodes nested within customer nodes. XML inclusion dependency $\mathcal{I}_2$ should be satisfied in XML tree $\mathcal{T}_5$, which is depicted in Figure 2.4a, since $\{$*NY*, *Park Avenue*, *30*$\}$ is both an invoice address and also the address of a customer. In contrast, XML inclusion dependency $\mathcal{I}_2$ should be violated in XML tree $\mathcal{T}_6$ depicted in Figure 2.4b since $\{$*NY*, *Park Avenue*, *30*$\}$ is an invoice address but not the address of any customer. Because none of the addresses in XML trees $\mathcal{T}_5$ and $\mathcal{T}_6$ state an apartment number, absent `ano` nodes must be handled in checking XML inclusion dependency $\mathcal{I}_2$.

Testing the ability of an approach to preserve relational data semantics: We are specifically interested in whether an approach to value-based XML integrity constraints allows to preserve the semantics of relational keys and foreign keys when relational data is first nested and then mapped to XML as illustrated in Section 1.2.2.

To keep the number of XML integrity constraints and XML trees in the benchmark test as small as possible, XML integrity constraints $\mathcal{K}_1$ and $\mathcal{I}_1$ as well as XML tree $\mathcal{T}_1$ are used for evaluating the ability of an approach to XML integrity constraints to preserve relational data semantics. XML tree $\mathcal{T}_1$ can be used since it results from mapping relations `Invoice` and `Phone` depicted in Figure 1.5, after tuples in relation `Invoice` have been nested on the charged amount (`amt`) together with the code (`code`) and number (`no`) of phones, as illustrated in Example 1.6. Also, XML key $\mathcal{K}_1$ and XML inclusion dependency $\mathcal{I}_1$ can be used since they have been derived from the key (`code`, `no`, `prd`) in relation `Invoice` and the foreign key (`cno`, `code`, `no`) from relation `Invoice` to relation `Phone`, respectively. Hence, an approach to XML integrity constraints allows to preserve relational data semantics, if XML key $\mathcal{K}_1$ (or the corresponding XML functional dependency $\mathcal{F}_1$) and XML inclusion dependency $\mathcal{I}_1$ are satisfied in XML tree $\mathcal{T}_1$ according to the semantics of XML integrity constraints in the particular approach.

Table 2.2 summarizes the test cases of our benchmark test. We use test cases $\mathcal{C}1$ and $\mathcal{C}2$ to evaluate the ability of an approach to handle multiple property nodes and also to evaluate the ability of an approach to preserve the semantics of relational data when it is transformed to XML data. The specific ability of XML keys to ensure uniqueness of property nodes is evaluated using test case $\mathcal{C}3$. The remaining test cases $\mathcal{C}4$ - $\mathcal{C}7$ are used to evaluate the ability of an approach to handle absent property nodes.

Evaluated Ability	*Test Case*			
	#	*Benchmark Constraint*	*XML Tree*	*Expected Result*
Handling of multiple property nodes /	$\mathcal{C}1$	$\mathcal{K}_1/\mathcal{F}_1$	$\mathcal{T}_1$	satisfied
Preserving relational semantics	$\mathcal{C}2$	$\mathcal{I}_1$	$\mathcal{T}_1$	satisfied
Ensuring uniqueness of property nodes	$\mathcal{C}3$	$\mathcal{K}_1$	$\mathcal{T}_2$	violated
Handling of absent property nodes	$\mathcal{C}4$	$\mathcal{K}_2/\mathcal{F}_2$	$\mathcal{T}_3$	satisfied
	$\mathcal{C}5$	$\mathcal{K}_2/\mathcal{F}_2$	$\mathcal{T}_4$	violated
	$\mathcal{C}6$	$\mathcal{I}_2$	$\mathcal{T}_5$	satisfied
	$\mathcal{C}7$	$\mathcal{I}_2$	$\mathcal{T}_6$	violated

Table 2.2: Benchmark tests for value-based XML integrity constraints.

We now review previous approaches to value-based XML integrity constraints and evaluate them using the benchmark test developed in this section. The interested reader is referred to the excellent surveys in [10, 63–65] for comparative analyses of value-based XML integrity constraints from other perspectives.

2.2 ID and IDREF constraints in DTD

Early identification and reference mechanisms are the ID and IDREF constraints provided by DTDs. An ID constraint asserts that the values of distinguished attribute nodes are unique within an entire XML document, and the IDREF constraint allows to reference such ID attributes. Document Type Definitions also offer a set-valued reference constraint in form of the IDREFS constraint. The value of an attribute node of type IDREFS is a whitespace-separated list of references to ID nodes.

ID and IDREF constraints have limited expressivity in that they are unary constraints which are neither typed nor scoped. In this regard typing means that an XML integrity constraint allows for constraining distinguished types of data entities, whereas the scope of an XML integrity constraint denotes the sub-hierarchy in an XML document in which the constraint takes effect. Because ID and IDREF constraints are neither typed nor scoped, they are somewhat similar to object identifiers and object references known from the object oriented data model.

ID and IDREF constraints do not allow to specify any of the benchmark constraints listed in Table 2.2 because of their limited expressivity. Also, problems caused by multiple or absent property nodes are sidestepped since ID and IDREF constraints are unary constraints.

2.3 Selector/Field XML Integrity Constraints

The commonality of selector/field constraints is that they are made up of pairs of the form $S, \{F_1, \ldots, F_n\}$, where S is called the selector and $F_1, \ldots, F_n$ are called fields. The selector is used for selecting entity nodes in an XML document. For each selected entity node, the fields $F_1, \ldots, F_n$ specify sets of property nodes, as illustrated in the next example.

Example 2.1 *Consider the selector/field pair* `Company.Invoices.Invoice`, $\{$`Line.code`, `Line.no`$\}$ *and XML tree* $\mathcal{T}_1$. *The set of nodes selected by* `Company.Invoices.Invoice` *in* $\mathcal{T}_1$ *contains the two* `Invoice` *nodes* v_3 *and* v_{15}. *Regarding selector node* v_3, *fields* `Line.code` *and* `Line.no` *specify the following sets of field nodes:* $\{v_6, v_7\}$, $\{v_6, v_{11}\}$, $\{v_{10}, v_7\}$, *and* $\{v_{10}, v_{11}\}$. *Each of these sets of field nodes contains one* `code` *node and one* `no` *node. Note that only the sets* $\{v_6, v_7\}$ *and* $\{v_{10}, v_{11}\}$ *are semantically correct combinations of field nodes, since* $\{v_6, v_{11}\}$ *as well as* $\{v_{10}, v_7\}$ *contain a* `code` *node and a* `no` *node which do not belong to the same invoice line.*

Selector/field XML keys and foreign keys realize the notions of entity identification and entity references in a straight and direct manner. Selector nodes and field nodes thereby parallel entity nodes and property nodes, respectively. A selector/field key, which is made up of a single selector/field pair as depicted in Table 2.3, aims at the identification of selected entity nodes by specified sets of field nodes. A selector/field key asserts in an XML document that there do not exist distinct selector nodes v and v' with sets of field nodes $v_1, \ldots, v_n$ for v and $v'_1, \ldots, v'_n$ for v' such that $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_n$ are value equal.

XML Key	$(S, (F_1, \ldots, F_n))$
XML Inclusion Dependency	$(S, (F_1, \ldots, F_n)) \subseteq (S', (F'_1, \ldots, F'_n))$

Table 2.3: General form of selector/field XML keys and XML inclusion dependencies.

A selector/field foreign key, and also the more general selector/field inclusion dependency, is made up of two selector/field pairs as depicted in Table 2.3. We will refer to these selector/field pairs as the left hand side (LHS) and right hand side (RHS) selector/field in the following. A selector/field inclusion dependency asserts for an XML document that for each LHS selector node v with field nodes $v_1, \ldots, v_n$, there exists a RHS selector node v' with field nodes $v'_1, \ldots, v'_n$ such that $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_n$ are value equal.

Concerning the challenges which we have identified in Section 1.2, the primary question related to selector/field constraints is how sets of field nodes are formed in case that either multiple nodes are available for some field with respect to a given selector node, or conversely that no node is present at all for some field. In this regard the individual proposals for selector/field constraints form the following classes: *selector/field constraints with restrictions on fields*, *selector/field constraints with restrictions on field nodes*, *unrestricted selector/field constraints*. We devote Subsections 2.3.1 - 2.3.3 to these classes of selector/field constraints.

2.3.1 Selector/Field Constraints with Restrictions on Fields

Fan & Libkin originally proposed selector/field constraints which are defined on element types in a DTD. In particular, the selector of a constraint is specified in form of an element type, and the fields are specified in form of attributes of the selected element type. In this setting Fan & Libkin proposed keys, inclusion dependencies, and foreign keys for XML [66], but also the more exotic types of XML non-identification constraints and XML exclusion constraints [15], which are roughly speaking negations of keys and inclusion dependencies, respectively. Arenas et al. proposed scoped versions of the keys, foreign keys, and inclusion dependencies in the class of selector/field constraints with restrictions on fields [16]. The scope of a constraint is thereby also specified in form of an element type.

The expressivity of selector/field constraints with restrictions on fields is rather low because of the requirement on fields to be attributes declared for the selected element type. Except for XML key $\mathcal{K}_2$, it is not possible to specify any of the benchmark constraints depicted in Table 2.2 in form of a selector/field constraint with restrictions on fields. To illustrate XML key $\mathcal{K}_2$ in form of a selector/field key with restrictions on fields, Figure 2.5 depicts DTD $\mathcal{D}_1$ which specifies the structure of XML tree $\mathcal{T}_3 - \mathcal{T}_6$. Attributes `city`, `strt`, `hno`, and `ano` are implied attributes in DTD $\mathcal{D}_1$. This permits `Addr` nodes in conforming XML trees to lack any of these attribute nodes, and so reflects possible heterogeneity in addresses. With respect to DTD $\mathcal{D}_1$, XML key $\mathcal{K}_2$ is of the form (`Addr`, (`city`, `strt`, `hno`, `ano`)), where `Addr` is the selected element type, and `city`, `strt`, `hno`, and `ano` are attributes declared for element type `Addr`.

```
<!DOCTYPE COMPANY[
  <!ELEMENT Company (Customer*,Invoice*)>
  <!ELEMENT Customer (Addr)>
  <!ELEMENT Invoice (Addr)>
  <!ELEMENT Addr EMPTY>
  <!ATTLIST Addr city CDATA #IMPLIED>
  <!ATTLIST Addr strt CDATA #IMPLIED>
  <!ATTLIST Addr hno CDATA #IMPLIED>
  <!ATTLIST Addr ano CDATA #IMPLIED>
  <!ATTLIST Addr id CDATA #REQUIRED>
]>
```

Figure 2.5: DTD $\mathcal{D}_1$ specifying the structure of XML trees $\mathcal{T}_3 - \mathcal{T}_6$.

We now detail on how selector/field constraints with restrictions on fields deal with multiple or absent field nodes.

Handling of multiple field nodes: According to the XML specification by the W3C, no two distinct attribute nodes are permitted to have the same label if they have the same parent node. From the point of view of an element node this means that for a given label at most one attribute node is permitted. In combination with the restriction on fields to be attributes of selected element types this means that for each selector node there is at most one field node per field and so multiple field nodes do not occur during the validation of selector/field constraints with restrictions on fields. Thus, the problems related to multiple property nodes are sidestepped by selector/field constraints with restrictions on fields.

Handling of absent field nodes: The approaches presented in [15, 16, 66] assume that all attributes declared in a DTD are 'required' attributes as opposed to 'implied' attributes which are used in DTD $\mathcal{D}_1$. Hence, DTD $\mathcal{D}_1$ and, as a consequence, XML trees $\mathcal{T}_3$ and $\mathcal{T}_4$ are invalid according to the approaches presented in [15, 16, 66]. Whereas benchmark XML key $\mathcal{K}_2$ can be syntactically specified in terms of a selector/field key with restrictions on fields, which we have illustrated previously, $\mathcal{K}_2$ is neither satisfied nor violated in XML trees $\mathcal{T}_3$ and $\mathcal{T}_4$ since these XML trees are, due to limited support of DTD, invalid in the approaches presented in [15, 16, 66].

In general, the assumption that all attributes in a DTD are required attributes, in combination with the restriction on fields to be attributes of selected element types implies that for each selected entity node there is exactly one field node per field. Therefore, absent field nodes do not occur during the validation of selector/field constraints with restrictions on fields. Hence, also the problems related to absent property nodes are sidestepped by selector/field constraints with restrictions on fields.

We conclude the discussion of selector/field constraints with restrictions on fields by giving an overview on related theoretical results. The class of selector/field XML integrity constraints with restrictions on fields has been extensively studied. The combined implication problem related to keys and inclusion dependencies has been shown to be undecidable in general [15], which is expected given the well known result on the combined implication problem for functional dependencies and inclusion dependencies in the relational setting. Under the primary key assumption, which is that there is at most one key for each element type in a DTD, the implication problem becomes tractable, and a set of sound and complete inference rules has been presented for this special case [15].

As for the implication problem, the consistency problem related to keys and inclusion dependencies in the class of selector/field constraints with restrictions on fields has been shown to be undecidable in the presence of conventional DTDs [15] and also in the presence of DTDs that allow for subtyping [67]. That is, it cannot be determined in general whether there exists at least one XML document that conforms to a DTD and satisfies a given set of keys and inclusion dependencies. The consistency problem becomes tractable for unary keys and inclusion dependencies [15].

2.3.2 Selector/Field Constraints with Restrictions on Field Nodes

The XML Schema specification provides scoped selector/field constraints where the scope of a constraint is specified in form of an element type definition in the XML Schema. The selector as well as the fields of a constraint are specified in form of restricted XPath expressions [61]. The permitted fragment of XPath allows roughly speaking for downward navigation with wildcards but excludes predicates and functions. The peculiarity of selector/field constraints in XML Schema is that the cardinality of field nodes is implicitly constrained in that *at most one* field node is permitted per field for a given selector node.

In this setting the XML Schema specification provides two types of XML keys and also an XML foreign key. The two types of XML keys, subsequently called 'XSD key' and 'XSD unique', parallel the notions of primary keys and candidate keys known from the relational setting. Tan et al. [68] proposed a scoped selector/field functional dependency which requires in accordance to the XML Schema constraints that there is at most one field node per field for a given selector node. Unlike XSD unique constraints and XML functional dependencies presented in [68], XSD keys as well as XSD foreign keys additionally require that there is *at least one* field node per field for a given selector node, and so there must be *exactly one* field node per field.

We now detail on how these implicit cardinality constraints effect the semantics of a selector/field constraint in case that multiple or absent field nodes occur in an XML document. In this regard, XSD unique constraints are equivalent to the XML functional dependencies presented in [68] since both types of constraints impose the same cardinality constraint on field nodes. We therefore do not explicitly address the latter in the subsequent discussion.

Compared to selector/field constraints with restrictions on fields, XSD constraints provide greater flexibility for specifying fields, and in fact allow to specify benchmark constraints $\mathcal{K}_1$, $\mathcal{K}_2$, $\mathcal{I}_1$, and $\mathcal{I}_2$, which are shown in Table 2.4 in form of XSD constraints. Since $\mathcal{K}_1$, $\mathcal{K}_2$,

$\mathcal{I}_1$, and $\mathcal{I}_2$ are all global constraints, the explicit specification of the scope of a constraint is omitted in Table 2.4 for reasons of brevity.

$\mathcal{K}_1$	$(//\texttt{Invoice}, (\texttt{prd}, \texttt{Line.code}, \texttt{Line.no}))$
$\mathcal{K}_2$	$(//\texttt{Addr}, (\texttt{city}, \texttt{strt}, \texttt{hno}, \texttt{ano}))$
$\mathcal{I}_1$	$(//\texttt{Invoice}, (\texttt{cno}, \texttt{Line.code}, \texttt{Line.no})) \subseteq (//\texttt{Phone}, (\texttt{cno}, \texttt{code}, \texttt{no}))$
$\mathcal{I}_2$	$(//\texttt{Invoice.Addr}, (\texttt{city}, \texttt{strt}, \texttt{hno}, \texttt{ano})) \subseteq (//\texttt{Customer.Addr}, (\texttt{city}, \texttt{strt}, \texttt{hno}, \texttt{ano}))$

Table 2.4: Benchmark constraints $\mathcal{K}_1$, $\mathcal{K}_2$, $\mathcal{I}_1$, and $\mathcal{I}_2$ specified as selector/field constraints.

Handling of multiple field nodes: Because XSD constraints require that there is at most one field node per field for a given selector node, an XSD constraint is violated if multiple field nodes occur. For example, according to the semantics of XSD constraints, $\mathcal{K}_1$ and $\mathcal{I}_1$ are both violated in XML tree $\mathcal{T}_1$, since `Invoice` node v_3, which is selected by both $\mathcal{K}_1$ and $\mathcal{I}_1$, obviously has multiple field nodes for fields `Line.code` and `Line.no`, i.e. nodes $\{v_6, v_{10}\}$ and $\{v_7, v_{11}\}$, respectively. Given that $\mathcal{K}_1$ and $\mathcal{I}_1$ should be satisfied in XML tree $\mathcal{T}_1$, XSD constraints do not adequately handle multiple field nodes.

Ensuring uniqueness of field nodes: A direct consequence of the implicit cardinality constraints on field nodes imposed by XSD keys and XSD unique constraints is that there is at most one set of field nodes for a given selector node, and so the uniqueness of field nodes is ensured. In fact, according to the semantics of XSD keys and XSD unique constraints, $\mathcal{K}_1$ is violated in XML tree $\mathcal{T}_2$, which is desired. In particular, the selected `Invoice` node v_3 in XML tree $\mathcal{T}_2$ has multiple field nodes for both fields `Line.code` and `Line.no`, i.e. nodes $\{v_5, v_9\}$ and $\{v_6, v_{10}\}$, and so $\mathcal{K}_1$ is violated in XML tree $\mathcal{T}_2$ according to the semantics of XSD keys and XSD unique constraints.

Handling of absent field nodes: The implicit cardinality constraints on field nodes imposed by XSD keys and XSD foreign keys prohibit absent field nodes, and so $\mathcal{K}_2$ is violated in both XML trees $\mathcal{T}_3$ and $\mathcal{T}_4$, as well as $\mathcal{I}_2$ is violated in both XML trees $\mathcal{T}_5$ and $\mathcal{T}_6$ according to the semantics of XSD constraints. The reason for this is that the selected `Addr` node v_3 does not have a field node for the field `ano` in each one of these XML trees. Given that $\mathcal{K}_2$ and $\mathcal{I}_2$ should be satisfied in XML trees $\mathcal{T}_3$ and $\mathcal{T}_5$, respectively, XSD keys and XSD foreign keys do not adequately handle absent field nodes.

In contrast to XSD keys, absent field nodes are permitted by XSD unique constraints, which only requires that there is at most one field node per field for a given selector node. Unfortunately, selector nodes which do not have field nodes for some fields are simply ignored when checking the satisfaction of an XSD unique constraint. For example, regarding benchmark constraint $\mathcal{K}_2$ and XML tree $\mathcal{T}_4$, the `Addr` nodes v_3 and v_9 are ignored since both nodes do not have a field node for the field `ano`. Consequently, $\mathcal{K}_2$ is satisfied in XML tree $\mathcal{T}_4$ according to the semantics of XSD unique constraints. Since this is not desired, also XSD unique constraints do not adequately handle absent property nodes.

We conclude the discussion of selector/field constraints with restrictions on field nodes by giving an overview on related theoretical results. Only little attention has been paid so far on the study of this class of XML integrity constraints. Regarding XSD keys and XSD foreign keys, the only result established so far is that the related consistency problem is undecidable in the presence of a DTD [67, 69]. For the selector/field functional dependency proposed by Tan et al. in [68], a validation algorithm based on tree-automata has been presented recently which allows to validate an XML document against a DTD and a set of functional dependencies in a single-pass [70].

2.3.3 Unrestricted Selector/Field Constraints

Buneman et al. originally proposed a scoped selector/field key which neither puts restrictions on fields nor on the cardinality of field nodes [17, 71]. For specifying the scope, the selector and the fields of an unrestricted selector/field key, Buneman et al. proposed a path language which basically has the same expressivity as the restricted form of XPath used for specifying XSD keys and XSD foreign keys (cf. Subsection 2.3.2). In the class of unrestricted selector/field constraints, Wang & Topor proposed global functional dependencies and global equality generating dependencies [72]. Essentially, an XML equality generating dependency as proposed in [72] is an XML functional dependency which asserts value equality between field nodes of *two distinct* sets of selector nodes. For this purpose, XML equality generating dependencies as proposed by Wang & Topor have separate selectors for the LHS and RHS of the constraint. A scoped functional dependency in the class of unrestricted selector/field constraints has been proposed by Ahmad & Ibrahim [24].

The selector/field key proposed by Buneman et al. in [71] is representative for the class of unrestricted selector/field constraints with regard to how multiple and absent field nodes are handled. We therefore use the selector/field key as proposed by Buneman et al. for the evaluation of unrestricted selector/field constraints.

Handling of multiple field nodes: Selector/field keys as proposed by Buneman et al. compare selected entity nodes on the basis of all possible sets of field nodes. Consequently, semantically incorrect sets of field nodes are potentially used for the comparison of selected entity nodes. Consider for example benchmark constraint $\mathcal{K}_1$, which is depicted in Table 2.4 in form of a selector/field key as proposed by Buneman et al. With respect to XML tree $\mathcal{T}_1$, `Invoice` nodes v_3 and v_{15} are selected by $\mathcal{K}_1$. Also, the field nodes for v_3 are the single node v_{14} for the field `prd`, nodes $\{v_6, v_{10}\}$ for the field `Line.code`, and nodes $\{v_7, v_{11}\}$ for the field `Line.no`. Hence, the selected node v_3 is compared to node v_{15} on the basis of the values of nodes (v_{14}, v_6, v_7), (v_{14}, v_6, v_{11}), (v_{14}, v_{10}, v_7), and (v_{14}, v_{10}, v_{11}). Since the semantically incorrect set of field nodes (v_{14}, v_6, v_{11}) is value equal to the single set of field nodes (v_{22}, v_{18}, v_{19}) for node v_{15}, benchmark constraint $\mathcal{K}_1$ is violated in XML tree $\mathcal{T}_1$ according to the semantics of selector/field keys as proposed by Buneman et al. Since $\mathcal{K}_1$ should be satisfied in XML tree $\mathcal{T}_1$, unrestricted selector/field constraints do not adequately handle multiple field nodes.

Ensuring uniqueness of field nodes: Selector/field keys as proposed by Buneman et al. are only violated in an XML document if *distinct* selector nodes have value equal combinations of field nodes. Consequently, uniqueness of field nodes is not ensured. For example, benchmark constraint $\mathcal{K}_1$ is satisfied in XML tree $\mathcal{T}_2$ according to the semantics of selector/field keys as proposed by Buneman et al. since the duplicate sets of field nodes (v_{13}, v_5, v_6) and (v_{13}, v_9, v_{10}) belong to the same selected `Invoice` node v_3.

Handling of absent field nodes: Selector/field keys as proposed by Buneman et al. compare selected entity nodes only on the basis of 'complete' sets of field nodes. A set of field nodes is thereby complete, if it contains one node for each field in the constraint. Hence, benchmark constraint $\mathcal{K}_2$, which is depicted in Table 2.4 in form of a selector/field key as proposed by Buneman et al., is satisfied in XML tree $\mathcal{T}_4$ since selected `Addr` nodes v_3 and v_9 are not compared on the basis of the sets of field nodes $\{v_4, v_5, v_6\}$ and $\{v_{10}, v_{11}, v_{12}\}$. Note that both sets of field nodes are missing an `ano` node and are therefore incomplete. Since $\mathcal{K}_2$ should be violated in $\mathcal{T}_4$, unrestricted selector/fied integrity constraints do not adequately handle absent field nodes.

We conclude the discussion of unrestricted selector/field constraints by giving an overview on related theoretical results. The study of unrestricted selector/field constraints focussed exclusively on the selector/field key proposed by Buneman et al. in [17]. Chen et al. presented a validation algorithm for the selector/field key as proposed by Buneman et al. This validation algorithm is based on SAX and has linear runtime complexity in the size of the XML document [73]. Buneman et al. studied the related implication problem and presented a decision algorithm as well as a set of inference rules [74, 75]. Unfortunately, one of the inference rules turned out to be incorrect, and Hartmann & Link developed a sound and complete set of inference rules for the subclass of selector/field keys as proposed by Buneman where only simple paths are permitted in order to specify the fields of a key [76, 77]. Hartmann & Link also developed a sound and complete set of inference rules for another subclass of these keys, called structural keys, where the selector coincides with the fields [78]. However, for the general class of keys as proposed by Buneman et al., the related implication problem is still unsolved.

2.4 Tuple-based XML Integrity Constraints

To translate the notion of a tuple from the relational model to the XML data model is the common for what we call tuple-based XML integrity constraints. In general, the syntax of a tuple-based constraint parallels the syntax of its relational counterpart, where paths take on the role of attributes. For illustration, Table 2.5 shows the general forms of tuple-based functional dependencies and inclusion dependencies.

Tuple-based Functional Dependency	$\{P_1, \ldots, P_m\} \rightarrow \{P'_1, \ldots, P'_n\}$
Tuple-based Inclusion Dependency	$[P_1, \ldots, P_n] \subseteq [P'_1, \ldots, P'_n]$

Table 2.5: General form of tuple-based functional dependencies and inclusion dependencies.

Apart from its syntax, also the semantics of a tuple-based constraint is inspired by the semantics of its relational counterpart. Checking an XML document against a tuple-based constraint means in general to form tuples of nodes w.r.t. the paths in the constraint, and to compare these tuples of nodes as required by the particular type of constraint. Hence, a tuple-based functional dependency asserts that whenever there exist distinct tuples of nodes $\langle v_1, \ldots, v_m, v'_1, \ldots, v'_n \rangle$ and $\langle \bar{v}_1, \ldots, \bar{v}_m, \bar{v}'_1, \ldots, \bar{v}'_n \rangle$ for the set of paths $\{P_1, \ldots, P_m, P'_1, \ldots, P'_n\}$ such that the LHS nodes $\langle v_1, \ldots, v_m \rangle$ and $\langle \bar{v}_1, \ldots, \bar{v}_m \rangle$ are value equal, then also the RHS nodes $\langle v'_1, \ldots, v'_n \rangle$ and $\langle \bar{v}'_1, \ldots, \bar{v}'_n \rangle$ are value equal. Likewise, a tuple-based inclusion dependency asserts that whenever there exists a tuple of nodes $\langle v_1, \ldots, v_n \rangle$ for the LHS paths $\{P_1, \ldots, P_n\}$, there also exists a tuple of nodes $\langle v'_1, \ldots, v'_n \rangle$ for the RHS paths $\{P'_1, \ldots, P'_n\}$ such that $\langle v_1, \ldots, v_n \rangle$ and $\langle v'_1, \ldots, v'_n \rangle$ are value equal.

In contrast to the notions of tuple-based functional dependencies and inclusion dependencies, there is no common notion of a tuple-based key. In some tuple-based approaches, a key is a special case of a functional dependency. In other tuple-based approaches, keys and functional dependencies are separately defined. We will therefore address the notion of a tuple-based key separately for each individual approach.

Concerning the challenges which we have identified in Section 1.2, the primary question related to tuple-based XML integrity constraints is how tuples of nodes are formed for the paths in a constraint in case that either multiple nodes are reachable over some path, or conversely that no node is reachable over some path at all. In this regard the individual approaches to tuple-based XML integrity constraints form four groups, to which the subsequent Sections 2.4.1 - 2.4.4 are devoted.

2.4.1 The Intersection Path Approach

The tuple-based functional dependency presented by Yan & Lv in [79] and also the tuple-based inclusion dependency presented by Vincent et al. in [80] follow what we call the intersection path approach. In this approach, a set of nodes $v_1, \ldots, v_n$ forms a tuple for a set of paths $P_1, \ldots, P_n$ if nodes $v_1, \ldots, v_n$ have a common ancestor node which is reachable over the intersection path of $P_1, \ldots, P_n$. In the approach to functional dependencies proposed by Yan & Lv in [79], it is additionally required that both the LHS nodes and the RHS nodes in a tuple have common ancestor nodes at the intersection path of the LHS paths and the RHS paths, respectively. The paths in an intersection path constraint are simple paths leading to attribute or text nodes, and so element nodes are not permitted to be used as property nodes. Also, a key has not been proposed so far for the intersection path approach.

We now evaluate the ability of intersection path constraints to handle the challenges which we have identified in Section 1.2. In particular, we use benchmark constraints $\mathcal{F}_1$ and $\mathcal{F}_2$ for the evaluation of the intersection path functional dependency defined in [79], and we use benchmark constraints $\mathcal{I}_1$ and $\mathcal{I}_2$ for the evaluation of the intersection path inclusion dependency defined in [80]. Table 2.6 lists these benchmark constraints in the common form of tuple-based functional dependencies and tuple-based inclusion dependencies.

Handling of multiple property nodes: The structural cohesion between nodes in a tuple is very low in the intersection path approach. Consequently, semantically incorrect tuples of nodes are formed. To see this, consider benchmark constraint $\mathcal{F}_1$ and XML tree $\mathcal{T}_1$. Table

$\mathcal{F}_1$	$\{$`Company.Invoices.Invoice.prd`, `Company.Invoices.Invoice.Line.code`, `Company.Invoices.Invoice.Line.no`$\} \rightarrow$ `Company.Invoices.Invoice.id`
$\mathcal{I}_1$	$[$`Company.Invoices.Invoice.cno`, `Company.Invoices.Invoice.Line.code`, `Company.Invoices.Invoice.Line.no`$] \subseteq$ $[$`Company.Phones.Phone.cno`, `Company.Phones.Phone.code`, `Company.Phones.Phone.no`$]$
$\mathcal{F}_2$	$\{$`Company.Customer.Addr.city`, `Company.Customer.Addr.strt`, `Company.Customer.Addr.hno`, `Company.Customer.Addr.ano`$\} \rightarrow$ `Company.Customer.Addr.id`
$\mathcal{I}_2$	$[$`Company.Invoice.Addr.city`, `Company.Invoice.Addr.strt`, `Company.Invoice.Addr.hno`, `Company.Invoice.Addr.ano`$] \subseteq$ $[$`Company.Customer.Addr.city`, `Company.Customer.Addr.strt`, `Company.Customer.Addr.hno`, `Company.Customer.Addr.ano`$]$

Table 2.6: Benchmark constraints $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{I}_1$, and $\mathcal{I}_2$ specified as tuple-based constraints.

2.7 lists the tuples of nodes in $\mathcal{T}_1$ which are formed for the paths in $\mathcal{F}_1$. Recall that an intersection path functional dependency requires the RHS and LHS nodes in a tuple to have common ancestor nodes at the intersection of the RHS paths and the LHS paths in the constraint. Regarding benchmark constraint $\mathcal{F}_1$ this means that the LHS nodes in a tuple must have a common ancestor node at path `Company.Invoices.Invoice` and that the RHS nodes in a tuple must have a common ancestor node at path `Company.Invoices.Invoice.id`. Note that $\mathcal{F}_1$ is a unary functional dependency and that therefore the RHS intersection path equals the single RHS path `Company.Invoices.Invoice.id`. The common ancestor nodes for the RHS nodes and LHS nodes are indicated by the gray node identifiers to the left and right of a tuple of nodes in Table 2.7. The gray node identifier to the far right of a tuple of nodes indicates the common ancestor node at the intersection of all paths in $\mathcal{F}_1$, i.e. `Company.Invoices.Invoice`. For reasons of clarity, the values of attribute nodes are given below each tuple in Table 2.7. Tuples $\langle v_{14}, v_6, v_{11}, v_4\rangle$ and $\langle v_{14}, v_{10}, v_7, v_4\rangle$ are semantically incorrect, since neither the combination (v_6, v_{11}) nor the combination (v_{10}, v_7) of a `code` and a `no` node belong to the same invoice line. Consequently, $\mathcal{F}_1$ is violated in $\mathcal{T}_1$ since the two tuples of nodes $\langle v_{14}, v_6, v_{11}, v_4\rangle$ and $\langle v_{22}, v_{18}, v_{19}, v_{16}\rangle$ have value equal LHS nodes as opposed to the RHS nodes v_4 and v_{16} which have distinct values.

Consider now benchmark constraint $\mathcal{I}_1$ and XML tree $\mathcal{T}_1$. Table 2.8 lists the tuples of nodes in $\mathcal{T}_1$ which are formed for the paths in $\mathcal{I}_1$. The gray node identifier to the right of a tuple thereby indicates the common ancestor node at the intersection path, which is path `Company.Invoices.Invoice` for the LHS tuples and path `Company.Phones.Phone` for the RHS tuples. Tuples $\langle v_{13}, v_6, v_{11}\rangle$ and $\langle v_{13}, v_{10}, v_7\rangle$ are semantically incorrect. The reason for this is again that neither nodes (v_6, v_{11}) nor nodes (v_{10}, v_7) belong to the same invoice line. Consequently, also $\mathcal{I}_1$ is violated in $\mathcal{T}_1$, since there are no RHS tuples of nodes which are value equal to the semantically incorrect tuples of nodes $\langle v_{13}, v_6, v_{11}\rangle$ and $\langle v_{13}, v_{10}, v_7\rangle$.

Company.Invoices.Invoice	Company.Invoices.Invoice.prd	Company.Invoices.Invoice.Line.code	Company.Invoices.Invoice.Line.no	Company.Invoices.Invoice.id	Company.Invoices.Invoice.id	Company.Invoices.Invoice
v_3	$\langle v_{14}$	v_6	v_7	$v_4\rangle$	v_4	v_3
	01/09	0660	1010	I1		
v_3	$\langle v_{14}$	v_6	v_{11}	$v_4\rangle$	v_4	v_3
	01/09	0660	2020	I1		
v_3	$\langle v_{14}$	v_{10}	v_{11}	$v_4\rangle$	v_4	v_3
	01/09	0990	2020	I1		
v_3	$\langle v_{14}$	v_{10}	v_7	$v_4\rangle$	v_4	v_3
	01/09	0990	1010	I1		
v_{15}	$\langle v_{22}$	v_{18}	v_{19}	$v_{16}\rangle$	v_{16}	v_{15}
	01/09	0600	2020	I2		

Table 2.7: Tuples of nodes in XML Tree $\mathcal{T}_1$ formed for benchmark constraint $\mathcal{F}_1$ according to the intersection path approach.

Since benchmark constraints $\mathcal{F}_1$ and $\mathcal{I}_1$ both should be satisfied in XML tree $\mathcal{T}_1$, multiple property nodes are not adequately handled in the intersection path approach.

Handling of absent property nodes: Incomplete tuples of nodes are simply ignored by intersection path functional dependencies and inclusion dependencies. A tuple of nodes is incomplete if it does not contain a node for each path in the constraint. Consequently, benchmark constraints $\mathcal{F}_2$ and $\mathcal{I}_2$ are satisfied in XML trees $\mathcal{T}_4$ and $\mathcal{T}_6$ according to the semantics of intersection path functional dependencies and inclusion dependencies. Since benchmark constraints $\mathcal{F}_2$ and $\mathcal{I}_2$ should be violated in XML trees $\mathcal{T}_4$ and $\mathcal{T}_6$, absent property nodes are not adequately handled in the intersection path approach.

We conclude the discussion of intersection path constraints by giving an overview on related theoretical results. Vincent et al. presented a sound and complete set of inference rules for the implication of intersection path inclusion dependencies [80], which implies that the related implication problem is decidable. The consistency problem related to intersection path inclusion dependencies has not been studied so far.

Regarding intersection path functional dependencies, it has been shown by Lv & Yan that they preserve the semantics of relational functional dependencies when relational data is transformed to XML data on the basis of nested tuples [40]. In establishing this result Lv & Yan considered the restricted case where only one nesting operation on a single attribute is permitted prior to the mapping of the relation to XML. The result in [40] is therefore in accordance with the finding in our evaluation that intersection path functional depen-

Company.Invoices.Invoice.cno	Company.Invoices.Invoice.Line.code	Company.Invoices.Invoice.Line.no	Company.Invoices.Invoice	Company.Phones.Phone.cno	Company.Phones.Phone.code	Company.Phones.Phone.no	Company.Phones.Phone
$\langle v_{13}$	v_6	$v_7\rangle$	v_3	$\langle v_{27}$	v_{25}	$v_{26}\rangle$	v_{24}
C1	0660	1010		C1	0660	1010	
$\langle v_{13}$	v_6	$v_{11}\rangle$	v_3	$\langle v_{31}$	v_{29}	$v_{30}\rangle$	v_{28}
C1	0660	2020		C1	0990	2020	
$\langle v_{13}$	v_{10}	$v_{11}\rangle$	v_3	$\langle v_{35}$	v_{33}	$v_{34}\rangle$	v_{32}
C1	0990	2020		C2	0660	2020	
$\langle v_{13}$	v_{10}	$v_7\rangle$	v_3				
C1	0990	1010					
$\langle v_{21}$	v_{18}	$v_{19}\rangle$	v_{15}				
C2	0600	2020					

Table 2.8: Tuples of nodes in XML Tree $\mathcal{T}_1$ formed for benchmark constraint $\mathcal{I}_1$ according to the intersection path approach.

dencies do not adequately handle multiple field nodes. Note that this finding suggests that intersection path functional dependencies do not allow to preserve the semantics of relational functional dependencies in the general case where arbitrary nesting operations are permitted prior to the mapping.

2.4.2 The Tree Tuple Approach

Arenas & Libkin originally proposed the 'tree tuple' approach when defining an XML functional dependency [23]. Conceptually, tuples of nodes are formed in the 'tree tuple' approach as follows. The XML document, which is expected to conform to a DTD, is transformed into a flat relation by the total unnesting of the XML document. The attributes of the resulting relation correspond to the paths indicated by the DTD, and the values in the relation represent the nodes in the XML document. A set of nodes in the XML document then forms a tuple, if the nodes were mapped to the same tuple in the relation. For the purpose of illustration, Table 2.9 lists the 'tree tuples' that result from the total unnesting of the `Invoices` subtrees of XML trees $\mathcal{T}_1$ and $\mathcal{T}_2$. We note that the `Phones` subtree in $\mathcal{T}_1$ has been omitted for reasons of brevity. The paths in the header of Table 2.9 are derived from DTD $\mathcal{D}_1$ (cf. Section 2.3.1), to which XML trees $\mathcal{T}_1$ and $\mathcal{T}_2$ conform to.

Yu & Yagdish extended the 'tree tuple' functional dependency developed by Arenas & Libkin [23] with the flavor of a set-valued constraint [81, 82]. Such an extended 'tree tuple'

`Company`	`Company.Invoices`	`Company.Invoices.Invoice`	`Company.Invoices.Invoice.id`	`Company.Invoices.Invoice.cno`	`Company.Invoices.Invoice.prd`	`Company.Invoices.Invoice.Line`	`Company.Invoices.Invoice.Line.amt`	`Company.Invoices.Invoice.Line.code`	`Company.Invoices.Invoice.Line.no`	
$\langle v_1$	v_2	v_3	v_4	v_{13}	v_{14}	v_5	v_8	v_6	$v_7\rangle$	*XML Tree $\mathcal{T}_1$*
			`I1`	`C1`	`01/09`		`$10`	`0660`	`1010`	
$\langle v_1$	v_2	v_3	v_4	v_{13}	v_{14}	v_9	v_{12}	v_{10}	$v_{11}\rangle$	
			`I1`	`C1`	`01/09`		`$20`	`0990`	`2020`	
$\langle v_1$	v_2	v_{15}	v_{16}	v_{21}	v_{22}	v_{17}	v_{20}	v_{18}	$v_{19}\rangle$	
			`I2`	`C2`	`01/09`		`$30`	`0660`	`2020`	
$\langle v_1$	v_2	v_3	v_{14}	v_{12}	v_{13}	v_4	v_7	v_5	$v_6\rangle$	*XML tree $\mathcal{T}_2$*
			`I1`	`C1`	`01/09`		`$20`	`0990`	`2020`	
$\langle v_1$	v_2	v_3	v_{14}	v_{12}	v_{13}	v_8	v_{11}	v_9	$v_{10}\rangle$	
			`I1`	`C1`	`01/09`		`$20`	`0990`	`2020`	

Table 2.9: Tree tuples in benchmark XML Trees $\mathcal{T}_1$ and $\mathcal{T}_2$.

functional dependency allows for example to express that the *set* of phones in the lines of an invoice determines the customer who is charged. The single-valued and the set-valued notions of a 'tree tuple' functional dependency deal with multiple and absent property nodes in the same manner. We therefore restrict our subsequent discussion on the original proposal by Arenas & Libkin [23].

Unlike for an intersection path functional dependency, the paths in a 'tree tuple' functional dependency may also lead to element nodes. The *identities* of element nodes are thereby used for the comparison of element nodes when the satisfaction of a 'tree tuple' functional dependency is checked in an XML document. That is, according to the 'tree tuple' approach, two element nodes are 'value' equal if they have the same identity. Hence, if a 'tree tuple' functional dependency has an RHS path that leads to element nodes, then it is satisfied in an XML document if whenever a pair of 'tree tuples' have value equal LHS nodes, then the RHS element nodes are the same node, and so LHS nodes identify the RHS nodes. Thus, a 'tree tuple' functional dependency has some flavor of an XML key if it has a single RHS path that leads to element nodes.

We now evaluate the ability of 'tree tuple' functional dependencies to handle the challenges which we have identified in Section 1.2. In particular, we use benchmark constraints $\mathcal{K}_1$ and $\mathcal{K}_2$ for the evaluation. Table 2.10 lists these benchmark constraints in the form of 'tree tuple' functional dependencies.

Handling of multiple property nodes: Consider $\mathcal{K}_1$ depicted in Table 2.10 and the 'tree tuples' in XML tree $\mathcal{T}_1$, which are depicted in the top of Table 2.9. The projection

$\mathcal{K}_1$	{`Company.Invoices.Invoice.prd`, `Company.Invoices.Invoice.Line.code`, `Company.Invoices.Invoice.Line.no`} $\rightarrow$ `Company.Invoices.Invoice`
$\mathcal{K}_2$	{`Company.Customer.Addr.city`, `Company.Customer.Addr.strt`, `Company.Customer.Addr.hno`, `Company.Customer.Addr.ano`} $\rightarrow$ `Company.Customer.Addr`

Table 2.10: Benchmark constraints $\mathcal{K}_1$ and $\mathcal{K}_2$ specified as 'tree tuple' functional dependencies.

of these 'tree tuples' on the paths in $\mathcal{K}_1$ yields tuples $\langle v_{14}, v_6, v_7, v_3\rangle$, $\langle v_{14}, v_{10}, v_{11}, v_3\rangle$, and $\langle v_{21}, v_{18}, v_{19}, v_{16}\rangle$, where no two combinations of LHS nodes, i.e. (v_{14}, v_6, v_7), (v_{14}, v_{11}, v_{10}), and (v_{21}, v_{18}, v_{19}), are value equal (cf. Table 2.9). As a consequence, $\mathcal{K}_1$ is satisfied in $\mathcal{T}_1$ according to the semantics of 'tree tuple' functional dependencies, and so multiple property nodes are adequately handled in the 'tree tuple' approach.

Ensuring uniqueness of property nodes: Whereas 'tree tuple' functional dependencies capture the identification property of an XML key, they do not capture the uniqueness property. To see this, consider XML key $\mathcal{K}_1$ and the 'tree tuples' in XML tree $\mathcal{T}_2$ which are given in the bottom of Table 2.9. The projections of these 'tree tuples' to the paths in $\mathcal{K}_1$ are $\langle v_{13}, v_5, v_6, v_3\rangle$ and $\langle v_{13}, v_{10}, v_9, v_3\rangle$, and so $\mathcal{K}_1$ is satisfied in $\mathcal{T}_2$ according to the semantics of 'tree tuple' functional dependencies. Since $\mathcal{K}_1$ should be violated in $\mathcal{T}_2$, 'tree tuple' functional dependencies do not ensure the uniqueness of property nodes.

Handling of absent property nodes: 'Tree tuple' functional dependencies apply a kind of weak satisfaction semantics to absent property nodes. In particular, absent property nodes are represented by special null nodes in a 'tree tuple'. For illustration, Table 2.11 lists the 'tree tuples' in XML trees $\mathcal{T}_3$ and $\mathcal{T}_4$, where absent `ano` nodes are denoted by symbol $\perp$. The value of a $\perp$ node is thereby assumed to be different to the value of any other node, which is indicated by the distinct subscripts of $\perp$ nodes in Table 2.11. Consider now benchmark constraint $\mathcal{K}_2$ depicted in Table 2.10. Given that $\perp$ nodes have distinct values, neither $\langle v_4, v_5, v_6, \perp_1, v_3\rangle$ and $\langle v_{10}, v_{11}, v_{12}, v_{13}, v_9\rangle$, nor $\langle v_4, v_5, v_6, \perp_1, v_3\rangle$ and $\langle v_{10}, v_{11}, v_{12}, \perp_2, v_9\rangle$ have value equal LHS nodes. As a consequence, $\mathcal{K}_2$ is satisfied in both XML trees $\mathcal{T}_3$ and $\mathcal{T}_4$ according to the semantics of 'tree tuple' functional dependencies. Since this is not desired for XML tree $\mathcal{T}_4$, the weak satisfaction semantics applied by 'tree tuple' functional dependencies is not adequate for handling absent property nodes.

We conclude the discussion of the 'tree tuple' approach with an overview on related theoretical results. Arenas & Libkin presented a normal form for XML documents related to 'tree tuple' functional dependencies, and developed a corresponding normalization algorithm for DTDs [23]. Arenas & Libkin also presented an information-theoretic approach that can be used for the justification of XML normalforms in general [83].

The implication problem related to 'tree tuple' functional dependencies has been studied by Kot & White [19]. Their main finding is that the implication of 'tree tuple' functional dependencies is in general undecidable because of the interaction with structural constraints

`Company`	`Company.Customer`	`Company.Customer.Addr`	`Company.Customer.Addr.id`	`Company.Customer.Addr.city`	`Company.Customer.Addr.strt`	`Company.Customer.Addr.hno`	`Company.Customer.Addr.ano`	
$\langle v_1$	v_2	v_3	v_7	v_4	v_5	v_6	$\bot_1\rangle$	*XML Tree* T_3
			`A1`	`NY`	`5th Av.`	`50`		
$\langle v_1$	v_8	v_9	v_{14}	v_{10}	v_{11}	v_{12}	$v_{13}\rangle$	
			`A2`	`NY`	`Park Av.`	`30`	`2B`	
$\langle v_1$	v_2	v_3	v_7	v_4	v_5	v_6	$\bot_1\rangle$	*XML Tree* T_4
			`A1`	`NY`	`5th Av.`	`50`		
$\langle v_1$	v_8	v_9	v_{13}	v_{10}	v_{11}	v_{12}	$\bot_2\rangle$	
			`A2`	`NY`	`5th Av.`	`50`		

Table 2.11: Tree tuples in benchmark XML Trees T_3 and T_4.

imposed by the DTD which is assumed to be present. The implication problem becomes tractable for the class of disjunction free DTDs, and a set of sound and complete inference rules for the implication of 'tree tuple' functional dependencies has been presented [19].

2.4.3 The Closest Node Approach

The central mechanism for building tuples of nodes in the 'closest node' approach is the *closest* property of nodes, which has been originally presented by Vincent et al. in defining an XML functional dependency [25]. A pair of nodes satisfy the *closest* property if the nodes have a common ancestor-or-self node which is reachable over the intersection of the paths leading to the nodes. A node v is thereby an ancestor-or-self node of a node v', if either v is an ancestor node of v' or v and v' are the same node.

Vincent et al. proposed two versions of the 'closest node' approach, which they call *weak* 'closest node' approach and *strong* 'closest node' approach. Even though both version of the 'closest node' approach are based on the *closest* property of nodes, only the strong 'closest node' approach adequately handles multiple property nodes, as will be made more clear soon. Because of this significant difference we will discuss the weak 'closest node' approach separately from the strong 'closest node' approach.

As for the 'tree tuple' approach, element nodes are compared by their identities also in the 'closest node' approach. Hence, a 'closest node' functional dependency has also some flavor of an XML key in case that the functional dependency has a single RHS path that leads to element nodes. The uniqueness of property nodes is not ensured by a 'closest node' functional dependency for reasons similar to which a 'tree tuple' functional dependency also does not have this property. Since we detailed on this issue already in Section 2.4.2, we do not address it again in the evaluation of 'closest node' functional dependencies.

Weak Closest Node Approach

Vincent et al. proposed the weak 'closest node' approach when defining a *unary* XML functional dependency [25]. In this approach a set of nodes $v_1, \ldots, v_n, v_{n+1}$ forms a tuple for the set of paths $\{P_1, \ldots, P_n, P_{n+1}\}$ in an XML functional dependency $\{P_1, \ldots, P_n\} \rightarrow P_{n+1}$, if for all $i \in \{1, \ldots, n\}$, the LHS node v_i and the RHS node v_{n+1} satisfy the *closest* property. In this setting Vincent et al. also defined an XML multivalued dependency [84].

We now evaluate the ability of the weak 'closest node' approach to handle the challenges which we have identified in Section 1.2. We use benchmark constraints $\mathcal{K}_1$ and $\mathcal{K}_2$ for this purpose. In terms of weak 'closest node' functional dependencies benchmark constraints $\mathcal{K}_1$ and $\mathcal{K}_2$ are of the same form as the corresponding 'tree tuple' functional dependencies which are given in Table 2.10.

Handling of multiple property nodes: Because of the rather loose structural cohesion within tuples of nodes in the weak 'closest node' approach, semantically incorrect tuples of nodes are potentially formed. Table 2.12 lists the tuples of nodes which are formed during the validation of XML tree $\mathcal{T}_1$ against $\mathcal{K}_1$. Note that in each of these tuples, the pairs of an LHS node and the RHS node satisfy the *closest* property. For instance, `Invoice` node v_3 is the RHS node in the tuple of nodes $\langle v_{14}, v_6, v_7, v_3 \rangle$. For each one of the pairs (v_3, v_{14}), (v_3, v_6), and (v_3, v_7), the intersection path is `Company.Invoices.Invoice`. Because node v_3 is reachable over path `Company.Invoices.Invoice`, and v_3 is an ancestor-or-self node of v_3 as well as an ancestor-or-self node of each one of the LHS nodes $\{v_{14}, v_6, v_7\}$, the *closest* property is satisfied by each one of the pairs (v_3, v_{14}), (v_3, v_6), and (v_3, v_7).

Company.Invoices.Invoice.prd	Company.Invoices.Invoice.Line.code	Company.Invoices.Invoice.Line.no	Company.Invoices.Invoice
$\langle v_{14}$	v_6	v_7	$v_3 \rangle$
01/09	*0660*	*1010*	
$\langle v_{14}$	v_6	v_{11}	$v_3 \rangle$
01/09	*0660*	*2020*	
$\langle v_{14}$	v_{10}	v_{11}	$v_3 \rangle$
01/09	*0990*	*2020*	
$\langle v_{14}$	v_{10}	v_7	$v_3 \rangle$
01/09	*0990*	*1010*	
$\langle v_{22}$	v_{18}	v_{19}	$v_{15} \rangle$
01/09	*0600*	*2020*	

Table 2.12: Tuples of nodes in benchmark XML Tree $\mathcal{T}_1$ formed for benchmark constraint $\mathcal{K}_1$ according to the weak 'closest node' approach.

The tuples of nodes $\langle v_{14}, v_6, v_{11}, v_3\rangle$ and $\langle v_{14}, v_{10}, v_7, v_3\rangle$ are semantically incorrect, since neither the combination (v_6, v_{11}) nor the combination (v_{10}, v_7) of a `code` and a `no` node belong to the same invoice line. As a consequence, $\mathcal{K}_1$ is violated in $\mathcal{T}_1$ since the two tuples of nodes $\langle v_{14}, v_6, v_{11}, v_3\rangle$ and $\langle v_{22}, v_{18}, v_{19}, v_{15}\rangle$ have value equal LHS nodes whereas the RHS element nodes v_3 and v_{15} have distinct 'values', i.e. identities.

Handling of absent property nodes: Weak 'closest node' functional dependencies apply a kind of strong satisfaction semantics to absent property nodes. As for the 'tree tuple' approach, absent property nodes are represented by special null nodes ($\perp$). The essential difference is that a $\perp$ node is value equal to every other node in the weak 'closest node' approach, whereas $\perp$ nodes have distinct values in the 'tree tuple' approach. For the purpose of illustration, Table 2.13 lists the tuples of nodes formed during the validation of XML trees $\mathcal{T}_3$ and $\mathcal{T}_4$ against benchmark constraint $\mathcal{K}_2$. Given that a $\perp$ node is value equal to every other node, the tuples of nodes $\langle v_4, v_5, v_6, \perp, v_3\rangle$ and $\langle v_{10}, v_{11}, v_{12}, \perp, v_9\rangle$ have value equal LHS nodes. Since RHS nodes v_3 and v_9 are distinct nodes, and are therefore not value equal according to the semantics in the weak 'closest node' approach, $\mathcal{K}_2$ is violated in $\mathcal{T}_4$, which is desired. Further, $\mathcal{K}_2$ is satisfied in $\mathcal{T}_3$ since tuples $\langle v_4, v_5, v_6, \perp, v_3\rangle$ and $\langle v_{10}, v_{11}, v_{12}, v_{13}, v_9\rangle$ do not have value equal LHS nodes, irrespective of the $\perp$ node. Hence, with respect to XML functional dependencies, the strong satisfaction semantics applied in the weak 'closest node' approach is an adequate manner for dealing with absent property nodes.

`Company.Customer.Addr.city`	`Company.Customer.Addr.strt`	`Company.Customer.Addr.hno`	`Company.Customer.Addr.ano`	`Company.Customer.Addr`	
$\langle v_4$	v_5	v_6	$\perp$	$v_3\rangle$	*XML Tree* $\mathcal{T}_3$
NY	*5th Av.*	*50*			
$\langle v_{10}$	v_{11}	v_{12}	v_{13}	$v_9\rangle$	
NY	*Park Av.*	*30*	*2B*		
$\langle v_4$	v_5	v_6	$\perp$	$v_3\rangle$	*XML Tree* $\mathcal{T}_4$
NY	*5th Av.*	*50*			
$\langle v_{10}$	v_{11}	v_{12}	$\perp$	$v_9\rangle$	
NY	*5th Av.*	*50*			

Table 2.13: Tuples of nodes in benchmark XML Trees $\mathcal{T}_3$ and $\mathcal{T}_4$ formed for benchmark constraint $\mathcal{K}_2$ according to the weak 'closest node' approach.

We conclude the discussion of weak 'closest node' constraints with an overview on related theoretical results. Weak 'closest node' functional dependencies have been thoroughly studied. A validation algorithm has been developed which requires linear time in the size of the XML document and the number of functional dependencies [85]. A set of sound inference rules has been presented in [25], which then have been shown to be also complete for unary weak 'closest node' functional dependencies, and a linear time decision algorithm for the related implication problem has been developed on basis of these inference rules [86]. Vincent et al. further developed a normal form with respect to weak 'closest node' functional dependencies, and established the result that this normal form is a necessary and sufficient condition for the elimination of redundancy in XML data [87, 88].

Vincent et al. presented a check algorithm for weak 'closest node' multivalued dependencies which requires linear time in the size of the XML document [89]. Vincent et al. also developed a normal form with respect to weak 'closest node' multivalued dependencies [90, 91].

Strong Closest Node Approach

In order to overcome the limitations of the weak 'closest node' approach with respect to the manner in which multiple property nodes are handled, Vincent et al. proposed the strong 'closest node' approach when defining an XML functional dependency [20]. In the strong 'closest node' approach a set of nodes $\{v_1, \ldots, v_n\}$ forms a tuple for a set of paths $\{P_1, \ldots, P_n\}$ if the nodes pairwise satisfy the *closest* property. Compared to the weak 'closest node' approach, the structural cohesion between the nodes in a tuple is substantially tighter and so semantically incorrect tuples are avoided in the validation of an XML document against a strong 'closest node' functional dependency.

Liu et al. adopted the strong 'closest node' approach when defining a scoped XML functional dependency [1]. Fassetti & Fazzinga also adopted the strong 'closest node' approach in order to define an approximate XML functional dependency, which is an XML functional dependency that is not necessarily strictly satisfied in an XML document [93]. An approximate XML functional dependency is intended for the discovery of erroneous or exceptional element nodes, which has been claimed to be useful for data cleaning and data integration [93]. In the approach by Fassetti & Fazzinga, two element nodes are approximately value equal if the tree edit distance (TED) for the subtrees rooted at these element nodes is less than a fixed threshold. The TED is thereby the minimal number of nodes and edges that need to be added or deleted in two trees such that the trees became isomorphic. Based on the notion of approximate value equality between element nodes, Fassetti & Fazzinga defined an XML functional dependency to be satisfied in an XML document if whenever the LHS nodes in a pair of tuples are approximately value equal, then also the RHS nodes in these tuples must be approximately value equal.

We now evaluate the ability of the strong 'closest node' approach to handle the challenges identified in Section 1.2. The XML functional dependency presented by Vincent et al. in [20] is representative for the proposals by Liu et al. [92] and Fassetti & Fazzinga [93] regarding the manner in which multiple property nodes are handled. We therefore restrict our subsequent discussion to the original proposal by Vincent et al [20].

Handling of multiple property nodes: In comparison to the weak 'closest node' approach, the requirement on a set of nodes to form a tuple is strictly stronger in the strong 'closest node' approach. To see this, consider the tuples of nodes listed in Table 2.12 which are formed according to the weak 'closest node' approach for the validation of XML tree $\mathcal{T}_1$ against benchmark constraint $\mathcal{K}_1$ (cf. Table 2.10). The semantically incorrect tuples of nodes $\langle v_{14}, v_6, v_{11}, v_3 \rangle$ and $\langle v_{14}, v_{10}, v_7, v_3 \rangle$ are not formed according to the strong 'closest node'

[1]In [92] two nodes that satisfy the *closest* property are said to be *fellows*.

approach. The reason for this is that not *every* pair of nodes in either one of these tuples satisfy the *closest* property. In particular, the pairs (v_6, v_{11}) and (v_{10}, v_7) of a `code` node and a `no` node do not satisfy the *closest* property since the nodes in both of these pairs do not have a common ancestor node at the intersection path `Company.Invoices.Invoice.Line`. The remaining tuples of nodes in Table 2.12 are semantically correct, and in fact every pair of nodes in each of these tuples satisfy the *closest* property. Given that only the semantically correct tuples of nodes $\langle v_{14}, v_6, v_7, v_3\rangle$, $\langle v_{14}, v_{10}, v_{11}, v_3\rangle$ and $\langle v_{22}, v_{18}, v_{19}, v_{15}\rangle$ are formed in the validation of XML tree $\mathcal{T}_1$ against $\mathcal{K}_1$ according to the strong 'closest node' approach, $\mathcal{K}_1$ is satisfied in XML tree $\mathcal{T}_1$ as desired. In contrast to the weak 'closest node' approach, multiple property nodes are thus adequately handled in the strong 'closest node' approach.

Handling of absent property nodes: Strong 'closest node' functional dependencies apply exactly the same kind of strong satisfaction semantics to absent property nodes as weak 'closest node' functional dependencies. Hence, in accordance with the weak 'closest node' approach, benchmark constraint $\mathcal{K}_2$ is satisfied in XML tree $\mathcal{T}_3$ and violated in XML tree $\mathcal{T}_4$ according to the semantics of a strong 'closest node' functional dependency.

We conclude the discussion of strong 'closest node' constraints with an overview on related theoretical results. Vincent et al. established the result that the semantics of strong 'closest node' functional dependencies coincides with the semantics of 'tree tuple' functional dependencies in the class of XML trees without missing information. That is, whereas absent property nodes are handled in different manners, strong 'closest node' functional dependencies and 'tree tuple' functional dependencies handle multiple property nodes in essentially the same sophisticated manner [20].

Vincent et al. moreover established the result that a strong 'closest node' functional dependency precisely preserves the semantics of a relational functional dependency in case that XML data is generated from relational data by applying an arbitrary sequence of nesting operations prior to the mapping [20].

Fassetti & Fazzinga presented an algorithm for inferring approximate 'closest node' XML functional dependencies from XML data and showed experimental results asserting the effectiveness of this algorithm [94].

2.4.4 The Hedge-based Approach

Shariar & Liu proposed scoped versions of an XML key [95], an XML functional dependency [22] and an XML inclusion dependency [96] in the setting of a 'tuple based' approach to XML integrity constraints where the structural information in a DTD is exploited in order to form tuples of nodes. Shariar & Liu proposed for this purpose the notion of a *minimal structure* for a pair of types[2] in a DTD, and also the notion of a *hedge* for a pair of nodes in an XML document. Roughly speaking, the minimal structure for a pair of types $\boldsymbol{\tau}$ and $\boldsymbol{\tau}'$ in a DTD is the smallest snippet of the DTD that contains $\boldsymbol{\tau}$ and $\boldsymbol{\tau}'$. Figure 2.6a depicts the minimal structure for element types `B` and `C` in the fairly simple DTD $\mathcal{D}_2$ for the purpose of illustration. Again roughly speaking, the hedge for a pair of sibling nodes v and v' is the sequence of child nodes of the common parent node of v and v' that ranges from v to v'. Consider for example XML tree $\mathcal{T}_7$ depicted in Figure 2.6b which conforms to DTD $\mathcal{D}_2$.

[2]A type is thereby either an element type, a declared attribute or the predefined text type.

The hedge for sibling nodes v_3 and v_5 is the sequence of nodes (v_3, v_4, v_5), which are child nodes of the common parent node v_1. Likewise, the sequence of nodes (v_4, v_5) is the hedge for nodes v_4 and v_5.

(a)

```
<!DOCTYPE [
  <!ELEMENT A ((B,C)+)>
  <!ELEMENT B (#PCDATA)>  } Minimal structure
  <!ELEMENT C (#PCDATA)>  } for B and C
]>
```

(b)

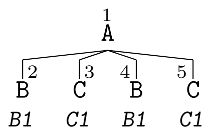


Figure 2.6: Minimal structure for a pair of element types in DTD $\mathcal{D}_2$, and XML tree $\mathcal{T}_7$ which conforms to DTD $\mathcal{D}_2$.

Based on the notions of a minimal structure for a pair of types in a DTD and the hedge for a pair of nodes, tuples are formed in the approach by Shariar & Liu as follows. A set of nodes $v_1, \ldots, v_n$ forms a tuple for a set of paths $P_1, \ldots, P_n$ if it holds true for every pair of nodes $(\bar{v}, \bar{v}')$, where $\bar{v}$ and $\bar{v}'$ are either nodes in $\{v_1, \ldots, v_n\}$ or ancestors of any nodes in $\{v_1, \ldots, v_n\}$, that (a) $\bar{v}$ and $\bar{v}'$ are the same node if $\bar{v}$ and $\bar{v}'$ are reachable over the same path; (b) if $\bar{v}$ and $\bar{v}'$ have a common parent node, then the hedge for $\bar{v}$ and $\bar{v}'$ conforms to the minimal structure for $\boldsymbol{\tau}$ and $\boldsymbol{\tau}'$, where $\boldsymbol{\tau}$ and $\boldsymbol{\tau}'$ are the types for nodes $\bar{v}$ and $\bar{v}'$.

There is a tight relationship between condition (a) and the *closest* property of nodes presented by Vincent et al. in [20]. Recall that a pair of nodes v and v' satisfy the *closest* property if and only if v and v' have a common ancestor at the intersection path. Now, if v and v' satisfy the *closest* property then this implies that any pair of ancestor nodes $\bar{v}$ of v and $\bar{v}'$ of v' are the same nodes if and only if $\bar{v}$ and $\bar{v}'$ are reachable over the same path. Hence, a pair of nodes satisfy the *closest* property if and only if these nodes satisfy condition (a). The effect of condition (a) on the nodes in a tuple is therefore equivalent with the effect of requiring the nodes in a tuple to pairwise satisfy the *closest* property.

To see the impact of condition (b) on the nodes in a tuple, and thus to see the difference between the manners in which tuples are built in the strong 'closest node' approach and in the hedge-based approach, consider again XML tree $\mathcal{T}_7$ and DTD $\mathcal{D}_2$ depicted in Figure 2.6. Suppose that tuples of nodes are to be formed for the paths `A.B` and `A.C`. Then, the tuples of nodes $\langle v_2, v_3 \rangle$, $\langle v_2, v_5 \rangle$, $\langle v_4, v_5 \rangle$, and $\langle v_4, v_3 \rangle$ are formed according to the strong 'closest node' approach, since every pair of a `B` node and a `C` node satisfies the *closest* property. Note that each of these pairs of nodes also satisfy condition (a). However, only tuples $\langle v_2, v_3 \rangle$ and $\langle v_4, v_5 \rangle$ also satisfy condition (b). The reason for this is that the minimal structure for types `B` and `C`, which is depicted in Figure 2.6a, requires conforming hedges to contain exactly one `B` node which is followed by exactly one `C` node. This is clearly not satisfied by the hedge (v_2, v_3, v_4, v_5) for the pair of nodes in tuple $\langle v_2, v_5 \rangle$. The hedge (v_3, v_4) for the pair of nodes in tuple $\langle v_4, v_3 \rangle$ does not conform to the minimal structure for types `B` and `C` because the `C` node v_3 precedes the `B` node v_4 in XML tree $\mathcal{T}_7$. That is, compared to the strong closest-node approach, the structural cohesion of nodes in a tuple is even stronger in the hedge-based approach. The reason for this is that condition (b) incorporates structural information in a DTD into the process of forming tuples of nodes.

A salient difference between the 'closest node' approach and the hedge-based approach is also that the value of an element node is the subtree rooted at this node in the hedge-based approach. Two element nodes are thereby value equal in the hedge-based approach if the subtrees rooted at these element nodes are isomorphic. As a consequence, a hedge-based functional dependency does not have the flavor of an XML key in case that the functional dependency has a single RHS path that leads to element nodes. Note that two element nodes need not necessarily be the same if the subtrees rooted at these nodes are isomorphic. Shariar & Liu however proposed a dedicated constraint for asserting the semantics of an XML key [95]. The general form of a hedge-based key is $(P_1, \ldots, P_n)$. Such an XML key asserts that if there is a pair of value equal tuples of nodes for the paths $P_1, \ldots, P_n$, then these tuples are made up of the same nodes. Table 2.14 gives benchmark constraints $\mathcal{K}_1$ and $\mathcal{K}_2$ in the form of hedge-based XML keys.

$\mathcal{K}_1$	(`Company.Invoices.Invoice.Line.code`, `Company.Invoices.Invoice.Line.no`, `Company.Invoices.Invoice.prd`)
$\mathcal{K}_2$	(`Company.Customer.Addr.city`, `Company.Customer.Addr.strt`, `Company.Customer.Addr.hno`, `Company.Customer.Addr.ano`)

Table 2.14: Benchmark constraints $\mathcal{K}_1$ and $\mathcal{K}_2$ specified as hedge-based keys.

The other salient difference between the 'closest node' approach and the hedge-based approach belongs to the manner in which absent property nodes are handled. Whereas the 'closest node' approach applies a kind of strong satisfaction semantics to absent property nodes, incomplete tuples of nodes are simply ignored in the hedge-based approach.

We now evaluate the ability of the hedge-based approach to handle the challenges identified in Section 1.2. We use benchmark constraints $\mathcal{I}_1$ and $\mathcal{I}_2$ for the evaluation of hedge-based inclusion dependencies. These benchmark constraints have already been illustrated in the common form of 'tuple based' XML inclusion dependencies within the discussion of the 'intersection path' approach (cf. Table 2.6 in Section 2.4.1). We use benchmark constraints $\mathcal{K}_1$ and $\mathcal{K}_2$ for the evaluation of hedge-based keys.

Handling of multiple property nodes: Table 2.15 lists the tuples of nodes formed for benchmark XML key $\mathcal{K}_1$ in XML tree $\mathcal{T}_1$ according to the hedge-based approach. Semantically incorrect combinations of property nodes are avoided. This is expected given that the nodes in each one of the tuples satisfy condition (a) and that therefore the nodes in each tuple pairwise satisfy the *closest* property.

We now illustrate that the tuples in Table 2.15 also satisfy condition (b) using the example of tuple $\langle v_6, v_7, v_{14}\rangle$. In doing this we refer to the types in DTD $\mathcal{D}_3$ depicted in Figure 2.7 which specifies the structure of XML tree $\mathcal{T}_1$. Consider first the pair of nodes (v_6, v_7). The ancestor nodes of both v_6 and v_7 are $\{v_5, v_3, v_2, v_1\}$, and so condition (b) must hold true for every pair of nodes in $\{v_6, v_5, v_3, v_2, v_1\} \times \{v_7, v_5, v_3, v_2, v_1\}$ that have a common parent node, i.e. (v_6, v_7), (v_5, v_5), (v_3, v_3), and (v_2, v_2). Note that condition (b)

Company.Invoices.Invoice.Line.code	Company.Invoices.Invoice.Line.no	Company.Invoices.Invoice.prd	
$\langle v_6$	v_7	$v_{14}\rangle$	
0660	*1010*	*01/09*	
$\langle v_{10}$	v_{11}	$v_{14}\rangle$	XML tree $\mathcal{T}_1$
0990	*2020*	*01/09*	
$\langle v_{18}$	v_{19}	$v_{22}\rangle$	
0660	*2020*	*01/09*	
$\langle v_5$	v_6	$v_{13}\rangle$	
0990	*2020*	*01/09*	XML tree $\mathcal{T}_2$
$\langle v_9$	v_{10}	$v_{13}\rangle$	
0990	*2020*	*01/09*	

Table 2.15: Tuples of nodes in benchmark XML Trees $\mathcal{T}_1$ and $\mathcal{T}_2$ formed for benchmark constraint $\mathcal{K}_1$ according to the hedge-based approach.

trivially holds true for a pair of nodes if it is made up of the same node. With respect to the remaining pair of nodes (v_6, v_7), condition (b) requires that the hedge for the `code` node v_6 and the `no` node v_7 conforms to the minimal structure for the types `code` and `no` in DTD $\mathcal{D}_3$. For reasons of brevity, we will subsequently use the fairly intuitive abstract syntax proposed by Shariar & Liu in [95] for formulating the minimal structure for a pair of types in a DTD. In this abstract syntax, the minimal structure for types `code` and `no` in DTD $\mathcal{D}_3$ is given by [`code`, `no`], which means that conforming hedges must be made up of exactly one `code` node followed by exactly one `no` node. This is clearly satisfied by the hedge (v_6, v_7) for the pair of nodes v_6 and v_7, and hence condition (b) holds true for this pair of nodes.

Consider now the pair of nodes (v_6, v_{14}) in tuple $\langle v_6, v_7, v_{14}\rangle$. The ancestor nodes of v_6 are $\{v_5, v_3, v_2, v_1\}$ and the ancestor nodes of v_{14} are $\{v_3, v_2, v_1\}$. Consequently, condition (b) takes effect for the pairs of nodes in $\{v_6, v_5, v_3, v_2, v_1\} \times \{v_{14}, v_3, v_2, v_1\}$ that have a common parent node, i.e. (v_5, v_{14}), (v_3, v_3), and (v_2, v_2). Again, condition (b) trivially holds true for the pairs of nodes (v_3, v_3) and (v_2, v_2). The hedge for the `Line` node v_5 and the `prd` node v_{14} in the remaining pair of nodes is (v_5, v_9, v_{14}), where v_9 is also a `Line` node. The minimal structure for the types `Line` and `prd` in DTD $\mathcal{D}_1$ is [$\texttt{Line}^+$, `prd`], which requires conforming hedges to be made up of a non-empty list of `Line` nodes followed by exactly one `prd` node. Given that in the hedge (v_5, v_9, v_{14}) nodes v_5 and v_9 are `Line` nodes followed by the `prd` node v_{14}, condition (b) holds true for this hedge and therefore also for the pair of nodes (v_6, v_{14}). For similar reasons condition (b) is also satisfied by the remaining pair of nodes v_7, v_{14} in tuple $\langle v_6, v_7, v_{14}\rangle$.

```
<!DOCTYPE COMPANY[
  <!ELEMENT Company (Invoices,Phones)>
  <!ELEMENT Invoices (Invoice+)>
  <!ELEMENT Invoice (Line+)>
  <!ELEMENT Line EMPTY>
  <!ELEMENT Phones (Phone+)>
  <!ELEMENT Phone EMPTY>
  <!ATTLIST Invoice prd CDATA #REQUIRED>
  <!ATTLIST Invoice cno CDATA #REQUIRED>
  <!ATTLIST Invoice id CDATA #REQUIRED>
  <!ATTLIST Line no CDATA #REQUIRED>
  <!ATTLIST Line code CDATA #REQUIRED>
  <!ATTLIST Line amt CDATA #REQUIRED>
  <!ATTLIST Phone no CDATA #REQUIRED>
  <!ATTLIST Phone code CDATA #REQUIRED>
  <!ATTLIST Phone amt CDATA #REQUIRED>
]>
```

Figure 2.7: DTD $\mathcal{D}_3$ specifying the structure of benchmark XML trees $\mathcal{T}_1$ and $\mathcal{T}_2$.

Because no two distinct tuples of nodes formed for benchmark XML key $\mathcal{K}_1$ in XML tree $\mathcal{T}_1$ are value equal (cf. Table 2.15), $\mathcal{K}_1$ is satisfied in XML tree $\mathcal{T}_1$ as desired according to the semantics of hedge-based keys.

As desired, benchmark XML inclusion dependency $\mathcal{I}_1$ is also satisfied in XML tree $\mathcal{T}_1$ according to the hedge-based approach. Table 2.16 lists the tuples of nodes formed for XML inclusion dependency $\mathcal{I}_1$ in XML tree $\mathcal{T}_1$ according to the hedge-based approach. Again, semantically incorrect tuples are not formed and so there is for each LHS tuple of nodes a corresponding RHS tuple of nodes such that the nodes in these tuples are value equal. Multiple property nodes are therefore adequately handled in the hedge-based approach.

Ensuring uniqueness of property nodes: XML keys in the hedge-based approach indeed ensure uniqueness of property nodes since no two distinct tuples of nodes are allowed to be value equal. Tuples $\langle v_5, v_6, v_{13}\rangle$ and $\langle v_9, v_{10}, v_{13}\rangle$ are formed in XML tree $\mathcal{T}_2$ for benchmark XML key $\mathcal{K}_2$ which is depicted in Table 2.14 in form of a hedge-based key. Because these two tuples are distinct but value equal, XML key $\mathcal{K}_2$ is violated in XML tree $\mathcal{T}_2$ according to the semantics of hedge-based XML keys.

Handling of absent property nodes: In the hedge-based approach incomplete tuples of nodes are simply ignored when checking the satisfaction of a constraints. Hence, benchmark XML key $\mathcal{K}_2$ is satisfied in both XML trees $\mathcal{T}_3$ and $\mathcal{T}_4$ as well as benchmark XML inclusion dependency $\mathcal{I}_2$ is satisfied in both XML trees $\mathcal{T}_5$ and $\mathcal{T}_6$. Since this result is not desired for XML trees $\mathcal{T}_4$ and $\mathcal{T}_6$, hedge-based constraints do not adequately handle absent property nodes.

Company.Invoices.Invoice.cno	Company.Invoices.Invoice.Line.code	Company.Invoices.Invoice.Line.no	Company.Phones.Phone.cno	Company.Phones.Phone.code	Company.Phones.Phone.no
$\langle v_{13}$	v_6	$v_7\rangle$	$\langle v_{27}$	v_{25}	$v_{26}\rangle$
C1	*0660*	*1010*	*C1*	*0660*	*1010*
$\langle v_{13}$	v_{10}	$v_{11}\rangle$	$\langle v_{31}$	v_{29}	$v_{30}\rangle$
C1	*0990*	*2020*	*C1*	*0990*	*2020*
$\langle v_{21}$	v_{18}	$v_{19}\rangle$	$\langle v_{35}$	v_{33}	$v_{34}\rangle$
C2	*0660*	*2020*	*C2*	*0660*	*2020*

Table 2.16: Tuples of nodes in benchmark XML Tree $\mathcal{T}_1$ formed for benchmark constraint $\mathcal{I}_1$ according to the hedge-based approach.

We conclude the discussion of hedge-based constraints with an overview on related theoretical results. Shariar & Liu showed that the semantics of hedge-based functional dependencies and keys can be preserved when transforming a DTD with basic transformation operations commonly found in approaches towards the transformation of DTDs [97, 98]. Shariar & Liu also presented check algorithms based on DOM for hedge-based keys and functional dependencies. These algorithms require linear time in the number of nodes in an XML document and the number of paths in the constraints to be checked [99, 100].

2.5 Subtree-based XML Integrity Constraints

Hartmann et al. originally proposed an XML functional dependency which compares combinations of property nodes by means of comparing entire subtrees that contain these combinations of property nodes [101]. For the purpose of specifying the subtrees that are to be compared in checking an XML functional dependency, the existence of a scheme tree, i.e. an a priori schema for the XML tree under consideration, is assumed to be present. Figures 2.8a and 2.8b depict the scheme trees for XML tree $\mathcal{T}_1$ as well as for XML trees $\mathcal{T}_3$ and $\mathcal{T}_4$, respectively. Symbols $*$, 1 and ? specify the cardinalities of nodes. In particular, 1 requires a node to occur at must once, ? requires a node to occur at most once, and $*$ specifies that a node may occur multiple times or not at all.

Based on the notion of a scheme tree, the general form of a subtree-based XML functional dependency is $X : Y \rightarrow Z$, where X is a subtree in a scheme tree, and Y and Z are rooted subtrees of X. The meaning of such an XML functional dependency is then roughly speaking that whenever there exist two XML subtrees T_X and T'_X that conform to X and contain at most one node for each path indicated by X, then the projections $\mathrm{T}_{X|Z}$ and $\mathrm{T}'_{X|Z}$ of T_X and T'_X on Z must be isomorphic if the projections $\mathrm{T}_{Y|Z}$ and $\mathrm{T}'_{Y|Z}$ of T_X and T'_X on Y are isomorphic and contain at least one node for each path indicated by Y. In particular, two projections of an XML tree are isomorphic in the subtree-based approach if they have the same structure and their leaf nodes are value equal.

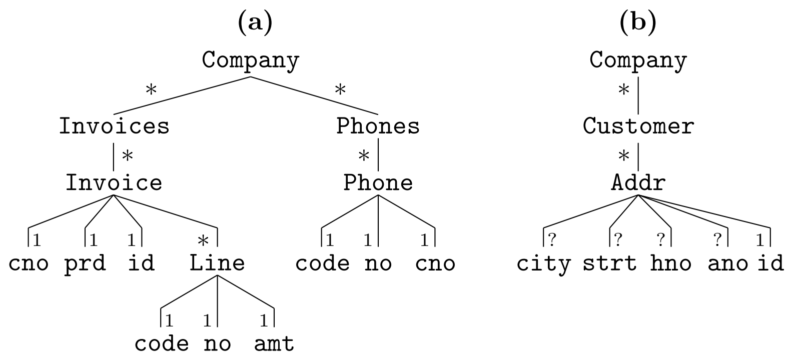


Figure 2.8: Scheme trees specifying the structure of XML trees $\mathcal{T}_1$, $\mathcal{T}_3$ and $\mathcal{T}_4$.

Hartmann et al. also extended their XML functional dependency with some flavor of set semantics [21]. Such an XML function dependency allows to express for example that the *set* of phones determines the customer number as opposed to the situation where each phone of a customer determines the customer number on its own. Lv & Yan proposed to extend subtree-based XML functional dependencies with a feature that allows application developers to individually specify the kind of agreement between subtrees on which an XML functional dependency takes effect [102].

We now evaluate the subtree-based approach on basis of the original proposal to XML functional dependencies by Hartmann et al. [101]. We use for this purpose benchmark XML functional dependencies $\mathcal{F}_1$ and $\mathcal{F}_2$ which are depicted in Figure 2.9 in form of subtree-based XML functional dependencies.

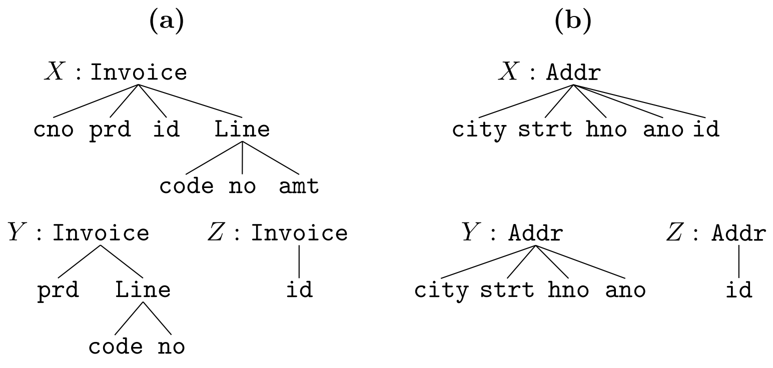


Figure 2.9: Benchmark constraints $\mathcal{F}_1$ and $\mathcal{F}_2$ in form of subtree-based XML functional dependencies.

Handling of multiple property nodes: When checking XML functional dependency $\mathcal{F}_1$ in XML tree $\mathcal{T}_1$, the three T_X-subtrees depicted in Figure 2.10 are found. Note that these subtrees conform to the X scheme tree specified by $\mathcal{F}_1$, which is the `Invoice` subtree of the scheme tree depicted in Figure 2.8. Also, each T_X-subtree contains at most one node reachable over any path indicated by X. The T_X-subtrees obviously do not agree on any projection on Y, i.e. the combinations of a `prd` node, a `code` node, and a `no` node have different values in each of the $T_{X|Y}$ projections, and so $\mathcal{F}_1$ is satisfied according to the semantics of subtree-based XML functional dependencies as desired. In general, subtree-based XML functional dependencies adequately deal with multiple property nodes, since structural relationships between property nodes are kept when comparing subtrees.

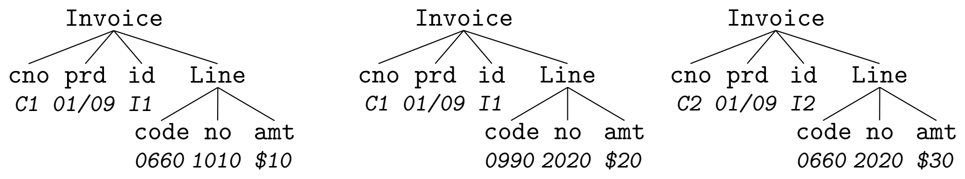


Figure 2.10: Subtrees in checking the satisfaction of benchmark constraint $\mathcal{F}_1$ in XML tree $\mathcal{T}_1$ in the subtree-based approach.

Handling of absent property nodes: Whereas subtree-based XML functional dependencies adequately handle multiple property nodes, they disregard incomplete combinations of property nodes and therefore do not adequately handle absent property nodes. To see this consider the validation of XML tree $\mathcal{T}_4$ against XML functional dependency $\mathcal{F}_2$. The T_X-subtrees are the subtrees of XML tree $\mathcal{T}_4$ which are rooted at the `Addr` nodes. Since the $T_{X|Y}$ projections in these subtrees are both missing an `ano` node, they do not satisfy the requirement to have one node for each path indicated by the Y scheme tree in $\mathcal{F}_2$. Hence, the Z subtrees, i.e. the `id` nodes, must not agree according to the semantics of subtree-based XML functional dependencies. As a consequence, $\mathcal{F}_1$ is satisfied in XML tree $\mathcal{T}_1$ according to the semantics of subtree-based XML functional dependencies, which is clearly not desired.

We conclude the discussion of subtree-based XML functional dependencies with an overview on related theoretical results. Hartmann et al. presented a sound and complete set of inference rules for subtree-based XML functional dependencies in [103]. Interestingly, the well known transitivity rule for relational functional dependencies is not sound for subtree-based XML functional dependencies. Lv & Yan developed a normalform based on their extended version of subtree-based XML functional dependencies and showed this normalform to be a necessary and sufficient condition for avoiding redundancies in XML data [104].

2.6 Formula-based XML Integrity Constraints

The commonality of what we call formula-based XML integrity constraints is that a constraint is specified using variables to which nodes and values in an XML document are bound in order to validate the XML document against an XML integrity constraint.

2.6.1 XML Template Functional Dependencies

Inspired by traditional template dependencies for the relational data model, Mok recently presented an XML template functional dependency [105]. Likewise to its relational antetype, an XML template functional dependency consists of a hypothesis and a conclusion. The hypothesis is made up of a set of lists of variables of the form $x_1. \cdots .x_n$. A variable x may have associated an element type $\boldsymbol{\tau}$ or attribute $\boldsymbol{\alpha}$ defined in a DTD, which is assumed to be present, in which case x is called an *element variable* or *attribute variable*, respectively. If a variable x has neither associated an element type nor an attribute, then x is a *text variable*. Attribute variables and text variables must not occur in any other position than the last in any list of variables in the hypothesis. The conclusion in an XML template functional dependency is made up of a single equality comparison between two variables of the hypothesis.

The variables in an XML template functional dependency are bound to nodes and values in an XML document, where element variables are bound to element nodes, as opposed to attribute variables and text variables, which are bound to the values of attribute nodes and text nodes, respectively. The binding of a list of variables $x_1. \cdots .x_n$ to a list of nodes and values $u_1, \ldots, u_n$ in an XML document is said to be valid, if for all $i \in \{1, \ldots, n\}$, (a) if x_i is an element variable having associated element type τ, then u_i is an element node of type τ, and u_{i-1} is the parent of node u_i if $i > 1$; (b) if x_i is an attribute variable having associated attribute α, then u_i is the value of an attribute node labeled α which is a child of node u_{i-1}; (c) if x_i is a text variable, then u_i is the value of a text node which is a child of node u_{i-1}. An XML document satisfies an XML template functional dependency, if every valid binding of all lists of variables in the hypothesis to nodes and values in the XML document satisfies the conclusion.

In case that the conclusion requires equality between element variables, an XML template functional dependency acts as a key constraint, since two element nodes are equal if and only if they are they same node according to [105]. We therefore use benchmark XML keys $\mathcal{K}_1$ and $\mathcal{K}_2$ for the evaluation of XML template functional dependencies.

With respect to the element types and attributes in DTDs $\mathcal{D}_1$ and $\mathcal{D}_3$ depicted in Figures 2.5 and 2.7, respectively, the benchmark XML keys $\mathcal{K}_1$ and $\mathcal{K}_2$ are specified in form of XML template functional dependencies as depicted in Table 2.17. The element type or attribute associated to a variable is thereby indicated by the name of the variable, in accordance with the notation used in [105].

Handling of multiple property nodes: The hypothesis of XML key $\mathcal{K}_1$ depicted in Table 2.17 contains only element variables and attribute variables but no text variable. In particular, $\texttt{Invoice}_1$ and $\texttt{Invoice}_2$ as well as $\texttt{Line}_1$ and $\texttt{Line}_2$ are element variables, and $\texttt{prd}_1$, $\texttt{code}_1$, and $\texttt{no}_1$ are attribute variables. Hence, variables $\texttt{Invoice}_1$ and $\texttt{Invoice}_2$ are bound to (not necessarily distinct) element nodes of type `Invoice`, and variables $\texttt{Line}_1$ and $\texttt{Line}_2$ are bound to (not necessarily distinct) element nodes of type `Line`. In contrast, attribute variables $\texttt{prd}_1$, $\texttt{code}_1$, and $\texttt{no}_1$ are bound to *values* of `prd` nodes, `code` nodes and `no` nodes, respectively.

The three lists of variables in the top of the hypothesis require for any valid binding that (i) the value in $\texttt{prd}_1$ is the value of a `prd` attribute node of the `Invoice` node in variable $\texttt{Invoice}_1$; (ii) the `Line` node in variable $\texttt{Line}_1$ is a child of the `Invoice` node in

$\mathcal{K}_1$	$\langle \texttt{Invoice}_1.\texttt{prd}_1 \rangle$	$\mathcal{K}_2$	$\langle \texttt{Addr}_1.\texttt{city}_1 \rangle$
	$\langle \texttt{Invoice}_1.\texttt{Line}_1.\texttt{code}_1 \rangle$		$\langle \texttt{Addr}_1.\texttt{strt}_1 \rangle$
	$\langle \texttt{Invoice}_1.\texttt{Line}_1.\texttt{no}_1 \rangle$		$\langle \texttt{Addr}_1.\texttt{hno}_1 \rangle$
	$\langle \texttt{Invoice}_2.\texttt{prd}_1 \rangle$		$\langle \texttt{Addr}_1.\texttt{ano}_1 \rangle$
	$\langle \texttt{Invoice}_2.\texttt{Line}_2.\texttt{code}_1 \rangle$		$\langle \texttt{Addr}_2.\texttt{city}_1 \rangle$
	$\langle \texttt{Invoice}_2.\texttt{Line}_2.\texttt{no}_1 \rangle$		$\langle \texttt{Addr}_2.\texttt{strt}_1 \rangle$
	$\texttt{Invoice}_1 = \texttt{Invoice}_2$		$\langle \texttt{Addr}_2.\texttt{hno}_1 \rangle$
			$\langle \texttt{Addr}_2.\texttt{ano}_1 \rangle$
			$\texttt{Addr}_1 = \texttt{Addr}_2$

Table 2.17: Benchmark constraints $\mathcal{K}_1$ and $\mathcal{K}_2$ specified as XML template functional dependencies.

variable $\texttt{Invoice}_1$, and the value in $\texttt{code}_1$ is the value of a `code` attribute node of the `Line` node in variable $\texttt{Line}_1$; (iii) the value in $\texttt{no}_1$ is the value of a `no` attribute node of the `Line` node in variable $\texttt{Line}_1$. The three lists of variables in the bottom of the hypothesis impose similar requirements on the nodes and values in variables $\texttt{Invoice}_2$ and $\texttt{Line}_2$. Note that the attribute variables $\texttt{prd}_1$, $\texttt{code}_1$, and $\texttt{no}_1$ are reused in the lists of variables in the bottom of the hypothesis. As a consequence, whenever element variables $\texttt{Invoice}_1, \texttt{Line}_1$ and $\texttt{Invoice}_2, \texttt{Line}_2$ are bound to two pairs of an `Invoice` node and a `Line` node, then the invoices must have the same invoice period, and the invoice lines must have the same combination of code and number. Combined with the condition in the conclusion, this means that if two pairs of an `Invoice` node and a `Line` node have the same values for the `prd`, `code`, and `no` nodes, then the `Invoice` nodes must be the same, which is the intended meaning of XML key $\mathcal{K}_1$.

There are three valid bindings in XML tree T_1, which are listed in Table 2.18. Because the same `Invoice` node is assigned to variables $\texttt{Invoice}_1$ and $\texttt{Invoice}_2$ in each of these bindings, XML key $\mathcal{K}_1$ is satisfied in XML tree T_1 according to the semantics of XML template functional dependencies as desired.

$\texttt{Invoice}_1$	$\texttt{prd}_1$	$\texttt{Line}_1$	$\texttt{code}_1$	$\texttt{no}_1$	$\texttt{Invoice}_2$	$\texttt{Line}_2$
v_3	*01/09*	v_5	*0660*	*1010*	v_3	v_5
v_3	*01/09*	v_9	*0990*	*2020*	v_3	v_9
v_{15}	*01/09*	v_{17}	*0660*	*2020*	v_{15}	v_{17}

Table 2.18: Bindings of the variables in the XML template functional dependency $\mathcal{K}_1$ to nodes and values in XML tree $\mathcal{T}_1$.

It is however worth being mentioned that an XML template functional dependency does not intrinsically avoid semantically incorrect combinations of property nodes when it is checked. For example, if XML template dependency $\mathcal{K}_1$ in Table 2.17 is specified in a slightly different way such that four separate `Line` variables are used, then $\{\texttt{Invoice}_1 = v_3$, $\texttt{prd}_1 =$ *01/09*, $\texttt{Line}_1 = v_5$, $\texttt{code}_1 =$ *0660*, $\texttt{Line}_2 = v_9$, $\texttt{no}_1 =$ *2020*, $\texttt{Invoice}_2 = v_{15}$, $\texttt{Line}_3 = v_{17}$, $\texttt{Line}_4 = v_{17}\}$ is a valid binding in XML tree $\mathcal{T}_1$. The binding of variables $\texttt{code}_1$ and $\texttt{no}_1$ to values *0660* and *2020* is however semantically incorrect, since *0660* and *2020* belong to different invoice lines with respect to the `Invoice` node v_3 assigned to variable $\texttt{Invoice}_1$. This slightly modified version of $\mathcal{K}_1$ is in fact violated in XML tree $\mathcal{T}_1$, because distinct `Invoice` nodes v_3 and v_{15} are assigned to variables $\texttt{Invoice}_1$ and $\texttt{Invoice}_2$, respectively, and so the conclusion is not satisfied. Hence, it is the responsibility of developers of XML applications to specify XML template dependencies in a way such that semantically incorrect combinations of property nodes are ignored and so multiple property nodes are adequately handled.

Ensuring uniqueness of property nodes: An XML template functional dependency does not ensure uniqueness of property nodes. To see this consider XML template functional dependency $\mathcal{K}_1$ depicted in Table 2.17 and XML tree $\mathcal{T}_2$. The valid bindings of the variables in the hypothesis of $\mathcal{K}_1$ to nodes and values in XML tree $\mathcal{T}_2$ are listed in Table 2.19. Because the `Invoice` node v_3 is assigned to both variables $\texttt{Invoice}_1$ and $\texttt{Invoice}_2$ in each of these bindings, $\mathcal{K}_1$ is satisfied in XML tree $\mathcal{T}_2$ according to the semantics of XML template functional dependencies, even though the combinations of property nodes v_5, v_6, v_{13} and v_9, v_{10}, v_{13}, i.e. *0990*, *2020*, *01/09*, are redundant.

$\texttt{Invoice}_1$	$\texttt{prd}_1$	$\texttt{Line}_1$	$\texttt{code}_1$	$\texttt{no}_1$	$\texttt{Invoice}_2$	$\texttt{Line}_2$
v_3	*01/09*	v_4	*0990*	*2020*	v_3	v_8
v_3	*01/09*	v_8	*0990*	*2020*	v_3	v_4

Table 2.19: Bindings of the variables in XML template functional dependency $\mathcal{K}_1$ to nodes and values in XML tree $\mathcal{T}_2$.

Generally speaking, in order to ensure uniqueness of property nodes it is necessary to assert that whenever there exist value equal combinations of property nodes, then these property nodes must be the same, and so it is necessary to compare property nodes both by means of value equality and also by means of existential equality. Translated to the setting of XML template dependencies this means that it would be necessary to bind variables in the hypothesis to values of nodes and to compare the identities of these nodes in the conclusion. Unfortunately, this is not permitted by XML template functional dependencies, and so uniqueness of property nodes cannot be ensured.

Handling of absent property nodes: The variables in an XML template functional dependencies are only bound to a set of nodes and values if there is exactly one node or value for each variable. Incomplete combinations of property nodes are therefore simply disregarded when the satisfaction of an XML template functional dependency is checked. For example with respect to XML template functional dependency $\mathcal{K}_2$ depicted in Table 2.17 and XML tree $\mathcal{T}_3$, $\{\texttt{Addr}_1 = v_9$, $\texttt{city}_1 =$ *NY*, $\texttt{strt}_1 =$ *Park Av.*, $\texttt{hno}_1 =$ *30*, $\texttt{ano}_1 =$ *2B*, $\texttt{Addr}_2 = v_9\}$ is the single valid binding such that each variable in $\mathcal{K}_2$ is bound. Because `Addr` node v_9 is assigned to both variables $\texttt{Addr}_1$ and $\texttt{Addr}_2$, the XML key $\mathcal{K}_2$ is satisfied in XML tree $\mathcal{T}_3$ according to the semantics of XML template functional dependencies. Whereas this result is intended in XML tree $\mathcal{T}_3$, $\mathcal{K}_2$ is also satisfied in XML tree $\mathcal{T}_4$, where not even one binding exists such that every variable is bound to a node or value. Hence, XML template functional dependencies do not adequately handle absent property nodes.

2.6.2 XML Disjunctive Embedded Dependencies

Deutsch & Tannen proposed an XML disjunctive embedded dependency, which is a direct extension of its relational antetype to the XML data model [106, 107]. An XML disjunctive embedded dependency is a first-order logic assertion of the form

$$\forall \mathcal{x}_1, \ldots, \mathcal{x}_n \left[\Phi(\mathcal{x}_1, \ldots, \mathcal{x}_n) \rightarrow \bigvee_{i=1}^{l} \exists \mathcal{y}_{i_1}, \ldots, \mathcal{y}_{i_m} \Psi_i(\mathcal{x}_1, \ldots, \mathcal{x}_n, \mathcal{y}_{i_1}, \ldots, \mathcal{y}_{i_m}) \right]$$

where $\mathcal{x}_1, \ldots, \mathcal{x}_n$ and $\mathcal{y}_{i_1}, \ldots, \mathcal{y}_{i_m}$ are lists of variables, and Φ and Ψ_i are conjunctions of *XPath atoms* of the form $[P](\mathcal{z})$ or $[P](\mathcal{z}, \mathcal{z}')$ and *(dis)equality atoms* of the form $(\mathcal{z} \neq \mathcal{z}')$ $\mathcal{z} = \mathcal{z}'$, where P is an XPath expression, and $\mathcal{z}$ and $\mathcal{z}'$ are variables or constants.

As for variables in XML template functional dependencies, the variables in an XML disjunctive embedded dependency are bound to nodes or values in an XML document. The binding of a variable $\mathcal{z}$ to a node or value u thereby satisfies a unary XPath atom $[P](\mathcal{z})$, if u is in the result of XPath expression P, when P is evaluated from the root of the XML document. A binary XPath atom $[P](\mathcal{z}, \mathcal{z}')$ is satisfied by the bindings of $\mathcal{z}$ to u and $\mathcal{z}'$ to u', if u' is in the result of XPath expression P, when P is evaluated relative to node u. An XML document satisfies an XML disjunctive embedded dependency, if for any binding of the variables in $\mathcal{x}_1, \ldots, \mathcal{x}_n$ that satisfies all atoms in Φ, there is some i, where $1 \leq i \leq l$, and some extension of this binding to the variables $\mathcal{y}_{i_1}, \ldots, \mathcal{y}_{i_m}$ such that all atoms in Ψ_i are satisfied by the extended binding [107].

Embedded dependencies for the relational data model allow to express most of the traditional integrity constraints, including functional dependencies, inclusion dependencies, and multivalued dependencies, but also cardinality and domain constraints [108]. Likewise, XML disjunctive embedded dependencies are a very powerful type of XML integrity constraint, and allow to specify XML keys, XML functional dependencies and XML inclusion dependencies for instance.

We now evaluate XML disjunctive embedded dependencies. For the purpose of illustration, Table 2.20 lists benchmark XML key $\mathcal{K}_1$ and XML inclusion dependency $\mathcal{I}_1$ in form of XML disjunctive embedded dependencies.

Handling of multiple property nodes: Consider first XML disjunctive embedded dependency $\mathcal{I}_1$ depicted in Table 2.20. The antecedent of $\mathcal{I}_1$ binds variables $\mathcal{A}$ and $\mathcal{B}$ to `Invoice` and `Line` nodes, respectively. Further, the $\mathcal{C}_i$ variables are bound to the values of `cno` nodes, `code` nodes, and `no` nodes, where the symbol '@' is standard XPath syntax for accessing the values of attribute nodes. In order to avoid semantically incorrect combinations of property nodes, XPath atom $[./\texttt{Line}](\mathcal{A}, \mathcal{B})$ together with XPath atoms $[@\texttt{code}](\mathcal{B}, \mathcal{C}_2)$ and $[@\texttt{no}](\mathcal{B}, \mathcal{C}_3)$ ensure that the pairs of values of a `code` node and a `no` node assigned to variables $\mathcal{C}_2$ and $\mathcal{C}_3$ belong to the same line in an invoice. The consequent of $\mathcal{I}_1$ then requires

$\mathcal{K}_1$	$\forall\ \mathcal{A}, \bar{\mathcal{A}}, \mathcal{B}, \bar{\mathcal{B}}, \mathcal{C}_1, \bar{\mathcal{C}}_1, \mathcal{C}_2, \bar{\mathcal{C}}_2, \mathcal{C}_3, \bar{\mathcal{C}}_3, \mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$ $[//\texttt{Invoice}](\mathcal{A}) \wedge [./\texttt{prd}](\mathcal{A}, \mathcal{C}_1) \wedge [\text{text()}](\mathcal{C}_1, \mathcal{D}_1)\wedge$ $[./\texttt{Line}](\mathcal{A}, \mathcal{B}) \wedge [./\texttt{code}](\mathcal{B}, \mathcal{C}_2) \wedge [\text{text()}](\mathcal{C}_2, \mathcal{D}_2) \wedge [./\texttt{no}](\mathcal{B}, \mathcal{C}_3) \wedge [\text{text()}](\mathcal{C}_3, \mathcal{D}_3)\wedge$ $[//\texttt{Invoice}](\bar{\mathcal{A}}) \wedge [./\texttt{prd}](\bar{\mathcal{A}}, \bar{\mathcal{C}}_1) \wedge [\text{text()}](\bar{\mathcal{C}}_1, \mathcal{D}_1)\wedge$ $[./\texttt{Line}](\bar{\mathcal{A}}, \bar{\mathcal{B}}) \wedge [./\texttt{code}](\bar{\mathcal{B}}, \bar{\mathcal{C}}_2) \wedge [\text{text()}](\bar{\mathcal{C}}_2, \mathcal{D}_2) \wedge [./\texttt{no}](\bar{\mathcal{B}}, \bar{\mathcal{C}}_3) \wedge [\text{text()}](\bar{\mathcal{C}}_3, \mathcal{D}_3)\wedge$ $\rightarrow \mathcal{A} = \bar{\mathcal{A}} \wedge \mathcal{C}_1 = \bar{\mathcal{C}}_1 \wedge \mathcal{C}_2 = \bar{\mathcal{C}}_2 \wedge \mathcal{C}_3 = \bar{\mathcal{C}}_3$
$\mathcal{I}_1$	$\forall\ \mathcal{A}, \mathcal{B}, \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ $[//\texttt{Invoice}](\mathcal{A}) \wedge [@\texttt{cno}](\mathcal{A}, \mathcal{C}_1) \wedge [./\texttt{Line}](\mathcal{A}, \mathcal{B}) \wedge [@\texttt{code}](\mathcal{B}, \mathcal{C}_2) \wedge [@\texttt{no}](\mathcal{B}, \mathcal{C}_3)$ $\rightarrow \exists\ \mathcal{D}\ [//\texttt{Phone}](\mathcal{D}) \wedge [@\texttt{cno}](\mathcal{D}, \mathcal{C}_1) \wedge [@\texttt{code}](\mathcal{D}, \mathcal{C}_2) \wedge [@\texttt{no}](\mathcal{D}, \mathcal{C}_3)$

Table 2.20: Benchmark constraints $\mathcal{K}_1$ and $\mathcal{I}_1$ specified as XML disjunctive embedded dependencies.

the existence of a `Phone` node, which is assigned to variable $\mathcal{D}$, such that this `Phone` node has a `cno`, a `code`, and a `no` attribute node with the values assigned to variables $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$, respectively. That is, the existential quantification of a `Phone` node in the consequent asserts the subset relationship between the values of `cno`, `code`, and `no` nodes nested within `Invoice` nodes and the values of `cno`, `code`, and `no` nodes nested within `Phone` nodes. Table 2.21 lists the bindings of variables in XML disjunctive embedded dependency $\mathcal{I}_1$ to nodes and values in XML tree $\mathcal{T}_1$. Because semantically incorrect combinations of property nodes are disregarded, there exists a $\mathcal{D}$ node for every binding of nodes and values to the variables in the antecedent of $\mathcal{I}_1$, and so $\mathcal{I}_1$ is satisfied in XML tree $\mathcal{T}_1$ according to the semantics of XML disjunctive embedded dependencies as desired.

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_3$	$\mathcal{D}$
v_3	v_5	*C1*	*0660*	*1010*	v_{24}
v_3	v_9	*C1*	*0990*	*2020*	v_{28}
v_{15}	v_{17}	*C2*	*0660*	*2020*	v_{32}

Table 2.21: Bindings of the variables in XML disjunctive embedded dependency $\mathcal{I}_1$ to nodes and values in XML tree $\mathcal{T}_1$

.

The rationale for XML disjunctive embedded dependency $\mathcal{K}_1$ depicted in Table 2.20 is essentially the same as the rationale for the XML template functional dependency depicted in Table 2.17, which has been illustrated in Section 2.6.1. Variables $\mathcal{A}$ and $\mathcal{B}$ as well as variables $\bar{\mathcal{A}}$ and $\bar{\mathcal{B}}$ bind pairs of an `Invoice` node and a `Line` node. An XML disjunctive embedded dependency permits in contrast to an XML template functional dependency to freely choose whether to bind a variable to the value of a node or to the identity of a node. This is the essential prerequisite for ensuring uniqueness of property nodes, as will be made more clear soon. In addition to the three $\mathcal{D}_i$ variables for binding the values of `prd` nodes, `code` nodes and `no` nodes, the XML disjunctive embedded dependency $\mathcal{K}_1$ binds the $\mathcal{C}_i$ and $\bar{\mathcal{C}}_i$ variables to the identities of `prd` nodes, `code` nodes, and `no` nodes. The $\mathcal{D}_i$ variables are thereby bound to the values of nodes using the standard XPath function text(). Note that each $\mathcal{D}_i$ variable is bound to the value of both the node in $\mathcal{C}_i$ and the node in $\bar{\mathcal{C}}_i$, and so each pair of nodes assigned to variables $\mathcal{C}_i$ and $\bar{\mathcal{C}}_i$ must have the same value, i.e. only pairwise value equal combinations of a `prd` node, a `code` node, and a `no` node are assigned to the $\mathcal{C}_i$ and $\bar{\mathcal{C}}_i$ variables.

The equality atoms to the right of the implication symbol parallel the conclusion of an XML template functional dependency. However, instead of only requiring the existential equality of the `Invoice` nodes assigned to variables $\mathcal{A}$ and $\bar{\mathcal{A}}$ in order to assert the identification of `Invoice` nodes, it is additionally required that the combinations of `prd` nodes, `code` nodes and `no` nodes assigned to the $\mathcal{C}_i$ and $\bar{\mathcal{C}}_i$ variables must be pairwise the same.

Table 2.22 lists the bindings of variables in XML disjunctive embedded dependency $\mathcal{K}_1$ to nodes and values in XML tree $\mathcal{T}_1$. Again, semantically incorrect combinations of property nodes are disregarded and so $\mathcal{K}_1$ is satisfied in $\mathcal{T}_1$ as desired.

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}_1$	$\mathcal{D}_1$	$\mathcal{C}_2$	$\mathcal{D}_2$	$\mathcal{C}_3$	$\mathcal{D}_3$	$\bar{\mathcal{A}}$	$\bar{\mathcal{B}}$	$\bar{\mathcal{C}}_1$	$\bar{\mathcal{C}}_2$	$\bar{\mathcal{C}}_3$
v_3	v_5	v_{14}	*01/09*	v_6	*0660*	v_7	*1010*	v_3	v_5	v_{14}	v_6	v_7
v_3	v_9	v_{14}	*01/09*	v_{10}	*0990*	v_{11}	*2020*	v_3	v_9	v_{14}	v_{10}	v_{11}
v_{15}	v_{17}	v_{22}	*01/09*	v_{18}	*0660*	v_{19}	*2020*	v_{15}	v_{17}	v_{22}	v_{18}	v_{19}

Table 2.22: Bindings of the variables in XML disjunctive embedded dependency $\mathcal{K}_1$ to nodes and values in XML tree $\mathcal{T}_1$

.

As for XML template functional dependencies, also XML disjunctive embedded dependencies do not intrinsically avoid semantically incorrect combinations of property nodes. It is again the responsibility of application developers to carefully define a constraint such that desired semantics is asserted.

Uniqueness of property nodes: XML disjunctive embedded dependencies allow to ensure uniqueness of property nodes in contrast to XML template functional dependencies. To see this consider the validation of XML tree $\mathcal{T}_2$ against XML disjunctive embedded dependency $\mathcal{K}_1$. Because the consequent of $\mathcal{K}_1$ requires nodes, which are bound to the $\mathcal{C}_i$ and $\bar{\mathcal{C}}_i$ variables, to be the same, it is effectively required that whenever there exist value equal combinations of a `prd` node, a `code` node, and a `no` node, then these nodes must be the same, i.e. the combinations of a `prd` node, a `code` node, and a `no` node must be unique. Table 2.23 lists the bindings of variables in XML disjunctive embedded dependency $\mathcal{K}_1$ to nodes and values in XML tree $\mathcal{T}_2$. Because different nodes are assigned to variables $\mathcal{C}_2$ and $\bar{\mathcal{C}}_2$ as well as $\mathcal{C}_3$ and $\bar{\mathcal{C}}_3$ in the bindings in the second and the fourth line of Table 2.23, the consequent of $\mathcal{K}_1$ is not satisfied. Hence, $\mathcal{K}_1$ is violated in XML tree $\mathcal{T}_2$ as desired. This desirable result is again achieved solely because of the deliberate specification of $\mathcal{K}_1$ in form of an XML disjunctive dependency.

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}_1$	$\mathcal{D}_1$	$\mathcal{C}_2$	$\mathcal{D}_2$	$\mathcal{C}_3$	$\mathcal{D}_3$	$\bar{\mathcal{A}}$	$\bar{\mathcal{B}}$	$\bar{\mathcal{C}}_1$	$\bar{\mathcal{C}}_2$	$\bar{\mathcal{C}}_3$
v_3	v_4	v_{13}	01/09	v_5	0990	v_6	2020	v_3	v_4	v_{13}	v_5	v_6
v_3	v_4	v_{13}	01/09	v_5	0990	v_6	2020	v_3	v_8	v_{13}	v_9	v_{10}
v_3	v_8	v_{13}	01/09	v_9	0990	v_{10}	2020	v_3	v_8	v_{13}	v_9	v_{10}
v_3	v_8	v_{13}	01/09	v_9	0990	v_{10}	2020	v_3	v_4	v_{13}	v_5	v_6

Table 2.23: Bindings of the variables in the XML disjunctive embedded dependency representing benchmark XML key $\mathcal{K}_1$ to nodes and variables in XML tree $\mathcal{T}_2$
.

Handling of absent property nodes: The variables in an XML disjunctive embedded dependency are only bound to nodes and values if there is exactly one node or value for each variable. As for XML template functional dependencies, also XML disjunctive embedded dependencies therefore do not allow to adequately deal with absent property nodes. In particular, benchmark constraint $\mathcal{K}_2$ is satisfied in both XML trees $\mathcal{T}_3$ and $\mathcal{T}_4$ as well as benchmark constraint $\mathcal{I}_2$ is satisfied in both XML trees $\mathcal{T}_5$ and $\mathcal{T}_6$ according to the semantics of XML disjunctive embedded dependencies. Since $\mathcal{K}_2$ and $\mathcal{I}_2$ should be violated in XML trees $\mathcal{T}_4$ and $\mathcal{T}_6$, respectively, XML disjunctive embedded dependencies do not adequately handle absent property nodes.

2.7 Summary

Table 2.24 summarizes the results of our evaluation of previous approaches to value-based XML integrity constraints. The symbols in Table 2.24 have the following meaning:

$+$ ...	The test case is passed.
$\times$ ...	The approach does not allow to specify the type of constraint used in the test case.
$-$ ...	The approach allows in general to specify the type of constraint used in the test case but either does not allow to specify the particular benchmark constraint or uses a model of XML data which does not allow to represent the XML tree in the test case.
$\ominus$ ...	The approach allows to specify the benchmark constraint but checking the satisfaction of the constraint in the benchmark XML tree yields a counter-intuitive result.

The strength of selector/field XML integrity constraints is clearly their intuitive syntax. Unfortunately, multiple and absent property nodes are not adequately handled by selector/field XML integrity constraints.

Compared to selector/field XML integrity constraints, the syntax of tuple-based XML integrity constraints is less intuitive since only the targeted property nodes are explicitly specified for tuple-based XML integrity constraints but not the targeted entity nodes. Nevertheless, tuple-based approaches as well as the subtree-based approach exploit structural relationships between property nodes and therefore disregard semantically incorrect combinations of property nodes in general. Absent property nodes are handled in these approaches either by means of applying strong/weak satisfaction semantics or by means of disregarding

Test Case / *Approach*	$\mathcal{C}1$	$\mathcal{C}2$	$\mathcal{C}3$	$\mathcal{C}4$	$\mathcal{C}5$	$\mathcal{C}6$	$\mathcal{C}7$
ID and IDREF	$-$	$-$	$-$	$-$	$-$	$-$	$-$
Subtree-based approach	$+$	$\times$	$\times$	$+$	$\ominus$	$\times$	$\times$
Selector/Field approaches							
With restrictions on fields	$-$	$-$	$-$	$-$	$-$	$-$	$-$
With restrictions on field nodes							
- XSD key and foreign key	$\ominus$	$\ominus$	$+$	$\ominus$	$+$	$\ominus$	$+$
- XSD unique constraint	$\ominus$	$\times$	$+$	$+$	$\ominus$	$\times$	$\times$
Unrestricted	$\ominus$	$\times$	$\ominus$	$+$	$\ominus$	$\times$	$\times$
Tuple-based approaches							
Intersection path approach	$\ominus$	$\ominus$	$\times$	$+$	$\ominus$	$+$	$\ominus$
Tree-tuple approach	$+$	$\times$	$\ominus$	$+$	$\ominus$	$\times$	$\times$
Weak closest-node approach	$\ominus$	$\times$	$\ominus$	$+$	$+$	$\times$	$\times$
Strong closest-node approach	$+$	$\times$	$\ominus$	$+$	$+$	$\times$	$\times$
Hedge-based approach	$+$	$+$	$+$	$+$	$\ominus$	$+$	$\ominus$
Formula-based approaches							
XML template functional dependencies	$+$	$\times$	$\ominus$	$+$	$\ominus$	$+$	$\ominus$
XML disjunctive embedded dependencies	$+$	$+$	$+$	$+$	$\ominus$	$+$	$\ominus$

Table 2.24: Evaluation of previous approaches to value-based XML integrity constraints.

incomplete combinations of property nodes at all. In general none of these techniques is adequate for handling absent property nodes as we have illustrated already in the introductory chapter. Applying strong satisfaction semantics is nevertheless an expedient manner for handling absent property nodes in checking the satisfaction of XML functional dependencies in particular. This is made evident by the results of both the strong and the weak 'closest node' approach for test cases $\mathcal{C}_4$ and $\mathcal{C}_5$ (cf. Table 2.2). But we note that applying strong satisfaction semantics leads to counter-intuitive results in checking the satisfaction of XML inclusion dependencies. This has been illustrated in Example 1.10.

Formula-based XML integrity constraints are highly expressive constraints. Especially, XML embedded disjunctive dependencies as proposed by Deutsch & Tannen. The limitations of formula-based XML integrity constraints are twofold. The first limitation is that it is left to the application developer to ensure that multiple property nodes are adequately handled, which is clearly undesirable. The second limitation is that formula-based XML integrity constraints do not allow for adequately handling absent property nodes.

Chapter 3

The Enhanced Closest node Approach

Contents

This chapter presents the definitions of XML keys and XML inclusion dependencies in the enhanced 'closest node' approach. For this purpose, Section 3.1 reviews the basics of graphs and trees and introduces some custom operators on these data structures. Section 3.2 presents our model of XML data, and finally Section 3.3 defines the syntax and semantics of the XML keys and XML inclusion dependencies proposed in this thesis.

3.1 Preliminaries

The purpose of this section is to review basics of graphs and trees, which we then use in Section 3.2 for defining our model of XML data. For detailed discussions of graphs and trees, we refer the reader to the excellent textbooks by Diestel [109] and Tutte [110].

3.1.1 Graphs and Trees

A graph is essentially a pair of a set of nodes and a set of edges connecting the nodes. Based on this general notion, we define a graph as follows.

Definition 3.1 (Graph) A *graph* G is defined to be

$$\mathrm{G} = (\boldsymbol{V}, \boldsymbol{E})$$

where $\boldsymbol{V}$ is the finite set of *nodes* and $\boldsymbol{E} \subseteq \boldsymbol{V} \times \boldsymbol{V}$ is the set of *edges* such that
(i) for all $(v, \bar{v}) \in \boldsymbol{E}$, $v \neq \bar{v}$;
(ii) for all $v \in \boldsymbol{V}$, if $\boldsymbol{V} \neq \{v\}$ then $\exists \bar{v} \in \boldsymbol{V}$ such that $(v, \bar{v}) \in \boldsymbol{E}$ or $(\bar{v}, v) \in \boldsymbol{E}$.

The graph G is defined to be the *empty graph* if $\boldsymbol{V} = \emptyset$, and G is defined to be a *trivial graph* if $|\boldsymbol{V}| = 1$.

A couple of comments on Definition 3.1 are appropriate. In contrast to graph models frequently used in general graph theory, our graph model requires graphs to consist of finitely many nodes and thus only permits *finite graphs*. The edges in a graph are a subset of the cross product of the nodes in the graph according to Definition 3.1. An edge is therefore an ordered pair of nodes and so the edges in a graph are *directed edges*. In particular, an edge $(v, \bar{v})$ leads *from* node v *to* node $\bar{v}$. Requirement (i) in Definition 3.1 means that an edge does not connect a node to itself. Loops are hence excluded in our graph model. Requirement (ii) in Definition 3.1 means that every node in a non-trivial graph must be connected to at least one other node. Our graph model thus only permits *connected graphs*.

A sequence of connected nodes in a graph is usually called a path. The term 'path' however has some special meaning in the context of XML. In order to avoid ambiguities we call a sequence of connected nodes in a graph a walk.

Definition 3.2 (Walk and Cycle) Let $\mathrm{G} = (\boldsymbol{V}, \boldsymbol{E})$ be a graph. A non-empty list of nodes $[v_1, \ldots, v_n] \subseteq \boldsymbol{V}$ is defined to be *walk* in G, denoted by $v_1 . \cdots . v_n$, if for all $i \in \{1, \ldots, n\}$, $(v_{i-1}, v_i) \in \boldsymbol{E}$ if $i > 1$. The walk $v_1 . \cdots . v_n$ is defined to be a *cycle* if $v_1 = v_n$ and $n > 1$.

Example 3.1 *Figure 3.1a depicts a graph consisting of three nodes. The number beside a node indicates its node identifier which we use for referencing individual nodes in examples. The edges connecting the nodes are depicted by solid lines in Figure 3.1a. The arrow at the end of an edge indicates its direction. For instance, the edge connecting nodes v_1 and v_2 leads from v_1 to v_2. Also, the sequence of connected nodes $v_1.v_2.v_3$ is a walk in this graph. In contrast, $v_1.v_3$ is not a walk in this graph because the edge that connects v_1 and v_3 leads from v_3 to v_1 and not in the opposite direction. Finally, the walk $v_1.v_2.v_3.v_1$ is a cycle.*

We now define a tree, a special case of a graph, since we model an XML document as a tree. The actual difference between a graph and a tree is that the former permits walks which are cycles in contrast to the latter.

Definition 3.3 (Tree) A graph G is defined to be a *tree* if no walk in G is a cycle.

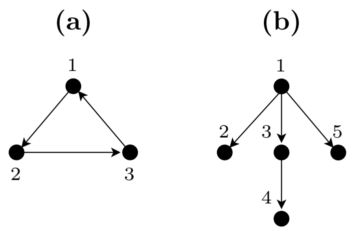


Figure 3.1: A graph and a tree.

Because edges are directed and cycles are prohibited in a tree, there exists exactly one node in every non-empty tree from which edges only emanate but to which no edge leads to. This node is usually called the root node of a tree. The next definition makes the notion of the root node of a tree more precise.

Definition 3.4 (Root Node) Let G = $(\boldsymbol{V}, \boldsymbol{E})$ be a non-empty tree. The root node of G, denoted by root(G), is defined to be

$$\text{root}(\text{G}) = v \in \boldsymbol{V} \mid \nexists\ \bar{v} \in \boldsymbol{V} \text{ such that } (\bar{v}, v) \in \boldsymbol{E}.$$

Example 3.2 *Figure 3.1b depicts a tree consisting of five nodes. The root node in this tree is node v_1. Hence, if we denote this tree by* G*, then* $\text{root}(G) = v_1$.

Because a tree is a connected graph and edges do not lead to the root node of a tree, there is a walk to every node in the tree which starts at the root node. Hence, there are the notions of above and below in trees, where above means closer to the root node and below means further away from the root node. We define next certain types of vertical relationships between nodes in a tree, which we call axes, in alignment with the terminology used in the XML specification by the W3C.

Definition 3.5 (Axes) Let G = $(\boldsymbol{V}, \boldsymbol{E})$ be a tree. With respect to a given node $v \in \boldsymbol{V}$,

$$\begin{aligned}
\text{parent}(v, \text{G}) =& \{\bar{v} \in \boldsymbol{V} \mid (\bar{v}, v) \in \boldsymbol{E}\} \\
\text{ancestor}(v, \text{G}) =& \text{parent}(v) \cup \text{ancestor}(\text{parent}(v, \text{root}(G)), \text{root}(G)) \\
\text{anc-or-self}(v, \text{G}) =& \text{ancestor}(v, \text{root}(G)) \cup \{v\} \\
\text{child}(v, \text{G}) =& \{\bar{v} \in \boldsymbol{V} \mid (v, \bar{v}) \in \boldsymbol{E}\} \\
\text{descendant}(v, \text{G}) =& \text{child}(v, \text{root}(G)) \cup \text{descendant}(\bar{v}, \text{root}(G)) \text{ for all } \bar{v} \in \text{child}(v, \text{root}(G)) \\
\text{desc-or-self}(v, \text{G}) =& \text{descendant}(v, \text{root}(G)) \cup \{v\}
\end{aligned}$$

With respect to a given node v, axes parent, ancestor, and anc-or-self relate nodes to v which are above v in the tree under consideration. These axes are therefore called *upward axes.* In contrast, axes child, descendant, and desc-or-self relate nodes to v which are below v in the tree under consideration. These axes are therefore called *downward axes.* We will omit explicitly stating the tree in which an axis is computed if the tree is understood from the context. Also, it is worth being mentioned that every node in a tree has exactly one parent node except for the root node which does not have a parent node at all.

Example 3.3 *Referring to the tree depicted in Figure 3.1b,* $\text{parent}(v_1) = \emptyset$ *whereas* $\text{parent}(v_2) = \{v_1\}$*. Also,* $\text{ancestor}(v_4) = \{v_3, v_1\}$ *and* $\text{anc-or-self}(v_4) = \{v_4, v_3, v_1\}$*. Next,* $\text{child}(v_5) = \emptyset$ *whereas* $\text{child}(v_3) = \{v_4\}$*. Finally,* $\text{descendant}(v_1) = \{v_2, v_3, v_4, v_5\}$ *and* $\text{desc-or-self}(v_1) = \{v_1, v_2, v_3, v_4, v_5\}$*.*

In addition to the notions of above and below in a tree, it is also convenient to have the notions of left and right. For this purpose we now introduce ordered trees.

Definition 3.6 (Ordered Tree) An *ordered tree* is defined to be

$$(\boldsymbol{V}, \boldsymbol{E}, \leq), \text{ where}$$

- $(\boldsymbol{V}, \boldsymbol{E})$ is a tree, and
- $\leq$ is a partial order on $\boldsymbol{V}$ such that for all $(v, v') \in \boldsymbol{V} \times \boldsymbol{V}$, nodes v and v' are related by $\leq$ iff $\text{parent}(v) = \text{parent}(v')$.

According to Definition 3.6, every set of nodes that have the same parent node are ordered by $\leq$. The child nodes of a node are therefore lists of nodes in ordered trees as opposed to unordered trees where child nodes are (unordered) sets. It is hence determined for each pair of child nodes $(v, \bar{v})$ of the same parent node whether v is to the left of $\bar{v}$ or vice versa. In the remainder of this thesis we assume that all trees are ordered trees even though we will omit to explicitly state the ordering function $\leq$ unless we need to refer to the ordering of child nodes.

Example 3.4 *Consider once more the tree depicted in Figure 3.1b and suppose that partial ordering $\leq$ is defined in correspondence with the arrangement of nodes in Figure 3.1b. Then, $v_2 \leq v_2$ as well as $v_2 \leq v_3$. In contrast, $v_5 \not\leq v_2$. Also, nodes v_4 and v_2 are not related by $\leq$ at all, i.e. $\leq$ is undefined for v_4 and v_2, since these nodes do not have the same parent node.*

3.1.2 Operators on Trees

We now define some tree operators that are needed later in the thesis. We start with defining the projection of a tree on an node v.

Definition 3.7 (Tree Projection) Let $\mathrm{G} = (\boldsymbol{V}, \boldsymbol{E})$ be a tree, and let $v \in \boldsymbol{V}$ be a node. The operation of projecting G on v, denoted by $\mathrm{G}[v]$, results in the tree $(\boldsymbol{V}', \boldsymbol{E}')$, where $\boldsymbol{V}' = \text{desc-or-self}(v, \mathrm{G})$ and $\boldsymbol{E}' = \{(\ddot{v}, \bar{v}) \in \boldsymbol{E} \mid \{\ddot{v}, \bar{v}\} \subseteq \boldsymbol{V}'\}$.

The projection of a tree on a node v results in the subtree rooted at v. Consider for example the tree depicted in the top of Figure 3.2a. If we denote this tree by G, then G$[v_3]$ results in the subtree of G depicted in the bottom of Figure 3.2a.

We define next the operation of adding a tree $\bar{\mathrm{G}}$ as principal subtree[1] to another tree G.

Definition 3.8 (Tree Add) Let $\mathrm{G} = (\boldsymbol{V}, \boldsymbol{E})$ and $\bar{\mathrm{G}} = (\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}})$ be trees such that $\boldsymbol{V} \cap \bar{\boldsymbol{V}} = \emptyset$. The operation of adding $\bar{\mathrm{G}}$ to G, denoted by $\mathrm{G} + \bar{\mathrm{G}}$, results in the tree $(\boldsymbol{V}', \boldsymbol{E}')$, where $\boldsymbol{V}' = \boldsymbol{V} \cup \bar{\boldsymbol{V}}$, and $\boldsymbol{E}' = \boldsymbol{E} \cup \bar{\boldsymbol{E}} \cup \{(\mathrm{root}(\mathrm{G}), \mathrm{root}(\bar{\mathrm{G}}))\}$.

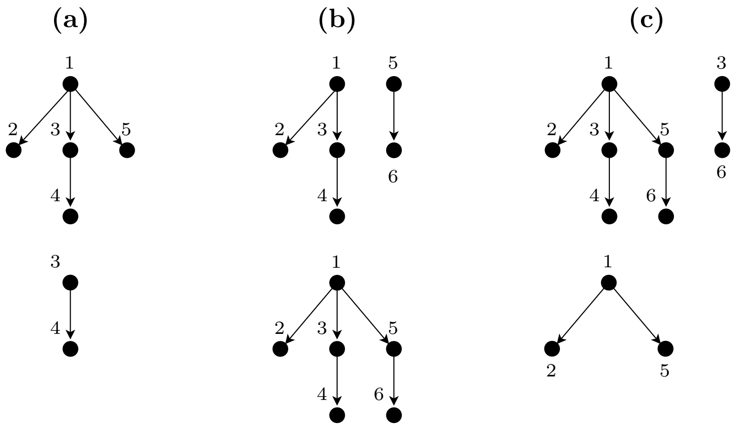


Figure 3.2: Trees resulting from the projection, add, and subtraction operator.

In order to ensure that adding a tree to another results in a tree in terms of our tree model, Definition 3.8 requires the two trees to have disjoint sets of nodes. For example, if we denote the trees depicted in the top left and right of Figure 3.2b by G and $\bar{\mathrm{G}}$, then G and $\bar{\mathrm{G}}$ have disjoint sets of nodes, as indicated by the disjoint sets of node identifiers. It is therefore valid to add $\bar{\mathrm{G}}$ as principal subtree to G. In particular, $\mathrm{G} + \bar{\mathrm{G}}$ yields the tree depicted in the bottom of 3.2b, which obviously contains $\bar{\mathrm{G}}$ as one of its principal subtrees.

We finally introduce the subtraction operator. Roughly speaking, this operator deletes all subtrees in a tree G which are rooted at nodes that are contained in another tree $\bar{\mathrm{G}}$.

Definition 3.9 (Tree Subtraction) Let $\mathrm{G} = (\boldsymbol{V}, \boldsymbol{E})$ and $\bar{\mathrm{G}} = (\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}})$ be trees. The operation of subtracting $\bar{\mathrm{G}}$ from G, denoted by $\mathrm{G} - \bar{\mathrm{G}}$, results in the tree $(\boldsymbol{V}', \boldsymbol{E}')$, where $\boldsymbol{V}' = \boldsymbol{V} - \{v \in \boldsymbol{V} \mid v \in \text{desc-or-self}(\bar{v}, \mathrm{G}) \wedge \bar{v} \in \bar{\boldsymbol{V}}\}$, and $\boldsymbol{E}' = \{(\ddot{v}, \bar{v}) \in \boldsymbol{E} \mid \{\ddot{v}, \bar{v}\} \subseteq \boldsymbol{V}'\}$.

We now illustrate the subtraction operator. Consider trees G and $\bar{\mathrm{G}}$ depicted in the top left and right of Figure 3.2c, respectively. The common nodes in trees G and $\bar{\mathrm{G}}$ are nodes v_3 and v_6. Hence, subtracting $\bar{\mathrm{G}}$ from G means to remove the subtrees of G which are rooted at nodes v_3 and v_6. The tree resulting from $\mathrm{G} - \bar{\mathrm{G}}$ is depicted in the bottom of Figure 3.2c.

[1]A principal subtree is one whose root node is a child of the root node of the original tree.

3.2 A Model of XML Data

We now present our model of XML data. We define XML trees in Section 3.2.1, and introduce the notion of a path in Section 3.2.2. In Section 3.2.3 we finally redefine the operators on trees introduced in Section 3.1.2 for the special class of XML trees.

3.2.1 XML Trees: The Primary Data Structure

Our intention is to model XML data as a tree of nodes which have assigned labels in order to represent the names of XML elements and XML attributes, and which may have assigned values in order to represent values of XML attributes as well as text enclosed by XML elements. We therefore assume the existence of a set of permitted values as well as a set of permitted labels, and we define these sets follows.

Definition 3.10 (Values and Labels) A *value* u is a member of the fixed, countably infinite set of values $\boldsymbol{U}$. A *label* l is a member of the fixed set of labels $\boldsymbol{L} = \boldsymbol{L}^E \cup \boldsymbol{L}^A \cup \{\mathtt{S}\}$, where $\boldsymbol{L}^E$ and $\boldsymbol{L}^A$ are countably infinite, disjoint sets of *element labels* and *attribute labels*, respectively, and $\mathtt{S}$ is the distinguished *text label*.

Our reason for distinguishing element labels, attribute labels and the text label is that we couple the kind of a node to the label of the node in our model of XML data, which allows for more compact definitions. That is, we assume that if a node has assigned an element label, an attribute label or a text label, then it is an element node, an attribute node or a text node, respectively. It is worth being mentioned that this assumption does not limit the generality of our model of XML data even though it is in contrast to other models like the Document Object Model (DOM) [111] for example. We now make these ideas more precise.

Definition 3.11 (XML Tree) An *XML tree* T is defined to be

$$\mathrm{T} = (\boldsymbol{V}, \boldsymbol{E}, \mathrm{lab}, \mathrm{val}), \text{ where}$$

(1) $(\boldsymbol{V}, \boldsymbol{E})$ is a tree in terms of Definition 3.3;
(2) the total function $\mathrm{lab} : \boldsymbol{V} \to \boldsymbol{L}$ assigns labels to the nodes in $\boldsymbol{V}$, such that a node $v \in \boldsymbol{V}$ is said to be an *element node* if $\mathrm{lab}(v) \in \boldsymbol{L}^E$, an *attribute node* if $\mathrm{lab}(v) \in \boldsymbol{L}^A$, and a text node if $\mathrm{lab}(v) = \mathtt{S}$;
(3) the partial function $\mathrm{val} : \{v \in \boldsymbol{V} \mid \mathrm{lab}(v) \notin \boldsymbol{L}^E\} \to \boldsymbol{U}$ assigns values to the attribute nodes and text nodes in $\boldsymbol{V}$;
(4) for all $(v, \bar{v}) \in \boldsymbol{E}$
  (i) $\mathrm{lab}(v) \in \boldsymbol{L}^E$, and
  (ii) if $\bar{v} \in \boldsymbol{L}^A$ then $\nexists(v, \bar{v}') \in \boldsymbol{E}$ such that $\bar{v} \neq \bar{v}'$ and $\mathrm{lab}(\bar{v}) = \mathrm{lab}(\bar{v}')$.

A couple of comments are appropriate. First, requirement (4.i) in Definition 3.11 means that only element nodes are inner nodes in an XML tree, which reflects the property of an XML document of being well-formed. Second, in correspondence with the XML specification by the W3C, (4.ii) in Definition 3.11 requires that an element node does not have two distinct child attribute nodes which have the same label assigned. Third, element nodes are allowed to have more than only one child text node. Our XML tree model therefore allows to represent mixed-content in XML documents.

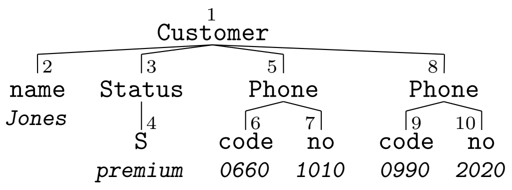


Figure 3.3: An XML tree conforming to Definition 3.11.

Example 3.5 (XML tree) *Figure 3.3 depicts an XML tree in terms of our tree model which represents information about customer* `Jones` *and his or her phones. Thereby,* $\{\texttt{Customer}, \texttt{Status}, \texttt{Phone}\} \subseteq \boldsymbol{E}$ *are element labels, and* $\{\texttt{name}, \texttt{code}, \texttt{no}\} \subseteq \boldsymbol{A}$ *are attribute labels. For instance,* $\mathrm{lab}(v_1) = \texttt{Customer}$ *and so* v_1 *is an element node, which is also the root node in this XML tree. Also,* $\mathrm{val}(v_2) =$ `Jones` *as well as* $\mathrm{val}(v_4) =$ `premium`*.*

3.2.2 Paths and Reachable Nodes

We proceed with defining the fundamental notion of a path in an XML tree.

Definition 3.12 (Path) A path P is defined to be

$$P = l_1. \cdots .l_n,$$

where for all $i \in \{1, \ldots, n\}$, $l_i \in \boldsymbol{L}$ and $l_i \in \boldsymbol{L}^E$ if $i \neq n$.

Because our model of XML data only permits element nodes as inner nodes in an XML tree, Definition 3.12 requires that a path does not contain attribute or text labels in any position other than the last. For instance, referring to Example 3.5, the sequence of labels `Customer.Phone` is a path, whereas `name.Phone` is not since it starts with attribute label `name`. We now introduce some frequently required properties of paths.

Definition 3.13 (Properties of Paths) Let P and $\bar{P}$ be given paths, where $P = l_1. \cdots .l_m$ and $\bar{P} = \bar{l}_1. \cdots .\bar{l}_n$.

Length: The length of P, denoted by $|P|$, is defined to be $|P| = m$.

First and Last Label: The first label in P, denoted by $\mathrm{first}(P)$, is defined to be $\mathrm{first}(P) = l_1$. The last label in P, denoted by $\mathrm{last}(P)$, is defined to be $\mathrm{last}(P) = l_m$.

Prefix and Strict Prefix: Path P is defined to be a prefix of $\bar{P}$, denoted by $P \subseteq \bar{P}$, if $m \leq n$ and for all $i \in \{1, \ldots, m\}$, $l_i = \bar{l}_i$. Path P is defined to be a strict prefix of $\bar{P}$, denoted by $P \subset \bar{P}$, if $P \subseteq \bar{P}$ and $m < n$.

Equality: Path P is defined to be equal to $\bar{P}$, denoted by $P = \bar{P}$, if $P \subseteq \bar{P}$ and also $\bar{P} \subseteq P$.
Parent Path: Path P is defined to be the parent path of $\bar{P}$, denoted by $P = \text{parent}(\bar{P})$, if $P \subset \bar{P}$ and $m + 1 = n$.
Intersection: If $l_1 = \bar{l}_1$, the intersection of P and $\bar{P}$, denoted by $P \cap \bar{P}$, is defined to be the longest path that is a prefix of both P and $\bar{P}$.

Again referring to Example 3.5, if we let $P =$ `Customer.Phone.code` then length$(P) = 3$, first$(P) =$ `Customer`, last$(P) =$ `code`, and parent$(P) =$ `Customer.Phone`. Also, `Customer.Phone` $\subset$ `Customer.Phone.code` and `Customer.Phone.code` $\cap$ `Customer.Phone.no` $=$ `Customer.Phone`. In contrast, `Phone.code` $\cap$ `Status.S` is undefined, since these two paths do not start with the same label. We define next the concatenation of two paths.

Definition 3.14 (Path Concatenation) Let $P = l_1.\cdots.l_m$ and $\bar{P} = \bar{l}_1.\cdots.\bar{l}_n$ be paths such that $l_m \in \boldsymbol{L}^E$. Then, the concatenation of P and $\bar{P}$, denoted by $P.\bar{P}$, is defined to result in the path $l_1.\cdots.l_m.\bar{l}_1.\cdots.\bar{l}_n$.

The requirement on a path P to end in an element label in case that a path $\bar{P}$ is concatenated to P ensures that path $P.\bar{P}$ conforms to Definition 3.12. For instance, referring once more to Example 3.5, the concatenation of paths `Customer.Phone` and `code` is legal and results in path `Customer.Phone.code`. In contrast, paths `Customer.Phone.code` and `no` cannot be concatenated since path `Customer.Phone.code` does not end in an element label. Note that the sequence of labels `Customer.Phone.code.no` is not a path in terms of Definition 3.12, since `code` is an attribute label in a position other than the last.

We now turn to the reachability of nodes in an XML tree by following a path. Intuitively, a node v is reachable over a path P if the labels of nodes in the walk from the root node of the XML tree to node v coincide with the labels in P. We now make this idea more precise.

Definition 3.15 (Walk on Path) Let $P = l_1.\cdots.l_n$ be a path, and let $v_1.\cdots.v_n$ be a walk in an XML tree $\text{T} = (\boldsymbol{V}, \boldsymbol{E}, \text{lab}, \text{val})$. The walk $v_1.\cdots.v_n$ is defined to be a *walk on path* P if for all $i \in \{1, \ldots, n\}$, $\text{lab}(v_i) = l_i$.

For the purpose of illustration, consider the walk $v_5.v_6$ in the XML tree depicted in Figure 3.3. The walk $v_5.v_6$ is a walk on path `Phone.code`, since lab$(v_5) =$ `Phone` and lab$(v_6) =$ `code`. We define next function nodes(P, T) which returns the set of nodes in an XML tree T that is reachable from the root node of T by following path P.

Definition 3.16 (Reachability of Nodes) Let $\text{T} = (\boldsymbol{V}, \boldsymbol{E}, \text{lab}, \text{val})$ be an XML tree and let $P = l_1.\cdots.l_n$ be a path. The set of nodes reachable from root(T) by following P, denoted by nodes(P, T), is defined to be

$$\begin{aligned}\text{nodes}(P, \text{T}) = \{v \in \boldsymbol{V} \mid\ & v_1.\cdots.v_n \text{ is a walk on } P \text{ in T such that} \\ & v_1 = \text{root}(\text{T}) \wedge \\ & v_n = v\}\end{aligned}$$

We will omit to explicitly state the XML tree to which function $\text{nodes}(P, \text{T})$ applies, if this XML tree is clear from context. We now illustrate function $\text{nodes}(P, \text{T})$. For this purpose, let P denote path `Customer.Phone.code` and let T be the XML tree depicted in Figure 3.3. Then $\text{nodes}(P, \text{T}) = \{v_6, v_9\}$. In contrast, if we let $P =$ `Phone.code`, then $\text{nodes}(P, \text{T})$ is the empty set, since the walks in T on path P do not start at the root node.

We note that it follows from our model of XML data that for every node v in an XML tree T there is exactly one walk from the root node in T to v and therefore $\text{nodes}(P) \cap \text{nodes}(\bar{P}) = \emptyset$ if $P \neq \bar{P}$. We therefore say that P is *the* path such that $v \in \text{nodes}(P)$.

3.2.3 Operators on XML trees

We proceed with redefining the operators on trees introduced in Section 3.1.2 for the more specific notion of XML trees. This actually means to specify the assignments of labels and values to the nodes in resulting XML trees by functions lab and val.

Definition 3.17 (XML Tree Projection) Let $\text{T} = (\boldsymbol{V}, \boldsymbol{E}, \text{lab}, \text{val})$ be an XML tree, and let $v \in \boldsymbol{V}$ be an element node. The operation of projecting T on v, denoted by $\text{T}[v]$, is defined to result in the XML tree $(\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}}, \bar{\text{lab}}, \bar{\text{val}})$, where $(\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}}) = (\boldsymbol{V}, \boldsymbol{E})[v]$ and $\bar{\text{lab}}$ and $\bar{\text{val}}$ are the restrictions of functions lab and val to $\bar{\boldsymbol{V}}$, respectively.

In order to ensure that the projection of an XML tree on a node v conforms to Definition 3.11, node v is required to be an element node. Otherwise, the root node of the resulting XML tree is not an element node, which clearly contradicts our model of XML data. The labels and values of nodes in the resulting XML tree directly correspond to the labels and values of nodes in the input XML tree.

This is also the case for the XML trees that result from applying the add operator and the subtraction operator. The definitions of these operators adopted to XML trees are given below.

Definition 3.18 (XML Tree Add) Let $\text{T} = (\boldsymbol{V}, \boldsymbol{E}, \text{lab}, \text{val})$ and $\bar{\text{T}} = (\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}}, \bar{\text{lab}}, \bar{\text{val}})$ be XML trees such that $\boldsymbol{V} \cap \bar{\boldsymbol{V}} = \emptyset$. The operation of adding an XML tree $\bar{\text{T}}$ to XML tree T, denoted by $\text{T} + \bar{\text{T}}$, is defined to result in the XML tree $(\ddot{\boldsymbol{V}}, \ddot{\boldsymbol{E}}, \ddot{\text{lab}}, \ddot{\text{val}})$, where $(\ddot{\boldsymbol{V}}, \ddot{\boldsymbol{E}}) = (\boldsymbol{V}, \boldsymbol{E}) + (\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}})$ and for all $\ddot{v} \in \ddot{V}$

$$\ddot{\text{lab}}(\ddot{v}) = \begin{cases} \text{lab}(\ddot{v}) & \ddot{v} \in \boldsymbol{V} \\ \bar{\text{lab}}(\ddot{v}) & \ddot{v} \in \bar{\boldsymbol{V}} \end{cases}$$

$$\ddot{\text{val}}(\ddot{v}) = \begin{cases} \text{val}(\ddot{v}) & \ddot{v} \in \boldsymbol{V} \\ \bar{\text{val}}(\ddot{v}) & \ddot{v} \in \bar{\boldsymbol{V}} \end{cases}$$

Definition 3.19 (Tree Subtraction) Let $\text{T} = (\boldsymbol{V}, \boldsymbol{E}, \text{lab}, \text{val})$ and $\bar{\text{T}} = (\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}}, \bar{\text{lab}}, \bar{\text{val}})$ be XML trees. The operation of subtracting $\bar{\text{T}}$ from T, denoted by $\text{T} - \bar{\text{T}}$, is defined to result in the XML tree $(\ddot{\boldsymbol{V}}, \ddot{\boldsymbol{E}}, \ddot{\text{lab}}, \ddot{\text{val}})$, where $(\ddot{\boldsymbol{V}}, \ddot{\boldsymbol{E}}) = (\boldsymbol{V}, \boldsymbol{E}) - (\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}})$ and $\ddot{\text{lab}}$ and $\ddot{\text{val}}$ are the restrictions of functions lab and val to $\ddot{\boldsymbol{V}}$, respectively.

3.3 XML Keys and XML Inclusion Dependencies

We now present the definitions of XML keys, XML inclusion dependencies and XML foreign keys in the enhanced 'closest node' approach. The syntax of enhanced 'closest node' constraints is presented in Section 3.3.1, and their semantics is defined in Section 3.3.2.

3.3.1 Defining the Syntax

The syntax of enhanced 'closest node' constraints adopts the intuitive syntactical framework of XML keys and foreign keys in XML Schema, which we call the selector/field framework. The selector is thereby used for selecting entity nodes in an XML document, and the fields are used to relate combinations of property nodes to the selected entity nodes for the purpose of value-based comparison of entity nodes.

Even though enhanced 'closest node' XML integrity constraints adopt the selector/field framework, their syntax slightly differs from the original syntax of selector/field constraints in XML Schema, which we call XSD constraints for short in the following. We now briefly outline these differences. (i) We only consider simple paths in the selectors and fields, whereas XSD constraints allow for a restricted form of XPath expressions. (ii) In contrast to enhanced 'closest node' constraints, XSD constraints allow for scoped constraints, where the constraint is only evaluated in part of the XML tree. (iii) We only allow for property nodes which are attribute or text nodes, whereas XSD constraints also allow for property nodes which are element nodes. Our reason for not considering these extensions is so that we can concentrate on the main contribution of this thesis, which is to apply different semantics to XML keys and XML inclusion dependencies so as to adequately handle multiple or absent property nodes. We now present the syntax of enhanced 'closest node' XML integrity constraints, starting with the syntax of an XML key.

Definition 3.20 (XML Key) An enhanced 'closest node' XML key (XKey) is a statement of the form

$$(S, \{F_1, \dots, F_n\}),$$

where S is a path, called selector, that ends in an element label, and $\{F_1, \dots, F_n\}$ is a non-empty set of paths, called fields, that end in attribute or text labels.

In general, the meaning of an XKey is that selected entity nodes are identified by the values in combinations of field nodes, i.e. property nodes which are related to the selected entity nodes by the fields of the XKey. For the purpose of illustration, Table 3.1 lists benchmark constraints $\mathcal{K}_1$ and $\mathcal{K}_2$ in terms of XKeys. These benchmark constraints have been introduced in Chapter 2 in order to evaluate the ability of previous approaches to XML keys to handle multiple or absent property nodes. We now repeat the meaning of benchmark constraints $\mathcal{K}_1$ and $\mathcal{K}_2$ for the convenience of the reader.

The XKey $\mathcal{K}_1$ asserts that invoices are identified by the invoice period together with the code and number of each phone for which phone charges are invoiced. In particular, the selector `Company.Invoices.Invoice` of $\mathcal{K}_1$ selects `Invoice` nodes in the XML document under consideration, and the fields {`prd`, `Line.code`, `Line.no`} relate combinations of `prd`, `code` and `no` nodes to the selected `Invoice` nodes for the purpose of value-based comparison. The XKey $\mathcal{K}_2$ asserts that customer addresses are identified by the combinations of city, street, house and apartment number. For this purpose, `Company.Customer.Addr` is the selector of $\mathcal{K}_2$, and {`city`, `strt`, `hno`, `ano`} are the fields of $\mathcal{K}_2$.

$\mathcal{K}_1$	(`Company.Invoices.Invoice`, {`prd`, `Line.code`, `Line.no`})
$\mathcal{K}_2$	(`Company.Customer.Addr`, {`city`, `strt`, `hno`, `ano`})

Table 3.1: Benchmark constraints $\mathcal{K}_1$ and $\mathcal{K}_2$ specified as XKeys.

We proceed with presenting the syntactic definition of an enhanced ‘closest node’ XML inclusion dependency, which is also the syntax for enhanced ‘closest node’ XML foreign keys.

Definition 3.21 (XML Inclusion Dependency) An enhanced ‘closest node’ XML inclusion dependency (XIND) is a statement of the form

$$(S, [F_1, \dots, F_n]) \subseteq (S', [F'_1, \dots, F'_n]),$$

where S and S' are paths, called LHS and RHS selector, that end in element labels, and $F'_1, \dots, F'_n$ and $F'_1, \dots, F'_n$ are non-empty lists of paths, called LHS and RHS fields, that end in attribute or text labels.

In general, the meaning of an XIND is that there is a subset relationship between the values in combinations of LHS field nodes related to selected LHS entity nodes, and the values in combinations of RHS field nodes related to selected RHS entity nodes. The XIND $(S, [F_1, \dots, F_n]) \subseteq (S', [F'_1, \dots, F'_n])$ is an XML foreign key (XFKey) if $(S', [F'_1, \dots, F'_n])$ is an XKey. In contrast to XKeys, which have unordered sets of fields, XINDs have *lists* of fields. The reason for XINDs to have lists of fields, as opposed to sets of fields, is so that it is clear from the positions of fields which specific pairs of an LHS field node and an RHS field node are to be compared when checking the satisfaction of an XIND in an XML tree.

For the purpose of illustration, Table 3.2 lists benchmark constraints $\mathcal{I}_1$ and $\mathcal{I}_2$ in terms of XINDs. As for benchmark constraints $\mathcal{K}_1$ and $\mathcal{K}_2$, also benchmark constraints $\mathcal{I}_1$ and $\mathcal{I}_2$ have been introduced in Chapter 2, and we now repeat their meaning.

$\mathcal{I}_1$	(`Company.Invoices.Invoice`, [`cno`, `Line.code`, `Line.no`]) $\subseteq$ (`Company.Phones.Phone`, [`cno`, `code`, `no`])
$\mathcal{I}_2$	(`Company.Invoice.Addr`, [`city`, `strt`, `hno`, `ano`]) $\subseteq$ (`Company.Customer.Addr`, [`city`, `strt`, `hno`, `ano`])

Table 3.2: Benchmark constraints $\mathcal{I}_1$ and $\mathcal{I}_2$ specified as XINDs.

The XIND $\mathcal{I}_1$ asserts that combinations of codes and numbers of phones together with the customer numbers in invoices are a subset of the combinations of codes and numbers of existing phones and the numbers of customers owning the phones. For this purpose, paths `Company.Invoices.Invoice` and `Company.Phones.Phone` are the LHS and RHS selectors of $\mathcal{I}_1$, and [`cno`, `Line.code`, `Line.no`] and [`cno`, `code`, `no`] are the lists of LHS and RHS fields of $\mathcal{I}_1$ which relate combinations of `cno`, `code`, and `no` nodes to the selected `Invoice` and `Phone` nodes. The XIND $\mathcal{I}_2$ asserts that invoice addresses are a subset of customer addresses. In particular, `Company.Invoice.Addr` and `Company.Customer.Addr` are the LHS and RHS selectors of $\mathcal{I}_2$ and [`city`, `strt`, `hno`, `ano`] is both the list of LHS and RHS fields.

3.3.2 Defining the Semantics

Dealing with multiple or absent field nodes has been identified in Chapter 1 as the primary challenge posed by the hierarchical and semi-structured nature of XML data to the design of value-based XML integrity constraints. Our focus in defining the semantics of enhanced 'closest node' XML integrity constraints is therefore on ensuring that our definitions capture the correct semantics in case that multiple or absent field nodes occur in an XML tree.

In order to adequately handle multiple field nodes it is necessary to disregard semantically incorrect combinations of field nodes in checking the satisfaction of an XKey or XIND. Our rationale for ensuring that semantically incorrect combinations of field nodes are disregarded relies on the assumption that the degree of structural coherence in a set of nodes is directly proportional to the degree of coherence in the represented information. Even though this assumption is arguable, it is by no means unusual and in fact frequently postulated in work within the context of XML data, as for example in the work on XML keyword search [43, 44]. Because of the correspondence between the structural coherence in a set of nodes on the one side and the coherence in the represented information on the other side, we argue that a combination of field nodes is semantically correct if the degree of structural coherence in the combination of nodes is maximal. Based on this rationale, semantically incorrect combinations of field nodes are disregarded in that only those combinations of field nodes are used for the value-based comparison of selected entity nodes, where the degree of structural coherence is maximal.

In order to determine whether the degree of structural coherence in a combination of field nodes is maximal, we use the *closest* property of nodes originally presented by Vincent et al. [20]. Intuitively, a pair of nodes satisfy the *closest* property if the nodes cannot be arranged more closely when taking into account the paths that lead to the nodes. Hence, if every pair of nodes in a combination of field nodes satisfy the *closest* property, then these nodes cannot be arranged more closely which means that the structural coherence in the combination of nodes is maximal and so that the combination is semantically correct. To be self-contained we now repeat the definition of the *closest* property of nodes.

Definition 3.22 (The *Closest* Property of Nodes) Let v_1 and v_2 be nodes in an XML tree $\mathrm{T} = (\boldsymbol{V}, \boldsymbol{E}, \mathrm{lab}, \mathrm{val})$. The boolean function $\mathrm{closest}(v_1, v_2)$ is defined to return true iff there exists node $v_2^1 \in \boldsymbol{V}$ such that
(i) $v_2^1 \in \text{anc-or-self}(v_1)$, and
(ii) $v_2^1 \in \text{anc-or-self}(v_2)$, and
(iii) $v_2^1 \in \text{nodes}(P_1 \cap P_2)$,
where P_1 and P_2 are the paths such that $v_1 \in \text{nodes}(P_1)$ and $v_2 \in \text{nodes}(P_2)$.

According to Definition 3.22, a pair of nodes satisfy the *closest* property, if the nodes have a common ancestor node which is reachable over the intersection of the paths that lead to the nodes. We now illustrate the *closest* property by an example, and explain how it facilitates to disregard semantically incorrect combinations of field nodes in checking XKey or XIND satisfaction.

Example 3.6 *Consider the XML tree $\mathcal{T}_1$ depicted in Figure 3.4, which has been introduced in Chapter 2 within our evaluation of previous approaches to value-based XML integrity constraints. If we want to test for instance whether the XKey $\mathcal{K}_1$ given in Table 3.1 is satisfied in XML tree $\mathcal{T}_1$, it is necessary to form combinations of* `cno`*,* `code`*,* `no` *nodes that are nested within* `Invoice` *nodes. The important point is thereby to disregard the semantically incorrect combinations (v_6, v_{11}) and (v_7, v_{10}) of a* `code` *node and a* `no` *node nested within* `Invoice` *node v_3. This is achieved by requiring the nodes in a combination of field nodes to pairwise satisfy the* closest *property, as we now illustrate. The* `code` *nodes $\{v_6, v_{10}\}$ are reachable over path* `Company.Invoices.Invoice.Line.code` *in XML tree $\mathcal{T}_1$ as well as the* `no` *nodes $\{v_7, v_{11}\}$ are reachable over path* `Company.Invoices.Invoice.Line.no`*. Hence,* `Company.Invoices.Invoice.Line.code` $\cap$ `Company.Invoices.Invoice.Line.no` $=$ `Company.Invoices.Invoice.Line` *and therefore* closest(v_6, v_{11}) = closest(v_7, v_{10}) = false*, since neither nodes v_6 and v_{11} nor nodes v_7 and v_{10} have a common* anc-or-self *node reachable over path* `Company.Invoices.Invoice.Line`*. In contrast, the semantically correct combinations (v_6, v_7) and (v_{10}, v_{11}) of a* `code` *node and a* `no` *node satisfy the* closest *property, since $\{v_5, v_9\} \subseteq$* nodes(`Company.Invoices.Invoice.Line`) *and $v_5 \in$* anc-or-self(v_6) $\cap$ anc-or-self(v_7) *as well as $v_9 \in$* anc-or-self(v_{10}) $\cap$ anc-or-self(v_{11})*.*

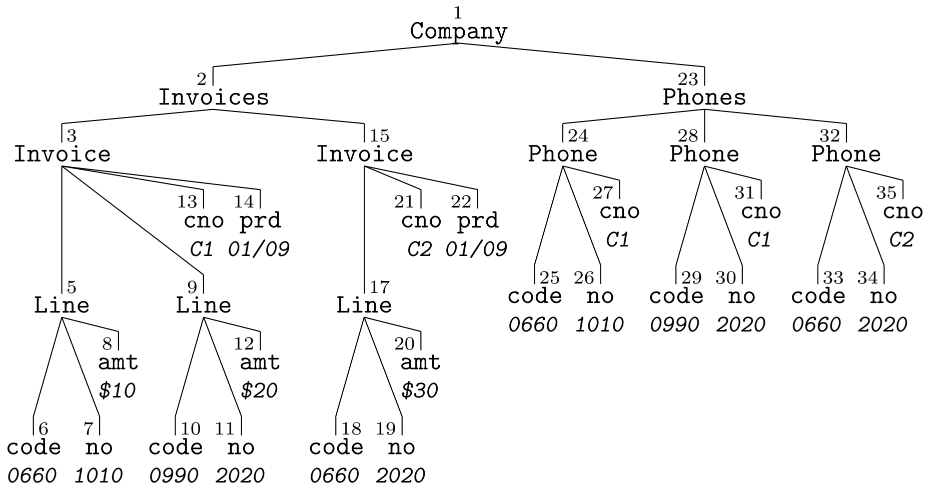


Figure 3.4: XML tree representing invoices and phones.

In order to adequately handle absent field nodes, it is necessary to compare selected entity nodes not only on the basis of values in complete combinations of field nodes, but also on the basis of values in incomplete combinations of field nodes. To compare selected entity nodes also on the basis of values in incomplete combinations of field nodes addresses XML documents where:

(i) information is complete and absent field nodes therefore reflect some degree of heterogeneity in real world entities, or

(ii) information is incomplete and absent field nodes are interpreted in terms of the *no information* interpretation as recommended in the XML specification by the W3C.

In both situations, it is necessary to check XKeys and XINDs solely on the basis of values in those combinations of field nodes that are present in an XML document, since using a completion of the XML document for this purpose leads to counter intuitive results, as has been illustrated in Example 1.10. In order to enable the comparison of entity nodes by the values in incomplete combinations of field nodes, we propose the novel concept of *maximum combinations of field nodes*. Intuitively, every semantically correct combination of field nodes that is maximal with respect to the number of nodes, is a maximum combination of field nodes. We now make this idea more precise.

Definition 3.23 (Maximum Combination of Field Nodes) Let S be a selector and let $\{F_1, \dots, F_n\}$ be a set of fields. Also, let T be an XML tree and let $v \in \text{nodes}(S, \text{T})$ be a selector node. A set of nodes $\{v_1, \dots, v_m\}$ in XML tree T is defined to be a *maximum combination of field nodes* for selector node v with respect to the fields $\{F_1, \dots, F_n\}$ if
(i) $\forall \{i, j\} \subseteq \{1, \dots, m\}$, $\text{closest}(v_i, v_j) = \text{true}$;
(ii) $\forall i \in \{1, \dots, m\}$, $v \in \text{ancestor}(v_i)$ and $v_i \in \text{nodes}(S.F_j)$ for some $j \in \{1, \dots, n\}$;
(iii) there does not exist node $\bar{v} \in \text{desc}(v)$ such that $\forall i \in \{1, \dots, m\}$, $\bar{v} \neq v_i$ and $\text{closest}(\bar{v}, v_i) = \text{true}$, and $\bar{v} \in \text{nodes}(S.F_j)$ for some $j \in \{1, \dots, n\}$.

A couple of comments are appropriate. To ensure that a maximum combination of field nodes is semantically correct, (i) in Definition 3.23 requires that the field nodes pairwise satisfy the *closest* property. Further, a maximum combination of field nodes is related to a selected entity node by the fields in an XKey or XIND, as precisely stated by (ii) in Definition 3.23. Finally, (iii) in Definition 3.23 requires the maximality of a combination of field nodes w.r.t. the number of nodes. Intuitively, (iii) in Definition 3.23 requires a maximum combination of field nodes to be no strict subset of any other semantically correct combination of field nodes. We now illustrate Definition 3.23 by an example.

Example 3.7 *Suppose that we want to check the satisfaction of XKey $\mathcal{K}_2$ (cf. Table 3.1) in XML tree $\mathcal{T}_3$ (cf. Figure 3.5a). Then, it is necessary to compare* `Addr` *nodes on the basis of the values in combinations of* `city`*,* `strt`*,* `hno`*, and* `ano` *nodes. Whereas the address represented by node v_9 is complete, the apartment number is absent for the address represented by node v_3. It is thus necessary to compare v_9 and v_3 on the basis of values in incomplete combinations of* `city`*,* `strt`*,* `hno`*,* `ano` *nodes. Thereby, w.r.t. the selected* `Addr` *node v_3 and the fields* {`city`, `strt`, `hno`, `ano`} *of XKey $\mathcal{K}_2$, nodes (v_4, v_5, v_6) are a maximum combination of field nodes since (i) these nodes pairwise satisfy the* closest *property, (ii) node v_3 is a common ancestor of nodes $\{v_4, v_5, v_6\}$ which are reachable over paths*

`Company.Customer.Addr.city`, `Company.Customer.Addr.strt`, `Company.Customer.Addr.hno`, *and (iii) nodes* $\{v_4, v_5, v_6\}$ *are no strict subset of any other semantically correct combination of field nodes. For instance, the superset* $\{v_4, v_5, v_6, v_{13}\}$ *is not a maximum combination of field nodes for* v_3 *w.r.t. the fields in XKey* $\mathcal{K}_2$ *for two reasons. First,* v_{13} *does not satisfy the* closest *property in combination with any node in* $\{v_4, v_5, v_6\}$ *and thus (i) in Definition 3.23 is violated. Second,* v_3 *is not an ancestor of node* v_{13} *and thus also (ii) in Definition 3.23 is violated. The set of nodes* $\{v_4, v_5\}$ *is also not a maximum combination of field nodes for* v_3 *w.r.t. the fields in XKey* $\mathcal{K}_2$*, since even though* v_4 *and* v_5 *satisfy (i) and (ii) in Definition 3.23, they violate (iii) for the obvious reason that* $\{v_4, v_5\} \subset \{v_4, v_5, v_6\}$*.*

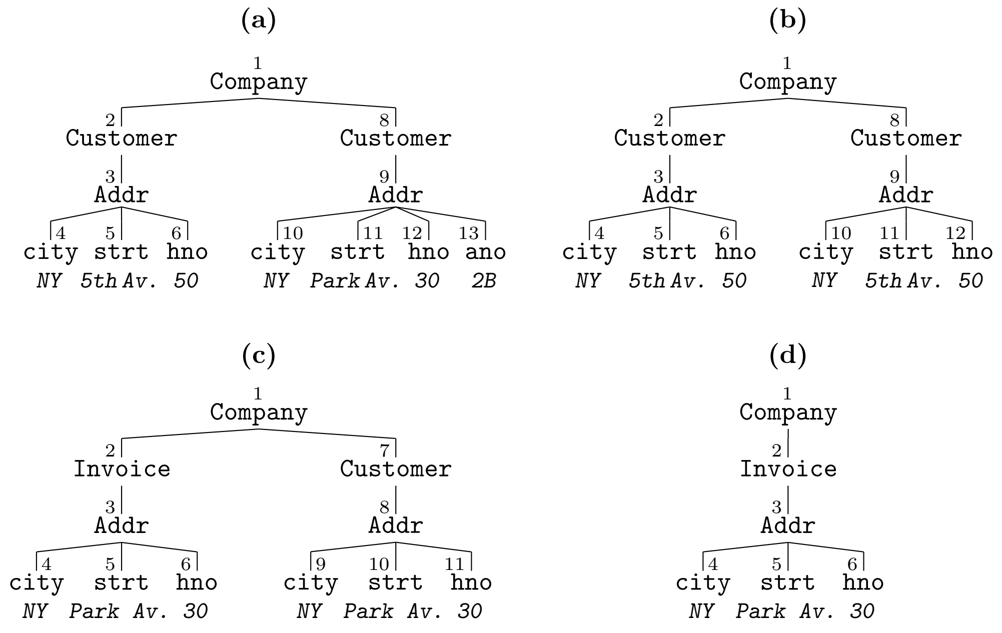


Figure 3.5: XML trees representing customer addresses and invoice addresses.

We are now ready to present our definitions of the semantics of XKeys and XINDs based on maximum combinations of field nodes. We start with defining the semantics of an XKey.

Definition 3.24 (XKey Semantics) Let $\sigma = (S, \{F_1, \ldots, F_n\})$ be an XKey and let T be an XML tree. The XML tree T satisfies σ, denoted by $\mathrm{T} \vDash \sigma$, if whenever there exist nodes $v, v' \in \text{nodes}(S, \mathrm{T})$, and, with respect to $\{F_1, \ldots, F_n\}$, there exist maximum combinations of field nodes $\{v_1, \ldots, v_m\}$ for selector node v, and $\{v'_1, \ldots, v'_m\}$ for selector node v', such that $\forall i \in \{1, \ldots, m\}$
(i) $\text{val}(v_i) = \text{val}(v'_i)$, and
(ii) there exists $j \in \{1, \ldots, n\}$ such that $\{v_i, v'_i\} \in \text{nodes}(S.F_j)$,
then $\forall i \in \{1, \ldots, m\}$, $v_i = v'_i$.

Intuitively, an XKey σ is satisfied in an XML tree, if there do not exist two distinct maximum combinations of field nodes that contain the same number of nodes and are value equal. Requirement (ii) in Definition 3.24 thereby ensures that only the values of field nodes that belong to the same field are compared, and so prevents comparing apples and oranges. Our definition of an XKey clearly captures the uniqueness property of a key, since if $\mathrm{T} \models \sigma$ then there cannot exist two distinct maximum combinations of field nodes for some selected entity nodes with respect to the fields in σ. Also, our definition captures the identification property of a key, i.e. the identification of selected entity nodes by maximum combinations of related field nodes. That is, if $\mathrm{T} \models \sigma$ and v, v' are selected entity nodes for σ, then $v = v'$ if there exist maximum combinations of field nodes $v_1, \ldots, v_m$ for v and $v'_1, \ldots, v'_m$ for v' with respect to the fields in σ such that (i) and (ii) in Definition 3.24 are satisfied. This is because if $v \neq v'$, then since T is a tree a field node cannot be a descendant of both v and v' and hence the maximum combinations of field nodes $v_1, \ldots, v_m$ and $v'_1, \ldots, v'_m$ must be distinct, which is a contradiction and so $v = v'$.

We now illustrate the semantics of an XKey using the test cases developed in Chapter 2 for evaluating the ability of previous approaches to XML keys to deal with multiple or absent field nodes.

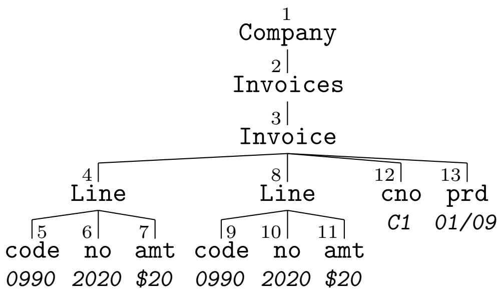


Figure 3.6: XML tree representing an invoice where phone charges are invoiced for the same phone twice.

Example 3.8 (XKey semantics) *Test case $\mathcal{C}1$ (cf. Table 2.2) addresses the handling of multiple field nodes and stipulates for this purpose that benchmark XML tree $\mathcal{T}_1$ depicted in Figure 3.4 is validated against benchmark constraint $\mathcal{K}_1$ which is given in Table 3.1 in terms of an XKey. According to Definition 3.24, $\mathcal{K}_1$ is satisfied in XML tree $\mathcal{T}_1$ for the following reason. The nodes v_3 and v_{15} are the entity nodes selected by XKey $\mathcal{K}_1$ in XML tree $\mathcal{T}_1$, and the only maximum combinations of field nodes are $\{(v_6, v_7, v_{14}), (v_{10}, v_{11}, v_{14})\}$ for v_3 and $\{(v_{18}, v_{19}, v_{22})\}$ for v_{15}. These maximum combinations of field nodes are obviously unique, since none of them are value equal, and therefore XML tree $\mathcal{T}_1$ satisfies $\mathcal{K}_1$ as desired. Because semantically incorrect combinations of field nodes are disregarded, our definition of an XKey passes test case $\mathcal{C}1$.*

Test case $\mathcal{C}3$ addresses the ability of an approach to XML keys to ensure the uniqueness of field nodes and stipulates for this purpose that benchmark XML tree $\mathcal{T}_2$ depicted in Figure 3.6 is validated against benchmark constraint $\mathcal{K}_1$ which is given in Table 3.1 in terms of *an XKey. The node v_3 is the single entity node selected by XKey $\mathcal{K}_1$ in XML tree $\mathcal{T}_2$, and the maximum combinations of field nodes for v_3 are (v_5, v_6, v_{13}) and (v_9, v_{10}, v_{13}). Because these maximum combinations of field nodes are value equal but distinct, XML tree $\mathcal{T}_2$ violates XKey $\mathcal{K}_1$ as desired, and so our definition of an XKey also passes test case $\mathcal{C}3$.*

Test cases $\mathcal{C}4$ and $\mathcal{C}5$ address the handling of absent field nodes and stipulate for this purpose that benchmark XML trees $\mathcal{T}_3$ and $\mathcal{T}_4$, which are depicted in Figures 3.5a and 3.5b, are validated against benchmark constraint $\mathcal{K}_2$ which is given in Table 3.1 in terms of an XKey. Regarding XML tree $\mathcal{T}_3$, nodes v_3 and v_9 are the entity nodes selected by XKey $\mathcal{K}_2$, and the maximum combinations of field nodes are (v_4, v_5, v_6) for v_3 and $(v_{10}, v_{11}, v_{12}, v_{13})$ for v_9. These maximum combinations of field nodes are obviously unique, since they do not contain the same number of nodes, and therefore XML tree $\mathcal{T}_3$ satisfies $\mathcal{K}_2$ as desired. Regarding XML tree $\mathcal{T}_4$, also nodes v_3 and v_9 are the entity nodes selected by XKey $\mathcal{K}_2$. However, the maximum combinations of field nodes are (v_4, v_5, v_6) for v_3 and (v_{10}, v_{11}, v_{12}) for v_9. Because these two maximum combinations of field nodes contain the same number of nodes, and are moreover value equal, XML tree $\mathcal{T}_4$ violates XKey $\mathcal{K}_2$, which is again desired. Because also incomplete combinations of field nodes are used for the comparison of entity nodes, our definition of an XKey also passes test cases $\mathcal{C}4$ and $\mathcal{C}5$.

We define next the semantics of an XIND.

Definition 3.25 (XIND Semantics) Let $\sigma = (S, [F_1, \ldots, F_n]) \subset (S', [F'_1, \ldots, F'_n])$ be an XIND and let T be an XML tree. The XML tree T satisfies σ, denoted by $\mathrm{T} \vDash \sigma$, if whenever there exists LHS selector node $v \in \text{nodes}(S, \mathrm{T})$ and a maximum combination of LHS field nodes $\{v_1, \ldots, v_m\}$ for v with respect to $[F_1, \ldots, F_n]$, then there exists RHS selector node $v' \in \text{nodes}(S', \mathrm{T})$ and a maximum combination of RHS field nodes $\{v'_1, \ldots, v'_m\}$ for v' with respect to $[F'_1, \ldots, F'_n]$, such that $\forall i \in \{1, \ldots, m\}$,
(i) $\text{val}(v_i) = \text{val}(v'_i)$ and
(ii) if $v_i \in \text{nodes}(S.F_j)$, where $j \in \{1, \ldots, n\}$, then $v'_i \in \text{nodes}(S'.F'_j)$.

Intuitively, an XIND is satisfied in an XML tree if the maximum combinations of field nodes for selected LHS nodes have value equal counterparts in maximum combinations of field nodes for selected RHS nodes. In analogy to the definition of the semantics of an XKey, requirement (ii) in Definition 3.25 ensures that only the values of field nodes that belong to corresponding LHS and RHS fields are compared. We now illustrate the semantics of an XIND. We use for this purpose again the test cases developed in Chapter 2 for evaluating the ability of previous approaches to XML inclusion dependencies to deal with multiple or absent field nodes.

Example 3.9 (XIND semantics) *Test case $\mathcal{C}2$ (cf. Table 2.2) addresses the handling of multiple field nodes and stipulates for this purpose that benchmark XML tree $\mathcal{T}_1$ depicted in Figure 3.4 is validated against benchmark constraint $\mathcal{I}_1$ which is given in Table 3.2 in terms of an XIND. According to Definition 3.25, the XIND $\mathcal{I}_1$ is satisfied in XML tree $\mathcal{T}_1$ as desired. This is because the only maximum combinations of field nodes for the selected LHS nodes v_3 and v_{15} are $\{(v_6, v_7, v_{13}), (v_{10}, v_{11}, v_{13})\}$ and $\{(v_{18}, v_{19}, v_{21})\}$, respectively, and these three combinations of field nodes are value equal to the maximum combinations of field* *nodes* (v_{25}, v_{26}, v_{27})*,* (v_{29}, v_{30}, v_{31}) *and* (v_{33}, v_{34}, v_{35}) *for the selected RHS nodes* v_{24}*,* v_{28} *and* v_{32}*, respectively. As for our XKeys, also an XIND disregards semantically incorrect combinations of field nodes, and so our definition of an XIND passes test case* $\mathcal{C}2$*.*

Test cases $\mathcal{C}6$ *and* $\mathcal{C}7$ *address the handling of absent field nodes and stipulate for this purpose that benchmark XML trees* $\mathcal{T}_5$ *and* $\mathcal{T}_6$*, which are depicted in Figures 3.5c and 3.5d, are validated against benchmark constraint* $\mathcal{I}_2$ *which is given in Table 3.2 in terms of an XIND. Regarding XML tree* $\mathcal{T}_5$*, the single maximum combination of field nodes* (v_4, v_5, v_6) *for the selected LHS entity node* v_3 *is value equal to the maximum combination of field nodes* (v_9, v_{10}, v_{11}) *for the selected RHS entity node* v_8*, and therefore XML tree* $\mathcal{T}_5$ *satisfies* $\mathcal{I}_2$ *as desired. Regarding XML tree* $\mathcal{T}_6$*, the single maximum combination of field nodes* (v_4, v_5, v_6) *obviously does not have a value equal counterpart in maximum combinations of field nodes for selected RHS nodes, simply because there is no node in XML tree* $\mathcal{T}_6$ *which is reachable over RHS selector path* `Company.Customer.Addr`*. Hence, XML tree* $\mathcal{T}_6$ *violates XIND* $\mathcal{I}_2$*. Because also incomplete combinations of field nodes are taken into account when testing the satisfaction of an XIND in an XML tree, our definition of an XIND also passes test cases* $\mathcal{C}6$ *and* $\mathcal{C}7$*.*

We conclude this chapter with the observation that our definitions XKeys and XINDs pass all test cases in the benchmark test developed in Chapter 2, which is in contrast to any other approach to value-based XML integrity constraints.

Chapter 4

Preserving Relational Semantics

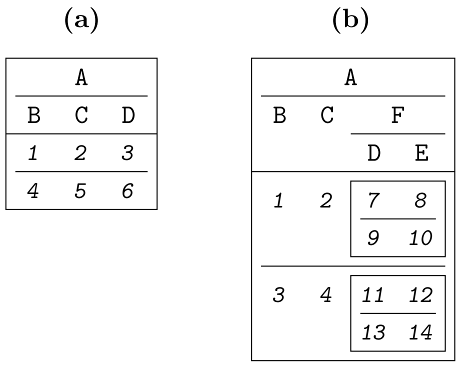
Contents

This chapter presents the procedure for transforming flat relational databases to XML, and also the major results on the preservation of relational data semantics by XKeys and XINDs. For this purpose, Section 4.1 presents our model of relational data, and introduces the fundamental operations of nesting and unnesting relations. Section 4.2 is devoted to the transformation of relational data to XML and presents the particular algorithms used in our transformation procedure. Finally, Section 4.3 reviews relational keys and inclusion dependencies and shows how to automatically derive XKeys and XINDs that preserve original data semantics.

4.1 A Model of Relational Data

Preliminary to the presentation of our model of relational data, we informally introduce the primary data structure used in the relational data model, i.e. a relation. Intuitively, a relation can be thought of as being a table, where the rows contain the actual data and the table header determines the structure of the data. The table header gives the name of the table together with the names of its columns, and determines the structure of the data, in that each row in the table is required to consist of one cell per column name.

In the *flat relational model*, a cell can contain only an atomic value such as a string or an integer. The restriction that a cell can contain only an atomic value is relaxed in the *nested relational model*, where a cell can contain either an atomic value or an entire table. The structure of tables that are nested within cells of a column in a table is determined by means of a nested table header that replaces the name of the column. We now illustrate flat and nested tables by an example.



(a)

A		
B	C	D
1	*2*	*3*
4	*5*	*6*

(b)

A			
B	C	F	
		D	E
1	*2*	*7*	*8*
		9	*10*
3	*4*	*11*	*12*
		13	*14*

Figure 4.1: A flat table and a nested table.

Example 4.1 (flat and nested table) *Figure 4.1a depicts a flat table named* `A` *with three columns named* `B`, `C` *and* `D`. *Table* `A` *contains two rows, each having exactly one cell per column name. Note that the values in the cells of table* `A` *are atomic values. Figure 4.1b again depicts a table named* `A` *with columns* `B`, `C` *and* `F`, *where column* `F` *is a table header on its own. The structure of tables nested inside* `F`*-cells is determined by the column names in* `F`, *i.e.* `D` *and* `E`. *Note that the values in columns* `B` *and* `C` *are again atomic, whereas the values in column* `F` *are tables.*

4.1.1 Relations: The Primary Data Structure

We now present our model of relational data, starting with the definition of a relational schema which is the formalization of a table header. Motivated by the fact that nested relations strictly generalize flat relations, we only model the more general case of nested relations explicitly. For this purpose, we first determine a set of allowed attributes, i.e. column names.

Definition 4.1 (Attribute) An *attribute* A is defined to be a member of the fixed, countably infinite set of attributes $\boldsymbol{A} = \boldsymbol{L}^E \cup \boldsymbol{L}^A$. Attribute A is said to be a *flat attribute* if $A \in \boldsymbol{L}^A$, and A is said to be a *nested attribute* otherwise.

Our reason for distinguishing flat attributes from nested attributes is to distinguish columns where the cells contain atomic values from those columns where the cells contain tables. For reasons which will be made clearer soon, we reuse the sets of attribute labels and element labels as the fixed sets of flat attributes and nested attributes, respectively, which we defined in our tree model of XML data in Section 3.2.

There is a variety of proposals in literature for modeling nested relational schemas, that range from modeling nested relational schemas as recursively nested sets [112] to modeling nested relational schemas as expressions conforming to some grammar [113]. In [114] a nested relational schema is modeled as a tree of attributes, which is particularly suitable for our purpose of mapping relations to XML trees. We therefore adopt the approach in [114] and define a relational schema as follows:

Definition 4.2 (Relational schema) A *relational schema* S is defined to be

$$\mathrm{S} = (\boldsymbol{V}, \boldsymbol{E}, \leq), \text{ where}$$

- $\boldsymbol{V} \subset \boldsymbol{A}$ is a finite set of attributes, and
- $(\boldsymbol{V}, \boldsymbol{E}, \leq)$ is an ordered tree in terms of Definition 3.6 such that $\forall (A, A') \in \boldsymbol{E}$, $A \in \boldsymbol{L}^E$.

Relational schema S is said to be a *flat relational schema* if for all $A \in \boldsymbol{V} - \{\mathrm{root(S)}\}$, $A \in \boldsymbol{L}^A$, and S is said to be a *nested relational schema*, otherwise.

A couple of comments are appropriate. A relational schema does not have an explicit name in our model of relational data. The name of a relational schema is however given by its root attribute. In particular, a relational schema is an *ordered* tree, and so the attributes in a relational schema are ordered from left to right. As for ordered trees, we will omit to explicitly write down ordering function $\leq$ in a relational schema unless we need to refer to the ordering of attributes explicitly. Next, because a relational schema is a tree, operators 'project', 'add', and 'subtract' defined in Section 3.1.2 are applicable to relational schemas. Clearly, if a relational schema S is projected on an attribute A, then A must be a nested attribute so that the resulting relational schema $\mathrm{S}[A]$ satisfies the requirement on edges in Definition 4.2. This requirement intuitively means that a flat attribute must not be nested inside another flat attribute. Also, since attributes in a relational schema are taken from the sets of element labels $\boldsymbol{L}^E$ and attribute labels $\boldsymbol{L}^A$, the requirement on edges in Definition 4.2 ensures that a walk in a relational schema is a path in terms of Definition 3.12. We now illustrate the notion of a relational schema.

Example 4.2 (relational schema) *The relational schema depicted in Figure 4.2a corresponds to the table header depicted in Figure 4.1a. The root attribute* `A` *indicates the relation name, and attributes* `B`, `C` *and* `D` *correspond to the column names of the table header depicted in Figure 4.1a. In particular,* `A` $\in \boldsymbol{L}^E$ *is a nested attribute and* $\{$`B`, `C`, `D`$\} \subseteq \boldsymbol{L}^A$ *are flat* *attributes. If we denote this relational schema by* S = $(\boldsymbol{V}, \boldsymbol{E})$*, then* $\boldsymbol{V} = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$*,* $\boldsymbol{E} = \{(\texttt{A},\texttt{B}), (\texttt{A},\texttt{C}), (\texttt{A},\texttt{D})\}$ *and* root(S) = `A`*. Also, the walk* `A.C` *in* S *is a path in terms of Definition 3.12, and* S *is a flat relational schema since root attribute* `A` *is the only nested attribute in* S*. In contrast, the relational schema depicted in Figure 4.2b, which corresponds to the table header depicted in Figure 4.1b, is a nested relational schema.*

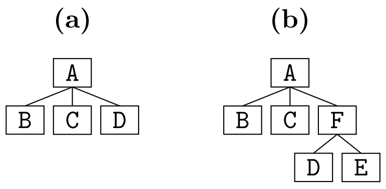


Figure 4.2: A flat relational schema and a nested relational schema.

The attributes in a relational schema that are children of the root attribute determine the overall structure of a relation. We call these child attributes the principal attributes of a relational schema in accordance with the terminology used in Section 3.2. We now make the notion of principal attributes more precise.

Definition 4.3 (Principal Attribute) The set of *principal attributes* in a relational schema $\mathrm{S} = (\boldsymbol{V}, \boldsymbol{E})$, denoted by att(S), is defined to be

$$\mathrm{att}(\mathrm{S}) = \mathrm{child}(\mathrm{root}(\mathrm{S})).$$

We define next a relation over a relational schema. For this purpose, we first define the domain of an attribute, which is the set of permitted values for an attribute in a relation.

Definition 4.4 (Domain and Tuple) Let S be a relational schema, and let A be an attribute in S. The *domain* of A, denoted by $\mathrm{dom}(A)$, is defined to be

$$\mathrm{dom}(A) = \begin{cases} \text{fixed subset of } \boldsymbol{U} & A \text{ is flat} \\ \mathcal{P}\Big(\{t\colon \{A_1, \ldots, A_n\} \to \{\mathrm{dom}(A_1) \cup \cdots \cup \mathrm{dom}(A_n)\} \mid & A \text{ is nested} \\ \quad \forall i \in \{1, \ldots, n\}, t(A_i) \in \mathrm{dom}(A_i)\}\Big) & \end{cases}$$

where $\{A_1, \ldots, A_n\} = \mathrm{att}(S[A])$ and mapping function t is said to be a *tuple* over S$[A]$.

Intuitively, the domain of a flat attribute is a primitive data type associated to the attribute. In Definition 4.4, the domain of a flat attribute is a subset of the set of values $\boldsymbol{U}$, which we have introduced in defining our model of XML data in Section 3.2.

As has been illustrated at the beginning of this section, the value of a nested attribute is an entire relation. Translated to the notion of a table, this means that the value of a nested attribute A is a set of rows. The rows in a table are called tuples in Definition 4.4, and the structure of a tuple is determined by the structure of the nested relational schema S[A], which is the relational schema obtained from projecting the overall relational schema S on A. To be more precise, a tuple is a mapping that assigns to every principal attribute $A_i \in$ att(S[A]) a value from the domain of A_i. The domain of a nested attribute is then defined to be the set of all sets of tuples over the relational schema S[A]. Because Definition 4.4 uses the powerset constructor to determine the set of possible sets of tuples over S[A], the empty set of tuples is a value in the domain of attribute A. Also, the domain of a nested attribute A is recursively defined in case that a principal attribute $A_i \in$ att(S[A]) is a nested attribute. This recursion is however always finite, since an attribute occurs exactly once in a relational schema according to Definition 4.2. We now illustrate the domains of flat and nested attributes by an example.

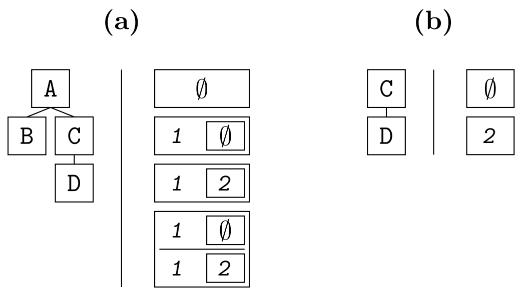


Figure 4.3: The domain of a nested attribute.

Example 4.3 (domain and tuple) *Let* S *be the relational schema depicted in the left of Figure 4.3a, and let* dom(B) = {*1*} *and* dom(D) = {*2*} *be the domains of flat attributes* B *and* D*. Then, the domain of nested attribute* A *is the set of all sets of tuples over relational schema* S[A]*, where* S[A] = S *since* A = root(S)*. In particular,* dom(A) *is the set of all sets of mappings from the principal attributes* B *and* C *of* S[A] *to the domains* dom(B) *and* dom(C)*, respectively. Since,* C *is again a nested attribute, the domain of* A *is recursively defined, and we now illustrate how the domain of* C *is constructed. According to Definition 4.4, the domain of* C *is the set of all sets of tuples over relational schema* S[C]*, which is depicted in the left of Figure 4.3b. Since* D *is the only principal attribute in* S[C]*, and since* dom(D) = {*2*}*, there is only one possible mapping from* D *to* dom(D)*. If we denote this tuple by* t_1 *then* t_1(D) = *2, and in common linear syntax* t_1 *is of the form* $t_1 = \langle 2 \rangle$*. The domain of* C *is then given by* $\mathcal{P}(t_1)$*. Hence* dom(C) = $\{\emptyset, \{\langle 2 \rangle\}\}$*, as depicted in the right of Figure 4.3b. We note that* $\emptyset$ *implicitly results from applying the powerset constructor. Given that* dom(C) = $\{\emptyset, \{\langle 2 \rangle\}\}$ *and that* dom(B) = {*1*}*, there are two possible tuples over* S[A]*. If we denote these tuples by* t_2 *and* t_3*, then* $t_2 = \langle 1, \emptyset \rangle$ *and* $t_3 = \langle 1, \{\langle 2 \rangle\} \rangle$*. Hence, the domain of attribute* A*, depicted in the right of Figure 4.3a, is finally given by* dom(A) = $\mathcal{P}(t_2, t_3)$ =$\{\emptyset,$ $\{\langle 1, \emptyset \rangle\}$, $\{\langle 1, \{\langle 2 \rangle\} \rangle\}$, $\{\langle 1, \emptyset \rangle, \langle 1, \{\langle 2 \rangle\} \rangle\}\}$.

We define next a relation over a relational schema as a finite set of tuples.

Definition 4.5 (Relation) Let S be a relational schema. A *relation* R over S is defined to be a finite set of tuples over S. The relation R is defined to be a *flat relation* if S is a flat relational schema, and R is defined to be a *nested relation* otherwise.

A couple of comments are appropriate. First, Definition 4.5 reflects the fact that the value of a nested attribute is a relation and that a relation R over a relational schema S is equally a set of tuples over $\mathrm{S} = (\boldsymbol{V}, \boldsymbol{E})$ and also a single value in the domain dom(root(S)). Second, according to Definition 4.4, a tuple assigns a value to every attribute in a relational schema, and therefore incomplete tuples are prohibited. As a consequence, our model of relational data only allows for complete relations.

We finally extend the definition of a single relation over a relational schema to a set of relations, i.e. a database, over a set of relational schemas, i.e. a database schema.

Definition 4.6 (Database and Database Schema) A *database schema* **S** is defined to be $\mathbf{S} = \{\mathrm{S}_1, \ldots, \mathrm{S}_n\}$, where $\forall i \in \{1, \ldots, n\}$, S_i is a relational schema. A *database* **R** over **S** is defined to be $\mathbf{R} = \{\mathrm{R}_1, \ldots, \mathrm{R}_n\}$, where $\forall i \in \{1, \ldots, n\}$, R_i is a relation over S_i.

4.1.2 Operators on Relations

We now introduce the most fundamental operators in the context of nested relations, i.e. the nest and unnest operator. We will use these operators in defining our procedure for transforming relational data to XML data. To define the nest and unnest operators, we require the well known projection operator, which we define first for a single tuple.

Definition 4.7 (Projection of a Tuple) Let t be a tuple over a relational schema S, and let $\{A_1, \ldots, A_n\} \subseteq \mathrm{att(S)}$ be a set of attributes. The *projection* of t on $\{A_1, \ldots, A_n\}$, denoted by $t[A_1, \ldots, A_n]$, is defined to result in tuple $\bar{t}$ over relational schema $\bar{\mathrm{S}}$ where

$$\begin{aligned} \bar{\mathrm{S}} &= \mathrm{S} - \mathrm{S}[A_i] \text{ for all } A_i \in \{\mathrm{att(S)} - \{A_1, \ldots, A_n\}\} \\ \bar{t} &= \text{restriction of } t \text{ to } \{A_1, \ldots, A_n\}. \end{aligned}$$

According to Definition 4.4, a tuple t over a relational schema S is a function that maps the principal attributes in S to values in the corresponding domains. Intuitively, to project a tuple t on a set of attributes $\{A_1, \ldots, A_n\}$ means to remove those attribute-value assignments from t, where the attribute is not in the projected set $\{A_1, \ldots, A_n\}$. This operation is commonly known as the restriction of a function, which is therefore used in Definition 4.7 for specifying the projection of a tuple on a set of attributes. A consequence of using the restriction of tuple in defining the projection operator is that the projection of a tuple t over relational schema S is only defined with respect to the principal attributes of S. Definition 4.7 therefore requires attributes $\{A_1, \ldots, A_n\}$ to be a subset of the principal attributes of S, i.e. $\{A_1, \ldots, A_n\} \subseteq \mathrm{att(S)}$ in Definition 4.7. Further, if $\bar{t}$ is the tuple that results from projecting a given tuple t over a relation schema S on attributes $\{A_1, \ldots, A_n\}$, then $\bar{t}$ is a tuple over the relational schema $\bar{\mathrm{S}}$ having $\{A_1, \ldots, A_n\}$ as principal attributes, where $\bar{\mathrm{S}}$ differs from S if $\{A_1, \ldots, A_n\} \subset \mathrm{att(S)}$. We now illustrate the projection of a tuple.

Example 4.4 (projection of a tuple) *Let* S *be the relational schema depicted in Figure 4.4a, and consider the tuple* $t = \langle 7, 8, \{\langle 9, 10\rangle\}\rangle$ *over* S. *Then, the projection of* t *on attributes* B *and* C *is legal, since* B *and* C *are principal attributes of* S. *If we let* $\bar{t}$ *be the tuple that results from* $t[\mathtt{B},\mathtt{C}]$*, then* $\bar{t}$ *is a tuple over the relational schema* $\bar{\mathrm{S}}$ *depicted in Figure 4.4b. Note that* $\bar{\mathrm{S}} = \mathrm{S} - \mathrm{S}[\mathtt{D}]$. *Therefore* B *and* C *are the only principal attributes in* $\bar{\mathrm{S}}$. *Since* $\bar{t}$ *is the restriction of* t *to* $\{\mathtt{B},\mathtt{C}\}$ *according to Definition 4.7, tuple* $\bar{t}$ *results from removing the assignment of attribute* D *to value* $\{\langle 9, 10\rangle\}$ *in* t. *Hence,* $\bar{t} = t[\mathtt{B},\mathtt{C}] = \langle 7, 8\rangle$. *In contrast to the projection of* t *on attributes* B *and* C*, the projection of* t *on attribute* E *is invalid, since* $\mathtt{E} \notin \mathrm{att}(\mathrm{S})$. *Note that tuple* t *does not assign a value to attribute* E.

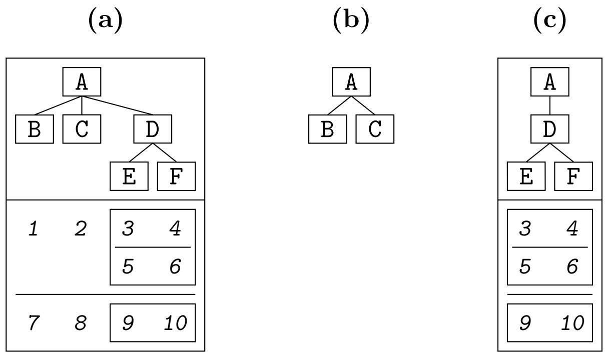


Figure 4.4: Projections of Tuples and Relations.

We now generalize the projection operator, so that it is applicable to a set of tuples. That is, we generalize the projection on a set of attributes from a single tuple to a relation.

Definition 4.8 (Projection of a Relation) Let R be a relation over a relational schema S, and let $\{A_1, \ldots, A_n\} \subseteq \mathrm{att}(\mathrm{S})$ be a set of principal attributes. The *projection* of relation R on attributes $A_1, \ldots, A_n$, denoted by $\mathrm{R}[A_1, \ldots, A_n]$, is defined to be the relation $\bar{\mathrm{R}}$ over the relational schema $\bar{\mathrm{S}} = (\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}})$ given by

$$\begin{aligned}\bar{\mathrm{S}} &= \mathrm{S} - \mathrm{S}[A_i] \text{ for all } A_i \in \{\mathrm{att}(S) - \{A_1, \ldots, A_n\}\} \\ \bar{\mathrm{R}} &= \{\bar{t} \mid \exists t \in \mathrm{R} \text{ such that } \bar{t} = t[A_1, \ldots, A_n]\}\end{aligned}$$

Intuitively, projecting a relation R over a relational schema S on a set of principal attributes $\{A_1, \ldots, A_n\}$ results in a relation $\bar{\mathrm{R}}$ which contains exactly the tuples in R when they are projected on $\{A_1, \ldots, A_n\}$. Hence, analogously to the projection of a tuple, also the projection of a relation R results in a relation $\bar{\mathrm{R}}$ over a new relational schema $\bar{\mathrm{S}}$ which has $\{A_1, \ldots, A_n\}$ as principal attributes. We now illustrate the projection of a relation.

Example 4.5 (projection of a relation) *Let* S *be the relational schema depicted in the top of Figure 4.4a, and let* R *be the relation over* S *depicted in the bottom of Figure 4.4a. Also, let* t_1 *and* t_2 *be the two tuples in* R. *Then, the projection of* R *on attribute* D *is legal* *since* $\mathtt{D} \in \mathrm{att}(\mathrm{S})$*. This projection results in relation* $\bar{\mathrm{R}}$ *over relational schema* $\bar{\mathrm{S}}$*, which are both depicted in Figure 4.4c. In particular, relation* $\bar{\mathrm{R}}$ *consists of tuples* t_1 *and* t_2 *when they are projected on* D*, and so* $\bar{\mathrm{R}}$ *contains tuples* $t_1[\mathtt{D}] = \langle\{\langle 3, 4\rangle, \langle 5, 6\rangle\}\rangle$ *and* $t_2[\mathtt{D}] = \langle\{\langle 9, 10\rangle\}\rangle$*.*

We define next the nest and unnest operators, starting with the nest operator. Roughly speaking, if R is a relation over a relational schema S, then nesting R on a set of principal attributes $\{A_1, \ldots, A_n\}$ in S means to group the tuples in R that agree on their values for attributes $\mathrm{att}(\mathrm{S}) - \{A_1, \ldots, A_n\}$. To represent the nested relations that result from the grouping, a new attribute A is added to S. In our definition of the nest operator, the new attribute A and the principal attributes $\{A_1, \ldots, A_n\}$ to be nested upon, are represented as a relational schema $\ddot{S}$. In particular, the root attribute of $\ddot{S}$ indicates the new attribute A, and the principal attributes of $\ddot{S}$ are the attributes to be nested upon.

Definition 4.9 (Nest Operator) Let R be a relation over a relational schema $\mathrm{S} = (\boldsymbol{V}, \boldsymbol{E})$. Also, let $\ddot{\mathrm{S}} = (\ddot{\boldsymbol{V}}, \ddot{\boldsymbol{E}})$ be a relational schema such that

(i) $\mathrm{root}(\ddot{\mathrm{S}}) \notin \boldsymbol{V}$, and
(ii) $\mathrm{att}(\ddot{\mathrm{S}}) \subset \mathrm{att}(\mathrm{S})$, and
(iii) $\forall A_i \in \mathrm{att}(\ddot{\mathrm{S}})$, $\ddot{\mathrm{S}}[A_i] = \mathrm{S}[A_i]$.

The operation of *nesting* relation R on attributes $\mathrm{att}(\ddot{\mathrm{S}})$ within attribute $\mathrm{root}(\ddot{\mathrm{S}})$, denoted by $\mathrm{nest}(\mathrm{R}, \ddot{\mathrm{S}})$, is defined to result in relation $\bar{\mathrm{R}}$ over relational schema $\bar{\mathrm{S}} = (\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}})$ given by

$$\begin{aligned}
\bar{\mathrm{S}} &= (\mathrm{S} - \mathrm{S}[A_i] \text{ for all } A_i \in \mathrm{att}(\ddot{\mathrm{S}})) + \ddot{\mathrm{S}} \\
\bar{\mathrm{R}} &= \{\text{tuple } \bar{t} \text{ over } \bar{\mathrm{S}} \mid \bar{t}\,[\mathrm{att}(\bar{\mathrm{S}}) - \{\mathrm{root}(\ddot{\mathrm{S}})\}] \in \mathrm{R}[\mathrm{att}(\mathrm{S}) - \mathrm{att}(\ddot{\mathrm{S}})] \wedge \\
&\qquad \bar{t}\,[\mathrm{root}(\ddot{\mathrm{S}})] = \{t[\mathrm{att}(\ddot{\mathrm{S}})] \mid t \in \mathrm{R} \wedge \\
&\qquad\qquad t[\mathrm{att}(\mathrm{S}) - \mathrm{att}(\ddot{\mathrm{S}})] = \bar{t}\,[\mathrm{att}(\bar{\mathrm{S}}) - \{\mathrm{root}(\ddot{\mathrm{S}})\}]\}\}.
\end{aligned}$$

We now briefly comment on requirements (i) - (iii) in Definition 4.9. First, the root attribute $\mathrm{root}(\ddot{\mathrm{S}})$ must not be contained within the attributes in S, since $\mathrm{root}(\ddot{\mathrm{S}})$ is added to S in order to obtain the relational schema $\bar{\mathrm{S}}$ of the resulting relation $\bar{\mathrm{R}}$, and duplicate attributes are not allowed in our model. Second, the principal attributes in $\ddot{\mathrm{S}}$ must be a subset of the principal attributes in S, for the obvious reason that nesting R on attributes outside of S is not possible. Third, the relational schemas rooted at the common principal attributes of S and $\ddot{\mathrm{S}}$ are required to be pairwise the same, which ensures that the resulting relation $\bar{\mathrm{R}}$ is indeed a relation over $\bar{\mathrm{S}}$. We now illustrate the nest operator.

Example 4.6 (nest operator) *Let* S *be the relational schema depicted in the top of Figure 4.5a, and let* R *be the relation over* S *depicted in the bottom of Figure 4.5a. Suppose now, that* R *is nested on attributes* C *and* D *inside attribute* E*, where* E *is the root attribute of relational schema* $\ddot{\mathrm{S}}$ *depicted in Figure 4.5b, where* $\{\mathtt{C}, \mathtt{D}\} = \mathrm{att}(\ddot{\mathrm{S}})$*. Note that this nesting is legal according to Definition 4.9 since (i)* $\mathtt{E} \notin \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$*, and (ii)* $\{\mathtt{C}, \mathtt{D}\} \subset \{\mathtt{B}, \mathtt{C}, \mathtt{D}\}$*, and (iii)* $\mathrm{S}[\mathtt{C}] = \ddot{\mathrm{S}}[\mathtt{C}]$ *as well as* $\mathrm{S}[\mathtt{D}] = \ddot{\mathrm{S}}[\mathtt{D}]$*. The relational schema* $\bar{\mathrm{S}}$ *that results from this nesting operation is given by* $\bar{\mathrm{S}} = (\mathrm{S} - \mathrm{S}[\mathtt{C}] - \mathrm{S}[\mathtt{D}]) + \ddot{\mathrm{S}}$*, which is the relational schema depicted in the top of Figure 4.5c. The relation* $\bar{\mathrm{R}}$ *over* $\bar{\mathrm{S}}$ *that results from this nesting is given by*

$$\begin{aligned}\bar{\mathrm{R}} = \{\textit{tuple } \bar{t} \textit{ over } \bar{\mathrm{S}} \mid &\ (a)\ \bar{t}\,[\mathtt{B}] \in \mathrm{R}[\mathtt{B}] \wedge \\ &\ (b)\ \bar{t}\,[\mathtt{E}] = \{t[\mathtt{C},\mathtt{D}] \mid t \in \mathrm{R} \wedge t[\mathtt{B}] = \bar{t}\,[\mathtt{B}]\}\}.\end{aligned}$$

Relation $\bar{\mathrm{R}}$ *is depicted in the bottom of Figure 4.5c. We now explain how the tuples in* $\bar{\mathrm{R}}$ *are constructed. From a bird eyes view, the tuples in* $\bar{\mathrm{R}}$ *are all those tuples over* $\bar{\mathrm{S}}$ *that satisfy requirements (a) and (b) above. A tuple* $\bar{t}$ *satisfies (a) if* $\bar{t}\,[\mathtt{B}] \in \{1, 2\}$ *since* $\mathrm{R}[\mathtt{B}] = \{\langle 1\rangle, \langle 2\rangle\}$*. Requirement (b) is a bit more involved and depends on the value of* $\bar{t}\,[\mathtt{B}]$*. In particular, if* $\bar{t}[\mathtt{B}] = 1$ *then* $\bar{t}\,[\mathtt{E}]$ *is required to contain the projections on attributes* C *and* D *of all those tuples in* R *that have* 1 *as value for* B*. So, if* $\bar{t}\,[\mathtt{B}] = 1$*, then* $\bar{t}\,[\mathtt{E}] = \{\langle 3, 4\rangle, \langle 5, 6\rangle\}$*. Analogously, if* $\bar{t}\,[\mathtt{B}] = 2$*, then* $\bar{t}\,[\mathtt{E}] = \{\langle 7, 8\rangle\}$*. Hence,* $\bar{t}$ *is a tuple in* $\bar{\mathrm{R}}$ *iff* $\bar{t}$ *is a tuple over* $\bar{\mathrm{S}}$ *and either* $\bar{t} = \langle 1, \{\langle 3, 4\rangle, \langle 5, 6\rangle\}\rangle$ *or* $\bar{t} = \langle 2, \{\langle 7, 8\rangle\}\rangle$*. From Definition 4.4, both* $\langle 1, \{\langle 3, 4\rangle, \langle 5, 6\rangle\}\rangle$ *and* $\langle 2, \{\langle 7, 8\rangle\}\rangle$ *are tuples over* $\bar{\mathrm{S}}$*, and therefore* $\bar{\mathrm{R}}$ *contains exactly these two tuples.*

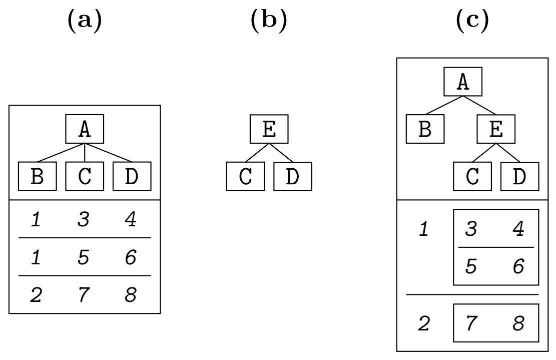


Figure 4.5: The operation of nesting a relation.

We now define the unnest operator, which is the inverse of the nest operator. The unnest operator takes as input a relation R over a relational schema S and obtains a new relation $\bar{\mathrm{R}}$ over $\bar{\mathrm{S}}$ by means of flatten the relations nested inside the tuples of R in an attribute A.

Definition 4.10 (Unnest Operator) Let R be a relation over a relational schema S, and let $A \in \mathrm{att(S)}$ be a nested attribute. The operation of *unnesting* A in R, denoted by $\mathrm{unnest}(\mathrm{R}, A)$, is defined to result in relation $\bar{\mathrm{R}}$ over relational schema $\bar{\mathrm{S}} = (\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}})$ given by

$$\begin{aligned}\bar{\mathrm{S}} &= \mathrm{S} - \mathrm{S}[A] + \mathrm{S}[A_i] \text{ for all } A_i \in \mathrm{att}(\mathrm{S}[A]) \\ \bar{\mathrm{R}} &= \{\text{tuple } \bar{t} \text{ over } \bar{\mathrm{S}} \mid \exists\, t \in \mathrm{R} \text{ such that} \\ &\qquad \bar{t}\,[\mathrm{att}(\bar{\mathrm{S}}) - \mathrm{att}(\mathrm{S}[A])] = t[\mathrm{att(S)} - \{A\}] \wedge \\ &\qquad \bar{t}\,[\mathrm{att}(\mathrm{S}[A])] \in t[A]\}\end{aligned}$$

We now illustrate the unnest operator by an example.

Example 4.7 (unnest operator) *Let* S *be the relational schema depicted in the top of Figure 4.5c, and let* R *be the relation over* S *depicted in the bottom of Figure 4.5c. Suppose now that attribute* E *is unnested in* R*. Then, the relational schema* $\bar{\mathrm{S}}$ *that results from this unnesting is given by* $\bar{\mathrm{S}} = \mathrm{S} - \mathrm{S}[\mathtt{E}] + \mathrm{S}[\mathtt{C}] + \mathrm{S}[\mathtt{D}]$*, which is the relational schema depicted in the top of Figure 4.5a. The resulting relation* $\bar{\mathrm{R}}$ *over* $\bar{\mathrm{S}}$ *is given by*

$$\bar{\mathrm{R}} = \{\textit{tuple } \bar{t} \textit{ over } \bar{\mathrm{S}} \mid \exists t \in \mathrm{R} \textit{ such that (a) } \bar{t}[\mathtt{B}] = t[\mathtt{B}] \wedge \\ \textit{(b) } \bar{t}[\mathtt{C},\mathtt{D}] \in t[\mathtt{E}]\}$$

Relation $\bar{\mathrm{R}}$ is depicted in the bottom of Figure 4.5a. We now explain how the tuples in $\bar{\mathrm{R}}$ are constructed. As for the nest operator, also the unnest operator determines the tuples in $\bar{\mathrm{R}}$ by means of restricting the set of all tuples over $\bar{\mathrm{S}}$. The requirement on a tuple $\bar{t}$ to be contained in $\bar{\mathrm{R}}$ is that there exists a tuple $t \in \mathrm{R}$ such that (a) and (b) above are satisfied. In particular, (a) is satisfied if $\bar{t}[\mathtt{B}] = t[\mathtt{B}]$. Thus, if $\bar{t}$ is a tuple in $\bar{\mathrm{R}}$ then $\bar{t}[\mathtt{B}] \in \{\langle 1\rangle, \langle 2\rangle\}$ since $\mathrm{R}[B] = \{\langle 1\rangle, \langle 2\rangle\}$. Requirement (b) depends on the choice of tuple t in (a), and states that the projection on attributes `C` *and* `D` *of $\bar{t}$ must be in the relation nested within attribute* `E` *in tuple t. Therefore, if $t[\mathtt{B}] = \langle 1\rangle$ then $t[\mathtt{E}] = \{\langle 3, 4\rangle, \langle 5, 6\rangle\}$, and $\bar{t}[\mathtt{C},\mathtt{D}]$ must either be of the form $\bar{t}[\mathtt{C},\mathtt{D}] = \langle 3, 4\rangle$ or $\bar{t}[\mathtt{C},\mathtt{D}] = \langle 5, 6\rangle$. If, instead, $t[\mathtt{B}] = \langle 2\rangle$ then $t[\mathtt{E}] = \{\langle 7, 8\rangle\}$, and so $\bar{t}[\mathtt{C},\mathtt{D}]$ must be of the form $\bar{t}[\mathtt{C},\mathtt{D}] = \langle 3, 4\rangle$. Hence, $\bar{t}$ is a tuple in $\bar{\mathrm{R}}$ iff $\bar{t}$ is a tuple over $\bar{\mathrm{S}}$ and $\bar{t}$ is of the form $\bar{t} = \langle 1, 3, 4\rangle$, or it is of the form $\bar{t} = \langle 1, 5, 6\rangle$, or it is of the form $\bar{t} = \langle 2, 7, 8\rangle$. It follows directly from Definition 4.4 that each of these three tuples is a tuple over $\bar{\mathrm{S}}$, and so these tuples are contained in $\bar{\mathrm{R}}$.*

4.2 Transforming Relational Data to XML Data

In this section we present our procedure for transforming a flat relational database to an XML tree. In our procedure the relations in the database are first transformed to separate XML trees, and these XML trees are then added as primary subtrees to the final XML tree.

For transforming a single relation to an XML tree, we adopt the transformation procedure originally proposed by Vincent et al. in [34]. This procedure allows the application developer to govern the restructuring of information during the transformation process by applying an arbitrary sequence of nesting operations to the initial flat relation prior to the mapping of the relation to an XML tree. Roughly speaking, the nested relation obtained from applying the sequence of nesting operations is mapped to XML so that each (nested) tuple is represented by an element node which has child attribute nodes that represent the values in the tuple.

To increase the usability of the transformation procedure in [34], we propose the following enhancements. First, the original transformation procedure requires the application developer to explicitly specify the sequence of nesting operations, which is likely to be cumbersome. In contrast, our transformation procedure only requires the application developer to specify the nested relational schema and automatically derives the necessary nesting operations. Second, when mapping the obtained nested relation to an XML tree, the original transformation procedure uses consecutive numbers as labels for element nodes that represent (nested) tuples. The labels of element nodes are therefore meaningless. In contrast, our transformation procedure reuses the nested attributes in the nested relational schema as labels for element nodes. We now illustrate these enhancements by an example.

Example 4.8 (procedures for transforming a relation) *Let S be the relational schema depicted in the top of Figure 4.6a, and let R be the initial flat relation over S depicted in the bottom of Figure 4.6a. Suppose now that relation R is transformed to XML, by first nesting the relation on attribute* `C` *inside new attribute* `D`*. The nested relation $\bar{\mathrm{R}}$ that results from this nesting is depicted in the bottom of Figure 4.6b, and the* *corresponding relational schema* $\bar{\mathrm{S}}$ *is depicted in the top of Figure 4.6b. When applying the original transformation procedure, one has to explicitly specify the nesting operation* $\mathrm{nest}(\mathrm{R}, \mathrm{S}[D])$*, whereas one can simply specify the relational schema* $\bar{\mathrm{S}}$ *when applying the enhanced transformation procedure.*

In the second step, nested relation $\bar{\mathrm{R}}$ *is mapped to an XML tree. The XML trees* T_O *and* T_E *that result from this mapping when applying the original and the enhanced transformation procedure, respectively, are depicted in Figures 4.6c and 4.6d. Both trees* T_O *and* T_E *contain one element node per (nested) tuple in* $\bar{\mathrm{R}}$*, and so the structure of trees* T_O *and* T_E *is the same. Nested tuples* $\langle$*c2*$\rangle$, $\langle$*c3*$\rangle$ *and* $\langle$*c5*$\rangle$ *for example, are each represented as an element node having a child attribute node labeled* `C` *with value c2, c3, and c5, respectively. The labels of these element nodes however differ between trees* T_O *and* T_E*. In particular, in tree* T_O *the meaningless label* `ID3` *is assigned to these element nodes, whereas in tree* T_E *the label* `D` *is used, since* $\langle$*c2*$\rangle$, $\langle$*c3*$\rangle$ *and* $\langle$*c5*$\rangle$ *are tuples over* $\bar{\mathrm{S}}[\mathtt{D}]$*. Analogously, the two overall tuples* $\langle$*b1*$, \{\langle$*c2*$\rangle, \langle$*c3*$\rangle\}\rangle$ *and* $\langle$*b4*$, \{\langle$*c5*$\rangle\}\rangle$ *in* $\bar{\mathrm{R}}$ *are represented in trees* T_E *and* T_O *by element nodes labeled* `A` *and* `ID2`*, respectively.*

Since, T_E *and* T_O *are both XML trees that conform to Definition 3.11, both XML trees have a single root node, which is created in addition to the element nodes for representing the (nested) tuples in* $\bar{\mathrm{R}}$*. The root node of tree* T_E *has the custom label* `E` *assigned, whereas the root node of tree* T_O *has the generated label* `ID1`*.*

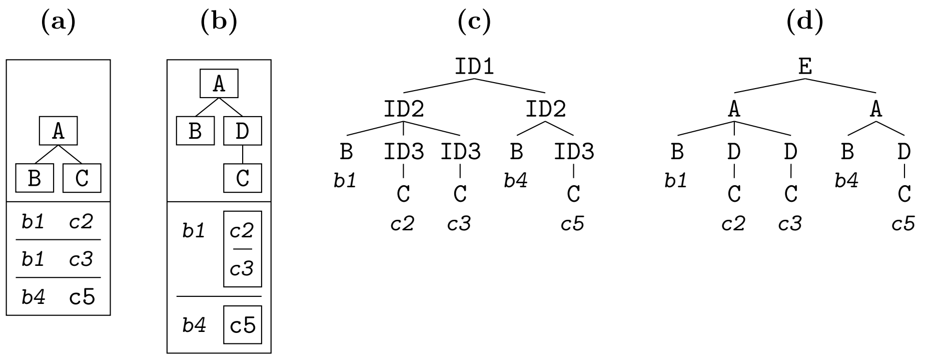


Figure 4.6: Two procedures for generating an XML tree from a flat relation..

We now specify our transformation procedure by precise algorithms in Subsections 4.2.1 - 4.2.4. In particular, Subsections 4.2.1 and 4.2.2 present algorithms FR2NR and NR2XML for converting a single relation to a nested relation by applying an arbitrary sequence of nesting operations, and for mapping a nested relation to an XML tree, respectively. Subsection 4.2.3 presents algorithm FR2XML which uses algorithms FR2NR and NR2XML and transforms a flat relation to an XML tree. Apart from the two enhancements illustrated above, algorithms FR2NR, NR2XML and FR2XML correspond to the transformation procedure proposed by Vincent et al. in [34]. Finally, Subsection 4.2.4 presents algorithm DB2XML which uses algorithm FR2XML in order to transform a relational database to an XML tree.

4.2.1 Converting Flat Relations to Nested Relations

Algorithm FR2NR essentially takes as input a flat relation, called source relation, together with a nested relational schema, called its target schema, and converts the source relation into a relation, possibly nested, over the target schema by recursively applying the nest operator.

Because converting a source relation to a target relation is done by recursively performing nest operations, it is in general not possible to obtain a target relation for arbitrary target schemas. In particular, the nest operator does not introduce new flat attributes, and therefore the source schema and the target schema must agree on the flat attributes. Also, the nest operator does not alter the structure of nested attributes in the source schema and so these nested attributes must be contained one-to-one in the target schema. We now make these requirements on a target schema more precise.

Definition 4.11 (Transformation) Let S and $\bar{\mathrm{S}}$ be relational schemas. The relational schema $\bar{\mathrm{S}}$ is defined to be a *transformation* of S if

(i) $A \in \mathrm{S}$ is a flat attribute iff $A \in \bar{\mathrm{S}}$ is a flat attribute, and
(ii) if $A \in \mathrm{att(S)}$ is a nested attribute, then $A \in \bar{\mathrm{S}}$ and $\mathrm{S}[A] = \bar{\mathrm{S}}[A]$.
(iii) if $\bar{A} \in \bar{\mathrm{S}}$ is a nested attribute, then $\exists A \in \mathrm{att}(\bar{A})$ which is a flat attribute.

If a target schema is a transformation of a source schema, and thus satisfies (i) - (iii) in Definition 4.11, then the target schema allows for obtaining a target relation by recursively applying nest operations. Requirements (i) and (ii) in Definition 4.11 reflect the requirements on target schemas discussed previously. The additional requirement in Definition 4.11 originates from the transformation procedure proposed by Vincent et al. in [34] and states that every nested attribute in the target schema must have at least one flat principal attribute. Our reason for including requirement (iii) in Definition 4.11 is so as to ensure that our transformation procedure creates XML trees that essentially have the same properties as those created by the original transformation procedure.

Algorithm FR2NR is a recursive algorithm that takes as input a source schema S, a source relation R over S, a target schema $\bar{\mathrm{S}}$ and a distinguished nested attribute $\bar{A}$ in $\bar{\mathrm{S}}$. In order to convert the source relation R to a target relation $\bar{\mathrm{R}}$ over $\bar{\mathrm{S}}$, the algorithm traverses the target schema $\bar{\mathrm{S}}$ in post order and performs nesting operations on R in correspondence to the nested attributes in $\bar{\mathrm{S}}$. The input attribute $\bar{A}$ specifies the nested attribute which is visited in an invocation of the algorithm during the traversal of $\bar{\mathrm{S}}$. Clearly, $\bar{A}$ is required to be the root attribute of $\bar{\mathrm{S}}$ when the algorithm is invoked first, in order to ensure that all nested attributes in $\bar{\mathrm{S}}$ are visited.

In every recursive invocation of algorithm FR2NR, only certain parts of the entire conversion are performed, and the algorithm therefore generates a temporary relational schema $\ddot{S}$ serving as target schema for the recursive invocation (cf. Line 3 in Algorithm 4.1). When algorithm FR2NR returns from a recursive invocation, the source schema S is replaced by the temporary target schema $\ddot{S}$ (cf. Line 5 in Algorithm 4.1), and so $\ddot{S}$ is the source schema for the next invocation of the algorithm.

Also, algorithm FR2NR does not perform a nest operation for the root attribute of $\bar{\mathrm{S}}$, since the nested attributes in $\bar{\mathrm{S}}$ are traversed in post order and therefore the conversion of

Algorithm 4.1 FR2NR - Convert Flat Relation to Nested Relation

```
in:  source relational schema $\mathrm{S} = (\boldsymbol{V}, \boldsymbol{E})$
     source relation $\mathrm{R}$ over $\mathrm{S}$
     target relational schema $\bar{\mathrm{S}} = (\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}})$ which is a transformation of $\mathrm{S}$
     nested attribute $\bar{A} \in \bar{\mathrm{S}}$ such that $\bar{A} = \mathrm{root}(\bar{\mathrm{S}})$ when FR2NR is first invoked
out: transformed relation $\bar{\mathrm{R}}$ over $\bar{\mathrm{S}}$
1: $\bar{\mathrm{R}} \leftarrow \mathrm{R}$
2: for each nested attribute $A \in \mathrm{att}(\bar{\mathrm{S}}[\bar{A}]) - \boldsymbol{V}$ do
3:     $\ddot{\mathrm{S}} \leftarrow \mathrm{S} - \bar{\mathrm{S}}[A] + \bar{\mathrm{S}}[A]$
4:     $\bar{\mathrm{R}} \leftarrow \mathrm{FR2NR}(\mathrm{S}, \bar{\mathrm{R}}, \ddot{\mathrm{S}}, A)$
5:     $\mathrm{S} \leftarrow \ddot{\mathrm{S}}$
6: end for
7: if $\bar{A} \neq \mathrm{root}(\bar{\mathrm{S}})$ then $\bar{\mathrm{R}} \leftarrow \mathrm{nest}(\bar{\mathrm{R}}, \bar{\mathrm{S}}[\bar{A}])$ end if
8: return $\bar{\mathrm{R}}$
```

R is finished before the root node of $\bar{\mathrm{S}}$ is visited. We now illustrate algorithm FR2NR by an example.

Example 4.9 (algorithm FR2NR) *Consider the source relation* R_1 *over relational schema* S_1 *depicted in Figure 4.7a, and the target schema* $\bar{\mathrm{S}}_1$ *depicted in Figure 4.7d. Also, suppose that* R_1 *is converted to a nested relation* $\bar{R}_1$ *by* $\mathrm{FR2NR}(\mathrm{S}_1, \mathrm{R}_1, \bar{\mathrm{S}}_1, \bar{A}_1)$, *where* $\bar{A}_1 = \mathtt{A}$ *is the root attribute in* $\bar{\mathrm{S}}_1$*, as required by the algorithm. Then,* $\bar{\mathrm{R}}_1 = \mathrm{R}_1$ *at Line 1 and the algorithm iterates over the principal nested attributes in* $\bar{\mathrm{S}}_1[\mathtt{A}]$ *which are not contained in* S_1*, i.e.* `C` *and* `G`*, within the loop at Line 2. Now, let* $\ddot{\mathrm{S}}_{1\mathrm{a}}$ *be the temporary target schema at Line 3 within the first iteration of the loop at Line 2. Then* $\ddot{\mathrm{S}}_{1\mathrm{a}} = \mathrm{S}_1 - \bar{\mathrm{S}}_1[\mathtt{C}] + \bar{\mathrm{S}}_1[\mathtt{C}]$*, which is depicted in Figure 4.7c. At Line 4, the algorithm is then recursively invoked by* $\mathrm{FR2NR}(\mathrm{S}_1, \bar{\mathrm{R}}_1, \ddot{\mathrm{S}}_{1\mathrm{a}}, \mathtt{C})$.

In this invocation, let $\mathrm{S}_2 = \mathrm{S}_1$, $\mathrm{R}_2 = \bar{\mathrm{R}}_1$, $\bar{\mathrm{S}}_2 = \ddot{\mathrm{S}}_{1\mathrm{a}}$ *and* $\bar{A}_2 = \mathtt{C}$ *denote the input for the algorithm. Note that* `C` *is not required to be the root of* S_2 *since the algorithm has been invoked recursively. Now, if* $\bar{\mathrm{R}}_2$ *is the target relation in the second invocation, then* $\bar{\mathrm{R}}_2 = \mathrm{R}_2$ *at Line 1, and the single principal nested attribute* `E` *in* $\bar{\mathrm{S}}_2[\mathtt{C}]$ *is iterated in the loop at Line 2. Thereby, if* $\ddot{\mathrm{S}}_2$ *is the temporary target schema at Line 3, then* $\ddot{\mathrm{S}}_2 = (\mathrm{S}_2 = \mathrm{S}_1) - \bar{\mathrm{S}}_2[\mathtt{C}] + \bar{\mathrm{S}}_2[\mathtt{C}]$*, which is depicted in Figure 4.7b. The algorithm is then again recursively invoked at Line 4 by* $\mathrm{FR2NR}(\mathrm{S}_2, \bar{\mathrm{R}}_2, \ddot{\mathrm{S}}_2, \mathtt{E})$.

In the third invocation, let $\mathrm{S}_3 = \mathrm{S}_2$, $\mathrm{R}_3 = \bar{\mathrm{R}}_2$, $\bar{\mathrm{S}}_3 = \ddot{\mathrm{S}}_2$ *and* $\bar{A}_3 = \mathtt{E}$ *denote the input for the algorithm. Then, the target relation* $\bar{\mathrm{R}}_3 = \mathrm{R}_3$ *at Line 1, and the algorithm proceeds with the test at Line 7, since there is no principal nested attribute in* $\bar{\mathrm{S}}_3[\mathtt{E}]$*. Because* $\bar{A}_3 = \mathtt{E} \neq \mathtt{A}$*, where* `A` *is the root attribute in* $\bar{\mathrm{S}}_3$ *(cf. Figure 4.7b), the test at Line 7 succeeds and so the nest operator is applied to* $\bar{\mathrm{R}}_3$*. In particular,* $\mathrm{nest}(\bar{\mathrm{R}}_3, \bar{\mathrm{S}}_3[\mathtt{E}])$ *is performed where* $\bar{\mathrm{R}}_3 = \mathrm{R}_2 = \mathrm{R}_1$*. That is, the tuples in* R_1 *that agree on their values in* `B`, `D` *and* `H` *are grouped together. The resulting relation is then returned at Line 8.*

Back in the second invocation, $\bar{\mathrm{R}}_2$ *is the relation depicted in Figure 4.7b at Line 4, and* $\mathrm{S}_2 = \ddot{\mathrm{S}}_2$ *at Line 5. The loop at Line 2 then exists, and because* $\bar{A}_2 = \mathtt{C} \neq \mathtt{A}$*, where* `A` *is the root attribute in* $\bar{\mathrm{S}}_2$ *(cf. Figures 4.7c), the test at Line 7 succeeds. Consequently,* $\mathrm{nest}(\bar{\mathrm{R}}_2, \bar{\mathrm{S}}_2[\mathtt{C}])$ *is returned at Line 8.*

Back in the first iteration of the loop at Line 2 within the first invocation of the algorithm, $\bar{\mathrm{R}}_1$ *is the relation depicted in Figure 4.7c at Line 4, and* $\mathrm{S}_1 = \ddot{\mathrm{S}}_{1\mathrm{a}}$ *at Line 5. So, in the second iteration of the loop at Line 2, the temporary target schema* $\ddot{\mathrm{S}}_{1\mathrm{b}} = (\mathrm{S}_1 = \ddot{\mathrm{S}}_{1a}) - \mathrm{S}_1[\mathtt{G}] + \bar{\mathrm{S}}_1[\mathtt{G}]$*, which is the relational schema in Figure 4.7d. At Line 4, algorithm* FR2NR *is then recursively invoked by* $\mathrm{FR2NR}(\ddot{\mathrm{S}}_{1\mathrm{a}}, \bar{\mathrm{R}}_1, \ddot{\mathrm{S}}_{1\mathrm{b}}, \mathtt{G})$*. Note that* $\mathrm{S}_1 = \ddot{\mathrm{S}}_{1\mathrm{a}}$ *directly before this invocation.*

In the fourth invocation, only a nesting operation is performed, since there are no principal nested attributes in $\ddot{\mathrm{S}}_{1\mathrm{b}}[\mathtt{G}]$ *and* $\mathtt{G} \neq \mathtt{A}$ *at Line 7. In particular, relation* $\bar{\mathrm{R}}_1$*, which is at this state the relation depicted in Figure 4.7c, is nested by performing* $\mathrm{nest}(\bar{\mathrm{R}}_1, \ddot{\mathrm{S}}_{1\mathrm{b}}[\mathtt{G}])$*. The resulting relation is depicted in Figure 4.7d.*

Next, $\mathrm{S}_1 = \ddot{\mathrm{S}}_{1\mathrm{b}}$ *at Line 5 within the second iteration of the loop at Line 2 in the first invocation, and the loop at Line 2 then exists. Because* $\bar{A}_1 = \mathtt{A}$*, which is the root attribute in* $\bar{\mathrm{S}}_1$*, the test at Line 7 fails. Hence, the algorithm finally returns at Line 8 the relation* $\bar{\mathrm{R}}_1$ *which has been returned by the fourth invocation. Note that this relation, which is depicted in Figure 4.7d, is indeed a relation over* $\bar{\mathrm{S}}_1$*.*

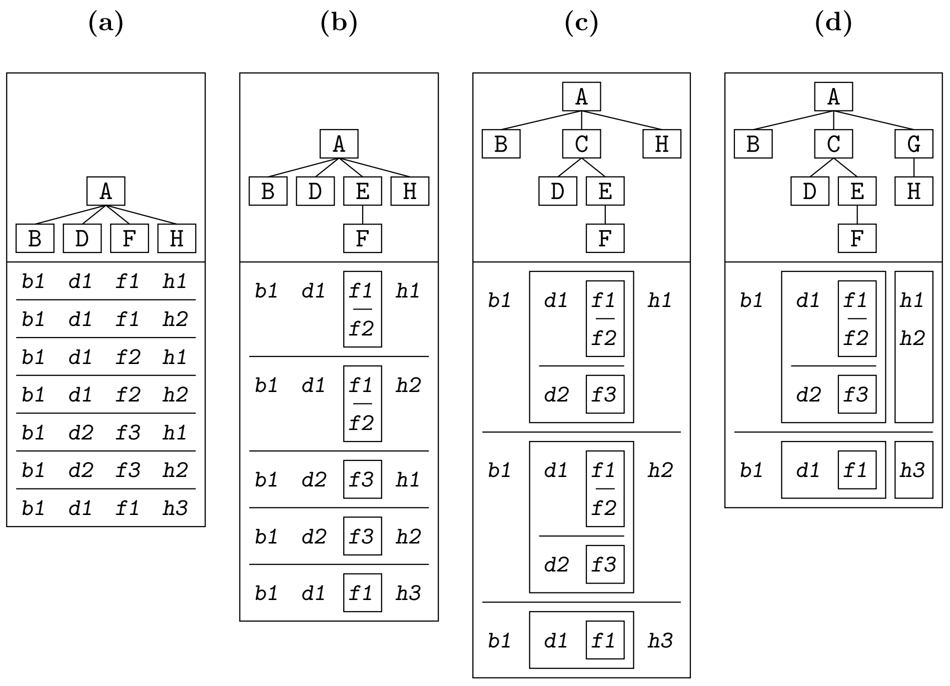


Figure 4.7: Steps in converting a flat relation into a nested relation using algorithm FR2NR.

Because the nest operator is not associative, the structure of tuples in the target relation depends not only on the performed nesting operations, but also on the specific order in which nesting operations are performed. Since algorithm FR2NR performs the nesting operations according to the nested attributes in the input target schema, the ordering of attributes in the target schema in fact determines the form of tuples in the target relation. We now illustrate this point by an example.

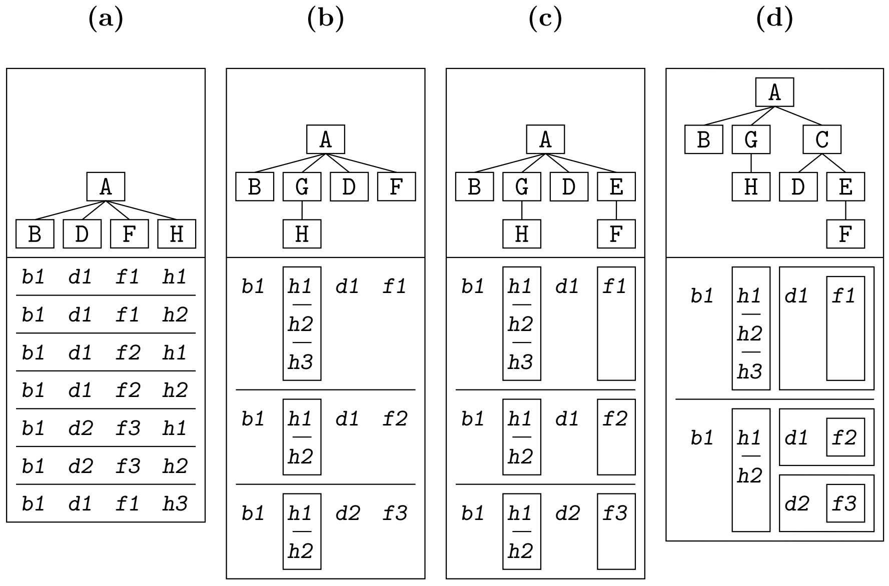


Figure 4.8: Impact of the order of nested attributes in target schemas on the result of algorithm FR2NR.

Example 4.10 (impact of attribute order on algorithm FR2NR) *Consider again the source relation* R_1 *over source schema* S_1 *from Example 4.9, and suppose this time that* R_1 *is to be converted to a target relation* $\bar{\mathrm{R}}_1$ *over target schema* $\bar{\mathrm{S}}_1$ *depicted in Figure 4.8d by calling* $\mathrm{FR2NR}(\mathrm{S}_1, \mathrm{R}_1, \bar{\mathrm{S}}_1, \mathtt{A})$*. Note that* $\bar{\mathrm{S}}_1$ *differs from the target schema in Example 4.9 only in the order of nested attributes. In particular, the nested attribute* `G` *this time precedes the nested attribute* `C` *in* $\bar{\mathrm{S}}_1$*. The tuples in the target relation* $\bar{\mathrm{R}}_1$*, depicted in Figure 4.8d, obviously differ from the tuples in the target relation in Example 4.9, which is depicted in Figure 4.7d. The reason for this is, that the order of nested attributes differs between* $\bar{\mathrm{S}}_1$ *and the target schema in Example 4.9.*

In particular, in Example 4.9 the first nesting operation grouped the tuples in R_1 *that agree on their values in* `B`*,* `D` *and* `H`*. Note that* R_1 *is depicted again in Figure 4.7a for reasons of clarity. In this example however the first nesting operation groups the tuples in* R_1 *that agree on their values in* `B`*,* `D` *and* `F`*, since this nesting operation is performed according to the nested attribute* `G`*. Note that the relations that result from the first nesting operation already differ between this example (cf. Figure 4.8b) and Example 4.9 (cf. Figure 4.7b).*

The second nesting operation then groups the tuples that agree on their values in `B`, `G` *and* `D` *(cf. Figure 4.8c), and the third nesting operation groups the tuples that agree on their values in* `D` *and* `E`*, which results in the final target relation depicted in Figure 4.8d.*

4.2.2 Mapping Nested Relations to XML Trees

Algorithm NR2XML essentially takes a relational schema S and a relation R over S as input and maps R to an XML tree T by representing each (nested) tuple by one element node which has child attribute nodes for representing the values in the tuple. The nested and flat attributes in S are used as labels for element and attribute nodes in T, respectively. Apart from relational schema S and relation R over S, algorithm NR2XML has input a custom element label l which is assigned to the root node of the resulting tree T. To obtain T from R, the algorithm initially creates a trivial XML tree that has label l assigned to the root node (cf. Line 1 in Algorithm 4.2). The actual mapping of the tuples in R is then performed by the recursive procedure convert() at Line 4 in Algorithm 4.2.

Algorithm 4.2 NR2XML - Map Nested Relation to XML

```
in:   relational schema S
      relation R over S
      element label $l$
out:  XML tree T representing R
1: let T = $(\boldsymbol{V}, \boldsymbol{E}, \mathrm{lab}, \mathrm{val})$ be a trivial XML tree where lab(root(T)) = $l$
2: convert(R, S, root(T))
3: return T
4: procedure convert(relational schema S, relation R, element node $v$)
5:     for each $t \in$ R do
6:         $\bar{v} \leftarrow$ newnode($\boldsymbol{V}$)
7:         $\boldsymbol{E} \leftarrow \boldsymbol{E} \cup \{(v, \bar{v})\}$
8:         lab($\bar{v}$) $\leftarrow$ root(S)
9:         for each $A \in$ att(S) do
10:            if $A$ is a nested attribute then
11:                convert($t[A]$, S[$A$], $\bar{v}$)
12:            else
13:                $\ddot{v} \leftarrow$ newnode($\boldsymbol{V}$)
14:                $\boldsymbol{E} \leftarrow \boldsymbol{E} \cup \{(v, \ddot{v})\}$
15:                lab($\ddot{v}$) $\leftarrow A$
16:                val($\ddot{v}$) $\leftarrow t[A]$
17:            end if
18:        end for
19:    end for
20: end procedure
```

Function newnode($\boldsymbol{V}$) is used at Lines 6 and 13 in Algorithm 4.2 for creating new nodes. This function in particular creates a new node v with respect to the input set of nodes $\boldsymbol{V}$, adds v to $\boldsymbol{V}$, and finally returns v. We now illustrate algorithm NR2XML by an example.

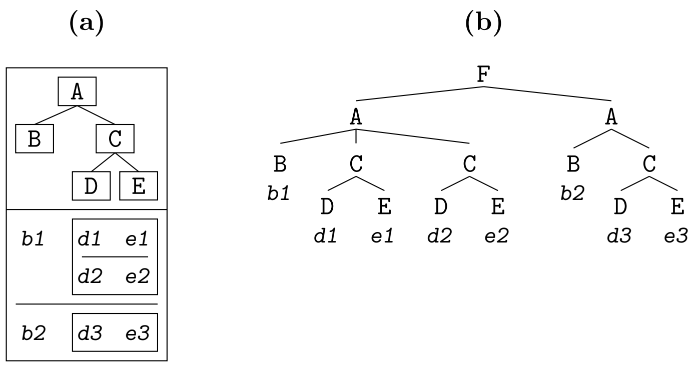


Figure 4.9: A nested relation and the XML tree obtained by algorithm NR2XML.

Example 4.11 (algorithm NR2XML) *Consider the relational schema* S_1*, and the relation* R_1 *over* S_1 *depicted in Figure 4.9a. Suppose now that* R_1 *is mapped to an XML tree* T *by calling* $\mathrm{NR2XML}(\mathrm{R}_1, \mathrm{S}_1, \mathtt{F})$*, where* F *is the custom label chosen for the root node of* T*, which is depicted in Figure 4.9b.*

Initially, T *is a trivial XML tree at Line 1 in Figure 4.2, and so* T *contains only the root node* root(T) *which has label* F *assigned. Procedure* convert() *is then invoked at Line 2, which actually performs the mapping of the tuples in* R_1 *and creates the subtrees rooted at the two* A *nodes in Figure 4.9b. In particular, procedure* convert *is invoked by* $\mathrm{convert}(\mathrm{R}_1, \mathrm{S}_1, \mathrm{root}(\mathrm{T}))$*. The tuples in* R_1*, i.e.* $t_1 = \langle \mathtt{b1}\{\langle \mathtt{d1}, \mathtt{e1}\rangle, \langle \mathtt{d2}, \mathtt{e2}\rangle\}\rangle$ *and* $t_1' = \langle \mathtt{b2}\{\langle \mathtt{d3}, \mathtt{e3}\rangle\}\rangle$*, are then iterated within the loop at Line 5. Consider first the mapping of tuple* t_1 *within the first iteration of the loop at Line 5. In order to represent the entire tuple, a node* $\bar{v}_1$ *with label* A *is created and added to* T *as child of* root(T) *at Lines 6 - 8. Note that* $\bar{v}_1$ *is an element node, since* A *is a nested attribute in* S*, and therefore* $\mathtt{A} \in \boldsymbol{L}^E$*. Next, in order to represent the values in* t_1*, the loop at Line 9 iterates over the principal attributes in* S*, i.e.* B *and* C*, and creates nodes for representing values* $t_1[\mathtt{B}]$ *and* $t_1[\mathtt{C}]$*.*

Concerning $t_1[\mathtt{B}]$*, the test at Line 10 fails because* B *is a flat attribute. Therefore, node* $\ddot{v}_1$ *with label* B *is created and added to* T *as a child of* root(T) *at Lines 13 - 15. In contrast to the previously created node* $\bar{v}_1$*, node* $\ddot{v}_1$ *is an attribute node, since* B *is a flat attribute, and therefore* $\mathtt{B} \in \boldsymbol{L}^A$*. Next, node* $\ddot{v}_1$ *gets the value* $t_1[\mathtt{B}]$ *assigned at Line 16, i.e.* $\mathrm{val}(\ddot{v}_1) = \mathtt{b1}$*.*

Concerning $t_1[\mathtt{C}]$*, the test at Line 10 succeeds because* C *is a nested attribute. Hence, procedure* convert() *is recursively invoked at Line 11 by* $\mathrm{convert}(\mathrm{S}_2, \mathrm{R}_2, \bar{v}_1)$*, where* $\mathrm{S}_2 = \mathrm{S}_1[\mathtt{C}]$*, and* $\mathrm{R}_2 = t_1[\mathtt{C}] = \{\langle \mathtt{d1}, \mathtt{e1}\rangle, \langle \mathtt{d2}, \mathtt{e2}\rangle\}$*. In this recursive invocation, tuples* $t_2 = \langle \mathtt{d1}, \mathtt{e1}\rangle$ *and* $t_2' = \langle \mathtt{d2}, \mathtt{e2}\rangle$ *are then iterated within the loop at Line 5, and because* C *is the root attribute in* S_2*, one element node labeled* C *is created at Lines 6 - 8 for each tuple. If* $\bar{v}_2$ *and* $\bar{v}_2'$ *denote these nodes, then* $\bar{v}_2$ *and* $\bar{v}_2'$ *are child nodes of* $\bar{v}_1$ *(cf. left of Figure 4.9b). Further, since the principal attributes* D *and* E *of* S_2 *are flat attributes, the test at Line 10* *fails for both attributes when tuples t_2 and t'_2 are iterated within the loop at Line 5, and therefore two pairs of attribute nodes $\ddot{v}_{2_1}, \ddot{v}_{2_2}$ and $\ddot{v}'_{2_1}, \ddot{v}'_{2_2}$ are created at Lines 13 - 16 such that $\{\ddot{v}_{2_1}, \ddot{v}_{2_2}\} = \text{child}(\bar{v}_2)$ and $\{\ddot{v}'_{2_1}, \ddot{v}'_{2_2}\} = \text{child}(\bar{v}'_2)$. Thereby, $\text{lab}(\ddot{v}_{2_1}) = \text{lab}(\ddot{v}'_{2_1}) = \texttt{D}$ and $\text{lab}(\ddot{v}_{2_2}) = \text{lab}(\ddot{v}'_{2_2}) = \texttt{E}$, and also $\text{val}(\ddot{v}_{2_1}) = \texttt{d1}$, $\text{val}(\ddot{v}_{2_2}) = \texttt{e1}$, $\text{val}(\ddot{v}'_{2_1}) = \texttt{d2}$ and $\text{val}(\ddot{v}'_{2_2}) = \texttt{e2}$ (cf. left of Figure 4.9b. The mapping of nested tuples t_2 and t'_2 is then finished, and hence also the mapping of the overall tuple t_1.*

The mapping of the remaining tuple t'_1 is then performed by first creating a node $\bar{v}'_1$ labeled A *at Lines 6 - 8 in order to represent the entire tuple t'_1, where node $\bar{v}'_1$ is added to* T *as a child node of* root(T)*. Next, an attribute node labeled* B *is created at Lines 13 - 16 and added to* T *as child of $\bar{v}'_1$ for representing the value $t'_1[\texttt{B}] = \texttt{b2}$. Procedure* convert() *is then recursively invoked at Line 11 by* $\text{convert}(t'_1[\texttt{C}], \text{S}[C], \bar{v}'_1)$ *in order to map the nested relation $t'_1[\texttt{C}] = \{\langle \texttt{d3}, \texttt{e3}\rangle\}$. In this final invocation, a node $\bar{v}_3$ labeled* C *is created at Lines 6 - 8 and added to* T *as child of $\bar{v}'_1$. Finally, two attribute nodes labeled* D *and* E *with values* d3 *and* e3 *are created and added to* T *as child nodes of $\bar{v}_3$. Algorithm* NR2XML *then terminates and returns at Line 3 the final tree* T*.*

4.2.3 Transforming Single Relations

Algorithm FR2XML takes as input a flat relation R over a relational schema S together with a relational schema $\bar{\text{S}}$ and an element label l. At Line 1 in Algorithm 4.3, relation R is converted to a nested relation $\bar{\text{R}}$ over $\bar{\text{S}}$ by invoking algorithm FR2NR, and $\bar{\text{R}}$ is then mapped to an XML tree T by invoking algorithm NR2XML at Line 2. The label l is thereby used as label for the root node of T, which is finally returned at Line 3. Note that the relational schema $\bar{\text{S}}$ serves both as target schema for converting R to $\bar{\text{R}}$, and as template for mapping $\bar{\text{R}}$ to T. In order to serve as target schema, $\bar{\text{S}}$ is required to be a transformation of S.

Algorithm 4.3 FR2XML - Transform Flat Relation to XML.

```
in:   relational schema S
      flat relation R over S
      relational schema $\bar{S} = (\boldsymbol{V}, \boldsymbol{E})$ which is a transformation of S
      element label $l$
out:  XML tree T representing R
1: $\bar{R} \leftarrow$ FR2NR(S, R, $\bar{S}$, root($\bar{S}$))
2: T $\leftarrow$ NR2NR($\bar{S}$, $\bar{R}$, $l$)
3: return T
```

We now illustrate algorithm FR2XML by an example.

Example 4.12 (algorithm FR2XML) *Consider the flat relation* R *over relational schema* S *depicted in Figure 4.10a, and let $\bar{\text{S}}$ be the relational schema depicted in Figure 4.10b. Suppose now, that* R *is transformed to XML by* $\text{FR2XML}(\text{S}, \text{R}, \bar{\text{S}}, \texttt{E})$*, where* E *is the custom label chosen for the root node of the resulting XML tree* T*, and $\bar{\text{S}}$ is a transformation of* S *as required by the algorithm. Then, at Line 1 in Algorithm 4.3, relation* R *is converted into the nested relation $\bar{\text{R}}$ depicted in Figure 4.10b by* $\text{FR2NR}(\text{S}, \text{R}, \bar{\text{S}}, \texttt{A})$*, where* A *is the root* *attribute in* $\bar{\mathrm{S}}$*. At Line 2, relation* $\bar{\mathrm{R}}$ *is then mapped to the XML tree* T*, depicted in Figure 4.10c, by* NR2XML($\bar{\mathrm{S}}, \bar{\mathrm{R}}$, `E`)*. XML tree* T *is finally returned at Line 3.*

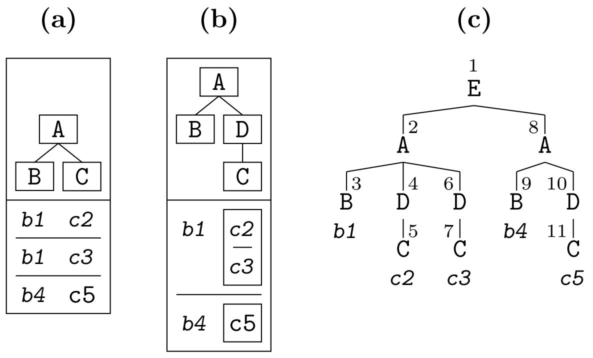


Figure 4.10: Flat and nested relation, and the XML tree obtained by algorithm FR2XML.

Vincent et al. established the important result that if a flat relation is mapped to an XML tree by first applying a sequence of nest operations and then mapping the relation directly to XML, two leaf nodes in the obtained XML tree satisfy the *closest* property iff the corresponding values appear in the same tuple in the initial relation [20]. Because algorithm FR2XML adopts the transformation procedure proposed by Vincent et al., the correspondence between leaf nodes that satisfy the *closest* property and values that appear in the same tuple also holds in the context of our transformation procedure. In order to be self-contained we now repeat this major result, which we will then use for showing that XKeys and XINDs preserve the semantics of relational keys and inclusion dependencies.

Lemma 4.1 (Closest Nodes in Generated XML Trees) Let flat relation R over relational schema S be mapped to XML tree $\mathrm{T} = (\boldsymbol{V}, \boldsymbol{E}, \mathrm{lab}, \mathrm{val})$ by FR2XML$(\mathrm{S}, \mathrm{R}, \bar{\mathrm{S}}, l)$, where l is an element label, and $\bar{\mathrm{S}}$ is a nested relational schema which is a transformation of S. Also, let $\{A_1, \ldots, A_n\} \subseteq \mathrm{att(S)}$ be a set of principal attributes in S, and let for all $i \in \{1, \ldots, n\}$, P_i be the walk in $\bar{\mathrm{S}}$ such that $\mathrm{first}(P_i) = \mathrm{root}(\bar{\mathrm{S}})$ and $\mathrm{last}(P_i) = A_i$. Then, there exist nodes $v_1, \ldots, v_n$ in T such that
(i) for all $i \in \{1, \ldots, n\}$, $v_i \in \mathrm{nodes}(l.P_i, \mathrm{T})$, and
(ii) for all $i, j \in \{1, \ldots, n\}$, $\mathrm{closest}(v_i, v_j) = \mathrm{true}$,
iff there exists tuple $t \in \mathrm{R}$ such that for all $i \in \{1, \ldots, n\}$, $t[A_i] = \mathrm{val}(v_i)$.

We do not repeat the proof of Lemma 4.1 here, but refer the interested reader to the proof of Lemma 21 in [20] instead. We now illustrate Lemma 4.1 by an example.

Example 4.13 (closest nodes in generated XML trees) *Referring to Example 4.12, consider again the transformation of relation* R*, depicted in Figure 4.10a, to XML tree* T*, depicted in Figure 4.10c. Regarding for instance the principal attributes* `B` *and* `C` *in* S*, depicted in Figure 4.10a,* `A.B` *and* `A.D.C` *are the walks in* $\bar{\mathrm{S}}$*, depicted in Figure 4.10b, that lead to* `B` *and* `C`*, respectively. Also, the custom root label* l *in the transformation of* R *to* T *is* *label* E *according to Example 4.12. So, from Lemma 4.1 there exist nodes v and v' in XML tree* T *which are reachable over paths* E.A.B *and* E.A.D.C*, respectively, and which satisfy the* closest *property, iff there exists tuple $t \in \mathrm{R}$ such that $t[\mathtt{B}] = \mathrm{val}(v)$ and $t[\mathtt{C}] = \mathrm{val}(v')$. Now, if we let $v = v_3$ and $v' = v_5$ in XML tree* T*, then $v_3 \in \mathrm{nodes}(\mathtt{E.A.B})$ and $v_5 \in \mathrm{nodes}(\mathtt{E.A.D.C})$ and also* $\mathrm{closest}(v_3, v_5) = \mathrm{true}$*. Hence, from Lemma 4.1, there exists tuple $t \in \mathrm{R}$ such that $t[\mathtt{B},\mathtt{C}] = \langle \mathtt{b1}, \mathtt{c2} \rangle$, which is obviously the case. If we let $v = v_3$ and $v' = v_{11}$ instead, then* $\mathrm{closest}(v_3, v_{11}) = \mathrm{false}$ *and in fact there does not exist tuple $t \in \mathrm{R}$ such that $t[\mathtt{B},\mathtt{C}] = \langle \mathtt{b1}, \mathtt{c5} \rangle$, which is expected according to Lemma 4.1.*

4.2.4 Transforming Relational Databases

The procedure of algorithm DB2XML is straightforward. The algorithm has input a flat database $\mathbf{R} = \{\mathrm{R}_1, \ldots, \mathrm{R}_n\}$ over a database schema $\mathbf{S} = \{\mathrm{S}_1, \ldots, \mathrm{S}_n\}$, together with a single element label l_{db}, a set of element labels $\{l_1, \ldots, l_n\}$, and a set of nested relational schemas $\{\bar{\mathrm{S}}_1, \ldots, \bar{\mathrm{S}}_n\}$ which are transformations of flat relational schemas $\{\mathrm{S}_1, \ldots, \mathrm{S}_n\}$. The algorithm first creates a trivial XML tree T at Line 1 (cf. Algorithm 4.4) for representing the entire database $\mathbf{R}$, where custom label l_{db} is the label for the root node in T. The algorithm then iterates the relations in $\mathbf{R}$ within the loop at Line 2, and creates for every relation $\mathrm{R}_i \in \mathbf{R}$ a separate tree $\bar{\mathrm{T}}_i$ that represents R_i. For this purpose algorithm FR2XML is invoked at Line 3 such that label $l_i \in \{l_1, \ldots, l_n\}$ is the label of the root node in XML tree $\bar{\mathrm{T}}_i$ returned by algorithm FR2XML. At Line 4, $\bar{\mathrm{T}}_i$ is then added to T as principal subtree. After adding a tree $\bar{\mathrm{T}}_i$ for every relation $\mathrm{R}_i \in \mathbf{R}$ to T, the algorithm terminates.

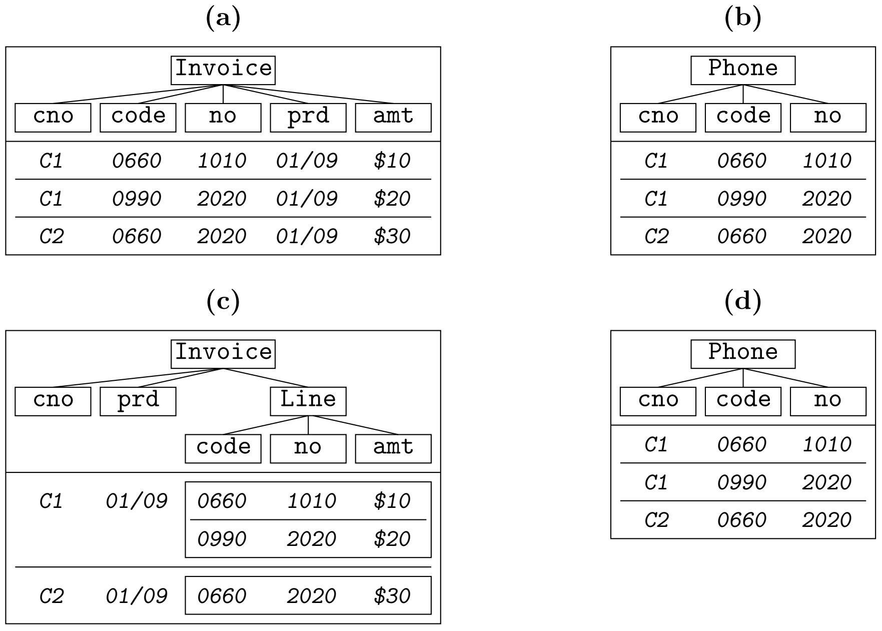
Algorithm 4.4 DB2XML - Map Relational Database to XML Tree.

```
in:   database schema $\mathbf{S} = \{\mathrm{S}_1, \ldots, \mathrm{S}_n\}$
      database $\mathbf{R} = \{\mathrm{R}_1, \ldots, \mathrm{R}_n\}$ over $\mathbf{S}$
      element label $l^{db}$
      set of element labels $\boldsymbol{l}^{rel} = \{l_1, \ldots, l_n\}$
      set of relational schemas $\{\bar{\mathrm{S}}_1, \ldots, \bar{\mathrm{S}}_n\}$ which are transformations of $\{\mathrm{S}_1, \ldots, \mathrm{S}_n\}$
out:  XML tree T representing $\mathbf{R}$

1: let $\mathrm{T} = (\boldsymbol{V}, \boldsymbol{E}, \mathrm{lab}, \mathrm{val})$ be a trivial XML tree where $\mathrm{lab}(\mathrm{root}(\mathrm{T})) = l^{db}$
2: for $i = 1$ to $n$ do
3:     $\bar{\mathrm{T}} \leftarrow \mathrm{FR2XML}(\mathrm{S}_i, \mathrm{R}_i, \bar{\mathrm{S}}_i, l_i)$
4:     $\mathrm{T} \leftarrow \mathrm{T} + \bar{\mathrm{T}}$
5: end for
6: return T.
```

We now illustrate the procedure of algorithm DB2XML in the context of our running example in the context of a phone company.

Example 4.14 (algorithm DB2XML) *Suppose that the flat relational database $\mathbf{R} = \{\mathrm{R}_1, \mathrm{R}_2\}$ over database schema $\mathbf{S} = \{\mathrm{S}_1, \mathrm{S}_2\}$ is transformed to XML by algorithm* DB2XML*, where* R_1 *and* R_2 *are relations* Invoice *and* Phone *over relational schemas* S_1 *and* S_2*, which are depicted in Figures 4.11a and 4.11b, respectively. Referring to Example 1.6,*



(a)

Invoice				
cno	code	no	prd	amt
C1	0660	1010	01/09	$10
C1	0990	2020	01/09	$20
C2	0660	2020	01/09	$30

(b)

Phone		
cno	code	no
C1	0660	1010
C1	0990	2020
C2	0660	2020

(c)

Invoice				
cno	prd	Line		
		code	no	amt
C1	01/09	0660	1010	$10
		0990	2020	$20
C2	01/09	0660	2020	$30

(d)

Phone		
cno	code	no
C1	0660	1010
C1	0990	2020
C2	0660	2020

Figure 4.11: Flat relations `Invoice` and `Phone` and the corresponding nested relations in an application of algorithm DB2XML.

it is intended that relation `Invoice` *is restructured during the transformation in contrast to relation* `Phone`*. In particular, it is expected that the invoices are grouped according to the invoice period and the addressed customer. The nested relational schemas* $\bar{\mathrm{S}}_1$ *and* $\bar{\mathrm{S}}_2$ *depicted in Figures 4.11c and 4.11d specify the intended restructuring of information in relations* `Invoice` *and* `Phone`*. Compared to* S_1*, the nested relational schema* $\bar{\mathrm{S}}_1$ *has the additional nested attribute* `Line` *which indicates that the* `code`*,* `no` *and* `amt` *of invoices which address the same customer and have the same invoice period, are to be represented as invoice lines. We note that* S_2 *equals* $\bar{\mathrm{S}}_2$ *since the information in relation* `Phone` *is not intended to be restructured during the transformation. Further, again referring to Example 1.6, the custom labels* `Invoices` *and* `Phones` *are intended as labels for the subtrees in the resulting XML tree* T *that represent entire relations* `Invoice` *and* `Phone`*, and the custom label* `Company` *is intended as the label of the root node of* T*. This transformation of* $\mathbf{R}$ *is achieved by* $\mathrm{DB2XML}(\{\mathrm{S}_1,\mathrm{S}_2\},\{\mathrm{R}_1,\mathrm{R}_2\},\texttt{Company},\{\texttt{Invoices},\texttt{Phones}\},\{\bar{\mathrm{S}}_1,\bar{\mathrm{S}}_2\})$*. In particular, initially at Line 1, XML tree* T *is a trivial XML tree, where the root node has the custom label* `Company` *assigned. Then, in the first iteration of the loop at Line 2, XML tree* $\bar{\mathrm{T}}_1$ *is obtained from relation* `Invoice` *by* $\mathrm{FR2XML}(\mathrm{S}_1,\mathrm{R}_1,\bar{\mathrm{S}}_1,\texttt{Invoices})$*, where flat relation* `Invoice` *is converted to the nested relation depicted in Figure 4.11c prior to the mapping. XML tree* $\bar{\mathrm{T}}_1$*, which is the subtree rooted at node* v_2 *in Figure 4.12, is then added as principal subtree to* T *at Line 4. In the second iteration of the loop at Line 2, XML tree* $\bar{\mathrm{T}}_2$ *is obtained from relation* `Phone` *by* $\mathrm{FR2XML}(\mathrm{S}_2,\mathrm{R}_2,\bar{\mathrm{S}}_2,\texttt{Phones})$*. Whereas flat relation* `Phone` *is converted to the nested relation depicted in Figure 4.11d prior to the mapping, it is not restructured, since* $\bar{\mathrm{S}}_1 = \mathrm{S}_1$ *and thus no nesting operation is performed. XML tree* $\bar{\mathrm{T}}_2$*, which is the subtree rooted at node* v_{23} *in Figure 4.12, is then added as principal subtree to* T *at Line 4, and XML tree* T *is finally returned at Line 6.*

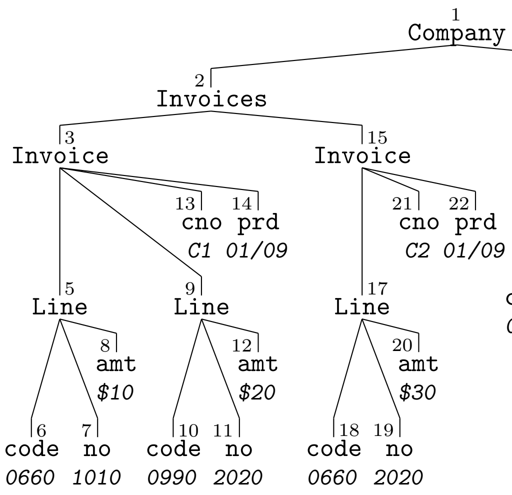


Figure 4.12: XML tree obtained from flat relations `Invoice` and `Phone` by algorithm DB2XML.

4.3 Translating Relational Semantics to XML

In this section we present our main result on preserving the semantics of relational keys and inclusion dependencies by XKeys and XINDs when a relational database is transformed to an XML tree by algorithm DB2XML. We briefly review relational keys and inclusion dependencies in Subsection 4.3.1. We then specify in Subsection 4.3.2 how XKeys and XINDs are derived from relational keys and inclusion dependencies, and we show that derived XKeys and XINDs in fact preserve the semantics or relational keys and inclusion dependencies.

4.3.1 Relational Keys and Inclusion Dependencies

A relational key asserts that every tuple in a relation is unique with respect to the values in specified attributes. Syntactically, a relational key is therefore specified as a set of attributes in a relational schema that identify the tuples in conforming relations. We now make this idea more precise.

Definition 4.12 (Relational Key) A *relational key* is defined to be

$$\{A_1, \ldots, A_n\} \rightarrow \mathrm{S},$$

where S is a relational schema and $\{A_1, \ldots, A_n\} \subseteq \text{att}(\text{S})$. A relation R over S satisfies the key $\sigma_{rel} = \{A_1, \ldots, A_n\} \rightarrow \text{S}$, denoted by $\text{R} \vDash \sigma_{rel}$, if whenever there exist tuples $t \in \text{R}$ and $t' \in \text{R}$ such that $t[A_1, \ldots, A_n] = t'[A_1, \ldots, A_n]$, then $t = t'$.

We now illustrate relational keys by an example.

Example 4.15 (relational key) *Referring to Examples 1.6 and 4.14, if we let* R *be the relation over relational schema* `Invoice` *depicted in Figure 4.11a, then the key for* R *is* $\{\texttt{prd}, \texttt{code}, \texttt{no}\} \rightarrow \texttt{Invoice}$*. This key asserts that at most one phone charge is invoiced for a certain phone in a certain period, and therefore requires that no two tuples in relation* `Invoice` *have the same values in these attributes. Relation* R *clearly satisfies this key.*

We define next a relational inclusion dependency which asserts a subset relationship between distinguished values in tuples of two not necessarily distinct relations.

Definition 4.13 (Relational Inclusion Dependency) A *relational inclusion dependency* (IND) is defined to be

$$\text{S}[A_1, \ldots, A_n] \subseteq \bar{\text{S}}[\bar{A}_1, \ldots, \bar{A}_n],$$

where S and $\bar{\text{S}}$ are relational schemas, and $[A_1, \ldots, A_n] \in \text{att}(\text{S})$ and $[\bar{A}_1, \ldots, \bar{A}_n] \in \text{att}(\bar{\text{S}})$ are lists of attributes. Relations R over S and $\bar{\text{R}}$ over $\bar{\text{S}}$ satisfy the IND $\sigma = \text{S}[A_1, \ldots, A_n] \subseteq \bar{\text{S}}[\bar{A}_1, \ldots, \bar{A}_n]$, denoted by $(\text{R}, \bar{\text{R}}) \vDash \sigma$, if for every tuple $t \in \text{R}$ there exists tuple $\bar{t} \in \bar{\text{R}}$ such that $t[A_1, \ldots, A_n] = \bar{t}[\bar{A}_1, \ldots, \bar{A}_n]$.

We now illustrate relational inclusion dependencies by an example.

Example 4.16 (relational inclusion dependency) *Referring again to Examples 1.6 and 4.14, there is the inclusion dependency* $\texttt{Invoice}[\texttt{cno}, \texttt{code}, \texttt{no}] \subseteq \texttt{Pone}[\texttt{cno}, \texttt{code}, \texttt{no}]$ *between relations* `Invoice` *and* `Phone` *depicted in Figures 4.11a and 4.11b, respectively. This inclusion dependency requires that the* `cno`*,* `code`*,* `no` *values in tuples of relation* `Invoice` *have matching counterparts in the tuples of relations* `Phone`*. Relations* `Invoice` *and* `Phone` *satisfy this inclusion dependency, since the only combinations of* `cno`*,* `code`*,* `no` *values in tuples of relation* `Invoice` *are* $\{C1, 0660, 1010\}$*, and* $\{C1, 0990, 2020\}$*, and* $\{C2, 0660, 2020\}$*, which obviously have matching counterparts in the tuples of relation* `Phone`*.*

4.3.2 Mapping Relational Constraints to XKeys and XINDs

We specify subsequently how to derive XKeys and XINDs from relational keys and inclusion dependencies. According to Definitions 4.12 and 4.13, relational keys and inclusion dependencies are syntactically specified on the basis of relational schemas and principal attributes in the relational schemas. Our intention in deriving XKeys and XINDs is to establish a correspondence between relational schemas and principal attributes in relational keys and inclusion dependencies on the one side, and selectors and fields in XKeys and XINDs on the other side. We now make this idea more precise.

Definition 4.14 (Derived Selector and Field) Let database $\mathbf{R}$ over database schema $\mathbf{S}$ be mapped to an XML tree by DB2XML$(\mathbf{S}, \mathbf{R}, l^{db}, \boldsymbol{l}^{rel}, \bar{\mathbf{S}})$, where l^{db} and $\boldsymbol{l}^{rel} = \{l_1, \ldots, l_n\}$ are element labels, and $\bar{\mathbf{S}} = \{\bar{\mathrm{S}}_1, \ldots, \bar{\mathrm{S}}_n\}$ are nested relational schemas. Then, with respect to a relational schema $\mathrm{S}_i \in \mathbf{S}$ and an attribute $A_j \in \mathrm{att}(\mathrm{S}_i)$,

(i) the selector obtained for S_i, denoted by $\mathrm{path}(\mathrm{S}_i)$, is defined to be

$$\mathrm{path}(\mathrm{S}_i) = l^{db}.l_i.\,\mathrm{root}(\bar{\mathrm{S}}_i).$$

(ii) the field obtained for A_j, denoted by $\mathrm{path}(\mathrm{S}_i, A_j)$, is defined to be

$$\mathrm{path}(\mathrm{S}_i, A_j) = \bar{A}_1.\cdots.\bar{A}_n,$$

where $\mathrm{root}(\bar{\mathrm{S}}_i).\bar{A}_1.\cdots.\bar{A}_n$ is the walk in $\bar{\mathrm{S}}_i$ such that $\bar{A}_n = A_j$.

In deriving selectors and fields from relational schemas and principal attributes, we take into account the restructuring of relational schemas during the transformation of a relational database to an XML tree by algorithm DB2XML. As will be made more clear soon, this achieves that if an XKey or XIND is derived from a relational key or inclusion dependency, then the syntax of the XKey or XIND directly reflects the intention of the relational key or inclusion dependency. We now illustrate how we derive selectors and fields from relational schemas and principal attributes by an example.

Example 4.17 (derived selector and field) *Referring to Example 4.14, suppose that a selector is to be derived for the flat relational schema* S_1 *depicted in Figure 4.11a, which is the schema for relation* `Invoice`*. Then, since* l^{db} = `Customer`*, and* l_1 = `Invoices`*, and* $\mathrm{root}(\bar{\mathrm{S}}_1)$ = `Invoice`*, where* $\bar{\mathrm{S}}_1$ *is the nested relational schema depicted in Figure 4.11c,* $\mathrm{path}(\mathrm{S}_1)$ = `Customer.Invoices.Invoice`*. Suppose further that fields are to be derived for the principal attributes* `cno` *and* `code` *in* S_1*. Then, the walks in* $\bar{\mathrm{S}}_1$ *that lead to* `cno` *and* `code` *are* `Invoice.cno` *and* `Invoice.Line.code`*, respectively. Hence, given that* $\mathrm{root}(\bar{\mathrm{S}}_1)$ = `Invoice`*,* $\mathrm{path}(\mathrm{S}_1, \texttt{cno})$ = `cno` *and* $\mathrm{path}(\mathrm{S}_1, \texttt{code})$ = `Line.code`*.*

Based on the notions of derived selectors and fields, we define next how XKeys and XINDs are derived from relational keys and inclusion dependencies.

Definition 4.15 (Derived XKey and XIND) Let database $\mathbf{R}$ over database schema $\mathbf{S}$ be mapped to an XML tree by DB2XML$(\mathbf{S}, \mathbf{R}, l^{db}, \boldsymbol{l}^{rel}, \bar{\mathbf{S}})$, where l^{db} and $\boldsymbol{l}^{rel} = \{l_1^{rel}, \ldots, l_n^{rel}\}$ are element labels, and $\bar{\mathbf{S}} = \{\bar{\mathrm{S}}_1, \ldots, \bar{\mathrm{S}}_n\}$ are nested relational schemas. The XKey derived from a relational key $\{A_1, \ldots, A_n\} \to \mathrm{S}$, where $\mathrm{S} \in \mathbf{S}$, is defined to be

$$\big(\mathrm{path}(\mathrm{S}), \{\,\mathrm{path}(\mathrm{S}, A_1), \ldots, \mathrm{path}(\mathrm{S}, A_n)\}\,\big).$$

The XIND derived from a relational inclusion dependency $\mathrm{S}[A_1, \ldots, A_n] \subseteq \bar{\mathrm{S}}[\bar{A}_1, \ldots, \bar{A}_n]$, where $\mathrm{S} \in \mathbf{S}$ and also $\bar{\mathrm{S}} \in \mathbf{S}$, is defined to be

$$\big(\mathrm{path}(\mathrm{S}),\ [\,\mathrm{path}(\mathrm{S}, A_1), \ldots, \mathrm{path}(\mathrm{S}, A_n)\,]\big) \subseteq \big(\mathrm{path}(\bar{\mathrm{S}}),\ [\,\mathrm{path}(\bar{\mathrm{S}}, \bar{A}_1), \ldots, \mathrm{path}(\bar{\mathrm{S}}, \bar{A}_n)\,]\big).$$

We now illustrate XKeys and XINDs that are derived from relational keys and inclusion dependencies.

Example 4.18 (derived XKey and XIND) *Referring once more to Example 4.14, we derive the XKey* κ = (`Company.Invoices.Invoice`, {`prd`, `Line.code`, `Line.no`}) *from the relational key* κ_{rel} = {`prd`, `code`, `no`} $\rightarrow$ `Invoice` *introduced in Example 4.15. Note that the intention of* κ_{rel} *is that invoices are identified by the combinations of* `prd`, `code`, `no`, *which is precisely expressed by the derived XKey* κ. *Also, from the relational inclusion dependency* σ_{rel} = `Invoice`[`cno`, `code`, `no`] $\subseteq$ `Pone`[`cno`, `code`, `no`] *introduced in Example 4.16, we derive the XIND* σ = (`Company.Invoices.Invoice`, [`cno`, `Line.code`, `Line.no`]) $\subseteq$ (`Company.Phones.Phone`, [`cno`, `code`, `no`]). *As for XKey* κ *and the relational key* κ_{rel}, *there is also an obvious correspondence between* σ_{rel} *and the derived XIND* σ.

We now present the first main result in this chapter, which is on preserving the semantics of relational keys when relational data is transformed to XML data by algorithm DB2XML.

Theorem 4.1 (Preserving the Semantics of Keys) *If database* $\mathbf{R}$ *over database schema* $\mathbf{S}$ *is mapped to XML tree* T *by algorithm* DB2XML *and a relation* R $\in \mathbf{R}$ *satisfies the relational key* σ_{rel} *then* T *satisfies the XKey obtained from* σ_{rel}.

Referring to Examples 4.14 and 4.18, consider for the purpose of illustration relation `Invoice` depicted in Figure 4.11a which satisfies the relational key κ_{rel} = {`prd`, `code`, `no`} $\rightarrow$ `Invoice`. The XML tree T depicted in Figure 4.12, which is obtained from relations `Invoice` and `Phone` by algorithm DB2XML, satisfies the derived XKey κ = (`Company.Invoices.Invoice`, {`prd`, `Line.code`, `Line.no`}), which is expected given Theorem 4.1. We note that the converse of Theorem 4.1 is not true. That is κ_{rel} may not be a key in a relation R but the derived XKey κ can be satisfied in the generated XML tree T. This is a consequence of the fact that the nest operator results in the removal of duplicates from the attributes not being nested upon, and so the nested relation will not necessarily contain duplicate values, even if R did.

In order to show Theorem 4.1, we first establish two preliminary results. The first preliminary result is on the existence of a common ancestor node for a set of nodes that pairwise satisfy the *closest* property, and the second preliminary result is on the correspondence between tuples in a relation and maximum combinations of field nodes.

Lemma 4.2 Let $P_1, \ldots, P_n$ be paths and $v_1, \ldots, v_n$ be nodes in an XML tree T such that $\forall i \in \{1, \ldots, n\}$, $v_i \in \text{nodes}(P_i)$ and $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(v_i, v_j) = \text{true}$. Then there exists node $\bar{v}$ such that
(i) $\bar{v} \in \text{nodes}(P_1 \cap \cdots \cap P_n)$, and
(ii) $\forall i \in \{1, \ldots, n\}$, $\bar{v} \in \text{anc-or-self}(v_i)$.

Proof (Lemma 4.2) It is shown subsequently by induction over the nodes in $v_1, \ldots, v_n$ that for all $k \in \{1, \ldots, n\}$, Lemma 4.2 holds true for the subset $v_1, \ldots, v_k$.

Base Case: The set of nodes containing the single node v_1 is used as base case for the induction, i.e. $k = 1$ in the base case. Now, given that $v_1 \in \text{nodes}(P_1)$, v_1 satisfies (i) in

Lemma 4.2, and v_1 also satisfies (ii) in Lemma 4.2, since $v_1 \in \text{anc-or-self}(v_1)$ per definition, which establishes the base case.

Inductive Step: Assume now that Lemma 4.2 holds true for some k, where $k \geq 1$, and note that the inductive step is established if Lemma 4.2 also holds true for the set of nodes $v_1, \ldots, v_{k+1}$, which is shown subsequently.

Note that the inductive assumption implies the existence of node $\tilde{v}$ such that

- $\tilde{v} \in \text{nodes}(P_1 \cap \cdots \cap P_k)$, and
- $\forall i \in \{1, \ldots, k\}$, $\tilde{v} \in \text{anc-or-self}(v_i)$.

Note further that $\text{closest}(v_k, v_{k+1})$ is true per assumption, and that therefore node $\hat{v}$ exists such that

- $\hat{v} \in \text{nodes}(P_k \cap P_{k+1})$, and
- $\hat{v} \in \text{anc-or-self}(v_k)$, and
- $\hat{v} \in \text{anc-or-self}(v_{k+1})$.

Now, given that $\{\hat{v}, \tilde{v}\} \subseteq \text{anc-or-self}(v_k)$ it follows that either $\tilde{v} \in \text{anc-or-self}(\hat{v})$ or $\hat{v} \in \text{anc-or-self}(\tilde{v})$.

Suppose first that $\tilde{v} \in \text{anc-or-self}(\hat{v})$. Then $\tilde{v} \in \text{anc-or-self}(v_{k+1})$ given that $\hat{v} \in \text{anc-or-self}(v_{k+1})$, and consequently for all $i \in \{1, \ldots, k+1\}$, $\tilde{v} \in \text{anc-or-self}(v_i)$, since $\tilde{v} \in \text{anc-or-self}(v_i)$ from the inductive assumption if $i \leq k$. Hence, $\tilde{v}$ satisfies (ii) in Lemma 4.2 w.r.t. nodes $v_1, \ldots, v_{k+1}$ and it therefore remains to be shown that $\tilde{v} \in \text{nodes}(P_1 \cap \cdots \cap P_{k+1})$. Thereby, since $\tilde{v} \in \text{nodes}(P_1 \cap \cdots \cap P_k)$ per assumption, $\tilde{v} \in \text{nodes}(P_1 \cap \cdots \cap P_{k+1})$ if $P_1 \cap \cdots \cap P_k = P_1 \cap \cdots \cap P_{k+1}$, which is shown next.

Note that $P_1 \cap \cdots \cap P_{k+1} \subseteq P_1 \cap \cdots \cap P_k$, since if to the contrary $P_1 \cap \cdots \cap P_{k+1} \not\subseteq P_1 \cap \cdots \cap P_k$, then $(P_1 \cap \cdots \cap P_k) \subset (P_1 \cap \cdots \cap P_k) \cap P_{k+1}$ according to Definition 3.13, and combining this with the fact that $(P_1 \cap \cdots \cap P_k) \cap P_{k+1} \subseteq (P_1 \cap \cdots \cap P_k)$ means that $P_1 \cap \cdots \cap P_k \subset P_1 \cap \cdots \cap P_k$, which is however clearly a contradiction.

Hence $P_1 \cap \cdots \cap P_{k+1} \subseteq P_1 \cap \cdots \cap P_k$ and therefore $P_1 \cap \cdots \cap P_k = P_1 \cap \cdots \cap P_{k+1}$ if $P_1 \cap \cdots \cap P_k \subseteq P_1 \cap \cdots \cap P_{k+1}$. Thereby, given that $\tilde{v} \in \text{nodes}(P_1 \cap \cdots \cap P_k)$ and that $\hat{v} \in \text{nodes}(P_k \cap P_{k+1})$ and that $\tilde{v} \in \text{anc-or-self}(\hat{v})$, it follows that $P_1 \cap \cdots \cap P_k \subseteq P_k \cap P_{k+1}$. Combining this with the fact that $P_k \cap P_{k+1} \subseteq P_{k+1}$ shows that $P_1 \cap \cdots \cap P_k \subseteq P_{k+1}$. Consequently, for all $i \in \{1, \ldots, k+1\}$, $P_1 \cap \cdots \cap P_k \subseteq P_i$, since if $i < k+1$, then $P_1 \cap \cdots \cap P_k \subseteq P_i$ per definition. Given that for all $i \in \{1, \ldots, k+1\}$, $P_1 \cap \cdots \cap P_k \subseteq P_i$, it follows that $P_1 \cap \cdots \cap P_k \subseteq P_1 \cap \cdots \cap P_{k+1}$, since $P_1 \cap \cdots \cap P_{k+1}$ is required to be the *longest* common prefix of paths $P_1, \ldots, P_{k+1}$ per definition.

Hence, $P_1 \cap \cdots \cap P_k = P_1 \cap \cdots \cap P_{k+1}$ and thus $\tilde{v}$ also satisfies (i) in Lemma 4.2 w.r.t. nodes $v_1, \ldots, v_{k+1}$, i.e. $\tilde{v} \in \text{nodes}(P_1 \cap \cdots \cap P_{k+1})$, which establishes the inductive step for the case that $\tilde{v} \in \text{anc-or-self}(\hat{v})$.

Suppose now instead that $\hat{v} \in \text{anc-or-self}(\tilde{v})$. Then $\forall i \in \{1, \ldots, k+1\}$, $\hat{v} \in \text{anc-or-self}(v_i)$, since $\hat{v} \in \text{anc-or-self}(v_{k+1})$ as shown above, and if $i < k+1$, then $\hat{v} \in \text{anc-or-self}(v_i)$ follows from $\hat{v} \in \text{anc-or-self}(\tilde{v})$ and $\tilde{v} \in \text{ancestor}(v_i)$. Hence $\hat{v}$ satisfies (ii) in Lemma 4.2 w.r.t. nodes $v_1, \ldots, v_{k+1}$ and it therefore remains to be shown that $\hat{v} \in \text{nodes}(P_1 \cap \cdots \cap P_{k+1})$. Thereby, since $\hat{v} \in \text{nodes}(P_k \cap P_{k+1})$ per assumption, $\hat{v} \in \text{nodes}(P_1 \cap \cdots \cap P_{k+1})$ if $P_k \cap P_{k+1} = P_1 \cap \cdots \cap P_{k+1}$, which is shown next.

Analogous to the previous argumentation, $P_1 \cap \cdots \cap P_{k+1} \subseteq P_k \cap P_{k+1}$, since if instead $P_1 \cap \cdots \cap P_{k+1} \not\subseteq P_k \cap P_{k+1}$, then $P_k \cap P_{k+1} \subset P_1 \cap \cdots \cap P_k \cap P_{k+1}$ according to Definition 3.12. Combining this with the fact that $P_1 \cap \cdots \cap P_k \cap P_{k+1} \subseteq P_k \cap P_{k+1}$ then means that $P_k \cap P_{k+1} \subset P_k \cap P_{k+1}$, which is however clearly a contradiction.

Hence $P_1 \cap \cdots \cap P_{k+1} \subseteq P_k \cap P_{k+1}$ and therefore $P_k \cap P_{k+1} = P_1 \cap \cdots \cap P_{k+1}$ if $P_k \cap P_{k+1} \subseteq P_1 \cap \cdots \cap P_{k+1}$. This however follows directly from the assumptions that $\hat{v} \in \text{nodes}(P_k \cap P_{k+1})$ and that $\tilde{v} \in \text{nodes}(P_1 \cap \cdots \cap P_{k+1})$ and that $\hat{v} \subseteq \tilde{v}$.

Consequently, $P_k \cap P_{k+1} = P_1 \cap \cdots \cap P_{k+1}$ and thus $\hat{v}$ also satisfies (i) in Lemma 4.2 w.r.t. nodes $v_1, \ldots, v_{k+1}$, i.e. $\hat{v} \in \text{nodes}(P_1 \cap \cdots \cap P_{k+1})$, which establishes the inductive step also for the case that $\hat{v} \in \text{anc-or-self}(\tilde{v})$. □

We now establish the second preliminary result on the correspondence between tuples in a relation and maximum combinations of field nodes.

Lemma 4.3 (Field Nodes in Generated XML Trees) Let database $\mathbf{R}$ over database schema $\mathbf{S}$ be mapped to XML tree T by algorithm DB2XML. Also, let S be the selector obtained from any relational schema $\mathrm{S}_x \in \mathbf{S}$, and let $\{F_1, \ldots, F_n\}$ be any set of fields such that for all $i \in \{1, \ldots, n\}$, F_i is obtained from attribute $A_i \in \text{att}(\mathrm{S}_x)$. Then, there exists a selector node $v \in \text{nodes}(S, \mathrm{T})$ and a maximum combination of field nodes $\{v_1, \ldots, v_m\}$ for v with respect to $\{F_1, \ldots, F_n\}$ iff $m = n$ and there exists tuple $t \in \mathrm{R}_x$, where R_x is the relation in $\mathbf{R}$ over relational schema S_x such that for all $i \in \{1, \ldots, m\}$, $t[A_i] = \text{val}(v_i)$.

Proof (Lemma 4.3) Suppose that database $\mathbf{R} = \{\mathrm{R}_1, \ldots, \mathrm{R}_z\}$ over database schema $\mathbf{S} = \{\mathrm{S}_1, \ldots, \mathrm{S}_z\}$ was mapped to XML in particular by DB2XML($\mathbf{S}, \mathbf{R}, l^{db}, \boldsymbol{l}^{rel}, \bar{\mathbf{S}}$), where l^{db} and $\boldsymbol{l}^{rel} = \{l_1, \ldots, l_z\}$ are element labels, and $\bar{\mathbf{S}} = \{\bar{\mathrm{S}}_1, \ldots, \bar{\mathrm{S}}_z\}$ are nested relational schemas such that for all $i \in \{1, \ldots, z\}$, $\bar{\mathrm{S}}_i$ is a transformation of S_i.

We first establish a correspondence between the selector S and the fields $F_1, \ldots, F_n$ on the one side, and the paths in T on the other side. We show in particular that $\forall i \in \{1, \ldots, n\}$, $S.F_i = l^{db}.l_x.P_i$, where P_i is the walk in $\bar{\mathrm{S}}_x$ such that $\text{first}(P_i) = \text{root}(\bar{\mathrm{S}}_x)$ and $\text{last}(P_i) = A_i$. We note that the walk P_i exists in $\bar{\mathrm{S}}_x$ since per assumption $\bar{\mathrm{S}}_x$ is a transformation of S_x and $A_i \in \text{att}(S_x)$. According to (i) in Definition 4.14, $S = l^{db}.l_x.\text{root}(\bar{\mathrm{S}}_x)$ given that S is obtained from S_x, and therefore $\forall i \in \{1, \ldots, n\}$, $l^{db}.l_x.P_i = S.F_i$ if $l^{db}.l_x.P_i = l^{db}.l_x.\text{root}(\bar{\mathrm{S}}_x).F_i$. Consequently, $\forall i \in \{1, \ldots, n\}$, $l^{db}.l_x.P_i = S.F_i$ if $P_i = \text{root}(\bar{\mathrm{S}}_x).F_i$. According to (ii) in Definition 4.14, $\text{root}(\bar{\mathrm{S}}_x).F_i$ is the walk in $\bar{\mathrm{S}}_x$ such that $\text{last}(\bar{\mathrm{S}}_x.F_i) = A_i$ given that F_i is obtained from A_i. From this and the assumption that P_i is the walk in $\bar{\mathrm{S}}_x$ such that $\text{first}(P_i) = \text{root}(\bar{\mathrm{S}}_x)$ and $\text{last}(P_i) = A_i$ it follows that $P_i = \text{root}(\bar{\mathrm{S}}_x).F_i$ since obviously $\text{first}(\bar{\mathrm{S}}_x.F_i) = \text{root}(\bar{\mathrm{S}}_x)$.

<u>If:</u> We show that if there exists tuple $t \in \mathrm{R}_x$, then there exists selector node $v \in \text{nodes}(S, \mathrm{T})$ and a maximum combination of field nodes $\{v_1, \ldots, v_n\}$ for v with respect to $\{F_1, \ldots, F_n\}$ such that for all $i \in \{1, \ldots, n\}$, $\text{val}(v_i) = t[A_i]$.

From the procedure of algorithm DB2XML, relation $\mathrm{R}_x \in \mathbf{R}$ has been mapped in iteration x of the loop at Line 2 in Algorithm 4.4, and so XML tree $T_x = \text{FR2XML}(\mathrm{S}_x, \mathrm{R}_x, \bar{\mathrm{S}}_x, l_x)$ was added as principal subtree to T at Line 4 in Algorithm 4.4. Then, from Lemma 4.1, there exist nodes $v_1, \ldots, v_n$ in T_x such that

(a) $\forall i \in \{1,\ldots,n\}$, $v_i \in \text{nodes}(l_x.P_i, \text{T}_x)$;
(b) $\forall i,j \in \{1,\ldots,n\}$, $\text{closest}(v_i, v_j) = \text{true}$;
(c) $\forall i \in \{1,\ldots,n\}$, $t[A_i] = \text{val}(v_i)$.

From (a) and the observation that T_x is a principal subtree of T it follows that $\forall i \in \{1,\ldots,n\}$, $v_i \in \text{nodes}(l^{db}.l_x.P_i, \text{T})$ since $\text{lab}(\text{root}(\text{T})) = l^{db}$ according to Line 1 in Algorithm 4.4. Therefore, $\forall i \in \{1,\ldots,n\}$, $v_i \in \text{nodes}(S.F_i, \text{T})$ given that $l^{db}.l_x.P_i = S.F_i$. Now, given (a) and (b), there exists a node $\hat{v} \in \text{nodes}(S.F_1 \cap \cdots \cap S.F_1, \text{T})$ such that $\forall i \in \{1,\ldots,n\}$, $\hat{v} \in \text{anc-or-self}(v_i)$ according to Lemma 4.2. From this and the observation that $S \subset S.F_1 \cap \cdots \cap S.F_1$ since $\forall i \in \{1,\ldots,n\}$, $S \subset S.F_1$, we deduce that there exists node $v \in \text{nodes}(S, \text{T})$ such that $v \in \text{ancestor}(\hat{v})$. Consequently, $\forall i \in \{1,\ldots,n\}$, $v \in \text{ancestor}(v_i)$ given that $\hat{v} \in \text{ancestor}(v_i)$. That is, there exists selector node $v \in \text{nodes}(S, \text{T})$ such that for all $\forall i \in \{1,\ldots,n\}$, $v \in \text{ancestor}(v_i)$ and $v_i \in \text{nodes}(S.F_i, \text{T})$. Hence, nodes v and $v_1,\ldots,v_n$ satisfy (ii) in Definition 3.23. According to (b) nodes $v_1,\ldots,v_n$ also satisfy (i) in Definition 3.23, and nodes $v_1,\ldots,v_n$ trivially satisfy (iii) in Definition 3.23 given that the number of nodes in $v_1,\ldots,v_n$ equals the number of fields $F_1,\ldots,F_n$. Hence, $v_1,\ldots,v_n$ is a maximum combination of field nodes for v with respect to $F_1,\ldots,F_n$ such that, according to (c), for all $i \in \{1,\ldots,n\}$, $t[A_i] = \text{val}(v_i)$.

Only If: We show that if there exists selector node $v \in \text{nodes}(S, \text{T})$ and a maximum combination of field nodes $\{v_1,\ldots,v_m\}$ for v with respect to $\{F_1,\ldots,F_n\}$, then $m = n$ and there exists tuple $t \in \text{R}_x$ such that for all $i \in \{1,\ldots,m\}$, $t[A_i] = \text{val}(v_i)$.

According to (ii) in Definition 3.23, $\forall i \in \{1,\ldots,n\}$, $v_i \in \text{nodes}(S.F_j, \text{T})$ for some $j \in \{1,\ldots,n\}$. For the ease of presentation but without loss of generality we assume subsequently that $\forall i \in \{1,\ldots,m\}$, $v_i \in \text{nodes}(S.F_i, \text{T})$. Then, $\forall i \in \{1,\ldots,m\}$, $v_i \in \text{nodes}(l^{db}.l_x.P_i, \text{T})$ given that $S.F_i = l^{db}.l_x.P_i$. From combining this with (i) in Definition 3.23 we deduce that there exists node $v_x \in \text{nodes}(l^{db}.l_x, \text{T})$ such that $\forall i \in \{1,\ldots,m\}$, $v_x \in \text{ancestor}(v_i)$. Now, since $\forall i \in \{1,\ldots,m\}$, $v_i \in \text{nodes}(l^{db}.l_x.P_i, \text{T})$ and $v_x \in \text{ancestor}(v_i)$, and given that $v_x \in \text{nodes}(l^{db}.l_x, \text{T})$, it follows that $\forall i \in \{1,\ldots,m\}$, $v_i \in \text{nodes}(l_x.P_i, \text{T}[v_x])$, where $\text{T}[v_x]$ is the subtree of T rooted at node v_x.

Further, $T[v_x]$ is a principal subtree of T given that $v_x \in \text{nodes}(l^{db}.l_x)$, and from the procedure of algorithm DB2XML, $\text{T}[v_x]$ has been created in iteration x of the loop at Line 2 in Algorithm 4.4, i.e. $\text{T}[v_x] = \text{FR2XML}(\text{S}_x, \text{R}_x, l_x, \bar{\text{S}}_x)$. From this together with the observation above that $\forall i \in \{1,\ldots,m\}$, $v_i \in \text{nodes}(l_x.P_i, \text{T}[v_x])$, and (i) in Definition 3.23, Lemma 4.1 applies to nodes $v_1,\ldots,v_m$. Consequently, there exists tuple $t \in \text{R}_x$ such that $\forall i \in \{1,\ldots,m\}$, $t[A_i] = \text{val}(v_i)$. Also, $m = n$ since $\text{T}[v_x] = \text{FR2XML}(\text{S}_x, \text{R}_x, l_x, \bar{\text{S}}_x)$ and therefore if instead $m < n$, then R_x contains at least one incomplete tuple, which clearly contradicts our model of relational data where only complete relations are permitted. □

Having established Lemma 4.3, the proof of Theorem 4.1 is now straightforward.

Proof (Theorem 4.1) Let $\sigma_{rel} = \{A_1,\ldots,A_n\} \to \text{S}$, where S is the relational schema in $\mathbf{S}$ such that R is the relation in $\mathbf{R}$ defined over S. Also, let $\sigma = (S, \{F_1,\ldots,F_n\})$ be the XKey obtained from σ_{rel}. We show that $\text{R} \vDash \sigma_{rel} \Rightarrow \text{T} \vDash \sigma$ by showing the contrapositive that $\text{T} \nvDash \sigma \Rightarrow \text{R} \nvDash \sigma_{rel}$.

Because σ is obtained from σ_{rel}, selector S is obtained from relational schema S, as well as for all $i \in \{1, \ldots, n\}$, field F_i is obtained from attribute $A_i \in \text{att(S)}$ according to Definition 4.15. Hence, from Lemma 4.3, whenever there exists selector node $\ddot{v} \in \text{nodes}(S, \text{T})$ and a maximum combination of field nodes $\{\ddot{v}_1, \ldots, \ddot{v}_m\}$ for $\ddot{v}$ with respect to $\{F_1, \ldots, F_n\}$, then $m = n$. Consequently, from Definition 3.24 and given that $\text{T} \nvDash \sigma$, there exist selector nodes $v, v' \in \text{nodes}(S, \text{T})$ and, with respect to $\{F_1, \ldots, F_n\}$, there exist maximum combinations of field nodes $\{v_1, \ldots, v_n\}$ for v and $\{v'_1, \ldots, v'_n\}$ for v' such that

(a) for all $i \in \{1, \ldots, n\}$, $\text{val}(v_i) = \text{val}(v'_i)$, and
(b) for at least one $z \in \{1, \ldots, n\}$, $v_z \neq v'_z$.

Then, from Lemma 4.3, there exist tuples t and t' in R such that for all $i \in \{1, \ldots, n\}$, $t[A_i] = \text{val}(v_i)$ and $t'[A_i] = \text{val}(v'_i)$. Consequently, for all $i \in \{1, \ldots, n\}$, $t[A_i] = t'[A_i]$ from (a), and therefore if $t \neq t'$ then $\text{R} \nvDash \sigma_{rel}$ according to Definition 4.12. From the procedure of Algorithm DB2XML, each value in R maps to a single node, and combining this with (b) we deduce that $t \neq t'$ and thus $\text{R} \nvDash \sigma_{rel}$. □

We now present the second main result in this chapter on preserving the semantics of relational inclusion dependencies when relational data is transformed to XML data by algorithm DB2XML.

Theorem 4.2 (Preserving the Semantics of INDs) *If database* **R** *over database schema* **S** *is mapped to XML tree* T *by algorithm* DB2XML*, relations* R *and* $\bar{\text{R}}$ *in* **R** *satisfy the relational IND* σ_{rel} *iff* T *satisfies the XIND obtained from* σ_{rel}.

It is worth mentioning that unlike for the case of keys, a database satisfies a relational inclusion dependency if and only if the generated XML tree satisfies the derived XIND. For the purpose of illustration, referring to Examples 4.14 and 4.18, consider relations `Invoice` and `Phone` depicted in Figure 4.11a which satisfy the relational inclusion dependency σ_{rel} = `Invoice[cno, code, no]` ⊆ `Pone[cno, code, no]`. The XML tree T depicted in Figure 4.12, which is obtained from relations `Invoice` and `Phone` by algorithm DB2XML, satisfies the derived XIND σ = (`Company.Invoices.Invoice`, [`cno`, `Line.code`, `Line.no`]) ⊆ (`Company.Phones.Phone`, [`cno`, `code`, `no`]), which is expected given Theorem 4.2.

As for Theorem 4.1, also the subsequent proof of Theorem 4.2 is based on Lemma 4.3.

Proof (Theorem 4.2) Let $\sigma_{rel} = \text{S}[A_1, \ldots, A_n] \subseteq \bar{\text{S}}[\bar{A}_1, \ldots, \bar{A}_n]$, where S and $\bar{\text{S}}$ are the relational schemas in **S** such that R and $\bar{\text{R}}$ are the relations in **R** defined over S and $\bar{\text{S}}$. Also, let $\sigma = (S, [F_1, \ldots, F_n]) \subseteq (\bar{S}, [\bar{F}_1, \ldots, \bar{F}_n])$ be the XIND obtained from σ_{rel}.

<u>If:</u> We establish that $\text{T} \vDash \sigma \Rightarrow (\text{R}, \bar{\text{R}}) \vDash \sigma_{rel}$ by showing the contrapositive that $(\text{R}, \bar{\text{R}}) \nvDash \sigma_{rel} \Rightarrow \text{T} \nvDash \sigma$. Given that $(\text{R}, \bar{\text{R}}) \nvDash \sigma_{rel}$, there exists tuple $t \in \text{R}$ such that for no tuple $\bar{t} \in \bar{\text{R}}$, $t\,[A_1, \ldots, A_n] = \bar{t}\,[\bar{A}_1, \ldots, \bar{A}_n]$ according to Definition 4.13. Because σ is obtained from σ_{rel}, selector S is obtained from relational schema S, as well as for all $i \in \{1, \ldots, n\}$, field F_i is obtained from attribute $A_i \in \text{att(S)}$ according to Definition 4.15. Hence, from Lemma 4.3, there exists selector node $v \in \text{nodes}(S, \text{T})$ and a maximum combination of field nodes $v_1, \ldots, v_n$ for v with respect to $F_1, \ldots, F_n$ such that for all $i \in \{1, \ldots, n\}$, $t\,[A_i] = \text{val}(v_i)$. Then, from Definition 3.25, if $\text{T} \vDash \sigma$, there exists selector node $\bar{v} \in \text{nodes}(\bar{S}, \text{T})$ and a

maximum combination of field nodes $\bar{v}_1, \ldots, \bar{v}_n$ for $\bar{v}$ with respect to $\bar{F}_1, \ldots, \bar{F}_n$ such that for all $i \in \{1, \ldots, n\}$, $\mathrm{val}(\bar{v}_i) = \mathrm{val}(v_i)$. Again, because σ is obtained from σ_{rel}, selector $\bar{S}$ is obtained from relational schema $\bar{\mathrm{S}}$, as well as for all $i \in \{1, \ldots, n\}$, field $\bar{F}_i$ is obtained from attribute $\bar{A}_i \in \mathrm{att}(\bar{\mathrm{S}})$ according to Definition 4.15. Then, from Lemma 4.3, there exists tuple $\bar{t} \in \bar{\mathrm{R}}$ such that for all $i \in \{1, \ldots, n\}$, $\bar{t}\,[\bar{A}_i] = \mathrm{val}(\bar{v}_i)$. This however clearly contradicts the fact that $(\mathrm{R}, \bar{\mathrm{R}}) \nvDash \sigma_{rel}$ since for all $i \in \{1, \ldots, n\}$, $\mathrm{val}(\bar{v}_i) = \mathrm{val}(v_i)$ and thus $t\,[A_i] = \bar{t}\,[\bar{A}_i]$ given that $\bar{t}\,[\bar{A}_i] = \mathrm{val}(\bar{v}_i)$ and $t\,[A_i] = \mathrm{val}(v_i)$. Hence, $\mathrm{T} \nvDash \sigma$.

Only If: We establish that $(\mathrm{R}, \bar{\mathrm{R}}) \vDash \sigma_{rel} \Rightarrow \mathrm{T} \vDash \sigma$ by showing the contrapositive that $\mathrm{T} \nvDash \sigma \Rightarrow (\mathrm{R}, \bar{\mathrm{R}}) \nvDash \sigma_{rel}$. Because σ is obtained from σ_{rel}, selector S is obtained from relational schema S, as well as for all $i \in \{1, \ldots, n\}$, field F_i is obtained from attribute $A_i \in \mathrm{att}(\mathrm{S})$ according to Definition 4.15. Hence, from Lemma 4.3, whenever there exists selector node $\ddot{v} \in \mathrm{nodes}(S, \mathrm{T})$ and a maximum combination of field nodes $\{\ddot{v}_1, \ldots, \ddot{v}_m\}$ for $\ddot{v}$ with respect to $\{F_1, \ldots, F_n\}$, then $m = n$. Consequently, from Definition 3.25 and given that $\mathrm{T} \nvDash \sigma$, there exist selector node $v \in \mathrm{nodes}(S, \mathrm{T})$ and a maximum combination of field nodes $\{v_1, \ldots, v_n\}$ for v with respect to $\{F_1, \ldots, F_n\}$, but there do not exist selector node $\bar{v} \in \mathrm{nodes}(\bar{S}, \mathrm{T})$ and maximum combination of field nodes $\{\bar{v}_1, \ldots, \bar{v}_n\}$ for $\bar{v}$ with respect to $\{\bar{F}_1, \ldots, \bar{F}_n\}$ such that for all $i \in \{1, \ldots, n\}$, $\mathrm{val}(v_i) = \mathrm{val}(\bar{v}_i)$.

Now, from Lemma 4.3, there exists tuple $t \in \mathrm{R}$ such that for all $i \in \{1, \ldots, n\}$, $t\,[A_i] = \mathrm{val}(v_i)$. Then, from Definition 4.13, if $(R, \bar{R}) \vDash \sigma_{rel}$, there exists tuple $\bar{t} \in \bar{\mathrm{R}}$ such that $t\,[A_1, \ldots, A_n] = \bar{t}\,[\bar{A}_1, \ldots, \bar{A}_n]$. Consequently, from Lemma 4.3, there exists selector node $\bar{v} \in \mathrm{nodes}(\bar{S}, \mathrm{T})$ and maximum combination of field nodes $\{\bar{v}_1, \ldots, \bar{v}_n\}$ for $\bar{v}$ with respect to $\{\bar{F}_1, \ldots, \bar{F}_n\}$ such that for all $i \in \{1, \ldots, n\}$, $\mathrm{val}(\bar{v}_i) = \bar{t}\,[\bar{A}_i]$. This however clearly contradicts the fact that $\mathrm{T} \nvDash \sigma$ since for all $i \in \{1, \ldots, n\}$, $t\,[A_i] = \bar{t}\,[\bar{A}_i]$ and thus $\mathrm{val}(v_i) = \mathrm{val}(\bar{v}_i)$ given that $\mathrm{val}(v_i) = t\,[A_i]$ and $\mathrm{val}(\bar{v}_i) = \bar{t}\,[A_i]$. Hence, $(\mathrm{R}, \bar{\mathrm{R}}) \nvDash \sigma_{rel}$. $\square$

Part II

Reasoning

Chapter 5

The Context for Reasoning about XKeys and XINDs

Contents

This chapter presents the context for our reasoning about XKeys and XINDs in Chapters 6 and 7, respectively. Section 5.1 introduces the class of complete XML trees which we will consider in solving the implication and consistency problems related to XKeys and XINDs. The class of complete XML trees has originally been proposed by Vincent et al. [20, 25], and Section 5.1 repeats central definitions related to complete XML trees in order for this thesis to be self-contained. Section 5.2 then revisits the definitions of XKeys and XINDs in the context of complete XML trees.

5.1 The Class of Complete XML Trees

As discussed in Chapter 1, our reason for considering complete XML trees in our reasoning is that they are a natural subclass of XML trees used in 'data-centric' business applications of XML that involve regularly structured XML data [7]. We start our discussion by making the concept of a complete XML tree more precise in Subsection 5.1.1. We then present in Subsections 5.1.2 and 5.1.3 certain properties of paths and nodes in complete XML trees which we will frequently require in our reasoning.

5.1.1 Conforming and Complete XML Trees

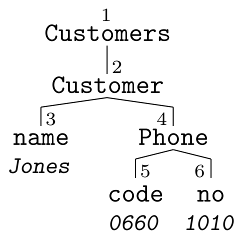
From a general point of view, before requiring the data in an XML tree to be complete, one first has to specify the structure of the information that the XML tree is expected to represent. We do this by requiring the a priori existence of a set of paths that are valid for an XML tree, and then define the notion of an XML tree conforming to such a set of paths as follows.

Definition 5.1 (Conforming XML Tree) An XML tree T *conforms* to a set of paths $\boldsymbol{P}$, if for every node v in T, if P is the path such that $v \in \text{nodes}(P)$ then $P \in \boldsymbol{P}$.

We note that the set of paths for an XML tree could be obtained from a DTD, if one exists, or from other schema information if no DTD exists. We now illustrate the conformance of an XML tree to a set of paths.

Example 5.1 (conforming XML tree) *If we use* T *to denote the XML tree in Figure 5.1a, then* T *conforms to the set of paths* $\boldsymbol{P}$ *given in Figure 5.1b since every node in* T *is reachable over a path in* $\boldsymbol{P}$*. In contrast,* T *does not conform to the set of paths* $\boldsymbol{P}' = \boldsymbol{P} - \{$`Customers.Customer.name`$\}$*, since* $v_3 \in \text{nodes}($`Customers.Customer.name`$)$ *but* `Customers.Customer.name` $\notin \boldsymbol{P}'$*.*

(a)



(b)

```
Customers
Customers.Customer
Customers.Customer.name
Customers.Customer.Phone
Customers.Customer.Phone.code
Customers.Customer.Phone.no
```

(c)

```
Customers
Customers.Customer
Customers.Customer.name
Customers.Customer.status
Customers.Customer.Phone
Customers.Customer.Phone.code
Customers.Customer.Phone.no
```

Figure 5.1: An XML tree and two sets of paths.

In order to clarify the intuition behind a complete XML tree, consider the set of paths $\boldsymbol{P}$ given in Figure 5.1c and the XML tree T in Figure 5.1a. Then T also conforms to $\boldsymbol{P}$, but we do not consider T to be complete with respect to $\boldsymbol{P}$ since the existence of path

`Customers.Customer.status` means that we expect every customer to have some status, like being a premium customer for example, and this is not satisfied for customer *`Jones`* in XML tree T. We now make this idea more precise.

Definition 5.2 (Complete XML Tree) If T is an XML tree that conforms to a set of paths $\boldsymbol{P}$, then T is defined to be *complete* with respect to $\boldsymbol{P}$ if whenever P and $\bar{P}$ are paths in $\boldsymbol{P}$ such that $P \subset \bar{P}$ and there exists node $v \in \text{nodes}(P)$, then there also exists a node $\bar{v} \in \text{nodes}(\bar{P})$ such that $v \in \text{ancestor}(\bar{v})$.

We now illustrate Definition 5.2 by an example.

Example 5.2 (complete XML tree) *Referring to Example 5.1, the XML tree* T *depicted in Figure 5.1a is not complete with respect to the set of paths* $\boldsymbol{P}$ *given in Figure 5.1c, since* $\boldsymbol{P}$ *contains path* `Customers.Customer.status` *but node* $v_2 \in$ nodes(`Customers.Customer`) *is not an ancestor of a node reachable over path* `Customers.Customer.status`. *The XML tree* T *is however complete with respect to the set of paths given in Figure 5.1b.*

Example 5.2 illustrates an important point. In the relational case, a relation conforms to exactly one relational scheme and either the relation is complete with respect to this relational scheme or not. The completeness of an XML tree is however only defined with respect to a specific set of paths and so, as we have just seen, an XML tree may conform to two different sets of paths, but may be complete with respect to one set but not the other.

5.1.2 Properties of Paths in Complete XML Trees

We now introduce a couple of properties of a set of valid paths $\boldsymbol{P}$ for a complete XML tree. First, if an XML tree is complete with respect to $\boldsymbol{P}$, then $\boldsymbol{P}$ has what we call the *downward-closed* property, which is defined as follows.

Definition 5.3 (Downward-closed Paths) A set of paths $\boldsymbol{P}$ is *downward-closed* if for every pair of paths P and P' in $\boldsymbol{P}$, if $P' \in \boldsymbol{P}$ and $P \subset P'$ then $P \in \boldsymbol{P}$.

For instance, if we use T to denote the XML tree in Figure 5.1a, then T is complete with respect to the set of paths $\boldsymbol{P}$ given in Figure 5.1b, which is downward-closed since every prefix of a path in $\boldsymbol{P}$ is also a path in $\boldsymbol{P}$. However, the set of paths $\boldsymbol{P'} = \boldsymbol{P} -$ {`Customers.Customer`} is not downward-closed, since, for instance, `Customers.Customer` $\subset$ `Customers.Customer.name` but `Customers.Customer` $\notin \boldsymbol{P'}$. Also, T is not complete with respect to $\boldsymbol{P'}$, since if T is complete with respect to a set of paths, then T must conform to this set of paths, which is obviously not the case for $\boldsymbol{P'}$.

Next, because all prefixes of paths are contained in a downward-closed set of paths $\boldsymbol{P}$, there is a distinguished path of length one which is a prefix of every path in $\boldsymbol{P}$. This path leads to the root node in XML trees that conform to $\boldsymbol{P}$, and we now make the idea of the root path in a downward-closed set of paths more precise.

Definition 5.4 (Root Path) Let $\boldsymbol{P}$ be a downward-closed set of paths, and let T be an XML tree that is complete with respect to $\boldsymbol{P}$. The *root path* in $\boldsymbol{P}$, denoted by $\rho(\boldsymbol{P})$, is defined to be

$$\rho(\boldsymbol{P}) = \{P \in \boldsymbol{P} \mid P \subseteq \bar{P} \text{ for all } \bar{P} \in \boldsymbol{P}\}.$$

We note that the root path $\rho(\boldsymbol{P})$ in a downward-closed set of paths $\boldsymbol{P}$ is of length 1. Also, we will omit to explicitly state $\boldsymbol{P}$ in denoting the root path if $\boldsymbol{P}$ is understood from the context. For instance, if we let $\boldsymbol{P}$ be the downward-closed set of paths in Figure 5.1b, then $\rho =$ `Customers`.

In general, two paths do not necessarily have an intersection path, which is the case if the paths do not start with the same label. Because the root path in a downward-closed set of paths $\boldsymbol{P}$ is a prefix of any path in $\boldsymbol{P}$, any pair of paths in $\boldsymbol{P}$ have an intersection path. This property of downward-closed paths will be useful in our reasoning, and we now make this property more precise.

Lemma 5.1 If P and $\bar{P}$ are paths in a downward-closed set of paths $\boldsymbol{P}$ then $P \cap \bar{P} \in \boldsymbol{P}$.

5.1.3 Properties of Nodes in Complete XML Trees

We now introduce a couple of properties related to nodes in a complete XML tree which satisfy the *closest* property.

Lemma 5.2 Let P_1, P_2, P_3 be paths such that $P_1 \cap P_3 \subseteq P_2 \cap P_3$, and let $v_1 \in \text{nodes}(P_1)$, $v_2 \in \text{nodes}(P_2)$, $v_3 \in \text{nodes}(P_3)$ be nodes in an XML tree T such that $\text{closest}(v_1, v_2) = \text{true}$ and $\text{closest}(v_2, v_3) = \text{true}$. Then $\text{closest}(v_1, v_3) = \text{true}$.

Proof (Lemma 5.2) Throughout this proof, let P_2^1, P_3^1 and P_3^2 denote paths $P_1 \cap P_2$, $P_1 \cap P_3$ and $P_2 \cap P_3$, respectively. Then, given that $\text{closest}(v_1, v_2) = \text{true}$, there exists node v_2^1 according to Definition 3.22 such that $v_2^1 \in \text{anc-or-self}(v_1)$, and $v_2^1 \in \text{anc-or-self}(v_2)$ and $v_2^1 \in \text{nodes}(P_2^1)$. Also, given that $\text{closest}(v_2, v_3) = \text{true}$, there exists node v_3^2 such that $v_3^2 \in \text{anc-or-self}(v_2)$, and $v_3^2 \in \text{anc-or-self}(v_3)$ and $v_3^2 \in \text{nodes}(P_3^2)$.

Further, given that $v_2^1 \in \text{anc-or-self}(v_2)$ and that $v_3^2 \in \text{anc-or-self}(v_2)$, it follows that either $v_2^1 \in \text{anc-or-self}(v_3^2)$ or that $v_3^2 \in \text{ancestor}(v_2^1)$. Assume first that $v_2^1 \in \text{anc-or-self}(v_3^2)$, which is illustrated in Figure 5.2(b). Then, in order to demonstrate that $\text{closest}(v_1, v_3) = \text{true}$, it is shown that node v_2^1 satisfies (i) - (iii) in Definition 3.22 w.r.t. nodes v_1 and v_3. In particular $v_2^1 \in \text{anc-or-self}(v_1)$ per assumption, which establishes (i) in Definition 3.22 with respect to nodes v_1 and v_3. Also, $v_2^1 \in \text{anc-or-self}(v_3)$ since $v_2^1 \in \text{anc-or-self}(v_3^2)$ per assumption and $v_3^2 \in \text{anc-or-self}(v_3)$ as shown above, which establishes (ii) in Definition 3.22 with respect to nodes v_1 and v_3.

Therefore, $\text{closest}(v_1, v_3) = \text{true}$ if $v_2^1 \in \text{nodes}(P_3^1)$. Note that if $P_2^1 = P_3^1$, which is shown next, then $v_2^1 \in \text{nodes}(P_3^1)$, since $v_2^1 \in \text{nodes}(P_2^1)$ as shown above. Thereby, $P_2^1 = P_3^1$ is established subsequently by first showing that $P_2^1 \subseteq P_3^1$ and then that $P_3^1 \subseteq P_2^1$.

Note that if P_2^1 is a common prefix of paths P_1 and P_3, then $P_2^1 \subseteq P_3^1$, since P_3^1 is required to be the longest common prefix of paths P_1 and P_3. Thereby, $P_2^1 \subseteq P_1$ follows directly from the definition of path intersection. It therefore remains to be shown that also

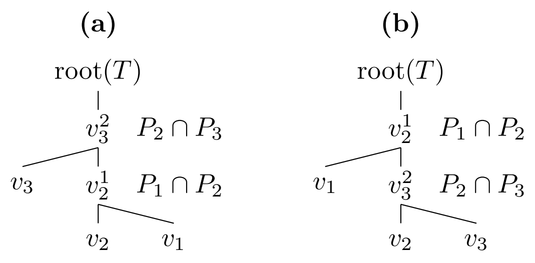


Figure 5.2: Possible constellations of nodes in Lemma 5.2.

$P_2^1 \subseteq P_3$. Thereby, $v_2^1 \in$ anc-or-self(v_3), since $v_2^1 \in$ anc-or-self(v_3^2) per assumption, and $v_3^2 \in$ anc-or-self(v_3) as show above. Given that $v_2^1 \in$ anc-or-self(v_3) it follows that $P_2^1 \subseteq P_3$, since $v_2^1 \in$ nodes(P_2^1) as shown above and $v_3 \in$ nodes(P_3) per assumption. Hence, $P_2^1 \subseteq P_3^1$.

To show that $P_3^1 \subseteq P_2^1$, it will be shown that $P_3^1 \subseteq P_1$ and $P_3^1 \subseteq P_2$, from which it follows that $P_3^1 \subseteq P_2^1$ since P_2^1 is by definition the longest common prefix of paths P_1 and P_2. First, $P_3^1 \subseteq P_1$ follows directly from the definition of path intersection. Also, $P_3^1 \subseteq P_2$ since $P_3^1 \subseteq P_3^2$ per assumption, and $P_3^2 \subseteq P_2$ according to the definition of path intersection. Hence, $P_2^1 = P_3^1$ and thus closest(v_1, v_3) is true in case that $v_2^1 \in$ anc-or-self(v_3^2).

Assume now instead that $v_3^2 \in$ ancestor(v_2^1), which is illustrated in Figure 5.2a. Then closest(v_1, v_3) = true, if v_3^2 satisfies (i) - (iii) in Definition 3.22 with respect to nodes v_1 and v_3, which is shown next. In particular $v_3^2 \in$ anc-or-self(v_1), since $v_3^2 \in$ ancestor(v_2^1) per assumption, and $v_2^1 \in$ anc-or-self(v_1) as shown above, which establishes (i) in Definition 3.22 with respect to nodes v_1 and v_3. Also, $v_3^2 \in$ anc-or-self(v_3) as shown above, which establishes (ii) in Definition 3.22 with respect to nodes v_1 and v_3.

Therefore, closest(v_1, v_3) = true if $v_3^2 \in$ nodes(P_3^1), which is shown next by showing that $P_3^2 = P_3^1$, from which it follows that $v_3^2 \in$ nodes(P_3^1) since $v_3^2 \in$ nodes(P_3^2) per assumption. Since $P_3^1 \subseteq P_3^2$ per assumption, to show that $P_3^2 = P_3^1$ it remains to verify that $P_3^2 \subseteq P_3^1$. First, $P_3^2 \subseteq P_3$ follows directly from the definition of path intersection. Next, $v_3^2 \in$ anc-or-self(v_1) since $v_3^2 \in$ ancestor(v_2^1) per assumption, and $v_2^1 \in$ anc-or-self(v_1) as shown above. Given that $v_3^2 \in$ anc-or-self(v_1) it follows that $P_3^2 \subseteq P_1$, since $v_3^2 \in$ nodes(P_3^2) and $v_1 \in$ nodes(P_1) per assumption. Thus P_3^2 is a prefix of both P_3 and P_1, and so $P_3^2 \subseteq P_1 \cap P_3$. Hence, $P_3^2 = P_3^1$ and thus closest(v_1, v_3) is true also in case that $v_3^2 \in$ anc-or-self(v_2^1), which establishes the result. □

Lemma 5.3 Let $\boldsymbol{P}$ be a downward-closed set of paths and let T be an XML tree that is complete with respect to $\boldsymbol{P}$. Also, let $\{P_1, \ldots, P_m\}$ and $\{P_{m+1}, \ldots, P_n\}$ be non-empty subsets of $\boldsymbol{P}$ and let $v_1, \ldots, v_m$ be a set of nodes in T such that
(a) $\forall i \in \{1, \ldots, m\}$, $v_i \in$ nodes(P_i), and
(b) for all $i, j \in \{1, \ldots, m\}$, closest$(v_i, v_j)$ = true.
Then, there exist nodes $v_{m+1}, \ldots, v_n$ in T such that
(i) for all $i \in \{m+1, \ldots, n\}$, $v_i \in$ nodes(P_i), and

(ii) for all $i, j \in \{1, \ldots, n\}$, closest$(v_i, v_j) =$ true.

Proof (Lemma 5.3) In order to demonstrate Lemma 5.3, the following preliminary result is established first: If $\{R_1, \ldots, R_x\} \subset \boldsymbol{P}$ and $R_{x+1} \in \boldsymbol{P}$ and there exist nodes $\bar{v}_1, \ldots, \bar{v}_x$ in T such that for all $i \in \{1, \ldots, x\}$, $\bar{v}_i \in$ nodes(R_i) and for all $i, j \in \{1, \ldots, x\}$, closest$(\bar{v}_i, \bar{v}_j)$ is true, then there exists node $\bar{v}_{x+1}$ in T such that

(A) $\bar{v}_{x+1} \in$ nodes(R_{x+1}), and
(B) for all $i, j \in \{1, x+1\}$, closest$(\bar{v}_i, \bar{v}_j) =$ true.

From Definition 3.13 and Lemma 5.1, for all $i \in \{1, \ldots, x\}$, $R_i \cap R_{x+1} \subseteq R_{x+1}$. Hence, one can choose $y \in [1, x]$ such that for all $i \in \{1, \ldots, x\}$, $R_i \cap R_{x+1} \subseteq R_y \cap R_{x+1}$. Then, since $\bar{v}_y \in$ nodes(R_y) per assumption, there exists node $\hat{v}$ such that $\hat{v} \in$ nodes$(R_y \cap R_{x+1})$, and $\hat{v} \in$ anc-or-self$(\bar{v}_y)$.

Further, $R_y \cap R_{x+1} \in \boldsymbol{P}$ since $R_y \cap R_{x+1} \subseteq R_{x+1}$ per definition, and $R_{x+1} \in \boldsymbol{P}$ and $\boldsymbol{P}$ is a downward-closed set of paths, per assumption. Therefore, given that T is complete w.r.t. $\boldsymbol{P}$ and that $\hat{v} \in$ nodes$(R_y \cap R_{x+1})$ it follows from Definition 5.2 that there exists node $\bar{v} \in$ nodes(R_{x+1}) such that $\hat{v} \in$ anc-or-self$(\bar{v}_{x+1})$, which establishes (A).

In order to verify (B), note that closest$(\bar{v}_{x+1}, \bar{v}_{x+1})$ is true trivially holds, since function closest is reflexive. It therefore remains to be shown that closest$(\bar{v}_{x+1}, \bar{v}_i)$ is true for all $i \in \{1, \ldots, x\}$, since closest$(\bar{v}_i, \bar{v}_j)$ is true for all $i, j \in \{1, \ldots, x\}$ per assumption.

It is shown first that closest$(\bar{v}_{x+1}, \bar{v}_y)$ is true. Thereby, closest$(\hat{v}, \bar{v}_y)$ is true and closest$(\hat{v}, \bar{v}_{x+1})$ is true follows directly from Definition 3.22 and the assumption that $\hat{v} \in$ anc-or-self$(\bar{v}_y)$ and $\hat{v} \in$ anc-or-self$(\bar{v}_{x+1})$, respectively. Consequently, closest$(\bar{v}_y, \bar{v}_{x+1})$ is true according to Lemma 5.2 (when choosing $P_1 = R_y$, $P_2 = (R_y \cap R_{x+1})$, $P_3 = R_{x+1}$, and $v_1 = \bar{v}_y$, $v_2 = \hat{v}$, $v_3 = \bar{v}_{x+1}$) since $R_y \cap R_{x+1} \subseteq (R_y \cap R_{x+1}) \cap R_{x+1}$ from Definition 3.13.

It is shown next that also closest$(\bar{v}_{x+1}, \bar{v}_i) =$ true for all $i \in \{1, \ldots, x\}$, where $i \neq y$. Thereby, for all $i \in \{1, \ldots, x\}$, closest$(\bar{v}_i, \bar{v}_y) =$ true per assumption and closest$(\bar{v}_y, \bar{v}_{x+1}) =$ true as shown above. Consequently, closest$(\bar{v}_{x+1}, \bar{v}_i) =$ true according to Lemma 5.2 (when choosing $P_1 = R_i$, $P_2 = R_y$, $P_3 = R_{x+1}$, and $v_1 = \bar{v}_i$, $v_2 = \bar{v}_y$, $v_3 = \bar{v}_{x+1}$) since per assumption $R_i \cap R_{x+1} \subseteq R_y \cap R_{x+1}$.

Now, Lemma 5.3 follows immediately, since one can choose for all $i \in \{m+1, \ldots, n\}$, v_i to be the node that exists according to the result just established. □

5.2 XKeys and XINDs in Complete XML Trees

We now revisit our definitions of XKeys and XINDs in the context of complete XML trees. In particular, Subsections 5.2.1 and 5.2.2 illustrate effects on the syntax and semantics of XKeys and XINDs in the context of complete XML trees.

5.2.1 Conforming XKeys and XINDs

Given that the expected information in a complete XML tree T is determined by the set of paths that the tree conforms to, it is natural to expect also that if an XKey or XIND σ is intended to apply to T, then the paths in σ should belong to the information represented by T. This leads us to the notion of XKeys and XINDs that conform to a set of paths, which we now make more precise by first defining the set of paths in an XKey or XIND.

Definition 5.5 (Paths in XKeys and XINDs) Let σ be an XKey of the general form $(S, \{F_1, \ldots, F_n\})$ or an XIND of the general form $(S, [F_1, \ldots, F_n]) \subseteq (\bar{S}, [\bar{F}_1, \ldots, \bar{F}_n])$. The set of paths in σ, denoted by paths(σ), is defined to be

$$\text{paths}(\sigma) = \begin{cases} \{S.F_1, \ldots, S.F_n\} & \sigma \text{ is an XKey} \\ \{S.F_1, \ldots, S.F_n\} \cup \{\bar{S}.\bar{F}_1, \ldots, \bar{S}.\bar{F}_n\} & \sigma \text{ is an XIND} \end{cases}$$

Using the definition of the set of paths in an XKey or XIND σ, we define next what it means for σ to conform to a set of paths $\boldsymbol{P}$.

Definition 5.6 (Conforming XKeys and XINDs) An XKey or XIND σ is defined to *conform* to a set of paths $\boldsymbol{P}$, if paths$(\sigma) \subseteq \boldsymbol{P}$. A set of XKeys or XINDs Σ conforms to a set of paths $\boldsymbol{P}$ if every $\sigma \in \Sigma$ conforms to $\boldsymbol{P}$.

We now illustrate Definition 5.6 by the example of an XKey.

Example 5.3 (conforming XKey) *Consider the XKey* σ = (`Customers.Customers`, {`Phone.code`, `Phone.no`}) *which asserts that the code and number of each phone identifies the customer who owns the phone. Applying* σ *to the XML tree* T *depicted in Figure 5.1a is plausible, given that* T *represents information about customers and their phones. In fact,* σ *conforms to the set of paths* $\boldsymbol{P}$ *depicted in Figure 5.1b which specifies the structure of* T*. In particular,* σ *conforms to* $\boldsymbol{P}$ *because* paths(σ) = {`Customers.Customers.Phone.code`, `Customers.Customers.Phone.no`} *and hence* paths$(\sigma) \subseteq \boldsymbol{P}$*. In contrast, the XKey* σ = (`Customers.Customers`, {`cno`}) *for instance does not conform to* $\boldsymbol{P}$*, which is however intuitively correct, since customer numbers* (`cno`) *are not represented in* T*, and hence cannot be used for the identification of customers.*

Next, because we require an XKey or XIND σ to conform to the set of paths $\boldsymbol{P}$ that specifies the structure of XML trees to which σ is intended to apply, the selectors of σ must start with $\rho(\boldsymbol{P})$. We now make this effect on the syntax of an XKey or XIND more precise.

Lemma 5.4 Let $\boldsymbol{P}$ be a downward-closed set of paths, and let σ be an XKey or XIND. Then, for every path $P \in$ paths(σ), first$(P) = \rho(\boldsymbol{P})$.

For instance, referring to Example 5.3, the XKey σ = (`Customers.Customers`, {`Phone.code`, `Phone.no`}) conforms to the downward-closed set of paths $\boldsymbol{P}$ depicted in Figure 5.1b, where $\rho(\boldsymbol{P})$ = `Customers` is also the first label in the paths `Customers.Customers.Phone.code` and `Customers.Customers.Phone.no` in σ.

5.2.2 Satisfaction of XKeys and XINDs in Complete XML Trees

We now review our definitions of the semantics of XKey and XINDs in the context of complete XML trees. In this regard, the peculiarity in the context of complete XML trees is that maximum combinations of field nodes are always complete, which makes checking the satisfaction of XKeys and XINDs easier. We now make this observation more precise.

Lemma 5.5 Let $\boldsymbol{P}$ be a downward-closed set of paths and let T be an XML tree that is complete with respect to $\boldsymbol{P}$. Also, let S be a selector and let $\{F_1, \dots, F_n\}$ be a set of fields such that for all $i \in \{1, \dots, n\}$, $S.F_i \in \boldsymbol{P}$. Then, a set of nodes $v_1, \dots, v_m$ is a maximum combination of field nodes for a selector node $v \in \text{nodes}(S)$ with respect to $\{F_1, \dots, F_n\}$ iff
(a) $m = n$
(b) $\forall \{1, \dots, n\}$, $v_i \in \text{nodes}(S.F_i)$, and
(c) $\forall i, j \in \{1, \dots, n\}$, $\text{closest}(v_i, v_j) = \text{true}$.

Proof (Lemma 5.5) If: It is shown that if there exists a selector node $v \in \text{nodes}(S)$ and a maximum combination of field nodes $v_1, \dots, v_m$ for v with respect to $F_1, \dots, F_n$ then nodes $v_1, \dots, v_m$ satisfy (a) - (c) in Lemma 5.5.

We establish $m = n$ by showing that if instead $m < n$, then this contradicts Lemma 5.3. Given that $m < n$, there does not exist node $\bar{v} \in \text{desc}(v)$ according to (iii) in Definition 3.23 such that

(i) $\forall i \in \{1, \dots, m\}$, $\bar{v} \neq v_i$;
(ii) $\forall i \in \{1, \dots, m\}$, $\text{closest}(v_i, \bar{v})$;
(iii) for some $y \in \{1, \dots, n\}$, $\bar{v} \in \text{nodes}(S.F_y)$.

Referring to (ii) in Definition 3.23, we now assume for ease of presentation but without loss of generality that for all $i \in \{1, \dots, m\}$, $v_i \in \text{nodes}(S.F_i)$. Then, since $\forall i \in \{1, \dots, m\}$, $S.F_i \in \boldsymbol{P}$ per assumption and $\forall i, j \in \{1, \dots, m\}$, $\text{closest}(v_i, v_j) = \text{true}$ according to (i) in Definition 3.23, Lemma 5.3 implies the existence of a node $\ddot{v} \in \text{nodes}(S.F_y)$, where $y \in \{m, \dots, n\}$, such that for all $i \in \{1, \dots, m\}$, $\text{closest}(v_i, \ddot{v}) = \text{true}$. Hence, if $\ddot{v} \in \text{desc}(v)$ then $\ddot{v}$ satisfies (ii) and (iii) above. Also, since $y \in \{m, \dots, n\}$ and $\ddot{v} \in \text{nodes}(S.F_y)$, $\ddot{v}$ satisfies (i) above, given that for all $i \in \{1, \dots, m\}$, $v_i \in \text{nodes}(S.F_i)$. That is, if $\ddot{v} \in \text{desc}(v)$ then this shows the desired contradiction. From Lemma 4.2, there exists node $\hat{v} \in \text{nodes}(S.F_y \cap S.F_1 \cap \cdots \cap S.F_m)$ such that $\hat{v} \in \text{anc-or-self}(\ddot{v})$ and $\forall i \in \{1, \dots, m\}$, $\hat{v} \in \text{anc-or-self}(v_i)$. From this and (ii) in Definition 3.23, i.e. that $\forall i \in \{1, \dots, m\}$, $v \in \text{ancestor}(v_i)$, we deduce that $v \in \text{ancestor}(\hat{v})$ since $S \subseteq (S.F_y \cap S.F_1 \cap \cdots \cap S.F_m)$ and T is a tree. Then, $\ddot{v} \in \text{desc}(v)$ follows directly from $v \in \text{ancestor}(\hat{v})$ and $\hat{v} \in \text{anc-or-self}(\ddot{v})$. Hence, nodes $v_1, \dots, v_m$ satisfy (a) in Lemma 5.5. Given that $m = n$, (ii) in Definition 3.23 implies that nodes $v_1, \dots, v_m$ satisfy (b) in Lemma 5.5, and (c) in Lemma 5.5 is given by (iii) in Definition 3.23.

Only If: It is shown that if there exist nodes $v_1, \dots, v_n$ that satisfy (a) - (c) in Lemma 5.5, then there exists selector node v and $v_1, \dots, v_n$ is a maximum combination of field nodes for v with respect to $F_1, \dots, F_n$.

Assume for the moment that there exists node $v \in \text{nodes}(S)$ such that for all $i \in \{1, \dots, n\}$, $v \in \text{ancestor}(v_i)$. Then, $v_1, \dots, v_n$ satisfy (i) and (ii) in Definition 3.23 because of (c) and (b) in Lemma 5.5, and $v_1, \dots, v_n$ trivially satisfy (iii) in Definition 3.23 given that the number of nodes $v_1, \ldots, v_n$ equals the number of fields $F_1, \ldots, F_n$. It therefore remains to be shown that there exists node $v \in \text{nodes}(S)$ such that for all $i \in \{1, \ldots, n\}$, $v \in \text{ancestor}(v_i)$.

From Lemma 4.2, there exists node $\ddot{v} \in \text{nodes}(S.F_1 \cap \cdots \cap S.F_n)$ such that for all $i \in \{1, \ldots, n\}$, $\ddot{v} \in \text{anc-or-self}(v_i)$. From this and the observation that $S \subseteq S.F_1 \cap \cdots \cap S.F_n$ we deduce that there exists node $v \in \text{nodes}(S)$ such that $v \in \text{anc-or-self}(\ddot{v})$ since T is a tree. Then, since for all $i \in \{1, \ldots, n\}$, $\ddot{v} \in \text{anc-or-self}(v_i)$ and $S \subset S.F_i$, also $v \in \text{ancestor}(v_i)$. □

As discussed at the beginning of this section, we do not need to consider the case of incomplete combinations of field nodes in our reasoning, where we only consider complete XML trees. We can even further simplify checking the satisfaction of XKeys and XINDs in complete XML trees by not explicitly considering selector nodes, because the existence of a selector node is implied by the existence of a maximum combination of field nodes. We now make these ideas more precise, and we present our result on the semantics of an XKey in the context of complete XML trees.

Lemma 5.6 (XKey Satisfaction in Complete XML Trees) Let T be an XML tree that is complete w.r.t. a downward-closed set of paths $\boldsymbol{P}$, and let $\sigma = (F, \{S_1, \ldots, S_n\})$ be an XKey conforming to $\boldsymbol{P}$. Then, $\text{T} \models \sigma$ iff whenever there exist nodes $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_n$ such that
(i) $\forall i \in \{1, \ldots, n\}$, $\{v_i, v'_i\} \subseteq \text{nodes}(S.F_i)$;
(ii) $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(v_i, v_j) = \text{closest}(v'_i, v'_j) = \text{true}$;
(iii) $\forall i \in \{1, \ldots, n\}$, $\text{val}(v_i) = \text{val}(v'_i)$,
then $\forall i \in \{1, \ldots, n\}, v_i = v'_i$.

Proof (Lemma 5.6) If: It is shown that if $\text{T} \models \sigma$, then whenever there exist nodes $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_n$ that satisfy (i) - (iii) in Lemma 5.6, then for all $i \in \{1, \ldots, n\}$, $v_i = v'_i$. We show in particular the contrapositive that if there exist nodes $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_n$ that satisfy (i) - (iii) in Lemma 5.6 and for at least one $i \in \{1, \ldots, n\}$, $v_i \neq v'_i$, then $\text{T} \not\models \sigma$.

From Lemma 5.5, there exist selector nodes $\{v, v'\} \in \text{nodes}(S)$ such that $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_n$ are maximum combinations of field nodes for v and v', respectively. Also, because of (iii) and (i) in Lemma 5.6, requirements (i) and (ii) in Definition 3.24 are satisfied, and therefore $\text{T} \not\models \sigma$ given that $v_i \neq v'_i$ for at least one $i \in \{1, \ldots, n\}$.

Only if: It is shown that if for all $i \in \{1, \ldots, n\}$, $v_i = v'_i$ whenever there exist nodes $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_n$ that satisfy (i) - (iii) in Lemma 5.6, then $\text{T} \models \sigma$. We show in particular the contrapositive that if $\text{T} \not\models \sigma$, then there exist nodes $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_n$ that satisfy (i) - (iii) in Lemma 5.6 such that for at least one $i \in \{1, \ldots, n\}$, $v_i \neq v'_i$.

Given that $\text{T} \not\models \sigma$, there exist selector nodes $\{v, v'\} \in \text{nodes}(S)$ and maximum combinations of field nodes $v_1, \ldots, v_m$ for v and $v'_1, \ldots, v'_m$ for v' with respect to $F_1, \ldots, F_n$ such that for all $i \in \{1, \ldots, m\}$, $\text{val}(v_i) = \text{val}(v'_i)$ and for at least one $i \in \{1, \ldots, m\}$, $v_i = v'_i$. From (a) in Lemma 5.5, $m = n$. Hence, because of (b) and (c) in Lemma 5.5, $v_1, \ldots, v_m$ and $v'_1, \ldots, v'_m$ satisfy (i) and (ii) in Lemma 5.6. Nodes $v_1, \ldots, v_m$ and $v'_1, \ldots, v'_m$ also satisfy (iii) in Lemma 5.6 given that $m = n$, since $i \in \{1, \ldots, m\}$, $\text{val}(v_i) = \text{val}(v'_i)$ per assumption. Finally, for at least one $i \in \{1, \ldots, m\}$, $v_i \neq v'_i$ per assumption, and so the result is established. □

As for XKeys, we also do not need to consider selector nodes or incomplete combinations of field nodes in checking XIND satisfaction in our reasoning, and we now present our result on the semantics of an XIND in the context of complete XML trees.

Lemma 5.7 (XIND Satisfaction in Complete XML Trees) Let T be an XML tree that is complete with respect to a downward-closed set of paths $\boldsymbol{P}$, and let $\sigma = (S, [F_1, \ldots, F_n]) \subseteq (S', [S'_1, \ldots, S'_n])$ be an XIND that conforms to $\boldsymbol{P}$. Then $\mathrm{T} \vDash \sigma$ iff whenever there exist nodes $v_1, \ldots, v_n$ such that
(i) for all $i \in \{1, \ldots, n\}$, $v_i \in \mathrm{nodes}(S.F_i)$, and
(ii) for all $i, j \in \{1, \ldots, n\}$, $\mathrm{closest}(v_i, v_j) = \mathrm{true}$,
then there exist nodes $v'_1, \ldots, v'_n$ such that
(i') for all $i \in \{1, \ldots, n\}$, $v'_i \in \mathrm{nodes}(S'.F'_i)$, and
(ii') for all $i, j \in \{1, \ldots, n\}$, $\mathrm{closest}(v'_i, v'_j) = \mathrm{true}$,
(iii') for all $i \in \{1, \ldots, n\}$, $\mathrm{val}(v_i) = \mathrm{val}(v'_i)$.

Proof (Lemma 5.7) If: It is shown that if $\mathrm{T} \vDash \sigma$, then whenever there exist nodes $v_1, \ldots, v_n$ that satisfy (i) and (ii) in Lemma 5.7, then there also exist nodes $v'_1, \ldots, v'_n$ that satisfy (i') - (iii') in Lemma 5.7. We show in particular the contrapositive that if there exist nodes $v_1, \ldots, v_n$ and there do not exist nodes $v'_1, \ldots, v'_n$, then $\mathrm{T} \nvDash \sigma$.

From Lemma 5.5, there exists selector node $v \in \mathrm{nodes}(S)$ such that $v_1, \ldots, v_n$ is a maximum combination of field nodes for v with respect to $F_1, \ldots, F_n$. Now, if $T \vDash \sigma$ then there exists selector node $v' \in \mathrm{nodes}(S')$ and a maximum combination of field nodes $v'_1, \ldots, v'_n$ for v' with respect to $F'_1, \ldots, F'_n$ such that for all $i \in \{1, \ldots, n\}$, $\mathrm{val}(v_i) = \mathrm{val}(v'_i)$. Then, from (b) and (c) in Lemma 5.5 there exist nodes $v'_1, \ldots, v'_n$ which satisfy (i') and (ii') in Lemma 5.7. This is however clearly a contradiction, since $v'_1, \ldots, v'_n$ also satisfy (iii') in Lemma 5.7 given that for all $i \in \{1, \ldots, n\}$, $\mathrm{val}(v_i) = \mathrm{val}(v'_i)$ if $\mathrm{T} \vDash \sigma$. Hence $\mathrm{T} \nvDash \sigma$.

Only if: It is shown that if there exist nodes $v'_1, \ldots, v'_n$ that satisfy (i') - (iii') in Lemma 5.7 whenever there exist nodes $v_1, \ldots, v_n$ that satisfy (i) and (ii) in Lemma 5.7, then $\mathrm{T} \vDash \sigma$. We show in particular the contrapositive that if $\mathrm{T} \nvDash \sigma$ then for at least one set of nodes $v_1, \ldots, v_n$ which satisfy (i) and (ii) in Lemma 5.7, there do not exist nodes $v'_1, \ldots, v'_n$ which satisfy (i') - (iii') in Lemma 5.7.

Given that $\mathrm{T} \nvDash \sigma$ there exists selector node $v \in \mathrm{nodes}(S)$ and a maximum combination of field nodes $v_1, \ldots, v_m$ for v with respect to $F_1, \ldots, F_n$, and there does not exist selector node $v' \in \mathrm{nodes}(S')$ and a maximum combination of field nodes $v'_1, \ldots, v'_m$ for v' with respect to $F'_1, \ldots, F'_n$ such that for all $i \in \{1, \ldots, m\}$, $\mathrm{val}(v_i) = \mathrm{val}(v'_i)$ and if $v_i \in \mathrm{nodes}(S.F_j)$, where $j \in \{1, \ldots, n\}$, then $v'_i \in \mathrm{nodes}(S'.F'_j)$. We now assume for ease of presentation but without loss of generality that for all $i \in \{1, \ldots, m\}$, $v_i \in \mathrm{nodes}(S.F_i)$.

From (a) in Lemma 5.5, $m = n$. Hence, because of (b) and (c) in Lemma 5.5, $v_1, \ldots, v_m$ satisfy (i) and (ii) in Lemma 5.7. Now, if there exist nodes $v'_1, \ldots, v'_m$ which satisfy (i') - (iii') in Lemma 5.7, then, from Lemma 5.5, there exists selector node $v' \in \mathrm{nodes}(S')$ such that $v'_1, \ldots, v'_m$ is a maximum combination of field nodes for v' with respect to $F_1, \ldots, F_n$. This however clearly contradicts that $\mathrm{T} \nvDash \sigma$, given that for all $i \in \{1, \ldots, n\}$, $\mathrm{val}(v_i) = \mathrm{val}(v'_i)$. Hence, $\mathrm{T} \nvDash \sigma$. □

Chapter 6

Reasoning about XKeys

Contents

This chapter presents our results on the consistency and implication problems related to XKeys in the context of complete XML trees. These important problems are addressed in Sections 6.1 and 6.2, respectively. Whereas it is straightforward to decide the consistency of XKeys, the implication problem is more involved. To solve the implication problem for XKeys, we first present a set of sound inference rules in Subsection 6.2.1 and then present a decision procedure which is based on our inference rules in Subsection 6.2.2. We finally establish the soundness and completeness of the decision procedure in Subsections 6.2.3 and 6.2.4, where we also establish completeness of our inference rules as a corollary of the completeness result for the decision procedure.

6.1 Consistency of XKeys

Intuitively, a set of XKeys is consistent if there exists at least one XML tree that satisfies the set of XKeys. This idea is now made more precise in the context of complete XML trees.

Definition 6.1 (Consistency of XKeys) A set of XKeys Σ is *consistent* if for every downward-closed set of paths $\boldsymbol{P}$ such that Σ conforms to $\boldsymbol{P}$, there exists an XML tree T which is complete with respect to $\boldsymbol{P}$ and also satisfies Σ.

It is important to note in Definition 6.1 that although the notion of a complete XML tree is only defined w.r.t. a specific set of paths, consistency is independent of a specific set of paths since it requires the existence of a complete XML tree for every set of paths that Σ conforms to. We have the following result on the consistency problem related to XKeys in the context of complete XML trees.

Theorem 6.1 (Consistency of XKeys) *Every set of XKeys is consistent.*

Proof (Theorem 6.1) The correctness of Theorem 6.1 follows from the observation that given Σ and any $\boldsymbol{P}$ that Σ conforms to, one can construct an XML tree $\mathrm{T}_{\boldsymbol{P},\Sigma}$ that is complete with respect to $\boldsymbol{P}$ and has distinct values for all leaf nodes. Then $\mathrm{T}_{\boldsymbol{P},\Sigma}$ satisfies Σ is a direct consequence of Lemma 5.6 and the fact that no two distinct leaf nodes in $\mathrm{T}_{\boldsymbol{P},\Sigma}$ have the same value. $\square$

6.2 Implication of XKeys

Intuitively, a single XKey σ is implied by a set of XKeys Σ if every XML tree that satisfies Σ also satisfies σ. We now make the implication of a single XKey by a set of XKeys more precise within the context of complete XML trees.

Definition 6.2 (Implication of XKeys) A set of XKeys Σ *implies* a single XKey σ, if for every downward-closed set of paths $\boldsymbol{P}$ such that $\Sigma \cup \{\sigma\}$ conforms to $\boldsymbol{P}$ and every XML tree T that is complete with respect to $\boldsymbol{P}$, if $\mathrm{T} \vDash \Sigma$ then $\mathrm{T} \vDash \sigma$.

As per our comment on the definition of XKey consistency, we note that our definition of XKey implication is independent of a specific set of paths that Σ and σ conform to.

On basis of Definition 6.2 we now formulate the implication problem as the question of whether $\Sigma \vDash \sigma$ is decidable. We answer this question in the following subsections. In particular, we first present a set of inference rules for XKey implication in Subsection 6.2.1, where we also establish the soundness of our inference rules. On the basis of the inference rules we then develop a decision procedure for the implication of XKeys in Subsection 6.2.2. In Subsections 6.2.3 and 6.2.4 we then show that the decision procedure is sound and complete for the implication of XKeys, i.e. we show that XKey implication is decidable.

6.2.1 Inference Rules for the Implication of XKeys

Table 6.1 gives a set of inference rules for the implication of XKeys. The symbol $\vdash$ denotes that the XKeys in the premise of a rule derive the XKeys in the conclusion of the rule. We also note that the downward-closed set of paths $\boldsymbol{P}$ that the XKeys in a rule conform to is not explicitly stated.

R1 **Upshift**
$(S, (R.F_1, \ldots, R.F_n)) \vdash (S.R, (F_1, \ldots, F_n))$

R2 **Downshift**
$(S.R, (F_1, \ldots, F_n)) \vdash (S, (R.F_1, \ldots, R.F_n))$

R3 **Split**
$(\rho, (\{F_1, \ldots, F_m\} \cup \{\bar{F}_1, \ldots, \bar{F}_n\})) \vdash (\rho, (F_1, \ldots, F_m))$
if $\rho.F_i \cap \rho.\bar{F}_j = \rho \; \forall (i,j) \in \{1, \ldots, m\} \times \{1, \ldots, n\}$

R4 **Union**
$(S, (F_1, \ldots, F_m\}) \wedge (S, (\bar{F}_1, \ldots, \bar{F}_n)) \vdash (S, (\{F_1, \ldots, F_m\} \cup \{\bar{F}_1, \ldots, \bar{F}_n\}))$

R5 **Augmentation**
$(S, (F_1, \ldots, F_m)) \wedge (S, (\{\bar{F}_1, \ldots, \bar{F}_n\} \cup \{\hat{F}_1, \ldots, \hat{F}_o\})) \vdash (S, (\{F_1, \ldots, F_m\} \cup \{\bar{F}_1, \ldots, \bar{F}_n\}))$
if $\exists k \in \{1, \ldots, m\}$ such that $S.\bar{F}_i \cap S.\hat{F}_j \subseteq S.F_k \; \forall (i,j) \in \{1, \ldots, n\} \times \{1, \ldots, o\}$

R6 **Expansion**
$(S, (F_1, \ldots, F_n\}) \vdash (S, (F_1, \ldots, F_n, \bar{F}))$
if $\text{last}(\bar{F}) \in \boldsymbol{L}^A$ and $\exists i \in \{1, \ldots, n\}$ such that $\text{parent}(S.\bar{F}) \subseteq S.F_i$

R7 **Unique Root Attribute**
$\vdash (\rho, (F))$ if $\text{last}(F) \in \boldsymbol{L}^A$ and $\text{parent}(\rho.F) = \rho$

Table 6.1: Inference rules for the implication of XKeys in complete XML trees.

We now illustrate rules R1 - R7. We note that as a consequence of achieving the identification of selector nodes by the uniqueness of field nodes, a set of field nodes in fact identifies every common ancestor node, which is accommodated by rules R1 and R2. In particular, rule R1 allows one to shift a path from the start of the fields up to the end of the selector. For instance, from the XKey (`Customers.Customer`, (`Phone.code`, `Phone.no`)), rule R1 derives (`Customers.Customer.Phone`, (`code`, `no`)), whereby the first label `Phone` in the fields has been shifted up to the end of the selector. Rule R2 is the reverse of rule R1, whereby a path from the end of the selector is shifted down to the start of the fields.

Rule R3 is a rule that, roughly speaking, allows one to derive two subkeys from a given XKey by means of splitting the fields of the given XKey in two subsets, provided that the fields in the subsets intersect only at the root path.

Further, rules R4 - R6 state three special cases where it is valid, unlike in the general case, to obtain a superkey from a given XKey. In particular, rule R4 allows one to union the fields of two given XKeys. For example, from the XKeys (`Customers.Customer`, (`Name.S`)) and (`Customers.Customer`, (`Phone.code`, `Phone.no`)), rule R4 derives the XKey (`Customers.Customer`, (`Name.S`, `Phone.code`, `Phone.no`)).

Rule R5 allows one to augment the set of fields of an XKey with a subset of the fields of another XKey. For example, given the XKeys (`Customers.Customer.Phone`, (`Serial.S`)) and (`Customers.Customer.Phone`, (`code`, `number`)), rule R5 allows to augment field `code` to the first XKey, which yields (`Customers.Customer.Phone`, (`Serial.S`, `code`)). Note that this application of rule R5 is valid, since `Customers.Customer.Phone.code` $\cap$ `Customers.Customer.Phone.no` = `Customers.Customer.Phone` and `Customers.Customer.Phone` $\subseteq$ `Customers.Customer.Phone.Serial.S`.

Rule R6 allows one to expand the set of fields of an XKey by a new field that ends in an attribute label if the parent path of this field is a prefix of at least one of the fields of the given XKey. For example, given that path `Customers.Customer.cno` ends in an attribute label and that (`Customers.Customer`, (`Phone.code`, `Phone.no`)) is an XKey, one can obtain the XKey (`Customers.Customer`, (`Phone.code`, `Phone.no`, `cno`)) since parent(`Customers.Customer.cno`) = `Customers.Customer` and `Customers.Customer` $\subseteq$ `Customers.Customer.Phone.code`. We note that rule R6 accommodates the restriction of the XML specification [5] on element nodes to have at most one child per attribute label (cf. (4.ii) in Definition 3.11). A consequence of this restriction is that every attribute node of the root node of a tree is unique, which is finally accommodated by rule R7.

It is also worth noting that our inference rules only contain a restricted type of superkey rule in the form of Rules R4 - R6, rather than an unrestricted superkey rule as in the relational case. That is, $(S, (F_1, \dots, F_n)) \not\vdash (S, (F_1, \dots, F_n, F'))$ for an arbitrary field F', whereas in the relational case if a set of attributes X is a key, then so is $X \cup Y$ for an arbitrary set of attributes Y. The reason for this difference is that the identification and uniqueness properties of a key coincide in the relational case, but not for XML which allows duplicate nodes. So, referring to the running example of the phone company, although the code and number of a phone identifies a customer, this does not necessarily imply that every combination of `code` and `no` and an additional text node, lets say a customer's address, is unique simply because in general a customer may have duplicate addresses. We note however that the absence of an unrestricted superkey rule does not contradict Theorem 4.1, since we consider the implication of XKeys in complete XML trees, which are a more general class of XML trees then those generated from complete relations.

We now denote by $\Sigma \vdash \sigma$ if there is a derivation sequence of σ from Σ using inference rules R1 - R7 in Table 6.1. We have the following result on the soundness of rules R1 - R7.

Theorem 6.2 (Soundness of Inference Rules for XKeys) *Given a set of XKeys Σ and a single XKey σ, if $\Sigma \vdash \sigma$ then $\Sigma \models \sigma$.*

Proof (Theorem 6.2) Rule R1: We show that given a downward-closed set of paths $\boldsymbol{P}$ and a conforming set of XKeys $\Sigma \cup \{\sigma\}$ where

- $\sigma = (S.R, (F_1, \dots, F_n))$, and
- $(S, (R.F_1, \dots, R.F_n)) \in \Sigma$,

there does not exist an XML tree T that is complete w.r.t. $\boldsymbol{P}$ such that T $\models \Sigma$ and T $\not\models \sigma$. For this purpose assume to the contrary that T $\not\models \sigma$. Then there exist nodes $v_1, \dots, v_n$ and $v'_1, \dots, v'_n$ such that

- $\forall i \in \{1, \ldots, n\}$, $\{v_i, v'_i\} \subseteq$ nodes($S.R.F_i$), and
- $\forall i, j \in \{1, \ldots, n\}$, closest($v_i, v_j$) is true and closest($v'_i, v'_j$) is true, and
- $\forall i \in \{1, \ldots, n\}$, val($v_i$) = val($v'_i$), and
- for at least one $i \in \{1, \ldots, n\}$, $v_i \neq v'_i$.

Then however $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_n$ clearly violate $(S, (R.F_1, \ldots, R.F_n))$ according to Lemma 5.6. Hence T $\not\models \Sigma$, which is a contradiction.

Rule R2: We show that given a downward-closed set of paths $\boldsymbol{P}$ and a conforming set of XKeys $\Sigma \cup \{\sigma\}$ where

- $\sigma = (S, (R.F_1, \ldots, R.F_n))$, and
- $(S.R, (F_1, \ldots, F_n)) \in \Sigma$,

there does not exist an XML tree T that is complete w.r.t. $\boldsymbol{P}$ such that T $\models \Sigma$ and T $\not\models \sigma$. For this purpose assume to the contrary that T $\not\models \sigma$. Then there exist nodes $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_n$ such that

- $\forall i \in \{1, \ldots, n\}$, $\{v_i, v'_i\} \subseteq$ nodes($S.R.F_i$), and
- $\forall i, j \in \{1, \ldots, n\}$, closest($v_i, v_j$) is true and closest($v'_i, v'_j$) is true, and
- $\forall i \in \{1, \ldots, n\}$, val($v_i$) = val($v'_i$), and
- for at least one $i \in \{1, \ldots, n\}$, $v_i \neq v'_i$.

Then however nodes $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_n$ clearly violate $(S.R, (F_1, \ldots, F_n))$ according to Lemma 5.6. Hence T $\not\models \Sigma$, which is a contradiction.

Rule R3: We show that given a downward-closed set of paths $\boldsymbol{P}$ and a conforming set of XKeys $\Sigma \cup \{\sigma\}$ where

- $\sigma = (\rho, (F_1, \ldots, F_m))$ and
- $(\rho, (\{F_1, \ldots, F_m\} \cup \{\bar{F}_1, \ldots, \bar{F}_n\})) \in \Sigma$ such that $\rho.F_i \cap \rho.\bar{F}_j = \rho$ $\forall (i, j) \in \{1, \ldots, m\} \times \{1, \ldots, n\}$,

there does not exist an XML tree T that is complete w.r.t. $\boldsymbol{P}$ such that T $\models \Sigma$ and T $\not\models \sigma$. For this purpose assume to the contrary that T $\not\models \sigma$. Then there exist nodes $v_1, \ldots, v_m$ and $v'_1, \ldots, v'_m$ such that

- $\forall i \in \{1, \ldots, m\}$, $\{v_i, v'_i\} \subseteq$ nodes($\rho.F_i$), and
- $\forall i, j \in \{1, \ldots, m\}$, closest($v_i, v_j$) is true and closest($v'_i, v'_j$) is true, and
- $\forall i \in \{1, \ldots, m\}$, val($v_i$) = val($v'_i$), and
- for at least one $i \in \{1, \ldots, m\}$, $v_i \neq v'_i$.

Note that Lemma 5.3 applies to the sets of paths $\{\rho.F_1, \ldots, \rho.F_m\}$ and $\{\rho.\bar{F}_1, \ldots, \rho.\bar{F}_n\}$ and the set of nodes $v_1, \ldots, v_m$, given that $v_i \in$ nodes($\rho.F_i$) for all $i \in \{1, \ldots, m\}$ and closest(v_i, v_j) is true for all $i, j \in \{1, \ldots, m\}$. Hence, there exist nodes $\hat{v}_1, \ldots, \hat{v}_n$ such that

- $\forall i \in \{1, \ldots, n\}$, $\hat{v}_i \in$ nodes($\rho.\bar{F}_i$), and
- $\forall i, j \in \{1, \ldots, n\}$, closest($\hat{v}_i, \hat{v}_j$) is true, and
- $\forall (i, j) \in \{1, \ldots, m\} \times \{1, \ldots, n\}$, closest($v_i, \hat{v}_j$) is true.

Consequently, if also closest$(v'_i, \hat{v}_j)$ is true for all $(i,j) \in \{1,\dots,m\} \times \{1,\dots,n\}$, then the sets of nodes $v_1,\dots,v_m,\hat{v}_1,\dots,\hat{v}_n$ and $v'_1,\dots,v'_m,\hat{v}_1,\dots,\hat{v}_n$ violate $(\rho, (\{F_1,\dots,F_m\} \cup \{\bar{F}_1,\dots,\bar{F}_n\}))$ in T, which establishes the desired contradiction that $\mathrm{T} \not\models \Sigma$ if $\mathrm{T} \not\models \sigma$. It is easily verified that for all $(i,j) \in \{1,\dots,m\} \times \{1,\dots,n\}$, closest$(v'_i, \hat{v}_j)$ is true since $v_\rho \in$ anc-or-self(v'_i) and also $v_\rho \in$ anc-or-self$(\hat{v}_j)$, and further $v_\rho \in$ nodes$(\rho.F_i \cap \rho.\bar{F}_j)$, given that $\rho.F_i \cap \rho.\bar{F}_j = \rho$.

Rule R4: We show that given a downward-closed set of paths $\boldsymbol{P}$ and a conforming set of XKeys $\Sigma \cup \{\sigma\}$ where

- $\sigma = (S, (\{F_1,\dots,F_m\} \cup \{\bar{F}_1,\dots,\bar{F}_n\}))$, and
- $\{(S,(F_1,\dots,F_m))\} \cup \{(S,(\bar{F}_1,\dots,\bar{F}_n))\} \subseteq \Sigma$,

there does not exist an XML tree T that is complete w.r.t. $\boldsymbol{P}$ such that $\mathrm{T} \models \Sigma$ and $\mathrm{T} \not\models \sigma$. For this purpose assume to the contrary that $\mathrm{T} \not\models \sigma$. Then there exist nodes $\{v_1,\dots,v_m\} \cup \{\bar{v}_1,\dots,\bar{v}_n\}$ and $\{v'_1,\dots,v'_m\} \cup \{\bar{v}'_1,\dots,\bar{v}'_n\}$ such that

- $\forall i \in \{1,\dots,m\}$, $\{v_i, v'_i\} \subseteq$ nodes$(S.F_i)$ and $\forall i \in \{1,\dots,n\}$, $\{\bar{v}_i, \bar{v}'_i\} \subseteq$ nodes$(S.\bar{F}_i)$, and
- $\forall i,j \in \{1,\dots,m\}$, closest$(v_i,v_j) =$ closest$(v'_i,v'_j) =$ true, and
- $\forall i,j \in \{1,\dots,n\}$, closest$(\bar{v}_i,\bar{v}_j) =$ closest$(\bar{v}'_i,\bar{v}'_j) =$ true, and
- $\forall (i,j) \in \{1,\dots,m\} \times \{1,\dots,n\}$, closest$(v_i,\bar{v}_j) =$ closest$(v'_i,\bar{v}'_j) =$ true, and
- $\forall i \in \{1,\dots,m\}$, val$(v_i) =$ val(v'_i) and $\forall i \in \{1,\dots,n\}$, val$(\bar{v}_i) =$ val$(\bar{v}'_i)$, and
- for at least one $i \in \{1,\dots,m\}$, $v_i \neq v'_i$ and/or for at least one $i \in \{1,\dots,n\}$, $\bar{v}_i \neq \bar{v}'_i$.

However, if $v_i \neq v'_i$ for some $i \in \{1,\dots,m\}$ then nodes $v_1,\dots,v_m$ and $v'_1,\dots,v'_m$ violate $(S,(F_1,\dots,F_m))$ in T. Also, if $\bar{v}_i \neq \bar{v}'_i$ for some $i \in \{1,\dots,n\}$ then nodes $\bar{v}_1,\dots,\bar{v}_n$ and $\bar{v}'_1,\dots,\bar{v}'_n$ violate $(S,(\bar{F}_1,\dots,\bar{F}_n))$ in T. Consequently, if $\mathrm{T} \not\models \sigma$ then this contradicts the assumption that $\mathrm{T} \models \Sigma$.

Rule R5: We show that given a downward-closed set of paths $\boldsymbol{P}$ and a conforming set of XKeys $\Sigma \cup \{\sigma\}$ where

- $\sigma = (S, (\{F_1,\dots,F_m\} \cup \{\bar{F}_1,\dots,\bar{F}_n\}))$, and
- $\{(S,(F_1,\dots,F_m))\} \cup \{(S,(\{\bar{F}_1,\dots,\bar{F}_n\} \cup \{\hat{F}_1,\dots,\hat{F}_o\}))\} \subseteq \Sigma$, and
- for all $(i,j) \in \{1,\dots,n\} \times \{1,\dots,o\}$ and at least one $k \in \{1,\dots,m\}$, $S.\bar{F}_i \cap S.\hat{F}_j \subseteq S.F_k$,

there does not exist a tree T that is complete w.r.t. $\boldsymbol{P}$ such that $\mathrm{T} \models \Sigma$ but $\mathrm{T} \not\models \sigma$. For this purpose assume to the contrary that $\mathrm{T} \not\models \sigma$. Then there exist sets of nodes $\{v_1,\dots,v_m\} \cup \{\bar{v}_1,\dots,\bar{v}_n\}$ and $\{v'_1,\dots,v'_m\} \cup \{\bar{v}'_1,\dots,\bar{v}'_n\}$ such that

- $\forall i \in \{1,\dots,m\}$, $\{v_i, v'_i\} \subseteq$ nodes$(S.F_i)$ and $\forall i \in \{1,\dots,n\}$, $\{\bar{v}_i, \bar{v}'_i\} \subseteq$ nodes$(S.\bar{F}_i)$, and
- $\forall i,j \in \{1,\dots,m\}$, closest$(v_i,v_j) =$ closest$(v'_i,v'_j) =$ true, and
- $\forall i,j \in \{1,\dots,n\}$, closest$(\bar{v}_i,\bar{v}_j) =$ closest$(\bar{v}'_i,\bar{v}'_j) =$ true, and
- $\forall (i,j) \in \{1,\dots,m\} \times \{1,\dots,n\}$, closest$(v_i,\bar{v}_j) =$ closest$(v'_i,\bar{v}'_j) =$ true , and
- $\forall i \in \{1,\dots,m\}$, val$(v_i) =$ val(v'_i) and $\forall i \in \{1,\dots,m\}$, val$(\bar{v}_i) =$ val$(\bar{v}'_i)$, and
- for at least one $i \in \{1,\dots,m\}$, $v_i \neq v'_i$ and/or for at least one $i \in \{1,\dots,n\}$, $\bar{v}_i \neq \bar{v}'_i$.

Note that nodes $v_1,\dots,v_m$ and $v'_1,\dots,v'_m$ violate $(S,(F_1,\dots,F_m))$ if $v_i \neq v'_i$ for at least one $i \in \{1,\dots,m\}$. Hence, $\mathrm{T} \not\models \sigma \Rightarrow \mathrm{T} \not\models \Sigma$ if $v_i \neq v'_i$ for at least one $i \in \{1,\dots,m\}$.

Assume now instead that $\bar{v}_i \neq \bar{v}'_i$ for at least one $i \in \{1,\ldots,n\}$. In order to verify that T $\not\models \Sigma$ also in this case, note that T is complete w.r.t. $\boldsymbol{P}$ per assumption, and that $\{S.F_k\} \cup \{S.\hat{F}_1,\ldots,S.\hat{F}_o\} \subseteq \boldsymbol{P}$, since both XKeys $(S,(F_1,\ldots,F_m))$ and $(S,(\bar{F}_1,\ldots,\bar{F}_n\} \cup \{\hat{F}_1,\ldots,\hat{F}_o\}))$ conform to $\boldsymbol{P}$ per assumption. Therefore, Lemma 5.3 applies to the unary set of nodes $\{v_k\} \subseteq \{v_1,\ldots,v_m\}$ and the sets of paths $\{S.F_k\}$ and $\{S.\hat{F}_1,\ldots,S.\hat{F}_o\}$. Consequently, there exist nodes $\hat{v}_1,\ldots,\hat{v}_o$ in T such that

- $\forall i \in \{1,\ldots,o\}$, $\hat{v}_i \in$ nodes($S.\hat{F}_i$), and
- $\forall i \in \{1,\ldots,o\}$, closest($v_k,\hat{v}_i$) is true, and
- $\forall i,j \in \{1,\ldots,o\}$, closest($\hat{v}_i,\hat{v}_j$) is true.

Observe that the sets of nodes $\{\bar{v}_1,\ldots,\bar{v}_n\} \cup \{\hat{v}_1,\ldots,\hat{v}_o\}$ and $\{\bar{v}'_1,\ldots,\bar{v}'_n\} \cup \{\hat{v}_1,\ldots,\hat{v}_o\}$ achieve the following properties:

- $\forall i \in \{1,\ldots,n\}$, $\{\bar{v}_i,\bar{v}'_i\} \subseteq$ nodes($S.\bar{F}_i$) and $\forall i \in \{1,\ldots,o\}$, $\hat{v}_i \in$ nodes($S.\hat{F}_i$), and
- $\forall i,j \in \{1,\ldots,n\}$, closest($\bar{v}_i,\bar{v}_j$) = closest($\bar{v}'_i,\bar{v}'_j$) = true, and
- $\forall i,j \in \{1,\ldots,o\}$, closest($\hat{v}_i,\hat{v}_j$) = true, and
- $\forall i \in \{1,\ldots,m\}$, val($\bar{v}_i$) = val($\bar{v}'_i$).

Therefore, the sets of nodes $\{\bar{v}_1,\ldots,\bar{v}_n\} \cup \{\hat{v}_1,\ldots,\hat{v}_o\}$ and $\{\bar{v}'_1,\ldots,\bar{v}'_n\} \cup \{\hat{v}_1,\ldots,\hat{v}_o\}$ violate $(S,(\{\bar{F}_1,\ldots,\bar{F}_n\} \cup \{\hat{F}_1,\ldots,\hat{F}_o\}))$ in T if closest($\bar{v}_i,\hat{v}_j$) = closest($\bar{v}'_i,\hat{v}_j$) = true for all $(i,j) \in \{1,\ldots,n\} \times \{1,\ldots,o\}$, since val($\hat{v}_i$) = val($\hat{v}_i$) trivially holds true for all $i \in \{1,\ldots,o\}$, and $\bar{v}_i \neq \bar{v}'_i$ for at least one $i \in \{1,\ldots,n\}$ per assumption.

In order to verify that closest($\bar{v}_i,\hat{v}_j$) = closest($\bar{v}'_i,\hat{v}_j$) = true for all $(i,j) \in \{1,\ldots,n\} \times \{1,\ldots,o\}$ recall first that

- $\forall j \in \{1,\ldots,o\}$, closest($\hat{v}_j,v_k$) is true, and
- $\forall i \in \{1,\ldots,m\}$, closest($\bar{v}_i,v_k$) is true, and
- $\forall i \in \{1,\ldots,m\}$, closest($\bar{v}'_i,v_k$) is true given that closest($\bar{v}'_i,v'_k$) is true and $v'_k = v_k$.

Hence, from Lemma 5.2, closest($\bar{v}_i,\hat{v}_j$) is true (where $P_1 = S.\bar{F}_i$, $P_2 = S.F_k$, $P_3 = S.\hat{F}_j$ and $v_1 = \bar{v}_i$, $v_2 = v_k$, $v_3 = \hat{v}_j$) and also that closest($\bar{v}'_i,\hat{v}_j$) is true (where $P_1 = S.\bar{F}_i$, $P_2 = S.F_k$, $P_3 = S.\hat{F}_j$ and $v_1 = \bar{v}'_i$, $v_2 = v_k$, $v_3 = \hat{v}_j$), if $S.\bar{F}_i \cap S.\hat{F}_j \subseteq S.F_k \cap S.\hat{F}_j$, which is what we show next.

Thereby, $S.\bar{F}_i \cap S.\hat{F}_j \subseteq S.F_k$ per assumption and $S.\bar{F}_i \cap S.\hat{F}_j \subseteq S.\hat{F}_j$ according to Definition 3.13. Hence $S.\bar{F}_i \cap S.\hat{F}_j$ is a common prefix of paths $S.F_k$ and $\hat{F}_j$. Clearly, $S.F_k \cap S.\hat{F}_j$ is also a common prefix of paths $S.F_k$ and $\hat{F}_j$, and therefore either $S.\bar{F}_i \cap S.\hat{F}_j \subseteq S.F_k \cap S.\hat{F}_j$ or $S.F_k \cap S.\hat{F}_j \subset S.\bar{F}_i \cap S.\hat{F}_j$.

If $S.F_k \cap S.\hat{F}_j \subset S.\bar{F}_i \cap S.\hat{F}_j$ then however $S.F_k \cap S.\hat{F}_j$ is not the *longest* common prefix of paths $S.F_k$ and $\hat{F}_j$, since length($S.F_k \cap S.\hat{F}_j$) < length($S.\bar{F}_i \cap S.\hat{F}_j$) given that $S.F_k \cap S.\hat{F}_j \subset S.\bar{F}_i \cap S.\hat{F}_j$. Therefore, if $S.F_x \cap S.\hat{F}_j \subset S.\bar{F}_i \cap S.\hat{F}_j$ then this contradicts Definition 3.13 and this case is therefore excluded. Hence, $S.\bar{F}_i \cap S.\hat{F}_j \subseteq S.F_x \cap S.\hat{F}_j$ and thus closest($\bar{v}_i,\hat{v}_j$) = closest($\bar{v}'_i,\hat{v}_j$) = true for all $i,j \in \{1,\ldots,n\} \times \{1,\ldots,o\}$, which establishes the result.

Rule R6: We show that given a downward-closed set of paths $\boldsymbol{P}$ and a conforming set of XKeys $\Sigma \cup \{\sigma\}$ where

- $\sigma = (S, (F_1, \ldots, F_n, \bar{F}))$, and
- $(S, (F_1, \ldots, F_n)) \in \Sigma$, and
- $\text{last}(\bar{F}) \in \boldsymbol{L}^A$ and $\text{parent}(S.\bar{F}) \subseteq S.F_k$ for at least one $k \in \{1, \ldots, n\}$,

there does not exist an XML tree T that is complete w.r.t. $\boldsymbol{P}$ such that $\text{T} \vDash \Sigma$ but $\text{T} \nvDash \sigma$. For this purpose assume to the contrary that $\text{T} \nvDash \sigma$. Then there exist sets of nodes $v_1, \ldots, v_n, \bar{v}$ and $v'_1, \ldots, v'_n, \bar{v}'$ such that

- $\forall i \in \{1, \ldots, n\}$, $\{v_i, v'_i\} \subseteq \text{nodes}(S.F_i)$ and $\{\bar{v}_i, \bar{v}'_i\} \subseteq \text{nodes}(S.\bar{F})$, and
- $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(v_i, v_j) = \text{closest}(v'_i, v'_j) = \text{true}$, and
- $\forall i \in \{1, \ldots, n\}$, $\text{closest}(v_i, \bar{v}) = \text{closest}(v'_i, \bar{v}') = \text{true}$, and
- $\forall i \in \{1, \ldots, n\}$, $\text{val}(v_i) = \text{val}(v'_i)$ and $\text{val}(\bar{v}) = \text{val}(\bar{v}')$, and
- for at least one $i \in \{1, \ldots, n\}$, $v_i \neq v'_i$ and/or $\bar{v} \neq \bar{v}'$.

Note that if $v_i \neq v'_i$ for at least one $i \in \{1, \ldots, n\}$, then nodes $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_n$ violate $(S, (F_1, \ldots, F_n))$ in T, which clearly contradicts the assumption that $\text{T} \vDash \Sigma$.

If instead $v_i = v'_i$ for all $i \in \{1, \ldots, n\}$ then $\bar{v} \neq \bar{v}'$, given that $\text{T} \nvDash \sigma$. In order to verify that this situation may not occur, let $S.F_k$ be a path in $\{S.F_1, \ldots, S.F_n\}$ such that $\text{parent}(S.\bar{F}) \subseteq S.F_k$. Note that at least one such path exists per assumption. Then, since $\text{closest}(v_k, \bar{v}) = \text{closest}(v'_k, \bar{v}') = \text{true}$ per assumption, there exist nodes $\hat{v}$ and $\hat{v}'$ according to Definition 3.22 such that

- $\{\hat{v}, \hat{v}'\} \subseteq \text{nodes}(S.F_k \cap S.\bar{F})$, and
- $\hat{v} \in \text{anc-or-self}(v_k)$ and $\hat{v}' \in \text{anc-or-self}(v'_k)$, and
- $\hat{v} \in \text{anc-or-self}(\bar{v})$ and $\hat{v}' \in \text{anc-or-self}(\bar{v}')$.

However, given that $v_i = v'_i$ for all $i \in \{1, \ldots, n\}$, also $v_k = v'_k$. Consequently, $\{\hat{v}, \hat{v}'\} \in \text{anc-or-self}(v_k)$ and hence $\hat{v} = \hat{v}'$, given that $\{\hat{v}, \hat{v}'\} \subseteq \text{nodes}(S.F_k \cap S.\bar{F})$ and that $v_k \in \text{nodes}(S.F_k)$. Further, $S.F_k \cap S.\bar{F} = \text{parent}(S.\bar{F})$ since per assumption $\text{parent}(S.\bar{F}) \subseteq S.F_k$ and $\text{last}(S.\bar{F})$ is an attribute label. Therefore, $\text{parent}(\bar{v}) = \text{parent}(\bar{v}') = \hat{v}$ follows from the assumptions that $\hat{v} \in \text{anc-or-self}(\bar{v})$, and that $\hat{v} \in \text{anc-or-self}(\bar{v}')$, and that $\hat{v} \in \text{nodes}(S.F_k \cap S.\bar{F})$. This however clearly contradicts (ii) in Definition 3.11, since $\bar{v}$ and $\bar{v}'$ are *distinct* attribute nodes and $\text{lab}(\bar{v}) = \text{lab}(\bar{v}') = \text{last}(S.\bar{F})$, given that $\{\bar{v}, \bar{v}'\} \in \text{nodes}(S.\bar{F})$ and that $\text{last}(S.\bar{F})$ is an attribute label. Hence, if $\hat{v} \neq \hat{v}'$ then T does not conform to the tree model in Definition 3.11, which establishes the contradiction that $\text{T} \nvDash \Sigma$ if $\text{T} \nvDash \sigma$.

Rule R7: We show that given a downward-closed set of paths $\boldsymbol{P}$ and a conforming XKey $\sigma = (\rho, (F))$, where $\text{last}(F) \in \boldsymbol{L}^A$ and $\text{parent}(\rho.F) = \rho$, there does not exist a tree T that is complete w.r.t. $\boldsymbol{P}$ such that $\text{T} \nvDash \sigma$. For this purpose assume to the contrary that $\text{T} \nvDash \sigma$. Then there exists nodes v, v' such that

- $\{v, v'\} \subseteq \text{nodes}(\rho.F)$, and
- $\text{closest}(v, v') = \text{true}$, and
- $\text{val}(v) = \text{val}(v')$, and
- $v \neq v'$.

Then $\text{parent}(v) = \text{parent}(v') = v_\rho$ given that $\text{parent}(\rho.F) = \rho$ and that $\{v, v'\} \subseteq \text{nodes}(\rho.F)$. This however clearly contradicts (ii) in Definition 3.11, since v and v' are *distinct* attribute nodes and $\text{lab}(v) = \text{lab}(v') = \text{last}(\rho.F)$, given that $\{v, v'\} \subseteq \text{nodes}(\rho.F)$ and that $\text{last}(F) \in \boldsymbol{L}^A$. Hence, T does not conform to Definition 3.11 if $\text{T} \nvDash \sigma$. Consequently, $\text{T} \vDash \sigma$. □

6.2.2 A Decision Procedure for XKey Implication

We now introduce algorithm DXKI, which is a decision procedure for the implication of XKeys in complete XML trees and is based on inference rules R1 - R7. Algorithm DXKI takes as input a set of XKeys Σ and a single XKey σ, and incrementally computes the largest subkey[1] $\hat{\sigma}$ of σ that is implied by Σ. The paths in $\hat{\sigma}$ are represented by the set $\boldsymbol{X}$, which is initially the empty set (cf. Line 1 in Algorithm 6.1). The algorithm adds paths from $\text{paths}(\sigma)$ to $\boldsymbol{X}$ at Lines 4, 7, and 10, and the algorithm terminates at Line 12 when no more paths from $\text{paths}(\sigma)$ can be added to $\boldsymbol{X}$. Then algorithm DXKI returns true at Line 12 if $\boldsymbol{X} = \text{paths}(\sigma)$, i.e. $\Sigma \vDash \sigma$, and false otherwise. Note that paths are never removed from $\boldsymbol{X}$ once added, and that $\text{paths}(\sigma)$, from which the paths in $\boldsymbol{X}$ are taken, is finite. Consequently, the number of paths in $\boldsymbol{X}$ is bounded from above by the number of paths in $\text{paths}(\sigma)$ and since during every iteration of the loop at Line 2 either the number of paths in $\boldsymbol{X}$ strictly increases or the loop exits, it follows that algorithm DXKI always terminates.

Algorithm 6.1 DXKI - Decide XKey Implication.

```
in:   a set of XKeys $\Sigma$ and a single XKey $\sigma$
out:  true if $\Sigma \vDash \sigma$ and false if $\Sigma \nvDash \sigma$
1:  $\boldsymbol{X} \leftarrow \emptyset$;
2:  repeat
3:      if $\exists \bar{X} \in \boldsymbol{X} \cup \{\rho\}$ and $R \in \text{paths}(\sigma) - \boldsymbol{X}$ such that $\text{last}(R) \in \boldsymbol{L}^A \wedge \text{parent}(R) \subseteq \bar{X}$ then
4:          $\boldsymbol{X} \leftarrow \boldsymbol{X} \cup \{R\}$;
5:      end if
6:      if $\exists \bar{\sigma} \in \Sigma$ such that $\text{paths}(\bar{\sigma}) \subseteq \text{paths}(\sigma) - \boldsymbol{X}$ then
7:          $\boldsymbol{X} \leftarrow \boldsymbol{X} \cup \text{paths}(\bar{\sigma})$;
8:      end if
9:      if $\exists \bar{X} \in \boldsymbol{X} \cup \{\rho\}$ and $\bar{\sigma} \in \Sigma$ and $\{R_1, \ldots, R_u\} \subseteq \text{paths}(\bar{\sigma}) \cap \text{paths}(\sigma) - \boldsymbol{X}$ such that
            $\forall R_i \in \{R_1, \ldots, R_u\}$ and $\forall \bar{R} \in \text{paths}(\bar{\sigma}) - \{R_1, \ldots, R_u\}$, $R_i \cap \bar{R} \subseteq \bar{X}$ then
10:         $\boldsymbol{X} \leftarrow \boldsymbol{X} \cup \{R_1, \ldots, R_u\}$;
11:     end if
12: until no more change to $\boldsymbol{X}$ is possible;
13: return $(\boldsymbol{X} = \text{paths}(\sigma))$;
```

We now illustrate the algorithm by an example.

Example 6.1 (algorithm DXKI) *Let $\Sigma = \{\sigma_1, \sigma_2\}$ be a set of XKeys, where $\sigma_1 = (\rho, (\texttt{A.B.C.S, D.E.S}))$ and $\sigma_2 = (\rho.\texttt{A}, (\texttt{B.E.S,B.D.S}))$, and let $\sigma = (\rho.\texttt{A}, (\texttt{B.C.S, B.E.S, a}))$ be a single XKey. Thereby,* `a` *is an attribute label and all other labels, with the exception of* `S`*, are element labels. We first note that* $\text{paths}(\sigma_1) = \{\rho.\texttt{A.B.C.S}, \rho.\texttt{D.E.S}\}$*, and that* $\text{paths}(\sigma_2) =$

[1]This means that the selector in $\hat{\sigma}$ is the same as the selector in σ, and the fields in $\hat{\sigma}$ are a subset of the fields in σ.

$\{\rho.\texttt{A.B.E.S}, \rho.\texttt{A.B.D.S}\}$*, and that* $\text{paths}(\sigma) = \{\rho.\texttt{A.B.C.S}, \rho.\texttt{A.B.E.S}, \rho.\texttt{A.a}\}$*. Consider then the operation of algorithm* DXKI*. Initially,* $\boldsymbol{X} = \emptyset$ *and so the first time that Line 3 is executed, the only path in* $\boldsymbol{X} \cup \{\rho\}$ *is* ρ *and so* $\bar{X} = \rho$*. So the test at Line 3 fails because there is no path* $R \in \text{paths}(\sigma)$ *such that* $\text{last}(R) \in \boldsymbol{L}^A$ *and* $\text{parent}(R) \subseteq \bar{X}$*. Hence the first time that Line 6 is executed,* $\boldsymbol{X} = \emptyset$ *and so the test at Line 6 fails because* $\text{paths}(\sigma_1) \not\subseteq \text{paths}(\sigma)$ *and* $\text{paths}(\sigma_2) \not\subseteq \text{paths}(\sigma)$*. Thus the first time that Line 9 is executed,* $\bar{X} = \rho$*. Also,* $\text{paths}(\sigma) - \boldsymbol{X} = \text{paths}(\sigma)$ *and so if we let* $\bar{\sigma} = \sigma_1$ *and* $R_1 = \rho.\texttt{A.B.C.S}$*, then* R_1 *satisfies the test at Line 9 if we let* $\bar{R} = \rho.\texttt{D.E.S}$ *since* $R_1 \cap \bar{R} = \rho$ *and* $\rho \subseteq \bar{X}$ *given that* $\bar{X} = \rho$*. Thus the test succeeds and so* $\boldsymbol{X} = \{\rho.\texttt{A.B.C.S}\}$ *at the end of the first iteration of the loop at Line 2.*

The second time that Line 3 is executed, $\boldsymbol{X} \cup \{\rho\} = \{\rho.\texttt{A.B.C.S}, \rho\}$ *and so the test succeeds when we let* $\bar{X} = \rho.\texttt{A.B.C.S}$ *and* $R = \rho.\texttt{A.a}$*. Thus* $\rho.\texttt{A.a}$ *is added to* $\boldsymbol{X}$*, i.e.* $\boldsymbol{X} = \{\rho.\texttt{A.B.C.S}, \rho.\texttt{A.a}\}$*. The test at Line 6 then fails since* $\text{paths}(\sigma_1) \not\subseteq \{\rho.\texttt{A.B.E.S}\}$ *and* $\text{paths}(\sigma_2) \not\subseteq \{\rho.\texttt{A.B.E.S}\}$*. So when Line 9 is next executed,* $\boldsymbol{X} = \{\rho.\texttt{A.B.C.S}, \rho.\texttt{A.a}\}$ *and so the test returns true when we let* $\bar{X} = \rho.\texttt{A.B.C.S}$*,* $\bar{\sigma} = \sigma_2$*,* $R_1 = \rho.\texttt{A.B.E.S}$ *and* $\bar{R} = \rho.\texttt{A.B.C.S}$*. Thus* $\rho.\texttt{A.B.E.S}$ *is added to* $\boldsymbol{X}$ *which becomes* $\boldsymbol{X} = \{\rho.\texttt{A.B.C.S}, \rho.\texttt{A.B.E.S}, \rho.\texttt{A.a}\}$*. Algorithm* DXKI *then terminates at Line 12, and returns true since* $\boldsymbol{X} = \text{paths}(\sigma)$*.*

6.2.3 Soundness of the Decision Procedure

We show subsequently that algorithm DXKI is sound, i.e. that if algorithm DXKI returns true for an input set of XKeys Σ and a single XKey σ, then $\Sigma \models \sigma$. To establish the soundness of algorithm DXKI, we first establish an important preliminary result.

Lemma 6.1 Let Σ be a set of XKeys, and let σ be a single XKey. If $\text{DXKI}(\Sigma, \sigma) = \text{true}$ then $\sigma \in \Sigma$ or $\Sigma \vdash \sigma$.

We now illustrate this result by an example and then present the proof of Lemma 6.1.

Example 6.2 *Consider the computation of* $\text{DXKI}(\Sigma, \sigma)$ *as given in Example 6.1. Initially* $\boldsymbol{X} = \emptyset$*. The first time that a path is added to* $\boldsymbol{X}$ *is at Line 9 when* $\rho.\texttt{A.B.C.S}$ *is added. In this case,* $\bar{X} = \rho$ *and so* $\Sigma \vdash (\rho.\texttt{A}, (\texttt{B.C.S}))$ *follows from rule* R3 *and* σ_1*. The next time that a path is added to* $\boldsymbol{X}$ *is at Line 3, when* $\rho.\texttt{A.a}$ *is added. In this case* $\Sigma \vdash (\rho.\texttt{A}, (\texttt{B.C.S}, \texttt{a}))$ *follows from Rule 6 and the previously derived XKey* $(\rho.\texttt{A}, (\texttt{B.C.S}))$*, since* $\text{parent}(\rho.\texttt{A.a}) = \rho.\texttt{A} \subseteq \rho.\texttt{A.B.C.S}$*. The final time that a path is added to* $\boldsymbol{X}$ *is at Line 9 when* $\rho.\texttt{A.B.E.S}$ *is added. In this case* $\Sigma \vdash (\rho.\texttt{A}, (\texttt{B.C.S}, \texttt{B.E.S}, \texttt{a})) = \sigma$ *follows from rule* R5 *and the previously derived XKey* $(\rho.\texttt{A}, (\texttt{B.C.S}, \texttt{a}))$*.*

We note that rules R1 *and* R2 *are also needed, but we do not explicitly show the application in this example for reasons of brevity. We also note that in the case where the test at Line 3 evaluates to true when* $\bar{X} = \rho$*, then soundness follows from Rule* R7 *and Rule* R4*. If the test at Line 6 evaluates to true, then soundness follows from the fact that* $\bar{\sigma} \in \Sigma$ *and rule* R4*, and if the test at Line 9 evaluates to true when* $\bar{X} = \rho$*, then soundness follows from* R3 *followed by* R4*.*

Proof (Lemma 6.1) Throughout this proof it is assumed that σ is of the general form $\sigma = (S, (F_1, \ldots, F_n))$. Referring to the XKey $\bar{\sigma}$ at Lines 6 and 9 in Algorithm 6.1, it is assumed that $\bar{\sigma}$ is of the form $\bar{\sigma} = (\bar{S}, (\bar{F}_1, \ldots, \bar{F}_m))$. Hence

$$\begin{aligned}\text{paths}(\sigma) &= \{S.F_1, \ldots, S.F_n\}\\ \text{paths}(\bar{\sigma}) &= \{\bar{S}.\bar{F}_1, \ldots, \bar{S}.\bar{F}_m\}.\end{aligned}$$

We observe that a set of paths is added to the set of paths $\boldsymbol{X}$ (cf. Line 1 in Algorithm 6.1) within the run DXKI(Σ, σ), then these paths are taken from paths(σ) (cf. Lines 4, 7, 10 in Algorithm 6.1). We therefore claim that whenever a set of paths $\{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\} \subseteq$ paths(σ) is added to $\boldsymbol{X}$ then Σ contains or derives the XKey $\hat{\sigma}$ such that

$$\hat{\sigma} = (S, (\{F_{\phi(1)}, \ldots, F_{\phi(y)}\} \cup \{F_{\varphi(1)}, \ldots, F_{\varphi(x)}\})),$$

where $\{S.F_{\phi(1)}, \ldots, S.F_{\phi(y)}\}$ are the paths such that $\boldsymbol{X} = \{S.F_{\phi(1)}, \ldots, S.F_{\phi(y)}\}$ before paths $\{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\}$ are added. We simply refer to this claim as *the* claim subsequently.

In order to demonstrate the claim, it is first established that if a set of paths $\{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\} \subseteq \sigma$ is added to $\boldsymbol{X}$ within the run DXKI(Σ, σ)

(A) at Line 4 and the condition at Line 3 is satisfied because $\bar{X} = \rho$, or
(B) at Line 7, or
(C) at Line 10 and the condition at Line 9 is satisfied because $\bar{X} = \rho$,

then Σ contains or derives the XKey

$$(S, (F_{\varphi(1)}, \ldots, F_{\varphi(x)})). \tag{6.1}$$

(A) Since paths $\{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\}$ are added to $\boldsymbol{X}$ at Line 4 per assumption, $\{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\}$ contains exactly one path. Hence (6.1) is of the form

$$(S, (F_{\varphi(1)})) \tag{6.2}$$

and it is therefore shown next that $\Sigma \vdash$ (6.2). Thereby, given that $S.F_{\varphi(1)}$ is added to $\boldsymbol{X}$ at Line 4, path $S.F_{\varphi(1)}$ satisfies the condition at Line 3 and therefore parent$(S.F_{\varphi(1)}) \subseteq \bar{X}$. This together with the assumption that $\bar{X} = \rho$ implies that parent$(S.F_{\varphi(1)}) = \rho$. Also, last$(S.F_{\varphi(1)})$ is an attribute label, given that the condition at Line 3 is satisfied, and therefore rule R7 (Unique Root Attribute) applies to path $S.F_{\varphi(1)}$. Hence, $\Sigma \vdash$ (6.2). Note that $S = \rho$ if parent$(S.F_{\varphi(1)}) = \rho$.

(B) Given that the set of paths $\{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\}$ is added to $\boldsymbol{X}$ at Line 7, the condition at Line 6 is satisfied. Thus, there exists XKey $\bar{\sigma} \in \Sigma$ such that paths$(\bar{\sigma}) \subseteq$ paths$(\sigma) - \boldsymbol{X}$ and paths$(\bar{\sigma}) = \{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\}$. Consequently, $\{\bar{S}.\bar{F}_{\theta(1)}, \ldots, \bar{S}.\bar{F}_{\theta(m)}\} = \{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\}$ since paths$(\bar{\sigma}) = \{\bar{S}.\bar{F}_{\theta(1)}, \ldots, \bar{S}.\bar{F}_{\theta(m)}\}$ per assumption, and it is therefore assumed subsequently without loss of generality that $\bar{S}.\bar{F}_{\theta(i)} = S.F_{\varphi(i)}$ for all $i \in \{1, \ldots, x\}$. Note that $x = m$. Now if $\bar{S} = S$, $\bar{\sigma}$ equals (6.1) and therefore (6.1) $\in \Sigma \vee \Sigma \vdash$ (6.1) since $\bar{\sigma} \in \Sigma$ per assumption. If instead $\bar{S} \neq S$, then either $S \subset \bar{S}$ or $\bar{S} \subset S$, given that $\bar{S}.\bar{F}_{\theta(i)} = S.F_{\varphi(i)}$ for all $i \in \{1, \ldots, x\}$. Suppose first that $S \subset \bar{S}$ and choose R such that $S.R = \bar{S}$. Then, $\bar{\sigma}$ is of the form

$$(S.R, (\bar{F}_{\theta(1)}, \ldots, \bar{F}_{\theta(x)})). \tag{6.3}$$

Note that applying rule R2 (Downshift) to (6.3) for path R yields (6.1) since for all $i \in \{1,\ldots,x\}$, $R.\bar{F}_{\theta(i)} = F_{\varphi(i)}$, given that $\bar{S} = S.R$ and that $\bar{S}.\bar{F}_{\theta(i)} = S.F_{\varphi(i)}$. Hence, (6.1) $\in \Sigma \vee \Sigma \vdash$ (6.1) in case that $S \subset \bar{S}$.

If instead $\bar{S} \subset S$, let $\tilde{R}$ be the path such that $\bar{S}.\tilde{R} = S$. Then, for all $i \in \{1,\ldots,x\}$, $S.F_{\varphi(i)} = \bar{S}.\tilde{R}.F_{\varphi(i)}$, and therefore $\bar{S}.\bar{F}_{\theta(i)} = \bar{S}.\tilde{R}.F_{\varphi(i)}$ since $\bar{S}.\bar{F}_{\theta(i)} = S.F_{\varphi(i)}$ per assumption. Consequently, $\bar{F}_{\theta(i)} = \tilde{R}.F_{\varphi(i)}$ for all $i \in \{1,\ldots,x\}$ and therefore $\bar{\sigma}$ is of the form

$$(\bar{S}, (\tilde{R}.F_{\varphi(1)},\ldots,\tilde{R}.F_{\varphi(x)})). \tag{6.4}$$

Note that applying rule R1 (Upshift) to (6.4) for path $\tilde{R}$ yields (6.1) since $\bar{S}.\tilde{R} = S$ per assumption. Hence, (6.1) $\in \Sigma \vee \Sigma \vdash$ (6.1) also in case that $\bar{S} \subset S$.

(C) Given that the set of paths $\{S.F_{\varphi(1)},\ldots,S.F_{\varphi(x)}\}$ is added to $\boldsymbol{X}$ at Line 10, the condition at Line 9 is satisfied and therefore $\{S.F_{\varphi(1)},\ldots,S.F_{\varphi(x)}\} \subseteq \text{paths}(\bar{\sigma}) \cap \text{paths}(\sigma) - \boldsymbol{X}$. Consequently, $\{S.F_{\varphi(1)},\ldots,S.F_{\varphi(x)}\} \subseteq \text{paths}(\bar{\sigma})$ and thus there exist the subsets $\{\bar{S}.\bar{F}_{\theta(1)},\ldots,\bar{S}.\bar{F}_{\theta(x)}\}$ and $\{\bar{S}.\bar{F}_{\vartheta(1)},\ldots,\bar{S}.\bar{F}_{\vartheta(z)}\}$ of paths($\bar{\sigma}$) such that

$$\{\bar{S}.\bar{F}_{\theta(1)},\ldots,\bar{S}.\bar{F}_{\theta(x)}\} = \{S.F_{\varphi(1)},\ldots,S.F_{\varphi(x)}\}$$
$$\{\bar{S}.\bar{F}_{\vartheta(1)},\ldots,\bar{S}.\bar{F}_{\vartheta(z)}\} = \text{paths}(\bar{\sigma}) - \{S.F_{\varphi(1)},\ldots,S.F_{\varphi(x)}\}$$

Now, note that $\text{paths}(\bar{\sigma}) = \{\bar{S}.\bar{F}_{\theta(1)},\ldots,\bar{S}.\bar{F}_{\theta(x)}\} \cup \{\bar{S}.\bar{F}_{\vartheta(x+1)},\ldots,\bar{S}.\bar{F}_{\vartheta(z)}\}$, and that therefore $\bar{\sigma}$ is of the form

$$(\bar{S}, (\{\bar{F}_{\theta(1)},\ldots,\bar{F}_{\theta(x)}\} \cup \{\bar{F}_{\vartheta(1)},\ldots,\bar{F}_{\vartheta(z)}\})). \tag{6.5}$$

Further, given that the condition at Line 9 is satisfied, $\bar{S}.\bar{F}_{\theta(i)} \cap \bar{S}.\bar{F}_{\vartheta(j)} \subseteq \bar{X}$ for all $(i,j) \in \{1,\ldots,x\} \times \{1,\ldots,z\}$. From this together with the assumption that $\bar{X} = \rho$, it follows that $\bar{S}.\bar{F}_{\theta(i)} \cap \bar{S}.\bar{F}_{\vartheta(j)} = \rho$ for all $(i,j) \in \{1,\ldots,x\} \times \{1,\ldots,z\}$. Consequently, $\bar{S} = \rho$ and therefore rule R3 (Split) applies to (6.5) which yields

$$(\rho,(\bar{F}_{\theta(1)},\ldots,\bar{F}_{\theta(x)})) \text{ and} \tag{6.6}$$
$$(\rho,(\bar{F}_{\vartheta(1)},\ldots,\bar{F}_{\vartheta(z)})). \tag{6.7}$$

Recall that $\{\bar{S}.\bar{F}_{\theta(1)},\ldots,\bar{S}.\bar{F}_{\theta(x)}\} = \{S.F_{\varphi(1)},\ldots,S.F_{\varphi(x)}\}$ and assume, without loss of generality, that for all $i \in \{1,\ldots,x\}$, $\bar{S}.\bar{F}_{\theta(i)} = S.F_{\varphi(i)}$. Then (6.6) equals (6.1) if $S = \rho$, since $\bar{S} = \rho$ as shown above. Hence $\Sigma \vdash$ (6.1) in case that $S = \rho$. If instead $S \neq \rho$, then there exists path R such that for all $i \in \{1,\ldots,x\}$, $\rho.\bar{F}_{\theta(i)} = \rho.R.F_{\varphi(i)}$, since $\bar{S}.\bar{F}_{\theta(i)} = \rho.\bar{F}_{\theta(i)} = S.F_{\varphi(i)}$ per assumption. Consequently, (6.6) is of the form

$$(\rho, (R.F_{\varphi(1)},\ldots,R.F_{\varphi(x)})) \tag{6.8}$$

Note that rule R1 (Upshift) applies to (6.8) for path R, which yields (6.1) since $\rho.R = S$, given that $\rho.R.F_{\varphi(i)} = S.F_{\varphi(i)}$ for all $i \in \{1,\ldots,x\}$. Hence, $\Sigma \vdash$ (6.1) also in case that $S \neq \rho$, which establishes the result.

Based on this preliminary result, the claim is now demonstrated by induction over the sets of paths that are added to $\boldsymbol{X}$ within the run DXKI(Σ, σ).

Base Case: The first time where a set of paths $\{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\}$ is added to $\boldsymbol{X}$ within the run DXKI(Σ, σ) is used as base case for the induction. Note that according to Line 1, $\boldsymbol{X} = \emptyset$ before a set of paths is added to $\boldsymbol{X}$ for the first time. Consequently, if the conditions at Lines 3 or 9 are satisfied then $\bar{X} \notin \boldsymbol{X}$ and therefore $\bar{X} = \rho$. Further, given that $\boldsymbol{X} = \emptyset$, also $\{S.F_{\phi(1)}, \ldots, S.F_{\phi(y)}\} = \emptyset$ since $\{S.F_{\phi(1)}, \ldots, S.F_{\phi(y)}\} = \boldsymbol{X}$ per assumption. Therefore, $\hat{\sigma}$ is of the form

$$(S, (F_{\varphi(1)}, \ldots, F_{\varphi(x)})) \tag{6.9}$$

in the base case, and it is therefore shown that (6.9) $\in \Sigma \vee \Sigma \vdash$ (6.9).

This is however easily verified, since if paths $\{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\}$ are added to $\boldsymbol{X}$ at Line 4, 7, or 10 then (A), (B) or (C) applies, respectively, given that $\bar{X} = \rho$, and obviously (6.9) equals (6.1). Hence, (6.9) $\in \Sigma \vee \Sigma \vdash$ (6.9) follows from the preliminary result that (6.1) $\in \Sigma \vee \Sigma \vdash$ (6.1).

Inductive Step: Assume now that the claim holds true until a certain, positive number of sets of paths were added to $\boldsymbol{X}$. The inductive step is then established by showing that the claim also holds true for the next set of paths that is added to $\boldsymbol{X}$.

Note that $S.F_{\phi(1)}, \ldots, S.F_{\phi(y)}$ are the paths that have already been added to $\boldsymbol{X}$ within the run of DXKI(Σ, σ) per assumption. Consequently, the inductive assumption implies that Σ contains or derives the XKey

$$(S, (F_{\phi(1)}, \ldots, F_{\phi(y)})). \tag{6.10}$$

Observe that if paths $S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}$ are added to $\boldsymbol{X}$ at Line 4, 7, or 10 and the conditions at Lines 3 and 9 are satisfied because $\bar{X} = \rho$, then (A), (B), or (C) applies, respectively, and $\hat{\sigma}$ equals the XKey that results from applying rule R4 (Union) to (6.1) and (6.10). Hence, $\hat{\sigma} \in \Sigma \vee \Sigma \vdash \hat{\sigma}$ if paths $S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}$ are added to $\boldsymbol{X}$ at Line 4, 7, or 10 and the conditions at Lines 3 and 9 are satisfied because $\bar{X} = \rho$.

In order to establish the inductive step it therefore remains to be shown that $\hat{\sigma} \in \Sigma \vee \Sigma \vdash \hat{\sigma}$ if paths $S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}$ are added to $\boldsymbol{X}$ at Line 4 or 10 and the conditions at Lines 3 or 9 are satisfied for some path $\bar{X} \in \boldsymbol{X}$. Suppose first that paths $S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}$ are added to $\boldsymbol{X}$ at Line 4. Then $S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}$ contains exactly one path and therefore $\hat{\sigma}$ is of the form

$$(S, (\{F_{\phi(1)}, \ldots, F_{\phi(y)}\} \cup \{F_{\varphi(1)}\})). \tag{6.11}$$

It has to be shown therefore that (6.11) $\in \Sigma \vee \Sigma \vdash$ (6.11). Thereby, given that the condition at Line 3 is satisfied, last$(S.F_{\varphi(1)})$ is an attribute label and parent$(S.F_{\varphi(1)}) \subseteq \bar{X}$. From this together with the observation that $\bar{X} \in$ paths((6.10)), since $\boldsymbol{X} = \{S.F_{\phi(1)}, \ldots, S.F_{\phi(y)}\}$ and $\bar{X} \in \boldsymbol{X}$ per assumption, it follows that rule R6 (Expansion) applies to (6.10) and path $S.F_{\varphi(1)}$, which yields (6.11). Hence, (6.11) $\in \Sigma \vee \Sigma \vdash$ (6.11) if paths $S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}$ are added to $\boldsymbol{X}$ at Line 4 and the condition at Line 3 is satisfied for some $\bar{X} \in \boldsymbol{X}$.

Suppose now that paths $\{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\}$ are added to $\boldsymbol{X}$ at Line 10 and that the condition at Line 9 is satisfied for some $\bar{X} \in \boldsymbol{X}$. Then $\{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\} \subseteq$ paths$(\bar{\sigma}) \cap$ paths$(\sigma) - \boldsymbol{X}$. Consequently, $\{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\} \subseteq$ paths$(\bar{\sigma})$ and thus there exist (again) the subsets $\{\bar{S}.\bar{F}_{\theta(1)}, \ldots, \bar{S}.\bar{F}_{\theta(x)}\}$ and $\{\bar{S}.\bar{F}_{\vartheta(1)}, \ldots, \bar{S}.\bar{F}_{\vartheta(z)}\}$ of paths$(\bar{\sigma})$ such that

$$\{\bar{S}.\bar{F}_{\theta(1)}, \ldots, \bar{S}.\bar{F}_{\theta(x)}\} = \{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\}$$
$$\{\bar{S}.\bar{F}_{\vartheta(1)}, \ldots, \bar{S}.\bar{F}_{\vartheta(z)}\} = \text{paths}(\bar{\sigma}) - \{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\}$$

Now, note that paths($\bar{\sigma}$) $= \{\bar{S}.\bar{F}_{\theta(1)}, \ldots, \bar{S}.\bar{F}_{\theta(x)}\} \cup \{\bar{S}.\bar{F}_{\vartheta(x+1)}, \ldots, \bar{S}.\bar{F}_{\vartheta(z)}\}$, and that therefore $\bar{\sigma}$ is of the form

$$(\bar{S}, (\{\bar{F}_{\theta(1)}, \ldots, \bar{F}_{\theta(x)}\} \cup \{\bar{F}_{\vartheta(1)}, \ldots, \bar{F}_{\vartheta(z)}\})). \tag{6.12}$$

Further, given that the condition at Line 9 is satisfied, $\bar{S}.\bar{F}_{\theta(i)} \cap \bar{S}.\bar{F}_{\vartheta(j)} \subseteq \bar{X}$ for all $(i,j) \in \{1, \ldots, x\} \times \{1, \ldots, z\}$. From this together with the observation that $\bar{X} \in$ paths(σ), since $\bar{X} \in \boldsymbol{X}$ and $\boldsymbol{X} = \{S.F_{\phi(1)}, \ldots, S.F_{\phi(y)}\}$ per assumption, it follows that rule R5 (Augmentation) applies to (6.10) and (6.12) if $\bar{S} = S$, which yields

$$(S, (\{F_{\phi(1)}, \ldots, F_{\phi(y)}\} \cup \{\bar{F}_{\theta(1)}, \ldots, \bar{F}_{\theta(x)}\})). \tag{6.13}$$

Note that (6.13) equals $\hat{\sigma}$, since $\{\bar{F}_{\theta(1)}, \ldots, \bar{F}_{\theta(x)}\} = \{F_{\varphi(1)}, \ldots, F_{\varphi(x)}\}$, given that $\{\bar{S}.\bar{F}_{\theta(1)}, \ldots, \bar{S}.\bar{F}_{\theta(x)}\} = \{S.F_{\varphi(1)}, \ldots, S.F_{\varphi(x)}\}$ and that $\bar{S} = S$.

It therefore remains to be shown that $\hat{\sigma} \in \Sigma \vee \Sigma \vdash \hat{\sigma}$ if $\bar{S} \neq S$. Assume for this purpose without loss of generality that for all $i \in \{1, \ldots, x\}$, $\bar{S}.\bar{F}_{\theta(i)} = S.F_{\varphi(i)}$, and observe that if $\bar{S} \neq S$ then either $S \subset \bar{S}$ or $\bar{S} \subset S$, given that $\bar{S}.\bar{F}_{\theta(i)} = S.F_{\varphi(i)}$ for all $i \in \{1, \ldots, x\}$.

Assume now that $S \subset \bar{S}$ and let R be the path such that $S.R = \bar{S}$. It follows then from (6.12) that $\bar{\sigma}$ is of the form

$$(S.R, (\{\bar{F}_{\theta(1)}, \ldots, \bar{F}_{\theta(x)}\} \cup \{\bar{F}_{\vartheta(1)}, \ldots, \bar{F}_{\vartheta(z)}\})). \tag{6.14}$$

and therefore, by applying rule R2 (Downshift) to (6.14), Σ derives the XKey

$$(S, (\{R.\bar{F}_{\theta(1)}, \ldots, R.\bar{F}_{\theta(x)}\} \cup \{R.\bar{F}_{\vartheta(1)}, \ldots, R.\bar{F}_{\vartheta(z)}\})). \tag{6.15}$$

Observe that rule R5 (Augmentation) applies to (6.10) and (6.15), since for all $(i,j) \in \{1, \ldots, x\} \times \{1, \ldots, z\}$, $S.R.\bar{F}_{\theta(i)} \cap S.R.\bar{F}_{\vartheta(j)} \subseteq \bar{X}$, given that $\bar{S}.\bar{F}_{\theta(i)} \cap \bar{S}.\bar{F}_{\vartheta(j)} \subseteq \bar{X}$ and that $S.R = \bar{S}$. Note that $\bar{X} \in$ paths($\bar{\sigma}$) as shown above. Further, the XKey that results from this application of rule R5 is of the form

$$(S, (\{F_{\phi(1)}, \ldots, F_{\phi(y)}\} \cup \{R.\bar{F}_{\theta(1)}, \ldots, R.\bar{F}_{\theta(x)}\})), \tag{6.16}$$

which equals $\hat{\sigma}$, since for all $i \in \{1, \ldots, x\}$, $R.\bar{F}_{\theta(i)} = F_{\varphi(i)}$, given that $\bar{S}.\bar{F}_{\theta(i)} = S.F_{\varphi(i)}$ and that $S.R = \bar{S}$.

Assume now instead that $\bar{S} \subset S$ and let $\tilde{R}$ be the path such that $\bar{S}.\tilde{R} = S$. Then (6.10) is of the form

$$(\bar{S}.\tilde{R}, (F_{\varphi(1)}, \ldots, F_{\varphi(x)})) \tag{6.17}$$

and therefore, by applying rule R2 (Downshift) to (6.18a), Σ derives the XKey

$$(\bar{S}, (\tilde{R}.F_{\phi(1)}, \ldots, \tilde{R}.F_{\phi(y)})) \tag{6.18}$$

Note that $\bar{X} \in$ paths((6.18)) since $\bar{X} \in \boldsymbol{X}$ per assumption and $\boldsymbol{X} = \{\bar{S}.\tilde{R}.F_{\phi(1)}, \ldots, \bar{S}.\tilde{R}.F_{\phi(y)}\}$, given that $\boldsymbol{X} = \{S.F_{\phi(1)}, \ldots, S.F_{\phi(1)}\}$ and $\bar{S}.\tilde{R} = S$. From this together with the result above that $\bar{S}.\bar{F}_{\theta(i)} \cap \bar{S}.\bar{F}_{\vartheta(j)} \subseteq \bar{X}$ for all $(i,j) \in \{1, \ldots, x\} \times \{1, \ldots, z\}$, it follows that rule R5 (Augmentation) applies to (6.18) and (6.12), which yields

$$(\bar{S}, (\{\tilde{R}.F_{\phi(1)}, \ldots, \tilde{R}.F_{\phi(y)}\} \cup \{\bar{F}_{\theta(1)}, \ldots, \bar{F}_{\theta(x)}\})) \tag{6.19}$$

Further, for all $i \in \{1,\ldots,x\}$, $\bar{F}_{\theta(i)} = \tilde{R}.F_{\varphi(i)}$ since $\bar{S}.\bar{F}_{\theta(i)} = \bar{S}.\tilde{R}.F_{\varphi(i)}$ given that $\bar{S}.\bar{F}_{\theta(i)} = S.F_{\varphi(i)}$ and $\bar{S}.\tilde{R} = S$. Consequently, (6.19) is of the form

$$(\bar{S}, (\{\tilde{R}.F_{\phi(1)},\ldots,\tilde{R}.F_{\phi(y)}\} \cup \{\tilde{R}.F_{\varphi(1)},\ldots,\tilde{R}.F_{\varphi(x)}\})) \tag{6.20}$$

Note finally that rule R1 (Upshift) applies to (6.20) and path $\tilde{R}$, which yields $\hat{\sigma}$ given that $\bar{S}.\tilde{R} = S$. Hence, the inductive step is established. □

We now present our result on the soundness of algorithm DXKI.

Theorem 6.3 (Soundness of Algorithm DXKI) *Given a set of XKeys* Σ *and a single XKey* σ*, if* DXKI(Σ,σ) = true *then* $\Sigma \models \sigma$*.*

Proof (Theorem 6.3) If DXKI(Σ,σ) = true then either $\sigma \in \Sigma$ or $\Sigma \vdash \sigma$ from Lemma 6.1. Now, if $\sigma \in \Sigma$ then $\Sigma \models \sigma$ from Definition 6.2, and if $\Sigma \vdash \sigma$ then $\Sigma \models \sigma$ from Theorem 6.2.□

6.2.4 Completeness of the Decision Procedure

We show next the completeness of algorithm DXKI, i.e. that if a set of XKeys Σ implies a single XKey σ then DXKI(Σ,σ) returns true. We then use this result to establish completeness of inference rules R1 - R7 for the implication of XKeys, i.e. if $\Sigma \models \sigma$ then $\Sigma \vdash \sigma$.

Theorem 6.4 (Completeness of Algorithm DXKI) *Given a set of XKeys* Σ *and a single XKey* σ*, if* $\Sigma \models \sigma$ *then* DXKI(Σ,σ) = true.

We first give an overview on the proof of Theorem 6.4. We establish completeness by showing the contrapositive that DXKI(Σ,σ) = false $\Rightarrow \Sigma \not\models \sigma$. Note that if DXKI$(\Sigma,\sigma)$ = false then $\boldsymbol{X} \subset$ paths(σ) when the algorithm terminates. We therefore construct a counter example tree T_σ that essentially contains duplicate nodes for the paths in paths$(\sigma) - \boldsymbol{X}$, and then show that $\mathrm{T}_\sigma \not\models \sigma$ but $\mathrm{T}_\sigma \models \Sigma$ and so $\Sigma \not\models \sigma$.

The counter example tree T_σ is generated by the algorithm GCEXKI (cf. Algorithm 6.2). The algorithm has input a downward-closed set of paths $\boldsymbol{P}$, a subset $\boldsymbol{D}$ of the paths in paths$(\sigma) - \boldsymbol{X}$ and a single path $\ddot{D}$ such that for all $D \in \boldsymbol{D}$, $\ddot{D} \subset D$. The choice of the specific input for algorithm GCEXKI is involved and depends on the set of XKeys $\Sigma \cup \{\sigma\}$ and the set of paths $\boldsymbol{X}$. We will detail on this issue later. Roughly speaking, $\boldsymbol{P}$ determines the structure of the counter example tree T_σ, and so algorithm GCEXKI iterates the paths in $\boldsymbol{P}$ and creates nodes correspondingly. In particular, if input path $\ddot{D}$ is a prefix of the intersection of a path $P \in \boldsymbol{P}$ and at least one path in $\boldsymbol{D}$, then algorithm GCEXKI creates two nodes, and so the final XML tree T_σ contains two isomorphic subtrees rooted at the specific node in T_σ which is reachable over path $\ddot{D}$. Let $\bar{\mathrm{T}}_\sigma$ and $\bar{\mathrm{T}}'_\sigma$ denote these isomorphic subtrees for the moment. Then, since $\ddot{D} \subset D$ for every path $D \in \boldsymbol{D}$, $\bar{\mathrm{T}}_\sigma$ and $\bar{\mathrm{T}}'_\sigma$ contain duplicate nodes reachable over the paths in $\boldsymbol{D}$, and since $\boldsymbol{D} \subseteq$ paths$(\sigma) - \boldsymbol{X}$, these duplicate nodes cause σ to be violated in T_σ. To ensure that the XKeys in Σ are satisfied, two nodes v in $\bar{\mathrm{T}}_\sigma$ and v' in $\bar{\mathrm{T}}'_\sigma$ that are reachable over the same path P have *distinct* values assigned if $P \notin \boldsymbol{D}$. We now illustrate algorithm GCEXKI by an example.

Algorithm 6.2 GCEXKI - Generate Counter Example for XKey Implication

```
in:   a downward-closed set of paths $\boldsymbol{P}$
      a single path $\ddot{D} \in \boldsymbol{P}$
      a set of paths $\boldsymbol{D} \subseteq \boldsymbol{P}$ such that $\forall D \in \boldsymbol{D}$
      - $\text{last}(D) \in \boldsymbol{L}^A \cup \{\texttt{S}\}$ and
      - $\text{last}(D) \notin \boldsymbol{L}^A$ if $\text{parent}(D) = \ddot{D}$
out:  XML tree $\text{T}_\sigma$ which is complete w.r.t. $\boldsymbol{P}$ and contains duplicate nodes on paths $\boldsymbol{D}$
1:  let $\boldsymbol{Z} = \boldsymbol{Z}' = \emptyset$
2:  let $\text{T}_\sigma = (\boldsymbol{V}, \boldsymbol{E}, \text{lab}, \text{val})$ be a trivial tree where $\text{lab}(\text{root}(\text{T}_\sigma)) = \rho(\boldsymbol{P})$
3:  let $\{R_1, \dots, R_m\} = \boldsymbol{P}$ such that $\forall i, j \in \{1, \dots, m\}$, $\text{length}(R_i) \le \text{length}(R_j)$ if $i \le j$
4:
5:  for $i = 2$ to $m$ do
6:      if $\exists D \in \boldsymbol{D}$ such that $\ddot{D} \subset (R_i \cap D)$ then          ▷ create duplicate nodes
7:          $v \leftarrow \text{newnode}(\boldsymbol{V})$; $v' \leftarrow \text{newnode}(\boldsymbol{V})$
8:          $\boldsymbol{Z} \leftarrow \boldsymbol{Z} \cup \{v\}$; $\boldsymbol{Z}' \leftarrow \boldsymbol{Z}' \cup \{v'\}$
9:          $\text{lab}(v) \leftarrow \text{last}(R_i)$; $\text{lab}(v') \leftarrow \text{last}(R_i)$
10:         if $\ddot{D} = \text{parent}(R_i)$ then
11:             let $\{\hat{v}\} = \text{nodes}(\text{parent}(R_i))$
12:             $\boldsymbol{E} \leftarrow \boldsymbol{E} \cup \{(\hat{v}, v), (\hat{v}, v')\}$
13:         else
14:             let $\{\hat{v}\} = \text{nodes}(\text{parent}(R_i)) \cap \boldsymbol{Z}$ and let $\{\hat{v}'\} = \text{nodes}(\text{parent}(R_i)) \cap \boldsymbol{Z}'$
15:             $\boldsymbol{E} \leftarrow \boldsymbol{E} \cup \{(\hat{v}, v), (\hat{v}', v')\}$
16:         end if
17:         if $R_i \in \boldsymbol{D}$ then
18:             $\text{val}(v) \leftarrow 1$; $\text{val}(v') \leftarrow 1$
19:         else if $\text{last}(R_i) \in \boldsymbol{L}^A \cup \{\texttt{S}\}$ then
20:             $\text{val}(v) \leftarrow 1$; $\text{val}(v') \leftarrow 2$
21:         end if
22:     else                                                              ▷ create single node
23:         $v \leftarrow \text{newnode}(\boldsymbol{V})$
24:         $\text{lab}(v) \leftarrow \text{last}(R_i)$
25:         let $\{\hat{v}\} = \text{nodes}(\text{parent}(R_i))$
26:         $\boldsymbol{E} \leftarrow \boldsymbol{E} \cup \{(\hat{v}, v)\}$
27:         if $\text{last}(R_i) \in \boldsymbol{L}^A \cup \{\texttt{S}\}$ then $\text{val}(v) \leftarrow 1$; end if
28:     end if
29: end for
30: return $\text{T}_\sigma$
```

Example 6.3 (algorithm GCEXKI) *Let $\Sigma = \{\sigma_1, \sigma_2\}$ be as in Example 6.1, so $\sigma_1 = (\rho,\ (\texttt{A.B.C.S},\ \texttt{D.E.S}))$ and $\sigma_2 = (\rho.\texttt{A},\ (\texttt{B.E.S},\ \texttt{B.D.S}))$, but let $\sigma = (\rho.\texttt{A},\ (\texttt{B.C.S},\ \texttt{B.F.f}))$. Then following Example 6.1, algorithm* DXKI(Σ, σ) *terminates with $\boldsymbol{X} = \{\rho.\texttt{A.B.C.S}\} \subset \text{paths}(\sigma)$ and algorithm* DXKI *returns false. The input to algorithm* GCEXKI *is then*

$\boldsymbol{P} = \{\rho,\ \rho.\mathtt{A},\ \rho.\mathtt{A.B},\ \rho.\mathtt{A.B.C},\ \rho.\mathtt{A.B.C.S},\ \rho.\mathtt{D},\ \rho.\mathtt{D.E},\ \rho.\mathtt{D.E.S},\ \rho.\mathtt{A.B.E},$
$\rho.\mathtt{A.B.E.S},\ \rho.\mathtt{A.B.D},\ \rho.\mathtt{A.B.D.S},\ \rho.\mathtt{A.B.F},\ \rho.\mathtt{A.B.F.f}\},$
$\ddot{D} = \rho.\mathtt{A.B},$
$\boldsymbol{D} = \{\rho.\mathtt{A.B.F.f}\}.$

Algorithm GCEXKI *proceeds as follows. At Line 3, tree* T_σ *contains only the root node and at Line 4 the paths in* $\boldsymbol{P}$ *are placed in the order* ρ, ρ.A, ρ.D, ρ.A.B, ρ.D.E, ρ.A.B.C, ρ.D.E.S, ρ.A.B.E, ρ.A.B.D, ρ.A.B.F, ρ.A.B.C.S, ρ.A.B.E.S, ρ.A.B.D.S, ρ.A.B.F.f*. At Line 5, all the paths in* $\boldsymbol{P}$ *are processed in the order just given. For each path in* $\boldsymbol{P}$ *before path* ρ.A.B.F*, the test at Line 6 fails and so Lines 23 - 27 are executed, resulting in the tree shown in Figure 6.1a.*

Consider then the case where $R_i = \rho.\mathtt{A.B.F}$*. Since* $D = \rho.\mathtt{A.B.F}.@F$*, this means that Line 6 evaluates to true and so Lines 7 - 9 are executed, and since the test at Line 10 evaluates to true, Lines 11 - 12 are executed and then neither test at Line 17 nor 18 evaluates to true. So at the end of this iteration of the loop at Line 29,* T_σ *is shown in Figure 6.1b.*

The test at Line 6 then fails for the paths ρ.A.B.C.S, ρ.A.B.E.S, ρ.A.B.D.S *in* $\boldsymbol{P}$*, but succeeds for* ρ.A.B.F.f*, the final path in* $\boldsymbol{P}$ *and so Lines 7 - 9 are executed. The test at Line 10 then fails, so Lines 14 - 15 are executed, and then the test at Line 17 evaluates to true and hence Line 18 is executed. At this stage the algorithm terminates with the final tree* T_σ *shown in Figure 6.1c, and it can easily be seen that* T_σ *is complete with respect to* $\boldsymbol{P}$ *and satisfies* Σ *but violates* σ *and so* $\Sigma \not\models \sigma$*.*

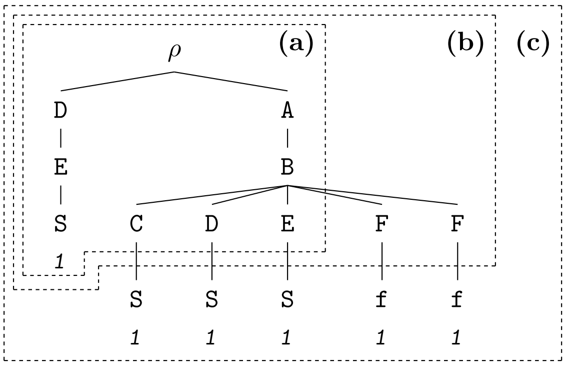


Figure 6.1: Counter example tree generated by algorithm GCEXKI.

In order to demonstrate Theorem 6.4, we now establish first some preliminary results related to the procedure of algorithm GCEXKI and the counter example tree T_σ. We will frequently need to refer to the root node of T_σ created at Line 2 in Algorithm 6.2. For ease of presentation, we will use v_ρ for denoting the root node of T_σ in the following. To be more precise, in the remainder of this chapter $v_\rho = \mathrm{root}(\mathrm{T}_\sigma)$.

Lemma 6.2 If P is a path in $\boldsymbol{P}$ within a run of algorithm GCEXKI$(\boldsymbol{P}, \boldsymbol{D}, \ddot{D})$, then
(i) parent(P) $= \ddot{D}$ if P satisfies the condition at Line 6 and length$(P) = 2$.
(ii) parent(P) satisfies the condition at Line 6 if P satisfies the condition at Line 6 and $\ddot{D} \neq$ parent(P).

(iii) parent(P) does not satisfy the condition at Line 6 if parent$(P) = \ddot{D}$.
(iv) $\bar{P}$ does not satisfy the condition at Line 6 if P does not satisfy the condition at Line 6 and $\bar{P} \subseteq P$.
(v) $\bar{P}$ satisfies the condition at Line 6 if P satisfies the condition at Line 6 and $P \subseteq \bar{P}$.
(vi) P does not satisfy the condition at Line 6 if $P = \rho$.

Proof (Lemma 6.2) (i) Given that P satisfies the condition at Line 6, there exists path $D \in \boldsymbol{D}$ such that $\ddot{D} \subset P \cap D$. Also, $P \cap D \subseteq P$ for all $D \in \boldsymbol{D}$ from Definition 3.13, and therefore length$(P \cap D) \leq 2$, given that length$(P) = 2$. Hence $\ddot{D} = \rho$, since ρ is the only strict prefix of any path of length 2 in a downward-closed set of paths. Further, parent$(P) = \rho$ given that $P \in \boldsymbol{P}$, which is a downward-closed set of paths per assumption, and consequently parent$(P) = \ddot{D}$.

(ii) Given that P satisfies the condition at Line 6, there exists path $D \in \boldsymbol{D}$ such that $\ddot{D} \subset P \cap D$. From this together with Definition 3.13 it follows that $\ddot{D} \subseteq$ parent$(P) \cap D$. Consequently, if parent(P) does not satisfy the condition at Line 6, then $\ddot{D} =$ parent$(P) \cap D$. Now suppose for the moment that if $\ddot{D} =$ parent$(P) \cap D$ then $\ddot{D} =$ parent(P). Then parent(P) must satisfy the condition at Line 6, since otherwise $\ddot{D} =$ parent(P), which clearly contradicts the assumption that $\ddot{D} \neq$ parent(P).

In order to verify that $\ddot{D} =$ parent(P) if $\ddot{D} =$ parent$(P) \cap D$, note that parent$(P) \cap D \subseteq P \cap D$ follows directly from Definition 3.13. Therefore either parent$(P) \cap D = P \cap D$ or parent$(P) \cap D \subset P \cap D$. If parent$(P) \cap D = P \cap D$, then $\ddot{D} = P \cap D$ follows from the observation above that $\ddot{D} =$ parent$(P) \cap D$, and $\ddot{D} = P \cap D$ clearly contradicts the assumption that $\ddot{D} \subset P \cap D$. Hence, parent$(P) \cap D \neq P \cap D$ and thus parent$(P) \cap D \subset P \cap D$.

Next, given that parent$(P) \cap D \subset P \cap D$, Definition 3.13 implies that $P \subseteq D$ and consequently $P \cap D = P$. Further, given that $P \cap D = P$, again Definition 3.13 implies that parent$(P) \cap D =$ parent(P) and thus $\ddot{D} =$ parent(P) follows from combining parent$(P) \cap D =$ parent(P) with the previous observation that $\ddot{D} =$ parent$(P) \cap D$.

(iii) Assume to the contrary that parent(P) satisfies the condition at Line 6. Then there exists path $D \in \boldsymbol{D}$ such that $\ddot{D} \subset D \cap$parent(P), and from this together with the assumption that $\ddot{D} =$ parent(P) it follows then that $\ddot{D}$ is a *strict* prefix of $D \cap \ddot{D}$. However this is a contradiction since when combined with the fact that $D \cap \ddot{D}$ is a prefix of $\ddot{D}$ from Definition 3.13, it would imply that $\ddot{D}$ is a strict prefix of $\ddot{D}$. Hence parent(P) does not satisfy the condition at Line 6.

(iv) Assume to the contrary that $\bar{P}$ satisfies the condition at Line 6. Then there exists path $D \in \boldsymbol{D}$ such that $\ddot{D} \subset D \cap \bar{P}$, and consequently also $\ddot{D} \subset D \cap P$ according to Definition 3.13, which however clearly contradicts the assumption that P does not satisfy the condition at Line 6.

(v) Given that P satisfies the condition at Line 6, there exists path $D \in \boldsymbol{D}$ such that $\ddot{D} \subset P \cap D$. It follows directly from Definition 3.13 that $P \cap D \subseteq \bar{P} \cap D$, given that $P \subseteq \bar{P}$ and hence combining with the fact that $\ddot{D} \subset D \cap P$ it follows $\bar{P}$ also satisfies the condition at Line 6.

(vi) If to the contrary $P = \rho$ satisfies the condition at Line 6, then there exists path $D \in \boldsymbol{D}$ such that $\ddot{D} \subset D \cap \rho$. However, since $D \in \boldsymbol{P}$, which is a downward-closed set of paths per assumption, first$(D) = \rho$, and therefore $D \cap \rho = \rho$. Hence, if ρ satisfies the condition at

Line 6, then $\ddot{D} \subset \rho$. This is however a contradiction since no path is a strict prefix of path ρ which is of length 1. Hence, P does not satisfy the condition at Line 6, if $P = \rho$. □

Lemma 6.3 In a run of algorithm GCEXKI($\boldsymbol{P}, \boldsymbol{D}, \ddot{D}$), let $\{R_1, \ldots, R_m\}$ be the paths at Line 3 in Algorithm 6.2. Then, in an iteration k of the loop at Line 5, where $1 \leq k < m$, the algorithm creates exactly

(i) two nodes v, v' such that $\{v\} = \text{nodes}(R_{k+1}) \cap \boldsymbol{Z}$ and $\{v'\} = \text{nodes}(R_{k+1}) \cap \boldsymbol{Z}'$ and anc-or-self$(v) \cap \boldsymbol{Z}' =$ anc-or-self$(v') \cap \boldsymbol{Z} = \emptyset$, where $\boldsymbol{Z}, \boldsymbol{Z}'$ are the sets of paths at Line 1, if path R_{k+1} satisfies the condition at Line 6;

(ii) one node v such that $\{v\} = \text{nodes}(R_{k+1})$ and anc-or-self$(v) \cap (\boldsymbol{Z} \cup \boldsymbol{Z}') = \emptyset$, if path R_{k+1} does not satisfy the condition at Line 6.

Proof (Lemma 6.3) The proof is by induction over the iterations of the loop at Line 5. Note that the root node v_ρ is the only node which is created before the loop at Line 5 is entered for the first time, and that $\{v_\rho\} = \text{nodes}(\rho)$ at Line 2.

Base Case: The first iteration of the loop at Line 5 is used as the base case for the induction. That is $k = 1$ in the base case. Note that the loop variable $i = 2$ in this iteration, and that therefore Lemma 6.3 holds true for the base case if

- two nodes v, v' are created such that $\{v\} = \text{nodes}(R_2) \cap \boldsymbol{Z}$ and $\{v'\} = \text{nodes}(R_2) \cap \boldsymbol{Z}'$ and anc-or-self$(v) \cap \boldsymbol{Z}' =$ anc-or-self$(v') \cap \boldsymbol{Z} = \emptyset$, if path R_2 satisfies the condition at Line 6;

- one node v is created such that $\{v\} = \text{nodes}(R_2)$ and anc-or-self$(v) \cap (\boldsymbol{Z} \cup \boldsymbol{Z}') = \emptyset$, if path R_2 does not satisfy the condition at Line 6.

Because paths $R_1, \ldots, R_m$ are downward-closed per assumption and ordered by length according to Line 3, length$(R_2) = 2$ and so parent$(R_2) = \rho$ since $R_2 \in \boldsymbol{P}$, which is a downward-closed set of paths per assumption.

Now, assume first that R_2 satisfies the condition at Line 6. Then distinct nodes v and v' are created at Line 7. Further, edges (v_ρ, v) and (v_ρ, v') are added at Line 12, since $\ddot{D} = \text{parent}(R_2)$ according to (i) in Lemma 6.2, and parent$(R_2) = \rho$ according to our previous observations. Recall that $\{v_\rho\} = \text{nodes}(\rho)$ at Line 2, and that therefore the assumption at Line 11 that nodes(parent(R_i)) contains exactly one node is met. The resulting path instances $v_\rho.v$ and $v_\rho.v'$ are defined over path R_2 since lab$(v) =$ lab$(v') =$ last(R_2) according to Line 9, and $R_2 = \rho.\text{last}(R_2)$, given that R_2 is a path of length 2. Consequently, $\{v, v'\} \subseteq \text{nodes}(R_2)$. Further, $v \in \boldsymbol{Z}$ and $v' \in \boldsymbol{Z}'$ according to Line 8. In particular $\{v\} = \boldsymbol{Z}$ and $\{v'\} = \boldsymbol{Z}'$, since nodes v and v' are the first nodes that are created within the loop at Line 5. Therefore, $\{v\} = \text{nodes}(R_2) \cap \boldsymbol{Z}$ and $\{v'\} = \text{nodes}(R_2) \cap \boldsymbol{Z}'$ and anc-or-self$(v) \cap \boldsymbol{Z}' =$ anc-or-self$(v') \cap \boldsymbol{Z} = \emptyset$. Hence, Lemma 6.3 holds true.

Assume now that R_2 does not satisfy the condition at Line 6. Then the algorithm creates the single node v at Line 23, and adds edge (v_ρ, v) at Line 26, since parent$(R_2) = \rho$, given that length$(R_2) = 2$. Likewise to the previous case $\{v_\rho\} = \text{nodes}(\rho)$, and therefore the assumption at Line 25 that nodes(parent(R_i)) contains exactly one node is met. The resulting path instance $v_\rho.v$ is defined over path R_2 since lab$(v) =$ last(R_2) according to Line 24, and $R_2 = \rho.\text{last}(R_2)$, since R_2 is a path of length 2. Consequently, $v \in \text{nodes}(R_2)$. Further, $\{v\} = \text{nodes}(R_2)$ follows again from the observation that node v is the first node created within the loop at Line 5. Also anc-or-self$(v) \cap (\boldsymbol{Z} \cup \boldsymbol{Z}') = \emptyset$ since anc-or-self$(v) = \{v_\rho, v\}$, given that $\{v\} = \text{nodes}(R_2)$, and the algorithm has neither added v_ρ nor v to $\boldsymbol{Z}$ or $\boldsymbol{Z}'$ and thus $\boldsymbol{Z} = \boldsymbol{Z}' = \emptyset$ from Line 1. Hence, Lemma 6.3 also holds true in case that R_2 does not satisfy the condition at Line 5, which establishes the base case.

Inductive Step: Assume now that Lemma 6.3 holds true until iteration x of the loop at Line 5, where $x \geq 1$. Then the inductive step is established if Lemma 6.3 also holds true for the iteration $x+1$ of the loop at Line 5, which is what we show next. Note that the inductive assumption implies that for every node v created in the loop at Line 5 before iteration $x+1$, $v \notin \text{nodes}(\rho)$ since paths $R_2, \ldots, R_m$ are ordered by length and therefore $\rho \notin \{R_2, \ldots, R_m\}$. From this together with the observation that $\{v_\rho\} = \text{nodes}(\rho)$ initially at Line 2 it follows that also $\{v_\rho\} = \text{nodes}(\rho)$ before iteration $x+1$ of the loop at Line 5 is entered.

Further, the loop variable $i = x+2$ in iteration $x+1$ of the loop at Line 5, and therefore Lemma 6.3 holds true for this iteration if

– two nodes v, v' are created such that $\{v\} = \text{nodes}(R_{x+2}) \cap \boldsymbol{Z}$ and $\{v'\} = \text{nodes}(R_{x+2}) \cap \boldsymbol{Z}'$ and anc-or-self$(v) \cap \boldsymbol{Z}' =$ anc-or-self$(v') \cap \boldsymbol{Z} = \emptyset$, if path R_{x+2} satisfies the condition at Line 6;
– one node v is created such that $\{v\} = \text{nodes}(R_{x+2})$ and anc-or-self$(v) \cap (\boldsymbol{Z} \cup \boldsymbol{Z}') = \emptyset$, if path R_{x+2} does not satisfy the condition at Line 6.

Assume first that R_{x+2} satisfies the condition at Line 6. Then it is assumed at Line 11 that there exists node $\hat{v}$ such that $\{\hat{v}\} = \text{nodes}(\text{parent}(R_{x+2}))$ in case that the condition at Line 10 is met, and otherwise it is assumed at Line 14 that there exist distinct nodes $\hat{v}$ and $\hat{v}'$ such that $\{\hat{v}\} = \text{nodes}(\text{parent}(R_{x+2})) \cap \boldsymbol{Z}$ and $\{\hat{v}'\} = \text{nodes}(\text{parent}(R_{x+2})) \cap \boldsymbol{Z}'$. The correctness of these assumptions is now verified.

Thereby, if the condition at Line 10 is met and $\text{parent}(R_{x+2}) = \rho$ then $\{\hat{v}\} = \text{nodes}(\text{parent}(R_{x+2}))$ follows from the previous observation that $\{v_\rho\} = \text{nodes}(\rho)$ before iteration $x+1$ is entered. If instead $\text{parent}(R_{x+2}) \neq \rho$, then $\text{parent}(R_{x+2}) \in \{R_2, \ldots, R_m\}$ and $\{\hat{v}\} = \text{parent}(R_{x+2})$ follows from the inductive assumption.

It follows in particular from (ii) in Lemma 6.3, since $\text{parent}(R_{x+2})$ does not satisfy the condition at Line 6 according to (iii) in Lemma 6.2, given that the condition at Line 10 is met and that therefore $\ddot{D} = \text{parent}(R_{x+2})$. Note that path $\text{parent}(R_{x+2})$ is iterated before path R_{x+2} within the loop at Line 5 since $\text{length}(\text{parent}(R_{x+2})) < \text{length}(R_{x+2})$ and paths $R_1, \ldots, R_m$ are ordered by length according to Line 3. The inductive assumption therefore indeed applies to path $\text{parent}(R_{x+2})$.

If instead the condition at Line 10 is not met, then the existence of nodes $\hat{v}$ and $\hat{v}'$ such that $\{\hat{v}\} = \text{nodes}(\text{parent}(R_{x+2})) \cap \boldsymbol{Z}$ and $\{\hat{v}'\} = \text{nodes}(\text{parent}(R_{x+2})) \cap \boldsymbol{Z}'$ is implied by the inductive assumption. In particular this is implied by (i) in Lemma 6.3, since $\text{parent}(R_{x+2})$ satisfies the condition at Line 6 according to (ii) in Lemma 6.2, given that the condition at Line 10 is not met and that therefore $\ddot{D} \neq \text{parent}(R_{x+2})$. Note that for the same reasons as above, the inductive assumption indeed applies to path $\text{parent}(R_{x+2})$.

Next, the algorithm creates distinct nodes v and v' at Line 7 and, in case that $\ddot{D} = \text{parent}(R_{x+2})$, the algorithm adds edges $(\hat{v}, v)$ and $(\hat{v}, v')$ at Line 12, where $\{\hat{v}\} = \text{nodes}(\text{parent}(R_{x+2}))$. If instead $\ddot{D} \neq \text{parent}(R_{x+2})$, then the algorithm adds edges $(\hat{v}, v)$ and $(\hat{v}', v')$ at Line 15, where $\{\hat{v}\} = \text{nodes}(\text{parent}(R_{x+2})) \cap \boldsymbol{Z}$ and $\{\hat{v}'\} = \text{nodes}(\text{parent}(R_{x+2})) \cap$

$\boldsymbol{Z}'$. In both cases, a walk over path R_{x+2} results, since both nodes $\hat{v}$ and $\hat{v}'$ are in nodes(parent(R_{x+2})) and lab(v) = lab(v') = last(R_{x+2}) according to Line 9. Note that $R_{x+2} = \text{parent}(R_{x+2}).\text{last}(R_{x+2})$. Consequently, $\{v, v'\} \in \text{nodes}(R_{x+2})$.

In order to verify that $\{v\} = \text{nodes}(R_{x+2}) \cap \boldsymbol{Z}$ and that $\{v'\} = \text{nodes}(R_{x+2}) \cap \boldsymbol{Z}'$, note that $v \in \boldsymbol{Z}$ and that $v' \in \boldsymbol{Z}'$ according to Line 8. From combining this with the previous result that $\{v, v'\} \in \text{nodes}(R_{x+2})$ it follows that $v \in \text{nodes}(R_{x+2}) \cap \boldsymbol{Z}$ and that $v' \in \text{nodes}(R_{x+2}) \cap \boldsymbol{Z}'$. Therefore, if $\{v\} \neq \text{nodes}(R_{x+2}) \cap \boldsymbol{Z}$, then there exists node $\bar{v}$ such that $\bar{v} \neq v$ and $\bar{v} \in \text{nodes}(R_{x+2}) \cap \boldsymbol{Z}$. Note that $\bar{v} \neq v_\rho$ since v_ρ is not added to $\boldsymbol{Z}$ by the algorithm. Therefore, if $\bar{v}$ exists, then $\bar{v}$ has been created for some path $\bar{R}$ within a previous iteration of the loop at Line 5. Then however $\bar{R} = R_{x+2}$ from the inductive assumption together with $\bar{v} \in \text{nodes}(R_{x+2})$, which clearly contradicts the assumption that $R_1, \ldots, R_m$ is a *set* of paths and therefore does not contain duplicates. This argumentation also applies to node v'. Thus $\{v\} = \text{nodes}(R_{x+2}) \cap \boldsymbol{Z}$ and also $\{v'\} = \text{nodes}(R_{x+2}) \cap \boldsymbol{Z}'$.

Further, in order to verify that anc-or-self(v) $\cap\, \boldsymbol{Z}' = \emptyset$, note that $v \in \boldsymbol{Z}$ according to Line 8, and that therefore anc-or-self(parent(v)) $\cap\, \boldsymbol{Z}' \neq \emptyset$ if anc-or-self(v) $\cap\, \boldsymbol{Z}' \neq \emptyset$.

Now, if parent(v) $= v_\rho$ then anc-or-self(parent(v)) $= \{v_\rho\}$ and therefore $\boldsymbol{Z}' \cap$ anc-or-self(parent(v)) $= \emptyset$ since algorithm GCEXKI does not add the root node v_ρ to $\boldsymbol{Z}'$. If instead parent(v) $\neq v_\rho$, then parent(v) has been created within a previous iteration of the loop at Line 5 and therefore the inductive assumption applies to parent(v). Thereby, if parent(R_{x+2}) does not satisfy the condition at Line 6 then anc-or-self(parent(v))$\cap(\boldsymbol{Z} \cup \boldsymbol{Z}') =$ $\emptyset$ according to (ii) in Lemma 6.3. If instead parent(R_{x+2}) satisfies the condition at Line 6 then anc-or-self(parent(v)) $\cap\, \boldsymbol{Z}' = \emptyset$ according to (i) in Lemma 6.3, since parent(v) $\in \boldsymbol{Z}$ according to Line 14. Thus, anc-or-self(parent(v))$\cap \boldsymbol{Z}' = \emptyset$. This argumentation also applies to node v' and thus also anc-or-self(parent(v')) $\cap\, \boldsymbol{Z} = \emptyset$. Hence, the inductive assumption holds true for iteration $x + 1$ in case that R_{x+2} satisfies the condition at Line 6.

Assume now instead that R_{x+2} does not satisfy the condition at Line 6. Then it is assumed at Line 25 that there exists node $\hat{v}$ such that $\{\hat{v}\} = \text{nodes}(\text{parent}(R_{x+2}))$. The correctness of this assumptions is now verified. In case that parent(R_{x+2}) $= \rho$, $\{\hat{v}\} =$ nodes(parent(R_{x+2})) follows from the previous observation that $\{v_\rho\} = \text{nodes}(v_\rho)$ before iteration $x+1$ is entered. If instead parent(R_{x+2}) $\neq \rho$, then parent(R_{x+2}) $\in \{R_2, \ldots, R_{x+1}\}$ and $\{\hat{v}\} = \text{parent}(R_{x+2})$ follows from the inductive assumption. It follows in particular from (ii) in Lemma 6.3, since R_{x+2} does not satisfy the condition at Line 6 per assumption and therefore also parent(R_{x+2}) does not satisfy this condition according to (iv) in Lemma 6.2. Note that path parent(R_{x+2}) is iterated before path R_{x+2} within the loop at Line 5 since length(parent(R_{x+2})) $<$ length(R_{x+2}) and paths $R_1, \ldots, R_m$ are ordered by length according to Line 3, i.e. the inductive assumption indeed applies to path parent(R_{x+2}).

Next, the algorithm creates the single node v at Line 23 and adds the edge $(\hat{v}, v)$ at Line 26, where $\{\hat{v}\} = \text{nodes}(\text{parent}(R_{x+2}))$. The resulting path instance is defined over path R_{x+2}, since $\hat{v} \in \text{nodes}(\text{parent}(R_{x+2}))$, and lab($v$) = last($R_{x+2}$) according to Line 24. Note that $R_{x+2} = \text{parent}(R_{x+2}).\text{last}(R_{x+2})$ and so $v \in \text{nodes}(R_{x+2})$.

In order to verify that $\{v\} = \text{nodes}(R_{x+2})$ note that if to the contrary $\{v\} \neq \text{nodes}(R_{x+2})$ then there exists node $\bar{v}$ such that $\bar{v} \neq v$ and $\bar{v} \in \text{nodes}(R_{x+2})$. Thereby, nodes($R_{x+2}$) $\neq$ ρ, given that $R_{x+2} \in \{R_2, \ldots, R_m\}$ and that paths $R_1, \ldots, R_m$ are downward-closed and ordered by length. Hence, $\bar{v} \neq v_\rho$ follows from the previous observation that $v_\rho \in \text{nodes}(\rho)$.

Consequently, if $\bar{v}$ exists then $\bar{v}$ has been created for some path $\bar{R}$ within a previous iteration of the loop at Line 5. Then however $\bar{R} = R_{x+2}$ from the inductive assumption, which clearly contradicts the assumption that $R_1, \ldots, R_m$ is a *set* of paths and therefore does not contain duplicates. Thus, $\{v\} = \text{nodes}(R_{x+2})$.

In order to verify that $\text{anc-or-self}(v) \cap (\boldsymbol{Z} \cap \boldsymbol{Z}') = \emptyset$, note that v has not been added to $\boldsymbol{Z}$ or $\boldsymbol{Z}'$ by the algorithm. Consequently, if $\text{anc-or-self}(v) \cap (\boldsymbol{Z} \cap \boldsymbol{Z}') \neq \emptyset$ then $\text{anc-or-self}(\text{parent}(v)) \cap (\boldsymbol{Z} \cap \boldsymbol{Z}') \neq \emptyset$. Again, if $\text{parent}(v) = v_\rho$, then $\text{anc-or-self}(\text{parent}(v)) = \{v_\rho\}$ and therefore $\text{anc-or-self}(\text{parent}(v)) \cap (\boldsymbol{Z} \cup \boldsymbol{Z}') = \emptyset$ since algorithm GCEXKI does neither add the root node v_ρ to $\boldsymbol{Z}$ nor to $\boldsymbol{Z}'$. If instead $\text{parent}(v) \neq v_\rho$, then $\text{parent}(v)$ has been created within a previous iteration of the loop at Line 5. Then however the inductive assumption implies, in particular (ii) in Lemma 6.3, that $\text{anc-or-self}(\text{parent}(v)) \cap (\boldsymbol{Z} \cup \boldsymbol{Z}') = \emptyset$ since $\text{parent}(R_{x+2})$ does not satisfy the condition at Line 6, which has been shown previously. Consequently, $\text{anc-or-self}(v) \cap (\boldsymbol{Z} \cap \boldsymbol{Z}') = \emptyset$. Hence, the inductive assumption also holds true for iteration $x + 1$ in case that R_{x+2} does not satisfy the condition at Line 6, which establishes Lemma 6.3. □

Lemma 6.4 Let $\boldsymbol{P}$ be a downward-closed set of paths, and let $\ddot{D} \in \boldsymbol{P}$ be a single path and $\boldsymbol{D} \subseteq \boldsymbol{P}$ be a set of paths such that for all $D \in \boldsymbol{D}$, $\text{last}(D) \in \boldsymbol{L}^A \cup \{\texttt{S}\}$ and $\text{last}(D) \notin \boldsymbol{L}^A$ if $\text{parent}(D) = \ddot{D}$. Then, $\text{GCEXKI}(\boldsymbol{P}, \boldsymbol{D}, \ddot{D})$ returns an XML tree T_σ such that
(i) T_σ conforms to Definition 3.11
(ii) T_σ is complete with respect to $\boldsymbol{P}$

Proof (Lemma 6.4) Note that Lemma 6.4 assumes that algorithm GCEXKI terminates. This is however easily verified since the input set of paths $\boldsymbol{P}$ is finite and hence the loop at Line 5 in Algorithm 6.2 is iterated only for a finite number of times.

(i) Note that algorithm GCEXKI iteratively creates tree T_σ by means of adding nodes and edges within the loop at Line 5, starting from the initial tree at Line 2. Now, let T_{σ_1} denote the tree after Line 2, and let for all $k \in \{2, \ldots, m\}$ tree T_{σ_k} denote the tree after iteration $k - 1$ of the loop at Line 5. Then, $\text{T}_{\sigma_1}, \ldots, \text{T}_{\sigma_m}$ denotes the sequence of trees generated within the run of algorithm GCEXKI. It is shown subsequently by induction over trees $\text{T}_{\sigma_1}, \ldots, \text{T}_{\sigma_m}$ that the final tree $\text{T}_\sigma = \text{T}_{\sigma_m}$ conforms to Definition 3.11. It is shown in particular, that for all $k \in \{1, \ldots, m\}$, tree T_{σ_k} satisfies requirements (1) - (4) in Definition 3.11, which means that

(1) $(\boldsymbol{V}, \boldsymbol{E})$ is a tree in terms of Definition 3.3 and hence
  (i) $\boldsymbol{V}$ is a finite set of nodes, from Definition 3.1
  (ii) $\forall (v, \bar{v}) \in \boldsymbol{E}$, $v \neq \bar{v}$, from Definition 3.1.i
  (iii) $\forall v \in \boldsymbol{V}$, if $\boldsymbol{V} \neq \{v\}$ then $\exists \bar{v} \in \boldsymbol{V}$ such that $(v, \bar{v}) \in \boldsymbol{E}$ or $(\bar{v}, v) \in \boldsymbol{E}$, from Definition 3.1.ii
  (iv) if there exist nodes $v_1 . \cdots . v_n \in \boldsymbol{V}$ such that for all $i \in \{1, \ldots, n\}$, $(v_{i-1}, v_i) \in \boldsymbol{E}$, then $v_1 \neq v_n$ if $n > 1$, from Definition 3.3
(2) $\forall v \in \boldsymbol{V}$, $\text{lab}(v) \in \boldsymbol{L}$.
(3) $\forall v \in \boldsymbol{V}$, if $\text{lab}(v) \in \boldsymbol{L}^A \cup \{\texttt{S}\}$ then $\text{val}(v) \in \boldsymbol{U}$ and $\text{val}(v)$ is undefined otherwise.
(4) $\forall (v, \bar{v}) \in \boldsymbol{E}$
  (i) $\text{lab}(v) \in \boldsymbol{L}^E$,

(ii) if $\bar{v} \in \boldsymbol{L}^A$ then $\nexists(v, \bar{v}') \in \boldsymbol{E}$ such that $\bar{v} \neq \bar{v}'$ and $\text{lab}(\bar{v}) = \text{lab}(\bar{v}')$

Tree T_{σ_1} is the base case for the induction. Given that T_{σ_1} is a trivial XML tree according to Line 2, T_{σ_1} satisfies (1) - (4) in Definition 3.11. In order to establish the inductive step we assume that XML tree T_{σ_k}, where $1 \leq k < m$, conforms to Definition 3.11 and show that also tree $\mathrm{T}_{\sigma_{k+1}}$ conforms to Definition 3.11. Because $\mathrm{T}_{\sigma_{k+1}}$ is created in iteration k of the loop at Line 5 given that $1 \leq k < m$, we show in particular that $\mathrm{T}_{\sigma_{k+1}}$ satisfies (1) - (4) in Definition 3.11 at the end of iteration k of the loop at Line 5.

(1.i) From the inductive assumption, $\boldsymbol{V}$ is finite in T_{σ_k}. Hence, $\boldsymbol{V}$ is also finite in $\mathrm{T}_{\sigma_{k+1}}$ if only a finite number of nodes is added to $\boldsymbol{V}$ in iteration k of the loop at Line 5. This is clearly the case given that at most two new nodes are added to $\boldsymbol{V}$ in iteration k of the loop at Line 5 (cf. Lines 7 and 23).

(1.ii) From the inductive assumption, $\forall(v, \bar{v}) \in \boldsymbol{E}$, $v \neq \bar{v}$ in T_{σ_k}. From the procedure of algorithm GCEXKI, edges that exist in XML tree T_{σ_k} are neither altered nor deleted. Hence, (1.ii) in Lemma 6.4 holds true for $\mathrm{T}_{\sigma_{k+1}}$ if $v \neq \bar{v}$ whenever an edge $(v, \bar{v})$ is added to $\boldsymbol{E}$ in iteration k of the loop at Line 5. From the procedure of algorithm GCEXKI, if an edge $(v, \bar{v})$ is added to $\boldsymbol{E}$ in iteration k, then v is a node that exists in XML tree T_{σ_k} and $\bar{v}$ is a node created in iteration k (cf. Lines 12, 15 and 26). Given that v is created in iteration k, node v does not exist in XML tree T_{σ_k}. Hence $v \neq \bar{v}$.

(1.iii) From the inductive assumption, $\forall v \in \boldsymbol{V}$ in XML tree T_{σ_k}, if $\boldsymbol{V} \neq \{v\}$ then there exists node $\bar{v} \in \boldsymbol{V}$ such that $(v, \bar{v}) \in \boldsymbol{E}$ or $(\bar{v}, v) \in \boldsymbol{E}$. From the procedure of algorithm GCEXKI, nodes and edges that exist in XML tree T_{σ_k} are neither altered nor deleted. Hence, (1.iii) in Lemma 6.4 holds true for $\mathrm{T}_{\sigma_{k+1}}$ if for every node v which is created in iteration k, there exists node $\bar{v} \in \boldsymbol{V}$ such that $(v, \bar{v}) \in \boldsymbol{E}$ or $(\bar{v}, v) \in \boldsymbol{E}$. From the procedure of algorithm GCEXKI, whenever a node v is created in iteration k, edge $(\bar{v}, v)$ is added to $\boldsymbol{E}$, where $\bar{v}$ is a node in T_{σ_k} (cf. Lines 12, 15 and 26), which establishes the result.

(1.iv) From the inductive assumption, there are no cycles in T_{σ_k}. Also, from the procedure of algorithm GCEXKI edges that exist in XML tree T_{σ_k} are neither altered nor deleted, and also no two nodes created in iteration k of the loop at Line 5 are connected to each other. From this we deduce that if $\mathrm{T}_{\sigma_{k+1}}$ contains a cycle, then a node v is created in iteration k and there exists a walk $\bar{v}_1.\cdots.\bar{v}_n$ in T_{σ_k} such that either $v.\bar{v}_1.\cdots.\bar{v}_n$ or $\bar{v}_1.\cdots.\bar{v}_n.v$ is a cycle. Now, if $v.\bar{v}_1.\cdots.\bar{v}_n$ is a cycle then $v = \bar{v}_n$ which clearly contradicts the assumption that $\bar{v}_n$ is a node in T_{σ_k} since v is a new node created in iteration k of the loop at Line 5. If instead $v.\bar{v}_1.\cdots.\bar{v}_n$ is a cycle, edge $(v, \bar{v}_1)$ has been added to $\boldsymbol{E}$ in iteration k. This however clearly contradicts the procedure of algorithm GCEXKI, since an edge that connects a new node v to an existing node does not start from v but only lead to v (cf. Lines 12, 15 and 26). Hence, also $\mathrm{T}_{\sigma_{k+1}}$ does not contain cycles.

(2) Whenever a node v is created within the loop at Line 5 (cf. Lines 7 and 23) then the final label of a path $R_{k+1} \in \{R_2, \ldots, R_m\}$ is assigned to v (cf. Lines 9 and 24), i.e. $\text{lab}(v) = \text{last}(R_{k+1})$. From this together with the observation that $\text{last}(R_{k+1}) \in \boldsymbol{L}$ according to Definition 3.12, $\text{lab}(v) \in \boldsymbol{L}$. Hence, $\mathrm{T}_{\sigma_{k+1}}$ satisfies (2) given that T_{σ_k} satisfies (2).

(3) From the inductive assumption, for every node $v \in \boldsymbol{V}$ in T_{σ_k}, $\mathrm{val}(v) \in \boldsymbol{U}$ iff v is an attribute or text node. From combining this with the observation that the assignment of values to nodes, which exist in tree T_{σ_k}, is not modified within the loop at Line 5, we deduce that $\mathrm{T}_{\sigma_{k+1}}$ satisfies (3) in Definition 3.11 if for every node v which is newly created in iteration k of the loop at Line 5, $\mathrm{val}(v) \in \boldsymbol{U}$ iff v is an attribute or text node.

Assume first that v is an element node which has been created at Line 7. Then, $\mathrm{last}(R_{k+1}) \in \boldsymbol{E}$ follows from the fact that $\mathrm{lab}(v) \in \boldsymbol{E}$ and $\mathrm{lab}(v) = \mathrm{last}(R_{k+1})$ according to Line 9. Therefore neither the condition at Line 17 nor the condition at Line 19 is met. We note that R_{k+1} does not satisfy the condition at Line 17 because for all $D \in \boldsymbol{D}$, $\mathrm{last}(D) \in \boldsymbol{L}^A \cup \{\mathtt{S}\}$ per assumption and therefore $R_{k+1} \notin \boldsymbol{D}$ if $\mathrm{last}(R_{k+1}) \in \boldsymbol{E}$. Hence, $\mathrm{val}(v)$ is undefined if v is an element node which has been created at Line 7. Next, if v is an attribute or text node which has been created at Line 7, then $\mathrm{val}(v) \in \boldsymbol{U}$ since either the condition at Line 17 or the condition at Line 19 is met. If instead v is a node created at Line 23, then $\mathrm{val}(v) \in \boldsymbol{U}$ iff v is an attribute or text node according to Line 27.

(4.i) From the inductive assumption, for all $(\bar{v}, v) \in \boldsymbol{E}$ in XML tree T_{σ_k}, $\mathrm{lab}(\bar{v}) \in \boldsymbol{L}^E$. From the procedure of algorithm GCEXKI, edges that exist in T_{σ_k} are not altered. Hence, $\mathrm{T}_{\sigma_{k+1}}$ satisfies (4.i) in Lemma 6.4 if whenever an edge $(\bar{v}, v)$ is added to $\boldsymbol{E}$ in iteration k of the loop at Line 5, $\mathrm{lab}(\bar{v}) \in \boldsymbol{L}^E$. To verify this, note that edge $(\bar{v}, v)$ is added to $\boldsymbol{E}$ either at Line 12, 15 or 26. From Lines 11, 14 and 25, $\bar{v} \in \mathrm{nodes}(\mathrm{parent}(R_{k+1}))$. Thus, if $\mathrm{lab}(\bar{v}) \notin \boldsymbol{L}^E$ then $\mathrm{last}(\mathrm{parent}(R_{k+1})) \notin \boldsymbol{L}^E$. Then, however, R_{k+1} contains an attribute label or the text label on a position other than the last, which clearly contradicts Definition 3.12. Hence, $\mathrm{T}_{\sigma_{k+1}}$ satisfies (4.i) in Lemma 6.4.

(4.ii) We show that there do not exist attribute nodes $\tilde{v}$ and $\tilde{v}'$ in $\mathrm{T}_{\sigma_{k+1}}$ such that $\mathrm{lab}(\tilde{v}) = \mathrm{lab}(\tilde{v}') \in \boldsymbol{L}^A$ and $\mathrm{parent}(\tilde{v}) = \mathrm{parent}(\tilde{v}')$. From the inductive assumption, such a pair of attributes nodes does not exist in T_{σ_k}. Combining this with the observation that edges which exist in T_{σ_k} are not altered, we deduce that if attribute nodes $\tilde{v}$ and $\tilde{v}'$ exist in $\mathrm{T}_{\sigma_{k+1}}$ then either $\tilde{v}$ is created in iteration k of the loop at Line 5 and $\tilde{v}'$ exists in T_{σ_k}, or both nodes $\tilde{v}$ and $\tilde{v}'$ are created in iteration k.

Consider first the case where node $\tilde{v}$ is created in iteration k and node $\tilde{v}'$ exists in T_{σ_k}, and let k' denote the iteration of the loop at Line 5 within which node $\tilde{v}'$ has been created. We note that $\tilde{v}'$ has been created within the loop at Line 5 because $\tilde{v}' \neq v_\rho$ given that $\tilde{v}'$ is an attribute node. Now, $\tilde{v} \in \mathrm{nodes}(R_{k+1})$ and $\tilde{v}' \in \mathrm{nodes}(R_{k'+1})$ according to Lemma 6.3. Hence, $R_{k+1} = R_{k'+1}$ since $\mathrm{lab}(\tilde{v}) = \mathrm{lab}(\tilde{v}')$ and $\mathrm{parent}(\tilde{v}) = \mathrm{parent}(\tilde{v}')$ per assumption. Given that $R_{k+1} = R_{k'+1}$, if $k \neq k'$ then this contradicts the assumption that $R_2, \ldots, R_m$ is a *set* of paths. However, if $k = k'$ then this contradicts the assumption that $\tilde{v}'$ exists in T_{σ_k}, and so $\mathrm{T}_{\sigma_{k+1}}$ satisfies (4.ii) with respect to every pair of attribute nodes $\tilde{v}$ and $\tilde{v}'$ where $\tilde{v}$ is a node created in iteration k of the loop at Line 5 and $\tilde{v}'$ is a node which exists in T_{σ_k}.

Consider now the case that both nodes $\tilde{v}$ and $\tilde{v}'$ are created in iteration k of the loop at Line 5. Then, given that $\tilde{v} \neq \tilde{v}'$, nodes $\tilde{v}$ and $\tilde{v}'$ have been created at Line 7, and consequently path R_{k+1} satisfies the condition at Line 6. Further, since $\mathrm{parent}(\tilde{v}) = \mathrm{parent}(\tilde{v}')$ per assumption, R_{k+1} satisfies the condition at Line 10. We note that if R_{k+1} does not satisfy the condition at Line 10 then $\mathrm{parent}(\tilde{v}) \neq \mathrm{parent}(\tilde{v}')$ according to Lines 14 and 15. Now, given that R_{k+1} satisfies the condition at Line 6, there exists path $D \in \boldsymbol{D}$ such that $\ddot{D} \subset$

$(R_{k+1} \cap D)$. Combining this with the previous observation that R_{k+1} satisfies the condition at Line 10, i.e. that $\ddot{D} = \text{parent}(R_{k+1})$, we deduce that $\text{parent}(R_{k+1}) \subset (R_{k+1} \cap D)$, and therefore $D = R_{k+1}$ since R_{k+1} ends in an attribute label given that $\tilde{v}$ and $\tilde{v}'$ are attribute nodes. Consequently, $R_{k+1} \in \boldsymbol{D}$, since $D \in \boldsymbol{D}$ per assumption. This however clearly contradicts the requirement on the input of algorithm GCEXKI that for all $D \in \boldsymbol{D}$, $\text{last}(D) \notin \boldsymbol{L}^A$ if $\text{parent}(D) = \ddot{D}$, since $\{\tilde{v}, \tilde{v}'\} \subseteq \text{nodes}(R_{k+1})$ from Lemma 6.3 and therefore $\text{last}(R_{k+1}) \in \boldsymbol{L}^A$ given that $\tilde{v}$ and $\tilde{v}'$ are attribute nodes. Hence, $\text{T}_{\sigma_{k+1}}$ also satisfies (4.ii) with respect to every pair of attribute nodes $\tilde{v}$ and $\tilde{v}'$ which are both created in iteration k of the loop at Line 5.

(ii) It is shown first that T_σ conforms to $\boldsymbol{P}$, i.e. that for every node v in T_σ, if P is the path such that $v \in \text{nodes}(P)$ then $P \in \boldsymbol{P}$. Note for this purpose, that $\rho \in \boldsymbol{P}$, since $\boldsymbol{P}$ is downward-closed per assumption, and that therefore T_σ conforms to $\boldsymbol{P}$ initially at Line 2. It therefore remains to be shown that whenever a node v is created in an iteration k of the loop at Line 5, then $P \in \boldsymbol{P}$. This is however easily verified, since $v \in \text{nodes}(R_{k+1})$ according to Lemma 6.3, and $R_{k+1} \in \boldsymbol{P}$, since $R_{k+1} \in \{R_2, \ldots, R_m\}$ and $\{R_2, \ldots, R_m\} \subset \boldsymbol{P}$.

Since T_σ conforms to $\boldsymbol{P}$, T_σ is complete w.r.t. $\boldsymbol{P}$ according to Definition 5.2 if whenever P and $\bar{P}$ are paths in $\boldsymbol{P}$ such that $P \subset \bar{P}$ and there exists node $v \in \text{nodes}(P)$ then there exists node $\bar{v}$ such that $\bar{v} \in \text{nodes}(\bar{P})$ and $v \in \text{anc-or-self}(\bar{v})$, which is what we show next.

Assume first that either $P = \rho$ or that P does not satisfy the condition at Line 6 in Algorithm 6.2. Note that if $P = \rho$ then $\text{nodes}(P)$ contains exactly one node, which is the root node v_ρ, since T_σ conforms to Definition 3.11 as shown previously. If instead P does not satisfy the condition at Line 6 in Algorithm 6.2 then $\text{nodes}(P)$ contains exactly one node according to (ii) in Lemma 6.3. Hence, if $P = \rho$ or P does not satisfy the condition at Line 6 in Algorithm 6.2, then there exists exactly one node v in T_σ such that $v \in \text{nodes}(P)$, i.e. $\{v\} = \text{nodes}(P)$. Further, given that $P \subset \bar{P}$ it follows that $\bar{P} \neq \rho$ and therefore either (i) or (ii) in Lemma 6.3 applies to $\bar{P}$. Consequently, $\text{nodes}(\bar{P}) \neq \emptyset$. Now, let $\bar{v}$ be a node in T_σ such that $\bar{v} \in \text{nodes}(\bar{P})$. Then, $v \in \text{anc-or-self}(\bar{v})$ follows directly from the observation above that $\{v\} = \text{nodes}(P)$ and the assumptions that $\bar{v} \in \text{nodes}(\bar{P})$ and that $P \subset \bar{P}$.

It therefore remains to be shown that node $\bar{v}$ exists also in case that P satisfies the condition at Line 6 in Algorithm 6.2. Given that P satisfies this condition and that $v \in \text{nodes}(P)$, (i) in Lemma 6.3 implies that there exists node v' such that $\{v\} = \text{nodes}(P) \cap \boldsymbol{Z}$ and $\{v'\} = \text{nodes}(P) \cap \boldsymbol{Z}'$ or vice versa. Note that v and v' are the only nodes in T_σ that are reachable over path P.

Assume first that $\{v\} = \text{nodes}(P) \cap \boldsymbol{Z}$. Note that $P \subset \bar{P}$ per assumption, and that therefore (v) in Lemma 6.2 implies that $\bar{P}$ satisfies the condition at Line 6 too. Consequently, (i) in Lemma 6.3 implies that there exist nodes $\bar{v}$ and $\bar{v}'$ such that $\{\bar{v}\} = \text{nodes}(\bar{P}) \cap \boldsymbol{Z}$ and $\{\bar{v}'\} = \text{nodes}(\bar{P}) \cap \boldsymbol{Z}'$. Now, given that $\{\bar{v}\} = \text{nodes}(\bar{P})$, and that $\{v, v'\} = \text{nodes}(P)$, and that $P \subset \bar{P}$, then either $v \in \text{anc-or-self}(\bar{v})$ or $v' \in \text{anc-or-self}(\bar{v})$. However, if $v' \in \text{anc-or-self}(\bar{v})$ then this contradicts (i) in Lemma 6.3, in particular that $\text{anc-or-self}(\bar{v}) \cap \boldsymbol{Z}' = \emptyset$ since $\bar{v} \in \boldsymbol{Z}$ per assumption. Consequently, $v \in \text{anc-or-self}(\bar{v})$.

This argumentation also applies to the case where $\{v\} = \text{nodes}(P) \cap \boldsymbol{Z}'$ and $\{v'\} = \text{nodes}(P) \cap \boldsymbol{Z}$, which establishes that T_σ is complete w.r.t. $\boldsymbol{P}$. $\square$

Lemma 6.5 Let T_σ be the tree returned by a run of algorithm GCEXKI($\boldsymbol{P}, \boldsymbol{D}, \ddot{D}$). Then for every path $P \in \boldsymbol{P}$, tree T_σ contains exactly

(i) two nodes v, v' such that $\{v\} = \text{nodes}(P) \cap \boldsymbol{Z}$ and $\{v'\} = \text{nodes}(P) \cap \boldsymbol{Z}'$ and $\text{anc-or-self}(v) \cap \boldsymbol{Z}' = \text{anc-or-self}(v') \cap \boldsymbol{Z} = \emptyset$, where $\boldsymbol{Z}, \boldsymbol{Z}'$ are the sets of paths at Line 1, if path P satisfies the condition at Line 6;

(ii) one node v such that $\{v\} = \text{nodes}(P)$ and $\text{anc-or-self}(v) \cap (\boldsymbol{Z} \cup \boldsymbol{Z}') = \emptyset$, if path P does not satisfy the condition at Line 6.

Proof (Lemma 6.5) Note that $\boldsymbol{P} = R_1, \ldots, R_m$ according to Line 3 in Algorithm 6.2. Also, since paths $R_1, \ldots, R_m$ are downward-closed and in increasing length order per assumption, $R_1 = \rho$. Hence, for every path $P \in \boldsymbol{P}$, either $P = \rho$ or $P \in \{R_2, \ldots, R_m\}$.

(i) Given that P satisfies the condition at Line 6, $P \neq \rho$ according to (vi) in Lemma 6.2 and therefore $P \in \{R_2, \ldots, R_m\}$ from the observation above. Hence, (i) in Lemma 6.3 applies to P, which establishes (i) in Lemma 6.5.

(ii) From the observation above, either $P = \rho$ or $P \in \{R_2, \ldots, R_m\}$. Suppose first that $P \in \{R_2, \ldots, R_m\}$. Then (ii) in Lemma 6.3 applies to P, which establishes (ii) in Lemma 6.5. If instead $P = \rho$, then $P \notin \{R_2, \ldots, R_m\}$ and therefore no node $\bar{v}$ is created within the loop at Line 5 in Algorithm 6.2 such that $\bar{v} \in \text{nodes}(P)$ according to Lemma 6.3. Hence the only node in tree T_σ that is reachable over path ρ is node v_ρ, which is created at Line 2 in Algorithm 6.2 and consequently $\{v_\rho\} = \text{nodes}(P)$. Also, $\text{anc-or-self}(v_\rho) = \{v_\rho\}$, since T_σ conforms to Definition 3.11 according to Lemma 6.4, and therefore $\text{anc-or-self}(v_\rho) \cap (\boldsymbol{Z} \cup \boldsymbol{Z}') = \emptyset$ since algorithm GCEXKI does not add v_ρ to $\boldsymbol{Z}$ or $\boldsymbol{Z}'$. □

Lemma 6.6 Let T_σ be the tree returned by a run of algorithm GCEXKI($\boldsymbol{P}, \boldsymbol{D}, \ddot{D}$). Also, let $P, \bar{P}$ be paths in $\boldsymbol{P}$ and let $v, \bar{v}$ be nodes in T_σ such that $v \in \text{nodes}(P)$ and $\bar{v} \in \text{nodes}(\bar{P})$. Then, $\text{closest}(v, \bar{v})$ is true if

(i) P does not satisfy the condition at Line 6 in Algorithm 6.2, or

(ii) P and $\bar{P}$ satisfy the condition at Line 6 in Algorithm 6.2, and either $\{v, \bar{v}\} \subseteq \boldsymbol{Z}$ or $\{v, \bar{v}\} \subseteq \boldsymbol{Z}'$, where $\boldsymbol{Z}, \boldsymbol{Z}'$ are the sets of paths at Line 1 in Algorithm 6.2.

Proof (Lemma 6.6) Since $v \in \text{nodes}(P)$ and $\bar{v} \in \text{nodes}(\bar{P})$ per assumption, it follows that there exist nodes $\{v'', \bar{v}''\} \subseteq \text{nodes}(P \cap \bar{P})$ such that $v'' \in \text{anc-or-self}(v)$ and $\bar{v}'' \in \text{anc-or-self}(\bar{v})$. Consequently, $\text{closest}(v, \bar{v})$ is true if $v'' = \bar{v}''$.

(i) Given that P does not satisfy the condition at Line 6, also $P \cap \bar{P}$ does not satisfy this condition according to (iv) in Lemma 6.2 since $P \cap \bar{P} \subseteq P$. Consequently, (ii) in Lemma 6.5 applies to path $P \cap \bar{P}$ and hence T_σ contains exactly one node that is reachable over path $P \cap \bar{P}$. Hence, $v'' = \bar{v}''$ follows from the observation above that $\{v'', \bar{v}''\} \subseteq \text{nodes}(P \cap \bar{P})$, and so $\text{closest}(v, \bar{v}) = \text{true}$.

(ii) Consider first the case that $\{v, \bar{v}\} \subseteq \boldsymbol{Z}$, and suppose that $P \cap \bar{P}$ does not satisfy the condition at Line 6. Then $\text{nodes}(P \cap \bar{P})$ contains exactly one node according to (ii) in Lemma 6.5 and so $v'' = \bar{v}''$, given that $\{v'', \bar{v}''\} \subseteq \text{nodes}(P \cap \bar{P})$, and hence $\text{closest}(v, \bar{v}) = \text{true}$.

If instead $P \cap \bar{P}$ satisfies the condition at Line 6, then, from (i) in Lemma 6.5, $\text{nodes}(P \cap \bar{P})$ contains exactly two nodes, call them $\tilde{v}$ and $\tilde{v}'$, such that $\{\tilde{v}\} = \text{nodes}(P \cap \bar{P}) \cap \boldsymbol{Z}$ and

$\{\tilde{v}'\} = \text{nodes}(P \cap \bar{P}) \cap \boldsymbol{Z}'$. Consequently, if $v'' \neq \bar{v}''$ then either $v'' = \tilde{v}'$ or $\bar{v}'' = \tilde{v}'$, since $\{v'', \bar{v}''\} \subseteq \text{nodes}(P \cap \bar{P})$ and $\text{nodes}(P \cap \bar{P}) = \{\tilde{v}, \tilde{v}'\}$ from the observations above. However, if $v'' = \tilde{v}'$ then $v'' \in \boldsymbol{Z}'$ which clearly contradicts (i) in Lemma 6.3, i.e. that $\text{anc-or-self}(v) \cap \boldsymbol{Z}' = \emptyset$. Note that (i) in Lemma 6.3 applies to v, since v must have been created within the loop at Line 5 given that $v \in \boldsymbol{Z}$. Likewise, if $\bar{v}'' = \tilde{v}'$ then $\bar{v}'' \in \boldsymbol{Z}'$ which contradicts that $\text{anc-or-self}(\bar{v}) \cap \boldsymbol{Z}' = \emptyset$. Hence, $v'' = \bar{v}''$ and so $\text{closest}(v, \bar{v})$ is true.

The argumentation applies analogously for the case where $\{v, \bar{v}\} \subseteq \boldsymbol{Z}'$. Hence (ii) in Lemma 6.6 is established. □

We are now ready to establish Theorem 6.4.

Proof (Theorem 6.4) It is shown that given a downward-closed set of paths $\boldsymbol{P}$ and a set of XKeys $\Sigma \cup \{\sigma\}$ which conform to $\boldsymbol{P}$, if $\text{DXKI}(\Sigma, \sigma)$ is false then there exists tree T_σ that is complete w.r.t. $\boldsymbol{P}$ such that $\text{T}_\sigma \models \Sigma$ but $\text{T}_\sigma \not\models \sigma$. Note that T_σ is a counter example for the implication of σ by Σ. Also, we assume in the following that σ is of the general form $\sigma = (S, (F_1, \ldots, F_n))$. Therefore

$$\text{paths}(\sigma) = (S.F_1, \ldots, S.F_n).$$

We now first show how the input for algorithm GCEXKI is determined.

Determining the input for the run of algorithm GCEXKI: The downward-closed set of paths $\boldsymbol{P}$ the XKeys $\Sigma \cup \{\sigma\}$ are conforming to, is used to determine the structure of tree T_σ and is therefore the first piece of input for the run of GCEXKI. In order to determine the choice of the set of paths $\boldsymbol{D}$ and the single path $\ddot{D}$, recall first that $\text{DXKI}(\Sigma, \sigma)$ is false per assumption and that therefore $\boldsymbol{X} \subset \text{paths}(\sigma)$ according to Line 13 in Algorithm 6.1. Hence, one can choose permutation $\chi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ and integer x, where $0 \leq x < n$, such that for all $i \in \{1, \ldots, n\}$

$$S.F_{\chi(i)} \in \begin{cases} \boldsymbol{X} & i \leq x \\ \text{paths}(\sigma) - \boldsymbol{X} & i > x \end{cases}$$

Note that $x < n$ since $\text{DXKI}(\Sigma, \sigma) = \text{false}$ per assumption, and therefore at least one path from $\text{paths}(\sigma)$ is missing in $\boldsymbol{X}$. Therefore, $\{S.F_{\chi(x+1)}, \ldots, S.F_{\chi(n)}\}$ is definitely non-empty. In contrast, $\{S.F_{\chi(1)}, \ldots, S.F_{\chi(x)}\}$ is the empty set in case that $x = 0$, i.e. if not even one path was added to $\boldsymbol{X}$ within the run $\text{DXKI}(\Sigma, \sigma)$. Further, let for all $i \in \{1, \ldots, x\}$, $\ddot{R}_{\chi(i)}$ be a prefix of $S.F_{\chi(i)}$ such that

$$\ddot{R}_{\chi(i)} = \text{ shortest prefix of } S.F_{\chi(i)} \mid S.F_{\chi(i)} \cap S.F_{\chi(j)} \subseteq \ddot{R}_{\chi(i)} \forall j \in \{x+1, \ldots, n\}.$$

The choices of $\ddot{D}$ and $\boldsymbol{D}$ are dependent on the paths $\{\ddot{R}_{\chi(1)}, \ldots, \ddot{R}_{\chi(x)}\}$ and on the subsets $\{S.F_{\chi(1)}, \ldots, S.F_{\chi(x)}\}$ and $\{S.F_{\chi(x+1)}, \ldots, S.F_{\chi(n)}\}$ of $\text{paths}(\sigma)$. In particular:

$$\ddot{D} = \begin{cases} \rho & \{S.F_{\chi(1)}, \ldots, S.F_{\chi(x)}\} = \emptyset \\ \ddot{R}_{\chi(k)} \mid \ddot{R}_{\chi(k)} \not\subset \ddot{R}_{\chi(i)} \forall i \in \{1, \ldots, x\} & \{S.F_{\chi(1)}, \ldots, S.F_{\chi(x)}\} \neq \emptyset \end{cases}$$

$$\boldsymbol{D} = \ \{S.F_{\chi(i)} \in \{S.F_{\chi(x+1)}, \ldots, S.F_{\chi(n)}\} \mid \ddot{D} \subset S.F_{\chi(i)}\}$$

The choice of $\ddot{D}$ is not deterministic in case that $\{S.F_{\chi(1)},\ldots,S.F_{\chi(x)}\} \neq \emptyset$ since in general more than one path $\ddot{R}_{\chi(k)} \in \{\ddot{R}_{\chi(1)},\ldots,\ddot{R}_{\chi(x)}\}$ may satisfy the condition that $\ddot{R}_{\chi(k)} \not\subset \ddot{R}_{\chi(i)}$ for all $i \in \{1,\ldots,x\}$. Note however that if $\{S.F_{\chi(1)},\ldots,S.F_{\chi(x)}\} \neq \emptyset$ then at least one $\ddot{R}_{\chi(k)}$ must satisfy this condition, since there are no circles within the *strict* prefix relationships in a set of paths, and therefore at least one $\ddot{R}_{\chi(k)}$ must not be a strict prefix of any path in $\{\ddot{R}_{\chi(1)},\ldots,\ddot{R}_{\chi(x)}\}$. Hence, the choice of $\ddot{D}$ is determined also in case that $\{S.F_{\chi(1)},\ldots,S.F_{\chi(x)}\} \neq \emptyset$.

Next, algorithm GCEXKI requires that for all $D \in \boldsymbol{D}$, last$(D) \in \boldsymbol{L}^A \cup \{\mathtt{S}\}$, and that last$(D) \notin \boldsymbol{L}^A$ if parent$(D) = \ddot{D}$ (cf. top of Algorithm 6.2). It is verified subsequently that $\boldsymbol{D}$ satisfies these requirements. In particular:

(i) for all $D \in \boldsymbol{D}$, last$(D) \in \boldsymbol{L}^A \cup \{\mathtt{S}\}$: Note that the fields of σ end in attribute or text labels, given that σ conforms to Definition 3.20. That is last$(P) \in \boldsymbol{L}^A \cup \{\mathtt{S}\}$ for all $P \in$ paths(σ). Also, $\boldsymbol{D} \subseteq$ paths(σ) according to the definition of $\boldsymbol{D}$ and consequently last$(D) \in \boldsymbol{L}^A \cup \{\mathtt{S}\}$ for all $D \in \boldsymbol{D}$.

(ii) for all $D \in \boldsymbol{D}$, last$(D) \notin \boldsymbol{L}^A$ if parent$(D) = \ddot{D}$: Assume to the contrary that for some $D \in \boldsymbol{D}$, last$(D) \in \boldsymbol{L}^A$ and parent$(D) = \ddot{D}$. Then, as shown next, this contradicts the assumption that no more change to $\boldsymbol{X}$ is possible within the run of DXKI(Σ,σ).

Note that $D \in$ paths$(\sigma) - \boldsymbol{X}$ since $\boldsymbol{D} \subseteq \{S.F_{\chi(x+1)},\ldots,S.F_{\chi(n)}\}$ according to the definition of $\boldsymbol{D}$, and $\{S.F_{\chi(x+1)},\ldots,S.F_{\chi(n)}\} =$ paths$(\sigma) - \boldsymbol{X}$ per assumption. From combining this with the assumption that last$(D) \in \boldsymbol{L}^A$ it follows that D satisfies the condition at Line 3 in Algorithm 6.1, and thus one more change to $\boldsymbol{X}$ is possible if there exists path $\bar{X} \in \boldsymbol{X} \cup \{\rho\}$ such that parent$(D) \subseteq \bar{X}$. In order to verify that path $\bar{X} \in \boldsymbol{X} \cup \{\rho\}$ exists, suppose first that $\{S.F_{\chi(1)},\ldots,S.F_{\chi(x)}\} = \emptyset$, and let $\bar{X} = \rho$. Then parent$(D) \subseteq \bar{X}$ follows because parent$(D) = \ddot{D}$ per assumption and $\ddot{D} = \rho$, given that $\{S.F_{\chi(1)},\ldots,S.F_{\chi(x)}\} = \emptyset$, and thus parent$(D) = \rho = \bar{X}$. Note that parent$(D) \subseteq \bar{X}$ if parent$(D) = \bar{X}$. Suppose now that $\{S.F_{\chi(1)},\ldots,S.F_{\chi(x)}\} \neq \emptyset$. Then $\ddot{D} \in \{\ddot{R}_{\chi(x+1)},\ldots,\ddot{R}_{\chi(n)}\}$ per definition. Now, let $\ddot{R}_{\chi(k)}$ be the path chosen for $\ddot{D}$, and let $\bar{X} = S.F_{\chi(k)}$. Then parent$(D) \subseteq \bar{X}$ follows because parent$(D) = \ddot{D}$ per assumption and therefore parent$(D) = \ddot{R}_{\chi(k)}$, given that $\ddot{D} = \ddot{R}_{\chi(k)}$, and thus parent$(D) \subseteq S.F_{\chi(k)} = \bar{X}$ since $\ddot{R}_{\chi(k)} \subseteq S.F_{\chi(k)}$ from the definition of $\ddot{R}_{\chi(k)}$.

The counter example tree T_σ **violates** σ: It is shown now that the tree T_σ which is returned by GCEXKI$(\boldsymbol{P},\boldsymbol{D},\ddot{D})$ violates the XKey σ.

Note first that $\boldsymbol{D}$ is a non-empty subset of $\{S.F_{\chi(x+1)},\ldots,S.F_{\chi(n)}\}$. This is because if $\ddot{D} = \rho$ then $\ddot{D} \subseteq S.F_{\chi(i)}$ for all $i \in \{x{+}1,\ldots,n\}$ and therefore $\boldsymbol{D} = \{S.F_{\chi(x+1)},\ldots,S.F_{\chi(n)}\}$, which is non-empty as shown previously. If instead $\ddot{D} = \ddot{R}_{\chi(k)}$ then, since $\ddot{R}_{\chi(k)}$ is defined to be the *shortest* prefix of $S.F_{\chi(k)}$ such that $S.F_{\chi(k)} \cap S.F_{\chi(j)} \subseteq \ddot{R}_{\chi(k)}$ for all $j \in \{x+1,\ldots,n\}$, it follows that there exists $S.F_{\chi(i)} \in \{S.F_{\chi(x+1)},\ldots,S.F_{\chi(n)}\}$ such that $S.F_{\chi(k)} \cap S.F_{\chi(i)} = \ddot{R}_{\chi(k)}$. From this together with the observation that $S.F_{\chi(k)}$ and $S.F_{\chi(i)}$ are legal paths that end in attribute or text labels since paths(σ) conforms to Definition 3.20 and $\{S.F_{\chi(k)},S.F_{\chi(i)}\} \subseteq$ paths(σ) per definition, it follows that $\ddot{R}_{\chi(k)} \subset S.F_{\chi(i)}$. Consequently, $S.F_{\chi(i)} \in \boldsymbol{D}$ from the definition of $\boldsymbol{D}$. Hence, $\boldsymbol{D}$ is non-empty also in case that $\ddot{D} = \ddot{R}_{\chi(k)}$.

Next, given that $\boldsymbol{D}$ is a non-empty subset of paths(σ), one can choose permutation $\delta : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ and integer d, where $1 \leq d \leq n$, such that $\forall i \in \{1, \ldots, n\}$

$$S.F_{\delta(i)} \in \begin{cases} \boldsymbol{D} & i \leq d \\ \text{paths}(\sigma) - \boldsymbol{D} & i > d \end{cases}$$

It is shown next that there exist nodes $\{v_{\delta(1)}, \ldots, v_{\delta(d)}\} \subseteq \boldsymbol{Z}$ and $\{v'_{\delta(1)}, \ldots, v'_{\delta(d)}\} \subseteq \boldsymbol{Z}'$ in tree T_σ, where $\boldsymbol{Z}$ and $\boldsymbol{Z}'$ denote the sets of paths at Line 1 in Algorithm 6.2, such that

– $\forall i \in \{1, \ldots, d\}$, $\{v_{\delta(i)}, v'_{\delta(i)}\} \subseteq \text{nodes}(S.F_{\delta(i)})$, and
– $\forall i, j \in \{1, \ldots, d\}$, $\text{closest}(v_{\delta(i)}, v_{\delta(j)}) = \text{closest}(v'_{\delta(i)}, v'_{\delta(j)}) = \text{true}$, and
– $\forall i \in \{1, \ldots, d\}$, $\text{val}(v_{\delta(i)}) = \text{val}(v'_{\delta(i)})$, and
– $\forall i \in \{1, \ldots, d\}$, $v_{\delta(i)} \neq v'_{\delta(i)}$.

In order to verify that nodes $v_{\delta(1)}, \ldots, v_{\delta(d)}$ and $v'_{\delta(1)}, \ldots, v'_{\delta(d)}$ exist, it is shown first that for all $i \in \{1, \ldots, d\}$, $S.F_{\delta(i)}$ satisfies the condition at Line 6 in Algorithm 6.2, i.e. it is shown that there exists path $D \in \boldsymbol{D}$ such that $\ddot{D} \subset S.F_{\delta(i)} \cap D$. Note that $S.F_{\delta(i)} \in \boldsymbol{D}$ per definition and thus, by taking $D = S.F_{\delta(i)}$, $\ddot{D} \subset S.F_{\delta(i)} \cap D$ if $\ddot{D} \subset S.F_{\delta(i)}$, which however follows immediately from the definition of $\boldsymbol{D}$.

Given that for all $i \in \{1, \ldots, d\}$, $S.F_{\delta(i)}$ satisfies the condition at Line 6 in Algorithm 6.2, from (i) in Lemma 6.5, there are distinct nodes $v_{\delta(i)}$ and $v'_{\delta(i)}$ in T_σ such that $\{v_{\delta(i)}\} = \text{nodes}(S.F_{\delta(i)}) \cap \boldsymbol{Z}$ and $\{v'_{\delta(i)}\} = \text{nodes}(S.F_{\delta(i)}) \cap \boldsymbol{Z}'$. Also, $\text{val}(v_{\delta(i)}) = \text{val}(v'_{\delta(i)})$ since $S.F_{\delta(i)} \in \boldsymbol{D}$ and therefore $v_{\delta(i)}$ and $v'_{\delta(i)}$ have been assigned the same value at Line 18 in Algorithm 6.2.

Finally, $\text{closest}(v_{\delta(i)}, v_{\delta(j)}) = \text{closest}(v'_{\delta(i)}, v'_{\delta(j)}) = \text{true}$ for all $i, j \in \{1, \ldots, d\}$ follows from (ii) in Lemma 6.6, given that $\{v_{\delta(i)}, v_{\delta(j)}\} \subseteq \boldsymbol{Z}$ and $\{v'_{\delta(i)}, v'_{\delta(j)}\} \subseteq \boldsymbol{Z}'$, and that $S.F_{\delta(i)}$ and $S.F_{\delta(j)}$ satisfy the condition at Line 6 because of our previous result that all paths in $S.F_{\delta(1)}, \ldots, S.F_{\delta(d)}$ satisfy this condition.

Now, observe that $\{S.F_{\delta(1)}, \ldots, S.F_{\delta(d)}\} = \text{paths}(\sigma)$ if $\boldsymbol{D} = \text{paths}(\sigma)$, and that therefore nodes $v_{\delta(1)}, \ldots, v_{\delta(d)}$ and $v'_{\delta(1)}, \ldots, v'_{\delta(d)}$ violate σ in T_σ according to Lemma 5.6 in this case.

It therefore remains to be shown that $\mathrm{T}_\sigma \not\models \sigma$ in case that $\boldsymbol{D} \subset \text{paths}(\sigma)$. For this purpose, it is shown first that if $\boldsymbol{D} \subset \text{paths}(\sigma)$ and therefore $\{S.F_{\delta(d+1)}, \ldots, S.F_{\delta(n)}\}$ is not the empty set, then there exist nodes $\bar{v}_{\delta(d+1)}, \ldots, \bar{v}_{\delta(n)}$ in T_σ such that

– $\forall i \in \{d+1, \ldots, n\}$, $\bar{v}_{\delta(i)} \in \text{nodes}(S.F_{\delta(i)})$, and
– $\forall i, j \in \{d+1, \ldots, n\}$, $\text{closest}(\bar{v}_{\delta(i)}, \bar{v}_{\delta(j)}) = \text{true}$.

The crucial prerequisite in order to verify the existence of nodes $\bar{v}_{\delta(d+1)}, \ldots, \bar{v}_{\delta(n)}$ is that for all $i \in \{d+1, \ldots, n\}$, path $S.F_{\delta(i)}$ does not satisfy the condition at Line 6 in Algorithm 6.2, which is established next.

Assume for this purpose to the contrary that for some $i \in \{d+1, \ldots, n\}$, path $S.F_{\delta(i)}$ satisfies the condition at Line 6. Then, since $\boldsymbol{D} = \{S.F_{\delta(1)}, \ldots, S.F_{\delta(d)}\}$ per assumption, there exists path $S.F_{\delta(j)} \in \{S.F_{\delta(1)}, \ldots, S.F_{\delta(d)}\}$ such that $\ddot{D} \subset S.F_{\delta(i)} \cap S.F_{\delta(j)}$.

It follows from the definition of $S.F_{\delta(i)}$ and $S.F_{\delta(j)}$ that $\{S.F_{\delta(i)}, S.F_{\delta(j)}\} \subseteq \text{paths}(\sigma)$. From this together with the definitions of the sets of paths $\{S.F_{\chi(1)}, \ldots, S.F_{\chi(x)}\}$ and

$\{S.F_{\chi(x+1)}, \dots, S.F_{\chi(n)}\}$ it follows then that $\{S.F_{\delta(i)}, S.F_{\delta(j)}\} \subseteq \{S.F_{\chi(1)}, \dots, S.F_{\chi(x)}, S.F_{\chi(x+1)}, \dots, S.F_{\chi(n)}\}$. Therefore, let $i', j' \in \{1, \dots, n\}$ be the indexes such that $S.F_{\delta(i)} = S.F_{\chi(i')}$ and $S.F_{\delta(j)} = S.F_{\chi(j')}$. Then, $\ddot{D} \subset S.F_{\chi(i')} \cap S.F_{\chi(j')}$ if path $S.F_{\delta(i)}$ satisfies the condition at Line 6 in Algorithm 6.2.

Note that $j' \in \{x+1, \dots, n\}$ since $S.F_{\delta(j)} \in \boldsymbol{D}$ per assumption and $\boldsymbol{D} \subseteq \{S.F_{\chi(x+1)}, \dots, S.F_{\chi(n)}\}$ per definition. Note further that $S.F_{\chi(i')} \notin \boldsymbol{D}$ since $S.F_{\chi(i')} = S.F_{\delta(i)}$ per assumption, and $S.F_{\delta(i)} \notin \boldsymbol{D}$ according to the definition of paths $S.F_{\delta(d+1)}, \dots, S.F_{\delta(n)}$. Now, if also $i' \in \{x+1, \dots, n\}$ then this contradicts the assumption that $S.F_{\chi(i')} \notin \boldsymbol{D}$ because $S.F_{\chi(i')} \in \boldsymbol{D}$ if $\ddot{D} \subset S.F_{\chi(i')}$ according to the definition of $\boldsymbol{D}$, and $\ddot{D} \subset S.F_{\chi(i')}$ is implied by the assumption that $\ddot{D} \subset S.F_{\chi(i')} \cap S.F_{\chi(j')}$. The case that $i' \in \{x+1, \dots, n\}$ is therefore excluded.

Assume now that $i' \in \{1, \dots, x\}$ instead and let $\ddot{R}_{\chi(k)}$ be the path in $\{\ddot{R}_{\chi(1)}, \dots, \ddot{R}_{\chi(x)}\}$ chosen for $\ddot{D}$. Note that if $i' \in \{1, \dots, x\}$, then $\{S.F_{\chi(1)}, \dots, S.F_{\chi(x)}\}$ is not the empty set and therefore $\ddot{D} \in \{\ddot{R}_{\chi(1)}, \dots, \ddot{R}_{\chi(x)}\}$ per definition. Then $\ddot{R}_{\chi(k)} \subset S.F_{\chi(i')} \cap S.F_{\chi(j')}$ if path $S.F_{\delta(i)}$ satisfies the condition at Line 6 in Algorithm 6.2. Consequently, $\ddot{R}_{\chi(k)} \subset S.F_{\chi(i')}$. Also, $\ddot{R}_{\chi(i')} \subseteq S.F_{\chi(i')}$ according to the definition of path $\ddot{R}_{\chi(i')}$, and hence either $\ddot{R}_{\chi(k)} \subset \ddot{R}_{\chi(i')}$ or $\ddot{R}_{\chi(i')} \subseteq \ddot{R}_{\chi(k)}$. If $\ddot{R}_{\chi(k)} \subset \ddot{R}_{\chi(i')}$ then this clearly contradicts the definition of $\ddot{D}$ since $\ddot{D} = \ddot{R}_{\chi(k)}$ per assumption. The case that $\ddot{R}_{\chi(k)} \subset \ddot{R}_{\chi(i')}$ is therefore excluded. If instead $\ddot{R}_{\chi(i')} \subseteq \ddot{R}_{\chi(k)}$ then $\ddot{R}_{\chi(i')} \subset S.F_{\chi(i')} \cap S.F_{\chi(j')}$ since $\ddot{R}_{\chi(k)} \subset S.F_{\chi(i')} \cap S.F_{\chi(j')}$ per assumption. Then however $S.F_{\chi(i')} \cap S.F_{\chi(j')} \not\subseteq \ddot{R}_{\chi(i')}$, which contradicts the definition of path $\ddot{R}_{\chi(i')}$. Therefore the case that $\ddot{R}_{\chi(i')} \subseteq \ddot{R}_{\chi(k)}$ is also excluded, which finally establishes that path $S.F_{\delta(i)}$ does not satisfy the condition at Line 6 in Algorithm 6.2.

Next, it follows then from (ii) in Lemma 6.3 that for all $i \in \{d{+}1, \dots, n\}$, exactly one node $\bar{v}_{\delta i}$ has been created within the run of GCEXKI$(\boldsymbol{P}, \boldsymbol{D}, \ddot{D})$ such that $\{\bar{v}_{\delta i}\} \in \text{nodes}(S.F_{\delta(i)})$. Further, according to (i) in Lemma 6.6, closest$(\bar{v}_{\delta i}, \bar{v}_{\delta j})$ is true for all $i, j \in \{d+1, \dots, n\}$.

Finally, observe that with respect to the sets of nodes $v_{\delta(1)}, \dots, v_{\delta(d)}, \bar{v}_{\delta(d+1)}, \dots, \bar{v}_{\delta(n)}$ and $v'_{\delta(1)}, \dots, v'_{\delta(d)}, \bar{v}_{\delta(d+1)}, \dots, \bar{v}_{\delta(n)}$, (i) in Lemma 6.6 implies that for all $(i,j) \in \{1, \dots, d\} \times \{d{+}1, \dots, n\}$, closest$(v_{\delta(i)}, \bar{v}_{\delta(j)}) =$ closest$(v'_{\delta(i)}, \bar{v}_{\delta(j)}) =$ true, since $\forall i \in \{d{+}1, \dots, n\}$, path $S.F_{\delta(i)}$ does not satisfy the condition at Line 6 in Algorithm 6.2. Hence, the sets of nodes $v_{\delta(1)}, \dots, v_{\delta(d)}, \bar{v}_{\delta(d+1)}, \dots, \bar{v}_{\delta(n)}$ and $v'_{\delta(1)}, \dots, v'_{\delta(d)}, \bar{v}_{\delta(d+1)}, \dots, \bar{v}_{\delta(n)}$ violate σ in T_σ also in case that $\boldsymbol{D} \subset \text{paths}(\sigma)$, which establishes the result.

The counter example tree T_σ satisfies Σ: $\mathrm{T}_\sigma \models \Sigma$ is demonstrated by showing the contradiction that if $\mathrm{T}_\sigma \not\models \Sigma$ then at least one more change to the set of paths $\boldsymbol{X}$ (cf. Lines 1 and 12 in Algorithm 6.1) is possible within the run DXKI(Σ, σ). For this purpose let $\bar{\sigma} \in \Sigma$ be an XKey such that $\mathrm{T}_\sigma \not\models \bar{\sigma}$ and let $\bar{\sigma}$ be of the general form

$$\bar{\sigma} = (\bar{S}, (\bar{F}_1, \dots, \bar{F}_m)).$$

Then, since $\mathrm{T}_\sigma \not\models \bar{\sigma}$ per assumption, there exist nodes $\bar{v}_1, \dots, \bar{v}_m$ and $\bar{v}'_1, \dots, \bar{v}'_m$ in tree T_σ according to Lemma 5.6 such that

- $\forall i \in \{1, \dots, m\}$, $\{\bar{v}_i, \bar{v}'_i\} \subseteq \text{nodes}(\bar{S}.\bar{F}_i)$, and
- $\forall i \in \{1, \dots, m\}$, $\text{val}(\bar{v}_i) = \text{val}(\bar{v}'_i)$, and

– $\forall i, j \in \{1, \ldots, m\}$, closest$(\bar{v}_i, \bar{v}_j)$ = true and closest$(\bar{v}'_i, \bar{v}'_j)$ = true, and
– $\bar{v}_i \neq \bar{v}'_i$ for at least one $i \in \{1, \ldots, m\}$.

Given that $\bar{v}_i \neq \bar{v}'_i$ for at least one $i \in \{1, \ldots, m\}$, one can choose permutation $\pi : \{1, \ldots, m\} \to \{1, \ldots, m\}$ and integer w, where $1 \leq w \leq m$, such that for all $i \in \{1, \ldots, m\}$

$$\bar{v}_{\pi(i)} \begin{cases} \neq \bar{v}'_{\pi(i)} & i \leq w \\ = \bar{v}'_{\pi(i)} & i > w \end{cases}$$

Then, (i) in Lemma 6.3 implies for all $i \in \{1, \ldots, w\}$ that $\bar{S}.\bar{F}_{\pi(i)}$ satisfies the condition at Line 6 in Algorithm 6.2, since $\bar{v}_{\pi(i)}$ and $\bar{v}'_{\pi(i)}$ are distinct nodes in T_σ such that $\{\bar{v}_{\pi(i)}, \bar{v}'_{\pi(i)}\} \subseteq$ nodes$(\bar{S}.\bar{F}_{\pi(i)})$. Further, since val$(\bar{v}_{\pi(i)})$ = val$(\bar{v}'_{\pi(i)})$ for all $i \in \{1, \ldots, w\}$, it follows that the string-values of $\bar{v}_{\pi(i)}$ and $\bar{v}_{\pi(i)}$ have been assigned at Line 18 in Algorithm 6.2 within the run GCEXKI$(\boldsymbol{P}, \boldsymbol{D}, \ddot{D})$, and therefore $\bar{S}.\bar{F}_{\pi(i)}$ satisfies the condition at Line 17. Consequently, $\bar{S}.\bar{F}_{\pi(i)} \in \boldsymbol{D}$, for all $i \in \{1, \ldots, w\}$.

Note that the definition of $\boldsymbol{D}$ implies that $\boldsymbol{D} \subseteq$ paths$(\sigma) - \boldsymbol{X}$. From this together with the observation above that for all $i \in \{1, \ldots, w\}$, $\bar{S}.\bar{F}_{\pi(i)} \in \boldsymbol{D}$ it follows then that $\{\bar{S}.\bar{F}_{\pi(1)}, \ldots, \bar{S}.\bar{F}_{\pi(w)}\} \subseteq$ paths$(\sigma) - \boldsymbol{X}$.

Now, if $w = m$ then $\{\bar{S}.\bar{F}_{\pi(1)}, \ldots, \bar{S}.\bar{F}_{\pi(w)}\}$ = paths$(\bar{\sigma})$ and consequently $\bar{\sigma}$ satisfies the condition at Line 6 in Algorithm 6.1 which establishes the desired contradiction that at least one more change to $\boldsymbol{X}$ is possible within the run DXKI(Σ, σ).

It is therefore shown next that this contradiction also applies in case that $w < m$. Note that $\{\bar{S}.\bar{F}_{\pi(1)}, \ldots, \bar{S}.\bar{F}_{\pi(w)}\} \subseteq$ paths$(\bar{\sigma})$ per definition. Also, $\{\bar{S}.\bar{F}_{\pi(1)}, \ldots, \bar{S}.\bar{F}_{\pi(w)}\} \subseteq$ paths$(\sigma) - \boldsymbol{X}$ as shown above, and therefore $\{\bar{S}.\bar{F}_{\pi(1)}, \ldots, \bar{S}.\bar{F}_{\pi(w)}\} \subseteq$ paths$(\bar{\sigma}) \cap$ paths$(\sigma) - \boldsymbol{X}$. Consequently, $\bar{\sigma}$ satisfies the condition at Line 9 in Algorithm 6.1 if there exists path $\bar{X} \in \boldsymbol{X} \cup \{\rho\}$ such that $\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)} \subseteq \bar{X}$ for all $(i, j) \in \{1, \ldots, w\} \times \{w + 1, \ldots, m\}$. Note that if path $\bar{X}$ exists then $\bar{\sigma}$ satisfies the condition at Line 9 in Algorithm 6.1, and hence establishes the desired contradiction that at least one more change to $\boldsymbol{X}$ is possible.

The crucial prerequisite in order to verify that path $\bar{X}$ exists, is that $\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)} \subseteq \ddot{D}$ for all $(i, j) \in \{1, \ldots, w\} \times \{w+1, \ldots, m\}$, which is shown next. Recall for this purpose that $\bar{S}.\bar{F}_{\pi(i)} \in \boldsymbol{D}$ as shown above. Therefore the definition of $\boldsymbol{D}$ implies that $\ddot{D} \subseteq \bar{S}.\bar{F}_{\pi(i)}$ for all $i \in \{1, \ldots, w\}$. Also, $\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)} \subseteq \bar{S}.\bar{F}_{\pi(i)}$ for all $j \in \{w + 1, \ldots, m\}$ and therefore either $\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)} \subseteq \ddot{D}$ or $\ddot{D} \subset \bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)}$. It therefore remains to exclude the case that $\ddot{D} \subset \bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)}$.

Note for this purpose that node $\bar{v}_{\pi(j)}$ has exactly one ancestor node at path $\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)}$ since $\bar{v}_{\pi(j)} \in$ nodes$(\bar{S}.\bar{F}_{\pi(j)})$ per assumption, and let $\hat{v}$ be this node. Then $\{\hat{v}\}$ = nodes$(\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)}) \cap$ anc-or-self$(\bar{v}_{\pi(j)})$. From this together with Definition 3.22, closest$(\bar{v}_{\pi(i)}, \bar{v}_{\pi(j)})$ = closest$(\bar{v}'_{\pi(i)}, \bar{v}_{\pi(j)})$ = true iff $\hat{v} \in$ anc-or-self$(\bar{v}_{\pi(i)})$ and $\hat{v} \in$ anc-or-self$(\bar{v}'_{\pi(i)})$. Conversely, if $\hat{v} \notin$ anc-or-self$(\bar{v}_{\pi(i)})$ or $\hat{v} \notin$ anc-or-self$(\bar{v}'_{\pi(i)})$ then closest$(\bar{v}_{\pi(i)}, \bar{v}_{\pi(j)})$ = false or closest$(\bar{v}'_{\pi(i)}, \bar{v}_{\pi(j)})$ = false, respectively. Hence, if $\hat{v} \notin$ anc-or-self$(\bar{v}_{\pi(i)})$ or $\hat{v} \notin$ anc-or-self$(\bar{v}'_{\pi(i)})$ then this contradicts the assumption that closest$(\bar{v}_{\pi(i)}, \bar{v}_{\pi(j)})$ = closest$(\bar{v}'_{\pi(i)}, \bar{v}_{\pi(j)})$ = true and therefore excludes the case that $\ddot{D} \subset \bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)}$, and so $\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)} \subseteq \ddot{D}$ $\forall (i, j) \in \{1, \ldots, w\} \times \{w + 1, \ldots, m\}$.

In order to verify that either $\hat{v} \notin$ anc-or-self$(\bar{v}_{\pi(i)})$ or $\hat{v} \notin$ anc-or-self$(\bar{v}'_{\pi(i)})$ in case that $\ddot{D} \subset \bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)}$, note first that $\bar{v}_{\pi(i)} \in \boldsymbol{Z}$ and $\bar{v}'_{\pi(i)} \in \boldsymbol{Z}'$, where $\boldsymbol{Z}, \boldsymbol{Z}'$ are the sets of paths at Line 1 in Algorithm 6.2, according to (i) in Lemma 6.3, since $\bar{S}.\bar{F}_{\pi(i)}$ satisfies the condition at Line 6 in Algorithm 6.2 as shown previously.

Further, path $(\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)})$ satisfies the condition at Line 6 in Algorithm 6.2, since $\ddot{D} \subset (\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)})$ per assumption, and clearly $\bar{S}.\bar{F}_{\pi(i)} \cap (\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)}) = (\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)})$, and therefore path $D \in \boldsymbol{D}$ exists such that $\ddot{D} \subset D \cap (\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)})$, where $D = \bar{S}.\bar{F}_{\pi(i)}$. Note that $\bar{S}.\bar{F}_{\pi(i)} \in \boldsymbol{D}$ per assumption.

Given that path $(\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)})$ satisfies the condition at Line 6 in Algorithm 6.2, (i) in Lemma 6.3 implies that either $\hat{v} \in \boldsymbol{Z}$ or $\hat{v} \in \boldsymbol{Z}'$. Now, if $\hat{v} \in \boldsymbol{Z}$ then this contradicts (i) in Lemma 6.3, in particular that anc-or-self$(\bar{v}'_{\pi(i)}) \cap \boldsymbol{Z} = \emptyset$, since $\bar{v}'_{\pi(i)} \in \boldsymbol{Z}'$ and $\hat{v} \in$ anc-or-self$(\bar{v}'_{\pi(i)})$ per assumption. If instead $\hat{v} \in \boldsymbol{Z}'$ then anc-or-self$(\bar{v}_{\pi(i)}) \cap \boldsymbol{Z}' \neq \emptyset$, since $\bar{v}_{\pi(i)} \in \boldsymbol{Z}$ as shown above and $\hat{v} \in$ anc-or-self$(\bar{v}_{\pi(i)})$ per assumption, and therefore (i) in Lemma 6.3 is again contradicted. Therefore, either $\hat{v} \notin$ anc-or-self$(\bar{v}_{\pi(i)})$ or $\hat{v} \notin$ anc-or-self$(\bar{v}'_{\pi(i)})$, as desired.

Given that $\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)} \subseteq \ddot{D}$ for all $(i, j) \in \{1, \ldots, w\} \times \{w+1, \ldots, m\}$ it is easily verified that path $\bar{X} \in \boldsymbol{X} \cup \{\rho\}$ exists such that $\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)} \subseteq \bar{X}$ for all $(i, j) \in \{1, \ldots, w\} \times \{w+1, \ldots, m\}$. For this purpose, assume first that $\boldsymbol{X} = \emptyset$. Then, $\ddot{D} = \rho$ per definition, and this together with the result that $\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)} \subseteq \ddot{D}$ for all $(i, j) \in \{1, \ldots, w\} \times \{w+1, \ldots, m\}$ implies that path $\bar{X}$ exists, where $\bar{X} = \ddot{D} = \rho$.

If instead $\boldsymbol{X} \neq \emptyset$, then $\ddot{D} = \ddot{R}_{\chi(k)}$ per definition, where $\ddot{R}_{\chi(k)}$ is a path in the set of prefix paths $\{R_{\chi(1)}, \ldots, R_{\chi(x)}\}$ of paths $\{S.F_{\chi(1)}, \ldots, S.F_{\chi(x)}\} = \boldsymbol{X}$. Then, since $\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)} \subseteq \ddot{D}$ for all $(i, j) \in \{1, \ldots, w\} \times \{w+1, \ldots, m\}$ it follows that $\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)} \subseteq \ddot{R}_{\chi(k)}$, and consequently also $\bar{S}.\bar{F}_{\pi(i)} \cap \bar{S}.\bar{F}_{\pi(j)} \subseteq S.F_{\chi(k)}$ since $\ddot{R}_{\chi(k)} \subseteq S.F_{\chi(k)}$ per definition. Therefore, path $\bar{X}$ also exists in case that $\boldsymbol{X} \neq \emptyset$, where $\bar{X} = S.F_{\chi(k)}$. Note that $S.F_{\chi(k)} \in \boldsymbol{X}$ per definition. Hence, $\mathrm{T}_\sigma \vDash \Sigma$. □

We now finally present our result on the completeness of inference rules R1 - R7.

Theorem 6.5 (Completeness of Inference Rules for XKeys) *Given a set of XKeys Σ and a single XKey σ, if $\Sigma \vDash \sigma$ then $\Sigma \vdash \sigma$.*

Proof (Theorem 6.5) If $\Sigma \vDash \sigma$ then DXKI(Σ, σ) = true from Theorem 6.4, and if DXKI(Σ, σ) = true then either $\sigma \in \Sigma$ or $\Sigma \vdash \sigma$ from Lemma 6.1. Now, if $\sigma \in \Sigma$ then $\Sigma \vdash \sigma$ trivially holds true, and thus $\Sigma \vDash \sigma \Rightarrow \Sigma \vdash \sigma$. □

Chapter 7

Reasoning about XINDs

Contents

Section 7.1 introduces the class of core XINDs, which excludes the small subset of XINDs which interact with structural constraints in the model of XML data presented in Chapter 3. Section 7.2 then presents a *chase* algorithm for core XINDs in complete XML trees, which is the primary tool for our reasoning. Using the *chase* algorithm, we then solve the consistency and implication problems related to core XINDs in complete XML trees in Section 7.3 and 7.4, respectively. In particular, Section 7.4 presents a set of sound and complete inference rules for core XINDs in complete XML trees, and also a decision procedure for the implication of core XINDs.

7.1 The Class of Core XINDs

Our motivation for defining the class of core XINDs is based on the belief that an XIND satisfaction should not imply, as a hidden side effect, that each node in a set of nodes in an XML tree T has the same value. We now justify this. Suppose that for the RHS selector of an XIND $\sigma = (S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n])$, $S' = \rho$, and for some $i \in \{1, \ldots, n\}$, the RHS field F'_i is an attribute label. Then since there is only one root node, and in turn at most one attribute node in nodes($S'.F'_i$, T), the semantics of σ means, that every node in nodes($S.F_i$, T) $\cup$ nodes($S'.F'_i$, T) must have the same value. We believe that this not the intent of an XIND, and that such a constraint should be specified instead explicitly in a DTD or XSD. Since the study of the interaction between structural constraints and integrity constraints is known to be a complex one [67], and outside the scope of this thesis, we exclude such an XIND and this leads to the following definition.

Definition 7.1 (Core XIND) An XIND $(S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n])$ is defined to be a *core* XIND, if in case that $S' = \rho$, then there does not exist a RHS field F'_i such that length(F'_i) = 1 and last(F'_i) $\in \boldsymbol{L}^A$.

7.2 The Chase for Core XINDs

The primary tool in our reasoning on XINDs is a chase algorithm given in Algorithm 7.1, which we will simply call *the chase* in the remainder of this chapter. The chase is a recursive algorithm that takes as input a set of paths $\boldsymbol{P}$, an XML tree T that is complete w.r.t $\boldsymbol{P}$, and a set of XINDs Σ that conforms to $\boldsymbol{P}$. The chase adds new nodes to T such that T $\models \Sigma$. From a bird-eyes view, the chase halts if the input XML tree T_s for a (recursive) step s satisfies Σ, and otherwise the chase:

(i) chooses an XIND $\sigma_s = (S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n])$ from Σ such that $\mathrm{T}_s \not\models \sigma_s$ because of a list of LHS field nodes $[v_1, \ldots, v_n]$

(ii) creates new nodes in T_s, such that the resulting XML tree T_{s+1} contains a list of RHS field nodes $v'_1, \ldots, v'_n$ that remove the violation of σ_s.

We now illustrate a step in the chase by an example.

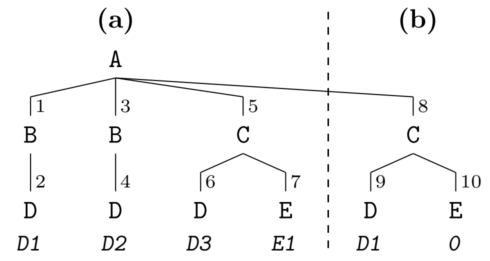


Figure 7.1: Step in the XIND chase.

Algorithm 7.1 CHASE - The Chase for XINDs.

```
in:   a downward-closed set of paths $\boldsymbol{P}$
      an XML tree $\mathrm{T} = (\boldsymbol{V}, \boldsymbol{E}, \mathrm{lab}, \mathrm{val})$ that is complete w.r.t. $\boldsymbol{P}$
      a set of core XINDs $\Sigma$ that conform to $\boldsymbol{P}$
out:  an XML tree $\bar{\mathrm{T}}$ that subsumes T, is complete w.r.t. $\boldsymbol{P}$ and satisfies $\Sigma$
1:  if $\mathrm{T} \models \Sigma$ then
2:      return T
3:  else
4:      let $\{R_1, \ldots, R_m\} = \boldsymbol{P}$ such that $\forall\, i, j \in \{1, \ldots, m\}$, length$(R_i) \leq$ length$(R_j)$ if $i \leq j$
5:      let $\sigma = (S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n])$ be an XIND in $\Sigma$ such that $\mathrm{T} \not\models \sigma$
6:      let $[v_1, \ldots, v_n]$ be a list of nodes that violates $\sigma$ in T
7:      let $\boldsymbol{X} = \{\mathrm{root(T)}\}$
8:      for $i := 1$ to $m$ do                                         ▷ remove violation
9:          if there exists path $F'_x \in [F'_1, \ldots, F'_n]$ such that $R_i \cap S'.F'_x \neq \rho$ then
10:             $\{\hat{v}\} \leftarrow$ nodes(parent$(R_i)$, T) $\cap\, \boldsymbol{X}$
11:             $v \leftarrow$ newnode$(\boldsymbol{V})$; $\boldsymbol{X} \leftarrow \boldsymbol{X} \cup \{v\}$
12:             lab$(v) \leftarrow$ last$(R_i)$
13:             $\boldsymbol{E} \leftarrow \boldsymbol{E} \cup (\hat{v}, v)$
14:             if there exists path $F'_y \in [F'_1, \ldots, F'_n]$ such that $R_i = S'.F'_y$ then
15:                 val$(v) \leftarrow$ val$(v_y)$;                     ▷ $v_y \in [v_1, \ldots, v_n]$
16:             else if last$(R_i) \in \boldsymbol{L}^A \cup \{\mathtt{S}\}$ then
17:                 val$(v) \leftarrow$ "0"
18:             end if
19:         end if
20:     end for
21:     return CHASE$(\boldsymbol{P}, \mathrm{T}, \Sigma)$
22: end if
```

Example 7.1 (chase step) *Consider the downward-closed set of paths* $\boldsymbol{P} = \{$`A`, `A.B`, `A.B.D`, `A.C`, `A.C.D`, `A.C.E`$\}$ *and the core XIND* $\sigma = (\mathtt{A.B}, [\mathtt{D}]) \subseteq (\mathtt{A.C}, [\mathtt{D}])$. *Then, XML tree* T_σ *depicted in Figure 7.1a violates* σ *because of both* $[v_2]$ *and* $[v_4]$. *Given that* $[v_2]$ *is chosen at Line 6, the chase creates nodes* v_8 *and* v_9 *in XML tree* $\mathrm{T}_{\sigma+1}$ *(cf. Figure 7.1b), which remove the violation, and then adds* v_{10} *as a child of* v_8 *in order that* $\mathrm{T}_{\sigma+1}$ *is complete w.r.t.* $\boldsymbol{P}$.

We now introduce the notion of XML tree subsumption, which we require in order to establish the completeness of our inference rules for core XINDs in Section 7.4.1.

Definition 7.2 (Subsumption of XML Trees) An XML tree $\mathrm{T} = (\boldsymbol{V}, \boldsymbol{E}, \mathrm{lab}, \mathrm{val})$ is defined to be *subsumed* within XML tree $\bar{\mathrm{T}} = (\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}}, \overline{\mathrm{lab}}, \overline{\mathrm{val}})$, denoted by $\mathrm{T} \simeq \bar{\mathrm{T}}$, if there exists the *subsumption mapping* $\alpha : \boldsymbol{V} \to \bar{\boldsymbol{V}}$ such that:

(i) $\alpha(\mathrm{root(T)}) = \mathrm{root}(\bar{\mathrm{T}})$, and
(ii) if $(v, \tilde{v}) \in \boldsymbol{E}$, then $(\alpha(v), \alpha(\tilde{v})) \in \bar{\boldsymbol{E}}$, and
(iii) for every node $v \in V$, $\mathrm{lab}(v) = \overline{\mathrm{lab}}\,(\alpha(v))$, and
(iv) for every attribute or text node $v \in V$, $\mathrm{val}(v) = \overline{\mathrm{val}}\,(\alpha(v))$.

We then have the following result on the chase.

Lemma 7.1 Let T be an XML tree that is complete with respect to a downward-closed set of paths $\boldsymbol{P}$, and let Σ be a set of XINDs which conform to $\boldsymbol{P}$. Then, CHASE($\boldsymbol{P}$, T, Σ) *terminates* and returns an XML tree $\bar{\mathrm{T}}$ such that:
(i) $\bar{\mathrm{T}}$ *subsumes* T;
(ii) $\bar{\mathrm{T}}$ is *complete* w.r.t. $\boldsymbol{P}$;
(iii) $\bar{\mathrm{T}}$ *satisfies* Σ.

We now establish a couple of preliminary results that we require in order to demonstrate Lemma 7.1.

Lemma 7.2 Let $\mathrm{T} = (\boldsymbol{V}, \boldsymbol{E}, \mathrm{lab}, \mathrm{val})$ and $\bar{\mathrm{T}} = (\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}}, \overline{\mathrm{lab}}, \overline{\mathrm{val}})$ be XML trees such that $\mathrm{T} \simeq \bar{\mathrm{T}}$, and let $\alpha : \boldsymbol{V} \to \bar{\boldsymbol{V}}$ be the subsumption mapping. Then, for every pair of nodes v and $\tilde{v}$ in V:
(i) if $v = \mathrm{parent}(\tilde{v})$ then $\alpha(v) = \mathrm{parent}(\alpha(\tilde{v}))$;
(ii) if $v \in \text{anc-or-self}(\tilde{v})$ then $\alpha(v) \in \text{anc-or-self}(\alpha(\tilde{v}))$;
(iii) if $v_1.\cdots.v_n$ is a walk in T where $v = v_n$, then $\alpha(v_1).\cdots.\alpha(v_n)$ is a walk in $\bar{\mathrm{T}}$;
(iv) if P is the path such that $v \in \mathrm{nodes}(P, \mathrm{T})$ then $\alpha(v) \in \mathrm{nodes}(P, \bar{\mathrm{T}})$;
(v) if $\mathrm{closest}(v, \tilde{v}) = \mathrm{true}$ then $\mathrm{closest}(\alpha(v), \alpha(\tilde{v})) = \mathrm{true}$.

Proof (Lemma 7.2) (i) Given that $v = \mathrm{parent}(\tilde{v})$, $(v, \tilde{v})$ is an edge in $\boldsymbol{E}$. Consequently, $(\alpha(v), \alpha(\tilde{v}))$ is an edge in $\bar{\boldsymbol{E}}$ from (ii) in Definition 7.2 and thus $\alpha(v) = \mathrm{parent}(\alpha(\tilde{v}))$.

(ii) Given that $v \in \text{anc-or-self}(\tilde{v})$, either $v = \tilde{v}$ or $v \in \mathrm{ancestor}(\tilde{v})$. If $v = \tilde{v}$ then $\alpha(v) = \alpha(\tilde{v})$ according to Definition 7.2 and thus $\alpha(v) \in \text{anc-or-self}(\alpha(\tilde{v}))$. If instead $v \in \mathrm{ancestor}(v)$, then $\alpha(v) \in \mathrm{ancestor}(\alpha(\tilde{v}))$ follows from (i) in Lemma 7.2 since function ancestor returns the transitive closure of parents of a node per definition. Therefore, $\alpha(v) \in \text{anc-or-self}(\alpha(\tilde{v}))$ also in case that $v \in \mathrm{ancestor}(\tilde{v})$.

(iii) From (i) in Lemma 7.2, $\alpha(v_{i-1}) = \mathrm{parent}(\alpha(v_i))$ for all $i \in \{1, \ldots, n\}$ where $i > 1$. From this together with (i) in Definition 7.2, $\alpha(v_1).\cdots.\alpha(v_n)$ is a walk in $\bar{\mathrm{T}}$.

(iv) Let $p = v_1.\cdots.v_n$ be the walk in T such that $v = \mathrm{last}(p)$. Then $\alpha(v_1).\cdots.\alpha(v_n)$ is a walk in $\bar{\mathrm{T}}$ according to (iii) in Lemma 7.2. Further, given that $v \in \mathrm{nodes}(P, \mathrm{T})$ and that $v = \mathrm{last}(p)$, path $P = l_1.\cdots.l_n$ where for all $i \in \{1, \ldots, n\}$, $l_i = \mathrm{lab}(v_i)$. Also, $\alpha(v) = \alpha(v_n)$ given that $v = \mathrm{last}(p) = v_n$, and thus $\alpha(v) = \alpha(v_n) \in \mathrm{nodes}(P, \bar{\mathrm{T}})$ since $\forall i \in \{1, \ldots, n\}$, $\mathrm{lab}(\alpha(v_i)) = \mathrm{lab}(v_i)$ from (iii) in Definition 7.2, and $\mathrm{lab}(v_i) = l_i$ as shown previously.

(v) Given that $\mathrm{closest}(v, \tilde{v}) = \mathrm{true}$, there exists node $\hat{v}$ in T according to Definition 3.22 such that $\hat{v} \in \text{anc-or-self}(v)$, and $\hat{v} \in \text{anc-or-self}(\tilde{v})$ and $\hat{v} \in \mathrm{nodes}(P \cap \tilde{P}, \mathrm{T})$, where P and $\tilde{P}$ are the paths such that $v \in \mathrm{nodes}(P, \mathrm{T})$ and $\tilde{v} \in \mathrm{nodes}(\tilde{P}, \mathrm{T})$. We now show that $\mathrm{closest}(\alpha(v), \alpha(\tilde{v})) = \mathrm{true}$ by showing that node $\alpha(\hat{v})$ satisfies (i) - (iii) in Definition 3.22 w.r.t. nodes $\alpha(v)$ and $\alpha(\tilde{v})$. Thereby, (i) and (ii) in Definition 3.22, i.e. that $\alpha(\hat{v}) \in \text{anc-or-self}(\alpha(v))$ and $\alpha(\hat{v}) \in \text{anc-or-self}(\alpha(\tilde{v}))$, follow from (ii) in Lemma 7.2 and the assumption that $\hat{v} \in \text{anc-or-self}(v)$ and $\hat{v} \in \text{anc-or-self}(\tilde{v})$, respectively. Further, (iii) in Definition 3.22, i.e. that $\alpha(\hat{v}) \in \mathrm{nodes}(P \cap \tilde{P}, \bar{\mathrm{T}})$, follows from (iv) in Lemma 7.2 and the assumption that $\hat{v} \in \mathrm{nodes}(P \cap \tilde{P}, \mathrm{T})$. □

We now introduce some terminology which we will use in the remainder of this chapter. The chase is recursive, and hence we use T_s to denote the input XML tree for a (recursive) step s in the algorithm. Also, in case that $\mathrm{T}_s \not\models \Sigma$ in step s, we use σ_s to denote the XIND chosen at Line 5, and $\boldsymbol{X}_s$ to denote the set of nodes at Line 7.

Lemma 7.3 Let T be an XML tree that is complete with respect to a downward-closed set of paths $\boldsymbol{P} = \{R_1, \ldots, R_m\}$, and let Σ be a set of core XINDs which conform to $\boldsymbol{P}$. Then, for every iteration $i \in \{1, \ldots, m\}$ of the loop at Line 8 in Algorithm 7.1 in a step s of CHASE($\boldsymbol{P}$, T, Σ)

(i) if R_i satisfies the condition at Line 9, then length$(R_i) \geq 2$;
(ii) if R_i satisfies the condition at Line 9 and length$(R_i) > 2$, then parent(R_i) satisfies the condition at Line 9;
(iii) if the condition at Line 9 is met for the first time within step s, then length$(R_i) = 2$;

Proof (Lemma 7.3) (i) Assume to the contrary, that length$(R_i) = 1$. Then $R_i = \rho$ and consequently for all $F'_x \in [F'_1, \ldots, F'_n]$, $R_i \cap S'.F'_x = \rho$. This however contradicts the assumption that R_i satisfies the condition at Line 9 and thus length$(R_i) \geq 2$.

(ii) Since path R_i satisfies the condition at Line 9 per assumption, there exists path $F'_x \in [F'_1, \ldots, F'_n]$, such that $R_i \cap S'.F'_x \neq \rho$. Now, if parent$(R_i) \neq \rho$, then also parent$(R_i) \cap S'.F'_x \neq \rho$, since parent$(R_i) \subset R_i$ per definition. Thereby, since length$(R_i) > 2$ per assumption, it follows that length(parent(R_i)) > 1 and consequently parent$(R_i) \neq \rho$. Thus, parent(R_i) satisfies the condition at Line 9.

(iii) Since R_i satisfies the condition at Line 9 per assumption, it follows that length$(R_i) \geq 2$ according to (i) in Lemma 7.3, and it therefore remains to show that $2 \leq$ length(R_i). For this purpose, assume to the contrary that length$(R_i) > 2$. Then, path parent(R_i) exists and also parent$(R_i) \in \{R_1, \ldots, R_m\}$, since paths $\{R_1, \ldots, R_m\}$ are downward-closed per assumption. Also, path parent(R_i) satisfies the condition at Line 9 according to (ii) in Lemma 7.3. This however contradicts the assumption that the condition at Line 9 is satisfied within step s for the first time in iteration i, since paths $\{R_1, \ldots, R_m\}$ are ordered by length according to Line 4, and therefore R_i succeeds path parent(R_i) within the ordered set of paths $\{R_1, \ldots, R_m\}$. □

Lemma 7.4 Let T be an XML tree that is complete with respect to a downward-closed set of paths $\boldsymbol{P} = \{R_1, \ldots, R_m\}$, and let Σ be a set of core XINDs which conform to $\boldsymbol{P}$. If a node v is created in a step s of CHASE($\boldsymbol{P}$, T, Σ),

(i) $\{v\} = \text{nodes}(R_i, \mathrm{T}_s) \cap \boldsymbol{X}_s$, where i denotes the iteration of the loop at Line 8 in which node v was created;
(ii) $\{\text{root}(\mathrm{T}_s)\} = \text{anc-or-self}(v) \cap \text{anc-or-self}(\tilde{v})$, if $\tilde{v}$ is a node in T_s such that $\tilde{v} \notin \boldsymbol{X}_s$;
(iii) closest$(v, \tilde{v})$ = true, if $\tilde{v}$ is a node in T_s such that $\tilde{v} \in \boldsymbol{X}_s$.

Proof (Lemma 7.4) We demonstrate Lemma 7.4 by induction over the iterations of the loop at Line 8. We note that v is created in an iteration i of the loop at Line 8 iff path R_i satisfies the condition at Line 9. Also, we use v_i to denote that node v was created in iteration i.

Base Case: We assume that the condition at Line 9 is met for the first time within iteration x of the loop at Line 8 in step s, and we use this iteration as base case.

(i) From (iii) in Lemma 7.3, $\text{length}(R_x) = 2$ and therefore $\text{parent}(R_x) = \rho$. Hence, $\hat{v} = \text{root}(\mathrm{T}_s)$ at Line 10. We note that $\text{root}(\mathrm{T}_s) \in \boldsymbol{X}_s$ according to Line 7. Given that $\hat{v} = \text{root}(\mathrm{T}_s)$, edge $(\text{root}(\mathrm{T}_s), v_x)$ is added to $\boldsymbol{E}$ at Line 13 and therefore $\text{parent}(v_x) = \text{root}(\mathrm{T}_s)$. The resulting walk $\text{root}(\mathrm{T}_s).v_x$ is a walk over path R_x, since R_x is of the form $\rho.\text{last}(R_x)$, given that $\text{length}(R_x) = 2$, and $\text{lab}(v_x) = \text{last}(R_x)$ according to Line 12. Consequently, $v_x \in \text{nodes}(R_x, \mathrm{T}_s)$. Also, $\{v_x\} = \text{nodes}(R_x, \mathrm{T}_s) \cap \boldsymbol{X}_s$, since v_x is the first node added to $\boldsymbol{X}_s$ in step s, given that the condition at Line 9 is met for the first time in iteration x.

(ii) We show that if $\tilde{v}$ is a node in T_s and $\tilde{v} \notin \boldsymbol{X}_s$, then $\text{anc-or-self}(v_x) \cap \text{anc-or-self}(\tilde{v}) = \{\text{root}(\mathrm{T}_s)\}$. Given that $\text{parent}(v_x) = \text{root}(\mathrm{T}_s)$ it follows that $\text{anc-or-self}(v_x) = \{\text{root}(\mathrm{T}_s), v_x\}$. Therefore, if $\text{anc-or-self}(\tilde{v}) \cap \text{anc-or-self}(v_x) \neq \{\text{root}(\mathrm{T}_s)\}$, then either $v_x = \tilde{v}$ or $v_x \in \text{ancestor}(\tilde{v})$. However, $\tilde{v} \neq v_x$ follows from the assumption that $\tilde{v} \notin \boldsymbol{X}_s$ and the observation that $v_x \in \boldsymbol{X}_s$ according to Line 11. Also, $v_x \notin \text{ancestor}(\tilde{v})$ since v_x is newly created per assumption and is therefore a leaf node in T_s within iteration x.

(iii) If $\tilde{v}$ is a node in T_s and $\tilde{v} \in \boldsymbol{X}_s$, then either $\tilde{v} = v_x$ or $\tilde{v} = \text{root}(\mathrm{T}_s)$. In both cases $\text{closest}(v_x, \tilde{v})$ follows directly from Definition 3.22.

Inductive Step: We now assume that Lemma 7.4 holds true until iteration y, where $y \geq x$, and establish the inductive step by showing that Lemma 7.4 also holds true for the iteration z, where $z > y$, within which a node is created next.

(i) We show that there exists node $\hat{v}$ at Line 10 such that $\{\hat{v}\} = \text{nodes}(\text{parent}(R_z), \mathrm{T}_s) \cap \boldsymbol{X}_s$. We note that either $\text{length}(R_z) = 2$ or $\text{length}(R_z) > 2$ according to (i) in Lemma 7.3. If $\text{length}(R_z) = 2$ then $\text{parent}(R_z) = \rho$, and therefore $\{\hat{v}\} = \text{nodes}(\text{parent}(R_z), \mathrm{T}_s) \cap \boldsymbol{X}_s$ follows, since $\hat{v} = \text{root}(\mathrm{T}_s)$ given that $\text{parent}(R_z) = \rho$. If instead $\text{length}(R_z) > 2$, then the inductive assumption applies, since $\text{parent}(R_z)$ satisfies the condition at Line 9 according to (ii) in Lemma 7.3, and $\text{parent}(R_z)$ precedes R_z within the ordered set of paths $R_1, \ldots, R_m$. Thus, $\{\hat{v}\} = \text{nodes}(\text{parent}(R_z), \mathrm{T}_s) \cap \boldsymbol{X}_s$ follows from (i) in Lemma 7.4 if $\text{length}(R_z) > 2$.

Given that $\{\hat{v}\} = \text{nodes}(\text{parent}(R_z), \mathrm{T}_s) \cap \boldsymbol{X}_s$ it follows that $\text{parent}(v_z) = \hat{v}$, since edge $(\hat{v}, v_z)$ is added to $\boldsymbol{E}$ at Line 13. Consequently, $v_z \in \text{nodes}(R_z, \mathrm{T}_s)$ since $\text{lab}(v_z) = \text{last}(R_z)$ according to Line 12. Further, $v_z \in \boldsymbol{X}_s$ according to Line 11, and therefore $v_z \in \text{nodes}(R_z, \mathrm{T}_s) \cap \boldsymbol{X}_z$. Also, $\{v_z\} = \text{nodes}(R_z, \mathrm{T}_s) \cap \boldsymbol{X}_z$, because if there exists to the contrary a node $\tilde{v} \in \boldsymbol{X}_s$ such that $\tilde{v} \in \text{nodes}(R_z, \mathrm{T}_s)$ and $\tilde{v} \neq v_z$, then either $\tilde{v} = \text{root}(\mathrm{T}_s)$ or $\tilde{v}$ was created in a previous iteration of the loop at Line 8 for some path $\tilde{R} \in [R_1, \ldots, R_m]$. We exclude the case that $\tilde{v} = \text{root}(\mathrm{T}_s)$ since then $R_z = \rho$, which contradicts (i) in Lemma 7.3. We also exclude the case that $\tilde{v}$ was previously created, since then $R_z = \tilde{R}$ which contradicts the assumption that paths $R_1, \ldots, R_m$ do not contain duplicates.

(ii) We show that if $\tilde{v}$ is a node in T_s and $\tilde{v} \notin \boldsymbol{X}_s$, then $\text{anc-or-self}(v_z) \cap \text{anc-or-self}(\tilde{v}) = \{\text{root}(\mathrm{T}_s)\}$. We note that $\text{root}(\mathrm{T}_s) \in \text{anc-or-self}(v_z)$, since $v_z \in \text{nodes}(R_z, \mathrm{T}_s)$ as shown above, and also that $\text{root}(\mathrm{T}_s) \in \text{anc-or-self}(\tilde{v})$, since initially T_s is a tree and $\tilde{v}$ exists in T_s before the first iteration of the loop at Line 8 given that $\tilde{v} \notin \boldsymbol{X}_s$. Therefore, if to the contrary $\text{anc-or-self}(v_z) \cap \text{anc-or-self}(\tilde{v}) \neq \{\text{root}(\mathrm{T}_s)\}$, then either $\text{anc-or-self}(\text{parent}(v_z)) \cap \text{anc-or-self}(\tilde{v}) \neq \{\text{root}(\mathrm{T}_s)\}$ or $v_z \in \text{anc-or-self}(\tilde{v})$. However, $v_z \notin \text{anc-or-self}(\tilde{v})$ since $v_z \neq \tilde{v}$ given that $v_z \in \boldsymbol{X}_s$ but $\tilde{v} \notin \boldsymbol{X}_s$, and also $v_z \notin \text{ancestor}(\tilde{v})$ given that v_z is newly created and therefore a leaf node in T_s within iteration z. It therefore remains to show that anc-or-self(parent(v_z)) $\cap$ anc-or-self($\tilde{v}$) $=$ $\{\text{root}(\mathrm{T}_s)\}$. Thereby, parent($v_z$) $= \hat{v}$ as shown previously, and therefore either parent(v_z) $=$ root(T_s) or parent(v_z) was newly created in a previous iteration of the loop at Line 8. Now, if parent(v_z) $=$ root(T_s) then anc-or-self(parent(v_z)) $=$ root(T_s) and consequently anc-or-self(parent(v_z)) $\cap$ anc-or-self($\tilde{v}$) $= \{\text{root}(\mathrm{T}_s)\}$. If instead parent($v_z$) $\neq$ root(T_s), then anc-or-self(parent(v_z)) $\cap$ anc-or-self($\tilde{v}$) $= \{\text{root}(\mathrm{T}_s)\}$ from the inductive assumption.

(iii) We show that if $\tilde{v}$ is a node in T_s and $\tilde{v} \in \boldsymbol{X}_s$, then closest($v_z, \tilde{v}$) $=$ true. If $\tilde{v} = v_z$, then closest($v_z, \tilde{v}$) $=$ true from Definition 3.22. We therefore assume that $v_z \neq \tilde{v}$, and show first that closest($v_z, \tilde{v}$) $=$ true, if parent(v_z) $\notin$ ancestor($\tilde{v}$). Then, from Definition 3.22, closest($v_z, \tilde{v}$) $=$ true iff closest(parent(v_z), $\tilde{v}$) $=$ true. Since parent(v_z) $\in \boldsymbol{X}_s$ per assumption, it follows from the inductive assumption that closest(parent(v_z), $\tilde{v}$) $=$ true, and hence also closest($v_z, \tilde{v}$) $=$ true.

If instead parent(v_z) $\in$ ancestor($\tilde{v}$), then $\{$parent(v_z)$\}$ $=$ anc-or-self($\tilde{v}$) $\cap$ anc-or-self(v_z), since v_z is newly created per assumption and v_z is therefore a leaf node in tree T_s within iteration z of the loop at Line 8. We note that $\tilde{v} \neq v_z$ per assumption. Given that $\{$parent(v_z)$\}$ $=$ anc-or-self($\tilde{v}$) $\cap$ anc-or-self(v_z), it follows that closest($v_z, \tilde{v}$) $=$ true if $R_z \cap \tilde{R} =$ parent(R_z), where $\tilde{R}$ is the path such that $\tilde{v} \in$ nodes($\tilde{R}, \mathrm{T}_s$). We observe that parent(R_z) $\subseteq R_z \cap \tilde{R}$, given that parent($v_z$) $\in$ anc-or-self(v_z)$\cap$anc-or-self($\tilde{v}$). Consequently, $R_z \cap \tilde{R} = R_z$ if parent(R_z) $\neq R_z \cap \tilde{R}$. Then, however, the assumption that $\{$parent(v_z)$\}$ $=$ anc-or-self($\tilde{v}$) $\cap$ anc-or-self(v_z) implies that there exists node $\bar{v} \in$ anc-or-self($\tilde{v}$), such that $\bar{v} \in$ nodes(R_z, T_s) and $\bar{v} \neq v_z$. Further, given that $\bar{v} \in$ anc-or-self($\tilde{v}$) and that $\tilde{v} \in \boldsymbol{X}_s$, it follows from the inductive assumption, in particular (ii) in Lemma 7.4, that $\bar{v} =$ root(T_s), if $\bar{v} \notin \boldsymbol{X}_s$. However, given that $\bar{v} \in$ nodes(R_z, T_s), $\bar{v} \neq$ root(T_s), since length(R_z) ≥ 2 according to (i) in Lemma 7.3. Now, given that $\bar{v} \in \boldsymbol{X}_s$ and that $\bar{v} \neq$ root(T_s), node $\bar{v}$ was created in a previous iteration of the loop at Line 8. This however clearly contradicts the inductive assumption, in particular (i) in Lemma 7.4, given that both $\bar{v}$ and v_z are nodes in nodes(R_z, T_s) and that $\bar{v} \neq v_z$. Consequently, $R_z \cap \tilde{R} =$ parent(R_z) and thus closest($v_z, \tilde{v}$) $=$ true. □

We now present a preliminary result on the properties of the XML tree T_{s+1} returned from a step s in the chase. This result is central to the subsequent proof of Lemma 7.1.

Lemma 7.5 Let T be an XML tree that is complete with respect to a downward-closed set of paths $\boldsymbol{P}$, and let Σ be a set of core XINDs which conform to $\boldsymbol{P}$. If $\mathrm{T}_s \not\models \Sigma$ in a step s of CHASE($\boldsymbol{P}$, T, Σ), then an XML tree T_{s+1} is returned such that
(i) T_{s+1} conforms to Definition 3.11;
(ii) T_{s+1} subsumes T_s;
(iii) T_{s+1} is complete w.r.t. $\boldsymbol{P}$.

Proof (Lemma 7.5) We note that a single step in the chase terminates since the input set of paths $\boldsymbol{P}$ is finite and hence the loop at Line 8 in Algorithm 7.1 is iterated only for a finite number of times.

(i) The chase iteratively creates XML tree T_{s+1} by means of adding nodes and edges within the loop at Line 8 to XML tree T_s. Now, let for all $i \in \{1, \ldots, m\}$, where m is the number of paths in $\boldsymbol{P}$, T_{s_i} be XML tree T_s after iteration i of the loop at Line 8. Then, from the procedure of the chase, $\mathrm{T}_{s_m} = \mathrm{T}_{s+1}$. We therefore establish (i) in Lemma 7.5 by induction over the sequence of XML trees $\mathrm{T}_{s_1}, \ldots, \mathrm{T}_{s_m}$ created in step s. We show in particular, that for all $k \in \{1, \ldots, m\}$, XML tree T_{s_k} satisfies requirements (1) - (4) in Definition 3.11, which means that

(1) $(\boldsymbol{V}, \boldsymbol{E})$ is a tree in terms of Definition 3.3 and hence
  (i) $\boldsymbol{V}$ is a finite set of nodes, from Definition 3.1
  (ii) $\forall(v, \bar{v}) \in \boldsymbol{E}$, $v \neq \bar{v}$, from Definition 3.1.i
  (iii) $\forall v \in \boldsymbol{V}$, if $\boldsymbol{V} \neq \{v\}$ then $\exists \bar{v} \in \boldsymbol{V}$ such that $(v, \bar{v}) \in \boldsymbol{E}$ or $(\bar{v}, v) \in \boldsymbol{E}$, from Definition 3.1.ii
  (iv) if there exist nodes $v_1. \cdots .v_n \in \boldsymbol{V}$ such that for all $i \in \{1, \ldots, n\}$, $(v_{i-1}, v_i) \in \boldsymbol{E}$, then $v_1 \neq v_n$ if $n > 1$, from Definition 3.3
(2) $\forall v \in \boldsymbol{V}$, $\mathrm{lab}(v) \in \boldsymbol{L}$.
(3) $\forall v \in \boldsymbol{V}$, if $\mathrm{lab}(v) \in \boldsymbol{L}^A \cup \{\mathtt{S}\}$ then $\mathrm{val}(v) \in \boldsymbol{U}$ and $\mathrm{val}(v)$ is undefined otherwise.
(4) $\forall(v, \bar{v}) \in \boldsymbol{E}$
  (i) $\mathrm{lab}(v) \in \boldsymbol{L}^E$,
  (ii) if $\bar{v} \in \boldsymbol{L}^A$ then $\nexists(v, \bar{v}') \in \boldsymbol{E}$ such that $\bar{v} \neq \bar{v}'$ and $\mathrm{lab}(\bar{v}) = \mathrm{lab}(\bar{v}')$

Base Case: XML tree T_{s_1} is the base case for the induction. Because the set of paths $R_1, \ldots, R_m$ at Line 4 is downward-closed and ordered by length, $R_1 = \rho$. Hence, for all $i \in \{1, \ldots, n\}$, $R_1 \cap S'.F'_x = \rho$. Consequently, the condition at Line 9 is not satisfied and therefore $\mathrm{T}_{s_1} = \mathrm{T}_s$. Because the input XML tree T_s satisfies (1) - (4) in Definition 3.11, the base case is established.

Inductive Step: In order to establish the inductive step we assume that XML tree $\mathrm{T}_{s_k} = (\boldsymbol{V}, \boldsymbol{E}, \mathrm{lab}, \mathrm{val})$, where $1 \leq k < m$, conforms to Definition 3.11 and we show that also XML tree $\mathrm{T}_{s_{k+1}}$ conforms to Definition 3.11. If the condition at Line 9 is not satisfied, then $\mathrm{T}_{s_{k+1}} = \mathrm{T}_{s_k}$, which establishes the inductive step since T_{s_k} satisfies (1) - (4) in Definition 3.11 per assumption. We therefore show next that $\mathrm{T}_{s_{k+1}}$ also satisfies (1) - (4) in Definition 3.11 if the condition at Line 9 is satisfied.

(1.i) From the inductive assumption, $\boldsymbol{V}$ is finite in T_{s_k}. Hence, $\boldsymbol{V}$ is also finite in $\mathrm{T}_{s_{k+1}}$ if only a finite number of nodes is added to $\boldsymbol{V}$ in iteration $k+1$ of the loop at Line 8. This is clearly the case since exactly one new node is added to $\boldsymbol{V}$ in iteration $k+1$ (cf. Line 10).

(1.ii) From the inductive assumption, $\forall(v, \bar{v}) \in \boldsymbol{E}$, $v \neq \bar{v}$ in T_{s_k}. From the procedure of the chase, edges that exist in XML tree T_{s_k} are neither altered nor deleted. Hence, (1.ii) in Lemma 6.4 holds true for $\mathrm{T}_{s_{k+1}}$ if $v \neq \bar{v}$ whenever an edge $(v, \bar{v})$ is added to $\boldsymbol{E}$ in iteration $k+1$. From the procedure of the chase, the only edge added to $\boldsymbol{E}$ in iteration $k+1$ is the edge $(\hat{v}, v)$ at Line 13. This edge connects a node $\hat{v}$ which exists in XML tree T_{s_k} to the node v created at Line 11 in iteration $k+1$. Given that v is created in iteration $k+1$, node v does not exist in XML tree T_{s_k}. Hence $\hat{v} \neq v$.

(1.iii) From the inductive assumption, $\forall v \in \boldsymbol{V}$ in XML tree T_{s_k}, if $\boldsymbol{V} \neq \{v\}$ then there exists node $\bar{v} \in \boldsymbol{V}$ such that $(v, \bar{v}) \in \boldsymbol{E}$ or $(\bar{v}, v) \in \boldsymbol{E}$. From the procedure of the chase, nodes and edges that exist in XML tree T_{s_k} are neither altered nor deleted. Hence, (1.iii) in Lemma 6.4 holds true for $\mathrm{T}_{s_{k+1}}$ if for the new node v created at Line 11 in iteration $k+1$, there exists node $\bar{v} \in \boldsymbol{V}$ such that $(v, \bar{v}) \in \boldsymbol{E}$ or $(\bar{v}, v) \in \boldsymbol{E}$. This is obviously the case, according to Line 13.

(1.iv) From the inductive assumption, there are no cycles in T_{σ_k}. Also, from the procedure of the chase edges that exist in XML tree T_{s_k} are neither altered nor deleted. Hence, if $\mathrm{T}_{s_{k+1}}$ contains a cycle, then this cycle must be caused by the new node v created at Line 11. Because a single node does not form a cycle according to Definition 3.2, if $\mathrm{T}_{s_{k+1}}$ contains a cycle, then there must be a walk $\bar{v}_1.\cdots.\bar{v}_n$ in T_{s_k} such that either $v.\bar{v}_1.\cdots.\bar{v}_n$ or $\bar{v}_1.\cdots.\bar{v}_n.v$ is a cycle. Now, if $v.\bar{v}_1.\cdots.\bar{v}_n$ is a cycle then $v = \bar{v}_n$ which clearly contradicts the assumption that $\bar{v}_n$ is a node in T_{s_k} since v is a new node created in iteration $k+1$. If instead $v.\bar{v}_1.\cdots.\bar{v}_n$ is a cycle, edge $(v, \bar{v}_1)$ has been added to $\boldsymbol{E}$ in iteration $k+1$. This however clearly contradicts the procedure of the chase, since the edge that connects the new node v to the existing node $\hat{v}$ does not start from v but only lead to v (cf. Line 13). Hence, also $\mathrm{T}_{s_{k+1}}$ does not contain cycles.

(2) From the procedure of the chase, the assignment of labels to nodes in XML tree T_{s_k} is not altered. Hence, XML tree $\mathrm{T}_{s_{k+1}}$ satisfies (2) in Definition 3.11 if $\mathrm{lab}(v) \in \boldsymbol{L}$. Since $\mathrm{lab}(v) = \mathrm{last}(R_{k+1})$ according to Line 12, $\mathrm{lab}(v) \in \boldsymbol{L}$ from Definition 3.12.

(3) From the inductive assumption, for every node $v \in \boldsymbol{V}$ in T_{s_k}, $\mathrm{val}(v) \in \boldsymbol{U}$ iff v is an attribute or text node. From combining this with the observation that the assignment of values to nodes, which exist in XML tree T_{s_k}, is not modified within the loop at Line 8, we deduce that $\mathrm{T}_{s_{k+1}}$ satisfies (3) in Definition 3.11 if with respect to the new node v created at Line 11, $\mathrm{val}(v) \in \boldsymbol{U}$ iff $\mathrm{lab}(v) \in \boldsymbol{L}^A \cup \{S\}$. If $\mathrm{lab}(v) \in \boldsymbol{L}^E$, $\mathrm{last}(R_{k+1}) \in \boldsymbol{L}^E$ according to (i) in Lemma 7.4. Consequently, neither the condition at Line 14 nor the condition at Line 16 is satisfied, given that σ conforms to Definition 3.21 and does therefore not contain fields that end in element labels. Thus, $\mathrm{val}(v)$ is undefined if $\mathrm{lab}(v) \notin \boldsymbol{L}^A \cup \{S\}$ is an element node. If instead $\mathrm{lab}(v) \in \boldsymbol{L}^A \cup \{S\}$ then $\mathrm{val}(v) \in \boldsymbol{U}$ since even if the condition at Line 14 is not satisfied, then at least the condition at Line 16 is satisfied.

(4.i) From the inductive assumption, for all $(\bar{v}, v) \in \boldsymbol{E}$ in XML tree T_{s_k}, $\mathrm{lab}(\bar{v}) \in \boldsymbol{L}^E$. From the procedure of the chase, edges that exist in T_{s_k} are not altered. Hence, $\mathrm{T}_{s_{k+1}}$ satisfies (4.i) in Lemma 6.4 if $\hat{v}$ is an element node at Line 13 in iteration $k+1$. Since $\mathrm{lab}(\hat{v}) = \mathrm{last}(\mathrm{parent}(R_{k+1}))$ according to Line 10, $\hat{v}$ is an element node given that R_{k+1} conforms to Definition 3.12.

(4.ii) We show that there do not exist attribute nodes $\tilde{v}$ and $\tilde{v}'$ in $\mathrm{T}_{s_{k+1}}$ such that $\mathrm{lab}(\tilde{v}) = \mathrm{lab}(\tilde{v}') \in \boldsymbol{L}^A$ and $\mathrm{parent}(\tilde{v}) = \mathrm{parent}(\tilde{v}')$. We show in particular that if the node v created at Line 11 is an attribute node, then there does not exist attribute node $\tilde{v}$ in T_{s_k} such that $\mathrm{lab}(v) = \mathrm{lab}(\tilde{v})$ and $\mathrm{parent}(v) = \mathrm{parent}(\tilde{v})$.

If $\tilde{v}$ exists, then v and $\tilde{v}$ are reachable over the same path given that $\mathrm{parent}(v) = \mathrm{parent}(\tilde{v})$. Combining this with (i) in Lemma 7.4 we deduce that if $\tilde{v}$ exists then $\tilde{v}$ is a node in T_s. Consequently, from (ii) in Lemma 7.4, $\text{anc-or-self}(\tilde{v}) \cap \text{anc-or-self}(v_i) = \{\mathrm{root}(\mathrm{T}_{s_{k+1}})\}$. From this and, again, (i) in Lemma 7.4 it then follows that $R_{k+1} = \rho.\mathrm{last}(R_{k+1})$, where $\mathrm{last}(R_{k+1}) \in \boldsymbol{L}^A$ given that v is an attribute node. Further, since v is created in iteration $k+1$, R_{k+1} satisfies the condition at Line 9. Thus, σ_s contains an RHS field F'_x such that $S'.F'_x = R_{k+1}$. This, however, clearly contradicts the assumption that σ_s conforms to Definition 7.1. Hence, $\tilde{v}$ does not exist in T_s, which establishes the result.

(ii) From the procedure of the chase, nodes and edges in T_s are not removed nor altered. Also, the values and labels of nodes in XML tree T_s are not modified. Hence, $\mathrm{T}_s \simeq \mathrm{T}_{s+1}$.

(iii) Given that T_s conforms to $\boldsymbol{P}$ also T_{s+1} conforms to $\boldsymbol{P}$ since if a node v_i is added to T_s in an iteration i of the loop at Line 8 then $v_i \in \text{nodes}(R_i, \mathrm{T}_s)$ according to (i) in Lemma 7.4, and $R_i \in \boldsymbol{P}$ per assumption.

Since T_{s+1} conforms to $\boldsymbol{P}$ and T_s is complete w.r.t. $\boldsymbol{P}$ per assumption, also T_{s+1} is complete w.r.t. $\boldsymbol{P}$ if for every new node v_i created in an iteration i of the loop at Line 8, there exists node $\tilde{v}$ in tree T_{s+1} such that $\tilde{v} \in \text{nodes}(\tilde{R}, \mathrm{T}_s)$ and $v \in \text{ancestor}(\tilde{v})$ whenever $\tilde{R} \in \boldsymbol{P}$ such that $R_i \subset \tilde{R}$.

Since v_i is created in iteration i, path R_i satisfies the condition at Line 9. From this and the assumptions that $R_i \subset \tilde{R}$ and $\tilde{R} \in \boldsymbol{P}$ we deduce that also $\tilde{R}$ satisfies the condition at Line 9. Consequently, from (i) in Lemma 7.4, there exists node $\tilde{v} \in \text{nodes}(\tilde{R}, \mathrm{T}_s) \cap \boldsymbol{X}_s$. It therefore remains to show that $v \in \text{ancestor}(\tilde{v})$. Since $\tilde{v} \in \text{nodes}(\tilde{R}, \mathrm{T}_s)$ and $R_i \subset \tilde{R}$ per assumption, it follows that there exists node $\hat{v} \in \text{nodes}(R_i, \mathrm{T}_s)$ such that $\hat{v} \in \text{ancestor}(\tilde{v})$. Now, if $\hat{v} \in \boldsymbol{X}_s$ then both v and $\hat{v}$ are nodes in $\text{nodes}(R_i, \mathrm{T}_s) \cap \boldsymbol{X}_s$ and $v = \hat{v}$ from (i) in Lemma 7.4. Hence, $v \in \text{ancestor}(\tilde{v})$ if $\hat{v} \in \boldsymbol{X}_s$. If instead $\hat{v} \notin \boldsymbol{X}_s$ then $\text{anc-or-self}(\hat{v}) \cap \text{anc-or-self}(\tilde{v}) = \{v_\rho\}$ from (ii) in Lemma 7.4. Consequently, $\hat{v} = v_\rho$ since $\hat{v} \in \text{ancestor}(\tilde{v})$ per assumption and thus $R_i = \rho$ given that $\hat{v} \in \text{nodes}(R_i, \mathrm{T}_s)$. This however contradicts (i) in Lemma 7.3 since R_i satisfies the condition at Line 9 per assumption. Consequently, $\hat{v} \notin \boldsymbol{X}_s$, which finally establishes (iii) in Lemma 7.4. □

We are now ready to establish Lemma 7.1.

Proof (Lemma 7.1) In order to show that $\text{CHASE}(\boldsymbol{P}, \Sigma, \mathrm{T})$ terminates, we show first that there exists a finite set of values $\bar{\boldsymbol{U}} \subset \boldsymbol{U}$, such that if u is a value in an XML tree T_s in a step s of $\text{CHASE}(\boldsymbol{P}, \Sigma, \mathrm{T})$, then $u \in \bar{\boldsymbol{U}}$. We note that u is said to be a value in T_s, if there exists an attribute or text node v in T_s such that $\text{val}(v) = u$. Let $\bar{\boldsymbol{U}}$ be the set given by

$$\bar{\boldsymbol{U}} = \{u \in \boldsymbol{U} \mid \text{ there exists node } v \text{ in T such that } \text{val}(v) = u\} \cup \{0\}.$$

We note that $\bar{\boldsymbol{U}}$ is finite since the number of attribute and text nodes in T is finite given that T conforms to Definition 3.11. We now show by induction over the sequence of XML trees $\mathrm{T}_1, \ldots, \mathrm{T}_k$ generated by the chase, where $\mathrm{T}_1 = \mathrm{T}$, that $u \in \bar{\boldsymbol{U}}$ if u is a value in T_s, $1 \leq s < k$. We use T_1 as the base case for the induction. Clearly, $u \in \bar{\boldsymbol{U}}$ if u is a value in T_1 given that $\mathrm{T}_1 = \mathrm{T}$. Assume now that $u \in \bar{\boldsymbol{U}}$ if u is a value in $\mathrm{T}_s \in \{\mathrm{T}_1, \ldots, \mathrm{T}_k\}$. We establish the inductive step by showing that $u \in \bar{\boldsymbol{U}}$ also if u is a value in T_{s+1}. Since $\mathrm{T}_s \simeq \mathrm{T}_{s+1}$, from (ii) in Lemma 7.5, either u is a value in T_s, and thus $u \in \bar{\boldsymbol{U}}$ from the inductive assumption, or u is the value of a node v created in step s. In the latter case $u = 0$ if u is assigned to v at Line 17, and u is the value of a node in T_s, if u is assigned to v at Line 15. In both cases $u \in \bar{\boldsymbol{U}}$. Hence $u \in \bar{\boldsymbol{U}}$, if u is a value in T_{s+1}.

We show next that the XML tree T_s in a step s of $\text{CHASE}(\boldsymbol{P}, \Sigma, \mathrm{T})$ satisfies any XIND $\sigma = (S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n])$ if for every list of values $u_1, \ldots, u_n \in \bar{\boldsymbol{U}}_1 \times \cdots \times \bar{\boldsymbol{U}}_n$, where $\forall i \in \{1, \ldots, n\}$, $\bar{\boldsymbol{U}}_i = \bar{\boldsymbol{U}}$, there exists a list of nodes $v'_1, \ldots, v'_n$ such that

(a) $\forall i \in \{1,\ldots,n\}$, $v'_i \in \text{nodes}(S'.F'_i, \text{T}_s)$
(b) $\forall i, j \in \{1,\ldots,n\}$, $\text{closest}(v'_i, v'_j) = \text{true}$
(c) $\forall i \in \{1,\ldots,n\}$, $\text{val}(v'_i) = u_i$.

From Lemma 5.7, $\text{T}_s \vDash \sigma$ if whenever there exist nodes $v_1,\ldots,v_n$ which satisfy (i) and (ii) in Lemma 5.7 with respect to σ, then there also exist nodes $v'_1,\ldots,v'_n$ which satisfy (i') and (ii') in Lemma 5.7 with respect to σ, and $\forall i \in \{1,\ldots,n\}$, $\text{val}(v_i) = \text{val}(v'_i)$. Now, if there exist nodes $v_1,\ldots,v_n$ which satisfy (i) and (ii) in Lemma 5.7 with respect to σ, then $\text{val}(v_1),\ldots,\text{val}(v_n) \in \bar{\boldsymbol{U}}_1 \times \cdots \times \bar{\boldsymbol{U}}_n$ since $\forall i \in \{1,\ldots,n\}$, $\text{val}(v_i)$ is a value in T_s according to the result established at the beginning of this proof and so $\text{val}(v_i) \in \bar{\boldsymbol{U}}$. Hence, if it is the case that for every list of values in $\bar{\boldsymbol{U}}_1 \times \cdots \times \bar{\boldsymbol{U}}_n$ there exists a list of nodes which satisfies (a) - (c) with respect to σ, then for the list of values $\text{val}(v_1),\ldots,\text{val}(v_n)$, there exist nodes $v'_1,\ldots,v'_n$ which satisfy (i') and (ii') in Lemma 5.7 with respect to σ because (a) and (b) are identical to (i') and (ii') in Lemma 5.7, respectively. Also, from (c), $\forall i \in \{1,\ldots,n\}$, $\text{val}(v'_i) = \text{val}(v_i)$ and therefore $\text{T}_s \vDash \sigma$.

Next, we say that T_s *misses a list of nodes in order to satisfy* σ, if for a list of values $u_1,\ldots,u_n \in \bar{\boldsymbol{U}}_1 \times \cdots \times \bar{\boldsymbol{U}}_n$, there do not exist nodes $v'_1,\ldots,v'_n$ in tree T_s which satisfy (a) - (c) with respect to σ. Also, from the procedure of the chase, a step $s+1$ is performed iff $\text{T}_s \nvDash \Sigma$, and so step $s+1$ is performed iff tree T_s misses at least one sequence of nodes in order to satisfy Σ. Hence, if the initial tree T_1 misses only a finite number of sequences of nodes in order to satisfy Σ and the number of missing sequences of nodes strictly decreases from a step s to step $s+1$, then only a finite number of steps is performed, and thus $\text{CHASE}(\boldsymbol{P}, \text{T}, \Sigma)$ terminates.

Because $\bar{\boldsymbol{U}}$ is a finite set of values per definition, T_1 misses only a finite number of sequences of nodes in order to satisfy Σ. To be more precise, T_1 misses at most $\sum_{i=1}^{x} |\bar{\boldsymbol{U}}|^{|\sigma_i|}$ sequences of nodes, where

- x is the number of XINDs in Σ
- $|\bar{\boldsymbol{U}}|$ is the number of values in $\bar{\boldsymbol{U}}$
- $|\sigma_i|$ is the number of LHS/RHS fields in σ_i

Hence, if the number of missing lists of nodes strictly decreases from a step s to step $s+1$ then $\text{CHASE}(\boldsymbol{P}, \text{T}, \Sigma)$ terminates after at most $\sum_{i=1}^{x} |\bar{\boldsymbol{U}}|^{|\sigma_i|}$ steps.

We now show that the number of missing lists of nodes indeed strictly decreases from a step s to step $s+1$. Let ω_s and ω_{s+1} be the number of missing sequences of nodes in order to satisfy Σ in trees T_s and T_{s+1}, respectively. Also, let α be the subsumption mapping establishing that $\text{T}_s \sqsubseteq \text{T}_{s+1}$. Then, whenever there exists a list of nodes $v'_1,\ldots,v'_{|\sigma|}$ in tree T_s, that satisfies (a) - (c) with respect to an XIND $\sigma \in \Sigma$ and a list of values $u_1,\ldots,u_{|\sigma|} \in \bar{\boldsymbol{U}}_1 \times \cdots \times \bar{\boldsymbol{U}}_{|\sigma|}$, then the list of nodes $\alpha(v'_1),\ldots,\alpha(v'_{|\sigma|})$ in T_{s+1} satisfies (a) and (b) according to (iv) and (v) in Lemma 7.2, and $\alpha(v'_1),\ldots,\alpha(v'_{|\sigma|})$ also satisfies (c) according to (iv) in Definition 7.2. Hence $\omega_{s+1} \leq \omega_s$.

It therefore remains to show that $\omega_{s+1} < \omega_s$. Given that step $s+1$ is performed, $\text{T}_s \nvDash \Sigma$ and therefore there exists a list of nodes $v_1,\ldots,v_n$ at Line 6 in Algorithm 7.1 which violate $\sigma_s = (S, [F_1,\ldots,F_n]) \subseteq (S', [F'_1,\ldots,F'_n])$ in T_s. So T_s misses a list of nodes $v'_1,\ldots,v'_n$

which satisfies (a) - (c) with respect to σ_s and the list of values $\mathrm{val}(v_1), \ldots, \mathrm{val}(v_n)$. Clearly, if T_{s+1} contains nodes $v'_1, \ldots, v'_n$ then $\omega_{s+1} < \omega_s$.

We show next that T_{s+1} indeed contains nodes $v'_1, \ldots, v'_n$ which satisfy (a) - (c) with respect to σ_s and the list of values $\mathrm{val}(v_1), \ldots, \mathrm{val}(v_n)$. Because Σ conforms to $\boldsymbol{P}$, $\{S'.F'_1, \ldots, S'.F'_n\} \subseteq [R_1, \ldots, R_m]$ and because $[R_1, \ldots, R_m]$ does not contain duplicates, there exists the unique mapping $\phi : \{1, \ldots, n\} \to \{1, \ldots, m\}$ such that for all $i \in \{1, \ldots, m\}$, $S'.F'_i = R_{\phi(i)}$. Also, for all $i \in \{1, \ldots, n\}$, $S'.F'_i \neq \rho$ according to Definition 3.21 and therefore $R_{\phi(i)} = S'.F'_i$ satisfies the condition at Line 9 within iteration $\phi(i)$ of the loop at Line 8 in Algorithm 7.1. Now, let for all $i \in \{1, \ldots, n\}$, v'_i be the node created in iteration $\phi(i)$ of the loop at Line 8. Then, from (i) in Lemma 7.4, $v'_i \in \mathrm{nodes}(R_{\phi(i)}, \mathrm{T}_s)$ when step s is finished, and thus for all $i \in \{1, \ldots, n\}$, $v'_i \in \mathrm{nodes}(S'.F'_i, \mathrm{T}_{s+1})$ given that $S'.F'_i = R_{\phi(i)}$. Hence, nodes $v'_1, \ldots, v'_n$ satisfy (a). Next, given that nodes $v'_1, \ldots, v'_n$ have been created in step s, for all $i, j \in \{1, \ldots, n\}$, $\mathrm{closest}(v'_i, v'_j) = \mathrm{true}$ according to (iii) in Lemma 7.4, and therefore nodes $v'_1, \ldots, v'_n$ also satisfy (b). Finally, since fields $[F'_1, \ldots, F'_n]$ do not contain duplicates, and for all $i \in \{1, \ldots, n\}$, path $R_{\phi(i)}$ meets the condition at Line 14 given that $R_{\phi(i)} = S'.F'_i$, it follows that for all $i \in \{1, \ldots, n\}$, $\mathrm{val}(v_i)$ is assigned to v'_i at Line 15 within iteration $\phi(i)$ of the loop at Line 8. Hence, nodes $v'_1, \ldots, v'_n$ also satisfy (c).

Given that CHASE($\boldsymbol{P}, \mathrm{T}, \Sigma$) terminates, it is now straightforward to establish that the final XML tree $\bar{\mathrm{T}}$ satisfies (i) - (iii) in Lemma 7.1. In particular, $\bar{\mathrm{T}} \vDash \Sigma$ from the procedure of the chase and so $\bar{\mathrm{T}}$ satisfies (iii) in Lemma 7.5. Also, from the preliminary results in (ii) and (iii) in Lemma 7.5, $\bar{\mathrm{T}}$ satisfies (i) and (ii) in Lemma 7.1. □

7.3 Consistency of Core XINDs

We now use the chase algorithm to solve the consistency problem related to core XINDs. Intuitively, a set of XINDs is consistent if there exists at least one XML tree that satisfies the XINDs. We now make this idea more precise within the context of complete XML trees.

Definition 7.3 (Consistency of XINDs) A set of XINDs Σ is *consistent* if for every downward-closed set of paths $\boldsymbol{P}$ that Σ conforms to, there exists an XML tree T that is complete w.r.t. $\boldsymbol{P}$ and also satisfies Σ.

As for the consistency of XKeys discussed in Section 6.1, the consistency of a set of XINDs is independent of a specific set of paths $\boldsymbol{P}$ since Definition 7.3 requires the existence of a complete XML tree for every set of paths that Σ conforms to. We have the following result on the consistency of core XINDs in complete XML trees.

Theorem 7.1 (Consistency of Core XINDs) *Every set of core XINDs is consistent.*

Proof (Theorem 7.1) The correctness of Theorem 7.1 follows from the fact that there always exists an XML tree $\tilde{\mathrm{T}}$ that is complete w.r.t. a given set of paths $\boldsymbol{P}$, and the result in Lemma 7.1 that the XML tree $\bar{\mathrm{T}}$ returned by CHASE($\boldsymbol{P}, \tilde{\mathrm{T}}, \Sigma$), is complete w.r.t. $\boldsymbol{P}$ and satisfies the given set of core XINDs Σ. □

7.4 Implication of Core XINDs

We now turn to the implication of core XINDs, and first make the notion of the implication of a single XIND by a set of XINDs more precise within the context of complete XML trees.

Definition 7.4 (Implication of XINDs) A set of XINDs Σ *implies* a single XIND σ, denoted by $\Sigma \vDash \sigma$, if for every downward-closed set of paths $\boldsymbol{P}$ that $\Sigma \cup \{\sigma\}$ conform to, and every XML tree T that is complete w.r.t. $\boldsymbol{P}$, if T $\vDash \Sigma$ then T $\vDash \sigma$.

We note that also our definition of XIND implication is independent of a specific set of paths that Σ and σ conform to. Based on Definition 7.4, we now formulate the specific implication problem related to core XINDs as the question of whether $\Sigma \vDash \sigma$ is decidable. We answer this question in the following subsections.

In particular, we first present a set of inference rules for the implication of core XINDs in complete XML trees in Subsection 7.4.1. In this subsection, we also establish the soundness and completeness of the inference rules using the chase. Based on our inference rules, we then develop a decision procedure for the implication of core XINDs Subsection 7.4.2.

7.4.1 Inference Rules for the Implication of Core XINDs

Table 7.1 gives a set of inference rules for the implication of XINDs, where symbol $\vdash$ denotes that the XINDs in the conclusion are derived from the XINDs in the premise. We note that the downward-closed set of paths $\boldsymbol{P}$, the XINDs in a rule conform to, is not explicitly stated.

Rules R1 - R3 correspond to the well known inference rules for relational inclusion dependencies [4], which is to be expected given Theorem 4.2 and the fact that XML trees generated from a complete relational database by algorithm DB2XML introduced in Section 4.2 are a subclass of complete XML trees. The remaining rules have no parallels in the inference rules for relational inclusion dependencies, and we now discuss them.

Rule R4 allows one to shift a path from the end of the RHS selector in an XIND down to the start of the RHS fields. For example, by applying R4 to the XIND (`Company.Invoices.Invoice`, [`cno`]) $\subseteq$ (`Company.Customers.Customer`, [`cno`]), we derive the XIND (`Company.Invoices.Invoice`, [`cno`]) $\subseteq$ (`Company.Customers`, [`Customer.cno`])), whereby the last label in the RHS selector `Company.Customers.Customer` has been shifted down to the start of the RHS fields. Rule R5 is the reverse of R4, whereby a path from the start of the RHS fields is shifted up to the end of the RHS selector.

Rule R6 is a rule that, roughly speaking, allows one to union the LHS fields and the RHS fields of two XINDs, provided that the RHS fields intersect only at the root path. For example, assuming that ρ = `Company` and given the XINDs (`Company.Invoices.Invoice`, [`cno`]) $\subseteq$ (`Company`, [`Customers.Customer.cno`]) and (`Company.Invoices.Invoice`, [`Line.code`, `Line.no`]) $\subseteq$ (`Company`, [`Phones.Phone.code`, `Phones.Phone.no`]), then by applying rule R6 we derive (`Company.Invoices.Invoice`, [`cno`, `Line.code`, `Line.no`]) $\subseteq$ (`Company`, [`Customers.Customer.cno`, `Phones.Phone.code`, `Phones.Phone.no`]) since `Company.Customers.Customer.cno` $\cap$ `Company.Phones.Phone.code` = `Company.Customers.Customer.no` $\cap$ `Company.Phones.Phone.no` = ρ. However, the

R1 **Reflexivity**
$\{\} \vdash (S, [F_1, \ldots, F_n]) \subseteq (S, [F_1, \ldots, F_n])$

R2 **Permutated Projection**
$(S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n]) \vdash$
$(S, [F_{\pi(1)}, \ldots, F_{\pi(m)}]) \subseteq (S', [F'_{\pi(1)}, \ldots, F'_{\pi(m)}])$ if $\{\pi(1), \ldots, \pi(m)\} \subseteq \{1, \ldots, n\}$

R3 **Transitivity**
$(S, [F_1, \ldots, F_n]) \subseteq (\bar{S}, [\bar{F}_1, \ldots, \bar{F}_n]) \wedge (\bar{S}, [\bar{F}_1, \ldots, \bar{F}_n]) \subseteq (S', [F'_1, \ldots, F'_n]) \vdash$
$(S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n])$

R4 **Downshift**
$(S, [F_1, \ldots, F_n]) \subseteq (S'.R, [F'_1, \ldots, F'_n]) \vdash (S, [F_1, \ldots, F_n]) \subseteq (S', [R.F'_1, \ldots, R.F'_n])$

R5 **Upshift**
$(S, [F_1, \ldots, F_n]) \subseteq (S', [R.F'_1, \ldots, R.F'_n]) \vdash (S, [F_1, \ldots F_n]) \subseteq (S'.R, [F'_1, \ldots, F'_n])$

R6 **Union**
$(S, [F_1, \ldots, F_m]) \subseteq (\rho, [F'_1, \ldots, F'_m]) \wedge (S, [F_{m+1}, \ldots, F_n]) \subseteq (\rho, [F'_{m+1}, \ldots, F'_n]) \vdash$
$(S, [F_1, \ldots, F_n]) \subseteq (\rho, [F'_1, \ldots, F'_n])$ if $\forall i, j \in \{1, \ldots, m\} \times \{m{+}1, \ldots, n\}$, $\rho.F'_i \cap \rho.F'_j = \rho$

Table 7.1: Inference rules for the implication of core XINDs in complete XML trees.

XINDs $(\texttt{Company.Invoices.Invoice}, [\texttt{cno}]) \subseteq (\texttt{Company}, [\texttt{Customers.Customer.cno}])$ and $(\texttt{Company.Invoices.Invoice}, [\texttt{address}]) \subseteq (\texttt{Company}, [\texttt{Customers.Customer.address}])$ rule R6 dos not derive the XIND $(\texttt{Company.Invoices.Invoice}, [\texttt{cno}, \texttt{address}]) \subseteq (\texttt{Company}, [\texttt{Customers.Customer.cno}, \texttt{Customers.Customer.address}])$ since obviously $\texttt{Company.Customers.Customer.cno} \cap \texttt{Company.Customers.Customer.address} \neq \rho$.

As for the derivation of a single XKey from a set of XKeys discussed in Section 6.2.1, we now denote by $\Sigma \vdash \sigma$ if there is a derivation sequence of a single XIND σ from a set of XINDs Σ using inference rules R1 - R6 in Table 7.1. We have the following result on the soundness of our inference rules.

Theorem 7.2 (Soundness of Inference Rules for Core XINDs) *Given a set of core XINDs Σ and a single core XIND σ, if $\Sigma \vdash \sigma$ then $\Sigma \models \sigma$.*

Proof (Theorem 7.2) Rule R1: We show that given a downward-closed set of paths $\boldsymbol{P}$ and a conforming set of XINDs $\Sigma \cup \{\sigma\}$ where $\sigma = (S, [F_1, \ldots, F_n]) \subseteq (S, [F_1, \ldots, F_n])$, there does not exist an XML tree T which is complete w.r.t. $\boldsymbol{P}$ such that $\mathrm{T} \models \Sigma$ but $\mathrm{T} \not\models \sigma$. We show in particular the strictly stronger result that there does not exist an XML tree T which is complete w.r.t. $\boldsymbol{P}$ and violates σ. For this purpose assume to the contrary that $\mathrm{T} \not\models \sigma$. Then there exist nodes $v_1, \ldots, v_n$ according to Lemma 5.7 such that

(a) $\forall i \in \{1, \ldots, n\}$, $v_i \in \text{nodes}(S.F_i, \mathrm{T})$
(b) $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(v_i, v_j) = \text{true}$

and there do not exist nodes $v'_1, \ldots, v'_n$ such that

(a') $\forall i \in \{1, \ldots, n\}$, $v'_i \in \text{nodes}(S.F_i, \text{T})$
(b') $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(v'_i, v'_j) = \text{true}$
(c') $\forall i \in \{1, \ldots, n\}$, $\text{val}(v'_i) = \text{val}(v_i)$.

If we let for all $i \in \{1, \ldots, n\}$, $v'_i = v_i$, then nodes $v'_1, \ldots, v'_n$ satisfy (a') and (b') because of (a) and (b), respectively, and (c') trivially holds true given that for all $i \in \{1, \ldots, n\}$, $v'_i = v_i$. Hence, the presence of nodes $v_1, \ldots, v_n$ contradicts the absence of nodes $v'_1, \ldots, v'_n$ and therefore $\text{T} \vDash \sigma$.

Rule R2: We show that given a downward-closed set of paths $\boldsymbol{P}$ and a conforming set of XINDs $\Sigma \cup \{\sigma\}$ where

- $(S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n]) \in \Sigma$
- $\sigma = (S, [F_{\pi(1)}, \ldots, F_{\pi(m)}]) \subseteq (S', [F'_{\pi(1)}, \ldots, F'_{\pi(m)}])$
- $\forall i \in \{1, \ldots, m\}$, $\pi(i) \in \{1, \ldots, n\}$,

there does not exist an XML tree T which is complete w.r.t. $\boldsymbol{P}$ such that $\text{T} \vDash \Sigma$ but $\text{T} \nvDash \sigma$. We show in particular that if $\text{T} \nvDash \sigma$ then $\text{T} \nvDash (S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n])$. For the ease of presentation but without loss of generality we assume now that for all $i \in \{1, \ldots, m\}$, $\pi(i) = i$. Then $\sigma = (S, [F_1, \ldots, F_m]) \subseteq (S', [F'_1, \ldots, F'_m])$.

Now, if $m = n$ then $\sigma = (S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n])$ and therefore $\text{T} \nvDash (S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n])$ if $\text{T} \nvDash \sigma$. If instead $m < n$ and given that $\text{T} \nvDash \sigma$, there exist nodes $v_1, \ldots, v_m$ according to Lemma 5.7 such that

(a) $\forall i \in \{1, \ldots, m\}$, $v_i \in \text{nodes}(S.F_i, \text{T})$
(b) $\forall i, j \in \{1, \ldots, m\}$, $\text{closest}(v_i, v_j) = \text{true}$

and there do not exist nodes $v'_1, \ldots, v'_m$ such that

(a') $\forall i \in \{1, \ldots, m\}$, $v'_i \in \text{nodes}(S'.F'_i, \text{T})$
(b') $\forall i, j \in \{1, \ldots, m\}$, $\text{closest}(v'_i, v'_j) = \text{true}$
(c') $\forall i \in \{1, \ldots, m\}$, $\text{val}(v'_i) = \text{val}(v_i)$.

Then, from Lemma 5.2, there exist nodes $v_{m+1}, \ldots, v_n$ such that

(d) $\forall i \in \{m+1, \ldots, n\}$, $v_i \in \text{nodes}(S.F_i, \text{T})$
(e) $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(v_i, v_j) = \text{true}$.

Now, if there do not exist nodes $\bar{v}'_1, \ldots, \bar{v}'_n$ such that

(a") $\forall i \in \{1, \ldots, n\}$, $\bar{v}'_i \in \text{nodes}(S'.F'_i, \text{T})$
(b") $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(\bar{v}'_i, \bar{v}'_j) = \text{true}$
(c") $\forall i \in \{1, \ldots, n\}$, $\text{val}(\bar{v}'_i) = \text{val}(v_i)$

then $\text{T} \nvDash (S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n])$ because of nodes $v_1, \ldots, v_n$ according to Lemma 5.7. It is easily verified that nodes $\bar{v}'_1, \ldots, \bar{v}'_n$ contradict the absence of nodes $v'_1, \ldots, v'_m$ since (a') - (c') holds true because of (a") - (c"), respectively. Hence, $\text{T} \nvDash (S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n])$ if $\text{T} \nvDash \sigma$.

Rule R3: We show that given a downward-closed set of paths $\boldsymbol{P}$ and a conforming set of XINDs $\Sigma \cup \{\sigma\}$ where

- $(S, [F_1, \ldots, F_n]) \subseteq (\bar{S}, [\bar{F}_1, \ldots, \bar{F}_n]) \in \Sigma$
- $(\bar{S}, [\bar{F}_1, \ldots, \bar{F}_n]) \subseteq (S', [F'_1, \ldots, F'_n]) \in \Sigma$
- $\sigma = (S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n])$,

there does not exist an XML tree T which is complete w.r.t. $\boldsymbol{P}$ such that $\mathrm{T} \vDash \Sigma$ but $\mathrm{T} \nvDash \sigma$. We show in particular that if $\mathrm{T} \nvDash \sigma$ then $\mathrm{T} \nvDash \{(S, [F_1, \ldots, F_n]) \subseteq (\bar{S}, [\bar{F}_1, \ldots, \bar{F}_n])\} \cup \{(\bar{S}, [\bar{F}_1, \ldots, \bar{F}_n]) \subseteq (S', [F'_1, \ldots, F'_n])\}$. Given that $\mathrm{T} \nvDash \sigma$ there exist nodes $v_1, \ldots, v_n$ according to Lemma 5.7 such that

(a) $\forall i \in \{1, \ldots, n\}$, $v_i \in \text{nodes}(S.F_i, \mathrm{T})$
(b) $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(v_i, v_j) = \text{true}$

and there do not exist nodes $v'_1, \ldots, v'_n$ such that

(a') $\forall i \in \{1, \ldots, n\}$, $v'_i \in \text{nodes}(S'.F'_i, \mathrm{T})$
(b') $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(v'_i, v'_j) = \text{true}$
(c') $\forall i \in \{1, \ldots, n\}$, $\text{val}(v'_i) = \text{val}(v_i)$.

Now, from Lemma 5.7, $\mathrm{T} \nvDash (S, [F_1, \ldots, F_n]) \subseteq (\bar{S}, [\bar{F}_1, \ldots, \bar{F}_n])$ because of nodes $v_1, \ldots, v_n$ if there do not exist nodes $\bar{v}_1, \ldots, \bar{v}_n$ such that

(e) $\forall i \in \{1, \ldots, n\}$, $\bar{v}_i \in \text{nodes}(\bar{S}.\bar{S}_i, \mathrm{T})$
(f) $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(\bar{v}_i, \bar{v}_j) = \text{true}$
(g) $\forall i \in \{1, \ldots, n\}$, $\text{val}(\bar{v}_i) = \text{val}(v_i)$.

If instead nodes $\bar{v}_1, \ldots, \bar{v}_n$ exist, then $\mathrm{T} \vDash (\bar{S}, [\bar{F}_1, \ldots, \bar{F}_n]) \subseteq (S', [F'_1, \ldots, F'_n])$ according to Lemma 5.7 iff there exist nodes $v'_1, \ldots, v'_n$ which satisfy (a') - (c'). Hence, $\mathrm{T} \nvDash \sigma \Rightarrow \mathrm{T} \nvDash \{(S, [F_1, \ldots, F_n]) \subseteq (\bar{S}, [\bar{F}_1, \ldots, \bar{F}_n])\} \cup \{(\bar{S}, [\bar{F}_1, \ldots, \bar{F}_n]) \subseteq (S', [F'_1, \ldots, F'_n])\}$.

Rule R4: We show that given a downward-closed set of paths $\boldsymbol{P}$ and a conforming set of XINDs $\Sigma \cup \{\sigma\}$ where

- $(S, [F_1, \ldots, F_n]) \subseteq (S'.R, [F'_1, \ldots, F'_n]) \in \Sigma$
- $\sigma = (S, [F_1, \ldots, F_n]) \subseteq (S', [R.F'_1, \ldots, R.F'_n])$,

there does not exist an XML tree T which is complete w.r.t. $\boldsymbol{P}$ such that $\mathrm{T} \vDash \Sigma$ but $\mathrm{T} \nvDash \sigma$. We show in particular that $\mathrm{T} \nvDash \sigma \Rightarrow \mathrm{T} \nvDash (S, [F_1, \ldots, F_n]) \subseteq (S'.R, [F'_1, \ldots, F'_n])$. Given that $\mathrm{T} \nvDash \sigma$, there exist nodes $v_1, \ldots, v_n$ according to Lemma 5.7 such that

(a) $\forall i \in \{1, \ldots, n\}$, $v_i \in \text{nodes}(S.F_i, \mathrm{T})$
(b) $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(v_i, v_j) = \text{true}$,

and there do not exist nodes $v'_1, \ldots, v'_n$ such that

(a') $\forall i \in \{1, \ldots, n\}$, $v'_i \in \text{nodes}(S'.R.F'_i, \mathrm{T})$
(b') $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(v'_i, v'_j) = \text{true}$
(c') $\forall i \in \{1, \ldots, n\}$, $\text{val}(v'_i) = \text{val}(v_i)$.

From Lemma 5.7, $\mathrm{T} \nvDash (S, [F_1, \ldots, F_n] \subseteq (S'.R, [F'_1, \ldots, F'_n])$ because of nodes $v_1, \ldots, v_n$ if there do not exist nodes $\bar{v}_1, \ldots, \bar{v}_n$ such that

(a") $\forall i \in \{1, \ldots, n\}$, $\bar{v}_i \in \text{nodes}(S'.R.F'_i, \mathrm{T})$
(b") $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(\bar{v}_i, \bar{v}_j) = \text{true}$

(c”) $\forall i \in \{1, \ldots, n\}$, $\text{val}(\bar{v}_i) = \text{val}(v_i)$.

Now, if nodes $\bar{v}_1, \ldots, \bar{v}_n$ exist then nodes $\bar{v}_1, \ldots, \bar{v}_n$ satisfy (a’) - (c’) given (a”) - (c”), which however clearly contradicts the assumption that nodes $v'_1, \ldots, v'_n$ do not exist. Hence, $\text{T} \nvDash \sigma \Rightarrow \text{T} \nvDash (S, [F_1, \ldots, F_n] \subseteq (S'.R, [F'_1, \ldots, F'_n])$.

Rule R5: We show that given a downward-closed set of paths $\boldsymbol{P}$ and a conforming set of XINDs $\Sigma \cup \{\sigma\}$ where

- $(S, [F_1, \ldots, F_n]) \subseteq (S', [R.F'_1, \ldots, R.F'_n]) \in \Sigma$
- $\sigma = (S, [F_1, \ldots, F_n]) \subseteq (S'.R, [F'_1, \ldots, F'_n])$,

there does not exist an XML tree T which is complete w.r.t. $\boldsymbol{P}$ such that $\text{T} \vDash \Sigma$ but $\text{T} \nvDash \sigma$. We show in particular that $\text{T} \nvDash \sigma \Rightarrow \text{T} \nvDash (S, [F_1, \ldots, F_n]) \subseteq (S', [R.F'_1, \ldots, R.F'_n])$. Given that $\text{T} \nvDash \sigma$, there exist nodes $v_1, \ldots, v_n$ according to Lemma 5.7 such that

(a) $\forall i \in \{1, \ldots, n\}$, $v_i \in \text{nodes}(S.F_i, \text{T})$
(b) $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(v_i, v_j) = \text{true}$

and there do not exist nodes $v'_1, \ldots, v'_n$ such that

(a’) $\forall i \in \{1, \ldots, n\}$, $v'_i \in \text{nodes}(S'.R.F'_i, \text{T})$
(b’) $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(v'_i, v'_j) = \text{true}$
(c’) $\forall i \in \{1, \ldots, n\}$, $\text{val}(v'_i) = \text{val}(v_i)$.

From Lemma 5.7, $\text{T} \nvDash (S, [F_1, \ldots, F_n] \subseteq (S', [R.F'_1, \ldots, R.F'_n])$ because of nodes $v_1, \ldots, v_n$ if there do not exist nodes $\bar{v}_1, \ldots, \bar{v}_n$ such that

(a”) $\forall i \in \{1, \ldots, n\}$, $\bar{v}_i \in \text{nodes}(S'.R.F'_i, \text{T})$
(b”) $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(\bar{v}_i, \bar{v}_j) = \text{true}$
(c”) $\forall i \in \{1, \ldots, n\}$, $\text{val}(\bar{v}_i) = \text{val}(v_i)$.

Now, if nodes $\bar{v}_1, \ldots, \bar{v}_n$ exist then nodes $\bar{v}_1, \ldots, \bar{v}_n$ satisfy (a’) - (c’) given (a”) - (c”), which however clearly contradicts the assumption that nodes $v'_1, \ldots, v'_n$ do not exist. Hence, $\text{T} \nvDash \sigma \Rightarrow \text{T} \nvDash (S, [F_1, \ldots, F_n]) \subseteq (S', [R.F'_1, \ldots, R.F'_n])$.

Rule R6: We show that given a downward-closed set of paths $\boldsymbol{P}$ and a conforming set of XINDs $\Sigma \cup \{\sigma\}$ such that

- $(S, [F_1, \ldots, F_m]) \subseteq (\rho, [F'_1, \ldots, F'_m]) \in \Sigma$
- $(S, [F_{m+1}, \ldots, F_n]) \subseteq (\rho, [F'_{m+1}, \ldots, F'_n])) \in \Sigma$
- $\forall i, j \in \{1, \ldots, m\} \times \{m{+}1, \ldots, n\}$, $\rho.F'_i \cap \rho.F'_j = \rho$
- $\sigma = (S, [F_1, \ldots, F_n]) \subseteq (\rho, [F'_1, \ldots, F'_n])$,

there does not exist an XML tree T which is complete w.r.t. $\boldsymbol{P}$ such that $\text{T} \vDash \Sigma$ but $\text{T} \nvDash \sigma$. We show in particular that if $\text{T} \nvDash \sigma$ then either $\text{T} \nvDash (S, [F_1, \ldots, F_m]) \subseteq (\rho, [F'_1, \ldots, F'_m])$ or $\text{T} \nvDash (S, [F_{m+1}, \ldots, F_n]) \subseteq (\rho, [F'_{m+1}, \ldots, F'_n])s$. Given that $\text{T} \nvDash \sigma$, there exist nodes $v_1, \ldots, \ldots, v_n$ according to Lemma 5.7 such that

(a) $\forall i \in \{1, \ldots, n\}$, $v_i \in \text{nodes}(S.F_i, \text{T})$
(b) $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(v_i, v_j) = \text{true}$,

and there do not exist nodes $v'_1, \ldots, v'_n$ such that

(a') $\forall i \in \{1, \ldots, n\}$, $v'_i \in \text{nodes}(S'.F'_i, \text{T})$
(b') $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(v'_i, v'_j) = \text{true}$
(c') $\forall i \in \{1, \ldots, n\}$, $\text{val}(v'_i) = \text{val}(v_i)$.

Now, from Lemma 5.7, $\text{T} \nvDash (S, [F_1, \ldots, F_m]) \subseteq (S', [F'_1, \ldots, F'_m])$ because of nodes $v_1, \ldots, v_m$ if there do not exist nodes $\bar{v}'_1, \ldots, \bar{v}'_m$ such that

(a") $\forall i \in \{1, \ldots, m\}$, $\bar{v}'_i \in \text{nodes}(S'.F'_i, \text{T})$
(b") $\forall i, j \in \{1, \ldots, m\}$, $\text{closest}(\bar{v}'_i, \bar{v}'_j) = \text{true}$
(c") $\forall i \in \{1, \ldots, m\}$, $\text{val}(\bar{v}'_i) = \text{val}(v_i)$.

Also, from Lemma 5.7, $\text{T} \nvDash (S, [F_{m+1}, \ldots, F_n]) \subseteq (S', [F'_{m+1}, \ldots, F'_n])$ because of nodes $v_{m+1}, \ldots, v_n$ if there do not exist nodes $\bar{v}'_{m+1}, \ldots, \bar{v}'_n$ such that

(a"') $\forall i \in \{m+1, \ldots, n\}$, $\bar{v}'_i \in \text{nodes}(S'.F'_i, \text{T})$
(b"') $\forall i, j \in \{m+1, \ldots, n\}$, $\text{closest}(\bar{v}'_i, \bar{v}'_j) = \text{true}$
(c"') $\forall i \in \{m+1, \ldots, n\}$, $\text{val}(\bar{v}'_i) = \text{val}(v_i)$.

Now, given that $\text{T} \vDash \Sigma$, nodes $\bar{v}'_1, \ldots, \bar{v}'_n$ exist which satisfy (a') and (c') given (a") and (a"') as well as (c") and (c"'). Further, given (b") and (b"'), nodes $\bar{v}'_1, \ldots, \bar{v}'_n$ also satisfy (b') if $\forall i, j \in \{1, \ldots, m\} \times \{m+1, \ldots, n\}$, $\text{closest}(\bar{v}'_i, \bar{v}'_j) = \text{true}$, which then however clearly contradicts the assumption that nodes $v'_1, \ldots, v'_n$ do not exist. Thus, if $\forall i, j \in \{1, \ldots, m\} \times \{m+1, \ldots, n\}$, $\text{closest}(\bar{v}'_i, \bar{v}'_j) = \text{true}$, which is what we show next, then $\text{T} \nvDash \sigma \Rightarrow \text{T} \nvDash \{(S, [F_1, \ldots, F_m]) \subseteq (S', [F'_1, \ldots, F'_m])\} \cup \{(S, [F_{m+1}, \ldots, F_n]) \subseteq (S', [F'_{m+1}, \ldots, F'_n])\}$.

In particular, $\forall i, j \in \{1, \ldots, m\} \times \{m+1, \ldots, n\}$, root(T) satisfies (i) - (iii) in Definition 3.22 with respect to nodes $\bar{v}'_i$ and $\bar{v}'_j$ because

(i) $\text{root(T)} \in \text{anc-or-self}(\bar{v}'_i)$ given that T conforms to Definition 3.11
(ii) $\text{root(T)} \in \text{anc-or-self}(\bar{v}'_j)$ given that T conforms to Definition 3.11
(iii) $\text{root(T)} \in \text{nodes}(S'.F'_i \cap S'.F'_j, \text{T})$ since $S'.F'_i \cap S'.F'_j = \rho$ by assumption.

Hence, $\forall i, j \in \{1, \ldots, m\} \times \{m+1, \ldots, n\}$, $\text{closest}(\bar{v}'_i, \bar{v}'_j) = \text{true}$. □

We now present our result on the completeness of our inference rules.

Theorem 7.3 (Completeness of Inference Rules for Core XINDs) *Given a set of core XINDs Σ and a single core XIND σ, if $\Sigma \vDash \sigma$ then $\Sigma \vdash \sigma$.*

A roadmap for the proof of Theorem 7.3 is as follows. We use algorithm GIXFC given in Algorithm 7.2 to construct a special initial XML tree T_σ, which essentially has LHS field nodes with distinct values w.r.t. the XIND σ and is empty elsewhere. We then show by induction that the only XINDs satisfied by any intermediate XML tree during the chase are those derivable from Σ using rules R1 - R6. That is, if $\bar{\text{T}}_\sigma \vDash \sigma$, where $\bar{\text{T}}_\sigma$ is the final tree returned by $\text{CHASE}(\boldsymbol{P}, \text{T}_\sigma, \Sigma)$, then $\Sigma \vdash \sigma$ and thus $\Sigma \vDash \sigma \Rightarrow \Sigma \vdash \sigma$, since $\bar{\text{T}}_\sigma \vDash \sigma$ if $\Sigma \vDash \sigma$ from Lemma 7.1.

We now present an immediate result on the properties of the initial XML tree T_σ constructed by algorithm GIXFC.

Algorithm 7.2 GIXFC - Generate Initial XML Tree For XIND Chase.

```
in:   a downward-closed set of paths $\boldsymbol{P} = [R_1, \ldots, R_m]$ ordered by length
      an XIND $\sigma = (S, [F_1, \ldots, F_n]) \subseteq (S', [F'_1, \ldots, F'_n])$ conforming to $\boldsymbol{P}$
out:  an XML tree $\mathrm{T} = (\boldsymbol{V}, \boldsymbol{E}, \mathrm{lab}, \mathrm{val})$ which is complete w.r.t. $\boldsymbol{P}$
1:  let T be a trivial XML tree where $\mathrm{lab}(\mathrm{root}(\mathrm{T})) = \rho(\mathrm{P})$
2:  **for** $i := 2$ **to** $m$ **do**
3:      $\{\hat{v}\} \leftarrow \mathrm{nodes}(\mathrm{parent}(R_i), \mathrm{T})$
4:      $v \leftarrow \mathrm{newnode}(\boldsymbol{V})$
5:      $\mathrm{lab}(v) \leftarrow \mathrm{last}(R_i)$
6:      $\boldsymbol{E} \leftarrow \boldsymbol{E} \cup \{(\hat{v}, v)\}$
7:      **if** there exists path $F_j \in [F_1, \ldots, F_n]$ such that $R_i = S.F_j$ **then**
8:          $\mathrm{val}(v) \leftarrow j$
9:      **else if** $\mathrm{last}(R_i) \in \boldsymbol{L}^A \cup \{\mathtt{S}\}$ **then**
10:         $\mathrm{val}(v) \leftarrow$ "0"
11:     **end if**
12: **end for**
13: **return** T;
```

Lemma 7.6 Let $\boldsymbol{P}$ be a downward-closed set of paths and let $\sigma = (S, [F_1, \ldots, F_n] \subseteq (S', [F'_1, \ldots, F'_n])$ be a core XIND. Also, let $\mathrm{T} = \mathrm{GIXFC}(\boldsymbol{P}, \sigma)$. Then, T is complete with respect to $\boldsymbol{P}$ and contains a list of nodes $v_1, \ldots, v_n$ such that
(i) $\forall i \in \{1, \ldots, n\}$, $v_i \in \mathrm{nodes}(S.F_i, \mathrm{T})$
(ii) $\forall i, j \in \{1, \ldots, n\}$, $\mathrm{closest}(v_i, v_j) = \mathrm{true}$
(iii) $\forall i \in \{1, \ldots, n\}$, $\mathrm{val}(v_i) = i$.

In order to demonstrate completeness of our inference rules, we first establish the following important preliminary result.

Lemma 7.7 Let $\boldsymbol{P}$ be a downward-closed set of paths, and let Σ be a set of core XINDs and σ be a single core XIND. Also, let $\mathrm{T} = (\boldsymbol{V}, \boldsymbol{E}, \mathrm{lab}, \mathrm{val})$ and $\bar{\mathrm{T}} = (\bar{\boldsymbol{V}}, \bar{\boldsymbol{E}}, \overline{\mathrm{lab}}, \overline{\mathrm{val}})$ be the XML trees such that $\mathrm{T} = \mathrm{GIXFC}(\boldsymbol{P}, \sigma)$ and $\bar{\mathrm{T}} = \mathrm{CHASE}(\boldsymbol{P}, \mathrm{T}, \Sigma)$. If $\bar{\mathrm{T}} \vDash \sigma$, then $\Sigma \vdash \sigma$.

Proof (Lemma 7.7) We show that $\Sigma \vdash \sigma$ by establishing the claim that whenever there exist nodes $\bar{v}_1, \ldots, \bar{v}_n$ in XML tree $\bar{\mathrm{T}}$ such that

- $\forall i, j \in \{1, \ldots, n\}$, $\mathrm{closest}(\bar{v}_i, \bar{v}_j) = \mathrm{true}$, and
- $\forall i \in \{1, \ldots, n\}$, $\mathrm{val}(\bar{v}_i) = i$,

then $\Sigma \vdash \bar{\sigma}$ of the form

$$\bar{\sigma} = (S, [F_1, \ldots, F_n]) \subseteq (\bar{S}, [\bar{F}_1, \ldots, \bar{F}_n]), \text{ where}$$

- $\forall i \in \{1, \ldots, n\}$, $\bar{S}.\bar{F}_i$ is the path such that $\bar{v}_i \in \mathrm{nodes}(\bar{S}.\bar{F}_i, \bar{\mathrm{T}})$, and
- $\bar{S} = \bar{S}.\bar{F}_1 \cap \cdots \cap \bar{S}.\bar{F}_n$.

We show first that the claim is sufficient, i.e. that if the claim holds true, then $\Sigma \vdash \sigma$ if $\bar{\mathrm{T}} \vDash \sigma$. From Lemma 7.6, the initial XML tree T contains nodes $v_1, \ldots, v_n$, such that

(a) $\forall i \in \{1, \dots, n\}$, $v_i \in \text{nodes}(S.F_i, \mathrm{T})$
(b) $\forall i, j \in \{1, \dots, n\}$, $\text{closest}(v_i, v_j) = \text{true}$
(c) $\forall i \in \{1, \dots, n\}$, $\text{val}(v_i) = i$.

From Lemma 7.1, $\mathrm{T} \simeq \bar{\mathrm{T}}$ and so there exists subsumption mapping $\alpha : \boldsymbol{V} \to \bar{\boldsymbol{V}}$. Further, from (iv) and (v) in Lemma 7.2 and (iv) in Definition 7.2, nodes $\alpha(v)_1, \dots, \alpha(v)_n$ satisfy (a) - (c) in the final XML tree $\bar{\mathrm{T}}$. Hence, assuming that $\bar{\mathrm{T}} \vDash \sigma$, from Lemma 5.7, there exist nodes $v'_1, \dots, v'_n$ in $\bar{\mathrm{T}}$, such that

(a') $\forall i \in \{1, \dots, n\}$, $v'_i \in \text{nodes}(S'.F'_i, \bar{\mathrm{T}})$
(b') $\forall i, j \in \{1, \dots, n\}$, $\text{closest}(v'_i, v'_j) = \text{true}$
(c') $\forall i \in \{1, \dots, n\}$, $\text{val}(v'_i) = \text{val}(v_i) = i$.

Now, given (b') and (c') and assuming that the claim holds true, Σ derives the XIND

$$(S, [F_1, \dots, F_n]) \subseteq (\tilde{S}, [\tilde{F}_1, \dots, \tilde{F}_n]), \text{ where} \tag{7.1}$$

- $\forall i \in \{1, \dots, n\}$, $\tilde{S}.\tilde{F}_i$ is the path such that $v'_i \in \text{nodes}(\tilde{S}.\tilde{F}_i, \bar{\mathrm{T}})$, and
- $\tilde{S} = \tilde{S}.\tilde{F}_1 \cap \dots \cap \tilde{S}.\tilde{F}_n$.

Because for all $i \in \{1, \dots, n\}$, $v'_i \in \text{nodes}(S'.F'_i, \bar{\mathrm{T}})$ (cf. (a')) and $v'_i \in \text{nodes}(\tilde{S}.\tilde{F}_i, \bar{\mathrm{T}})$ per assumption, $S'.F'_i = \tilde{S}.\tilde{F}_i$. Consequently, $S' \subseteq \tilde{S}$ since $\tilde{S} = \tilde{S}.\tilde{F}_1 \cap \dots \cap \tilde{S}.\tilde{F}_n$ per assumption. If $S' = \tilde{S}$ then for all $i \in \{1, \dots, n\}$, $F'_i = \tilde{F}_i$ and thus σ equals (7.1). Hence, $\Sigma \vdash \sigma$ if $S' = \tilde{S}$. If instead $S' \subset \tilde{S}$ then let R be the path such that $S'.R = \tilde{S}$. Then, applying rule R5 (Downshift) to (7.1) yields

$$(S, [F_1, \dots, F_n]) \subseteq (S', [R.\tilde{F}_1, \dots, R.\tilde{F}_n]) \tag{7.2}$$

Since $S'.R = \tilde{F}$ and for all $i \in \{1, \dots, n\}$, $S'.S'_i = \tilde{S}.\tilde{F}_i$, $S'.F'_i = S'.R.\tilde{F}_i$. Consequently, for all $i \in \{1, \dots, n\}$, $R.\tilde{F}_i = F'_i$ and therefore σ equals (7.2). Hence, $\Sigma \vdash \sigma$ also if $S' \subset \tilde{S}$.

We now demonstrate the claim by induction over the sequence of XML trees generated by the chase. We assume now for this purpose, that CHASE($\boldsymbol{P}, \mathrm{T}, \Sigma$) terminates after f steps. Then, the sequence of input trees for the steps in CHASE($\boldsymbol{P}, \mathrm{T}, \Sigma$) is given by $\mathrm{T}_1, \dots, \mathrm{T}_f$, where $\mathrm{T}_1 = \mathrm{T}$ and $\mathrm{T}_f = \bar{\mathrm{T}}$.

Base Case: XML tree T_1 is the base case in our induction. We show first that if the claim applies to a set of nodes $\bar{v}_1, \dots, \bar{v}_n$ in XML tree T_1 then for all $i \in \{1, \dots, n\}$, $\bar{v}_i \in \text{nodes}(S.F_i, \mathrm{T}_1)$. Given that the claim applies to nodes $\bar{v}_1, \dots, \bar{v}_n$, for all $i \in \{1, \dots, n\}$, $\text{val}(\bar{v}_i) = i$ and thus $\text{val}(\bar{v}_i) \neq 0$. Further, since $\mathrm{T}_1 = \mathrm{T} = \text{GIXFC}(\boldsymbol{P}, \sigma)$ per assumption, for all $i \in \{1, \dots, n\}$, $\text{val}(\bar{v}_i)$ has been set at Line 8 in Algorithm 7.2. Also, because the list of RHS fields $[F_1, \dots, F_n]$ in σ does not contain duplicates per definition, and all $i \in \{1, \dots, n\}$, $\text{val}(\bar{v}_i) = i$ per assumption, the procedure of algorithm GIXFC implies that for all $i \in \{1, \dots, n\}$, F_i is the particular RHS field in $[F_1, \dots, F_n]$ which satisfies the condition at Line 7 in Algorithm 7.2 and so $\bar{v}_i \in \text{nodes}(S.F_i, \mathrm{T}_1)$.

Next, by applying rule R1 (Reflexivity) we derive

$$(S, [F_1, \dots, F_n]) \subseteq (S, [F_1, \dots, F_n]) \tag{7.3}$$

Because for all $i \in \{1, \ldots, n\}$, $\bar{v}_i \in \text{nodes}(S.F_i, \mathrm{T}_1)$ and $\bar{v}_i \in \text{nodes}(\bar{S}.\bar{F}_i, \mathrm{T}_1)$ per assumption, $S.F_i = \bar{S}.\bar{F}_i$. Combining this with the assumption that $\bar{S} = \bar{S}.\bar{F}_1 \cap \cdots \cap \bar{S}.\bar{F}_n$ we deduce that $\bar{S} \subseteq S$. Now, if $\bar{S} = S$ then $\bar{\sigma}$ equals (7.3) and so $\Sigma \vdash \bar{\sigma}$. If instead $S \subset \bar{S}$ then let R be the path such that $S.R = \bar{S}$. Then, because of the previous observation that $\forall i \in \{1, \ldots, n\}$, $\bar{S}.\bar{F}_i = S.F_i$, $S.R.\bar{F}_i = S.F_i$ and therefore $R.\bar{F}_i = F_i$. Hence, by applying rule R4 (Upshift) to XIND (7.3) Σ derives the XIND

$$(S, [F_1, \ldots, F_n]) \subseteq (S.R, [\bar{F}_1, \ldots, \bar{F}_n]) \tag{7.4}$$

Since $S.R = \bar{S}$ per assumption, $\bar{\sigma}$ equals (7.4) and therefore $\Sigma \vdash \bar{\sigma}$ also in case that $S \subset \bar{S}$, which establishes the base case.

Inductive Step: We now assume that the claim holds true until step s in CHASE($\boldsymbol{P}$, T, Σ), and show that the claim also holds true for step $s + 1$, which establishes the inductive step. From the procedure of the chase, if $\bar{v}_1, \ldots, \bar{v}_n$ is a list of nodes in XML tree T_{s+1} then for all $i \in \{1, \ldots, n\}$, either $\bar{v}_i$ is a node in XML tree T_s or $\bar{v}_i$ is created in step s and so does not exist in XML tree T_s. We distinguish the following cases:

(A) $\forall i \in \{1, \ldots, n\}$, $\bar{v}_i$ is a node in T_s
(B) $\forall i \in \{1, \ldots, n\}$, $\bar{v}_i$ is created in step s
(C) there exists mapping $\mu : \{1, \ldots, n\} \to \{1, \ldots, n\}$ and integer m, where $1 \leq m < n$, such that for all $i \in \{1, \ldots, n\}$
- if $i \leq m$, node $\bar{v}_{\mu(i)}$ is created in step s
- if $i > m$, node $\bar{v}_{\mu(i)}$ is a node in T_s.

(A) $\Sigma \vdash \bar{\sigma}$ from the inductive assumption given that nodes $\bar{v}_1, \ldots, \bar{v}_n$ exist in tree T_s.

(B) Let σ_s be the XIND at Line 5 in Algorithm 7.1 chosen in step s of CHASE($\boldsymbol{P}$, T, Σ), and let σ_s be of the form

$$\sigma_s = (\tilde{S}, [\tilde{F}_1, \ldots, \tilde{F}_k]) \subseteq (\tilde{S}', [\tilde{F}'_1, \ldots, \tilde{F}'_k]).$$

Also, let $\phi : \{1, \ldots, n\} \to \{1, \ldots, m\}$, where m is the number of paths in $\boldsymbol{P}$, be the mapping such that for all $i \in \{1, \ldots, n\}$, $\bar{v}_i$ is created in iteration $\phi(i)$ of the loop at Line 8 in step s. Then, from the procedure of the chase, for all $i \in \{1, \ldots, n\}$, val($\bar{v}_i$) has been set at Line 15 in iteration $\phi(i)$ of the loop at Line 8 since val($\bar{v}_i$) $= i$ per assumption and thus val($\bar{v}_i$) $\neq 0$. Consequently, for all $i \in \{1, \ldots, n\}$, the condition at Line 14 is met in iteration $\phi(i)$ of the loop at Line 8, and therefore $\bar{R}_{\phi(i)} \in [\tilde{S}'.\tilde{F}'_1, \ldots, \tilde{S}'.\tilde{F}'_k]$, where $\bar{R}_{\phi(i)} \in \boldsymbol{P}$ is the path in iteration $\phi(i)$ of the loop at Line 8.

Since $\{\bar{R}_{\phi(1)}, \ldots, \bar{R}_{\phi(n)}\} \subseteq \{\tilde{S}'.\tilde{F}'_1, \ldots, \tilde{S}'.\tilde{F}'_k\}$, there exists mapping $\pi : \{1, \ldots, n\} \to \{1, \ldots, k\}$ such that for all $i \in \{1, \ldots, n\}$, $\bar{R}_{\phi(i)} = \tilde{S}'.\tilde{F}'_{\pi(i)}$. By applying rule R2 (Permutated Projection) to σ_s, Σ derives the XIND

$$(\tilde{S}, [\tilde{F}_{\pi(1)}, \ldots, \tilde{F}_{\pi(n)}]) \subseteq (\tilde{S}', [\tilde{F}'_{\pi(1)}, \ldots, \tilde{F}'_{\pi(n)}]) \tag{7.5}$$

Further, for all $i \in \{1, \ldots, n\}$, $\bar{R}_{\phi(i)} = \bar{S}.\bar{F}_i$ since $\bar{v}_i \in \text{nodes}(\bar{R}_{\phi(i)}, \mathrm{T}_{s+1})$ according to (i) in Lemma 7.4 and $\bar{v}_i \in \text{nodes}(\bar{S}.\bar{F}_i, \mathrm{T}_{s+1})$ per assumption. Combining this with the previous observation that $\forall i \in \{1,\ldots,n\}$, $\bar{R}_{\phi(i)} = \tilde{S}'.\tilde{F}'_{\pi(i)}$ we deduce that $\bar{S}.\bar{F}_i = \tilde{S}'.\tilde{F}'_{\pi(i)}$. Hence, $\bar{S}.\bar{F}_1 \cap \cdots \cap \bar{S}.\bar{F}_n = \tilde{S}'.\tilde{F}'_{\pi(1)} \cap \cdots \cap \tilde{S}'.\tilde{F}'_{\pi(n)}$ and thus also $\tilde{S}' \subseteq \bar{S}$ since $\tilde{S}' \subseteq \tilde{S}'.\tilde{F}'_{\pi(1)} \cap \cdots \cap \tilde{S}'.\tilde{F}'_{\pi(1)}$ trivially holds true and $\bar{S} = \bar{S}.\bar{F}_1 \cap \cdots \cap \bar{S}.\bar{F}_n$ per assumption.

Suppose now that $\tilde{S}' \subset \bar{S}$ and let $\tilde{R}$ be the path such that $\tilde{S}'.\tilde{R} = \bar{S}$. Then, for all $i \in \{1,\ldots,n\}$, $\tilde{S}'.\tilde{R}.\bar{F}_i = \tilde{S}'.\tilde{F}'_{\pi(i)}$ since $\bar{S}.\bar{F}_i = \tilde{S}'.\tilde{F}'_{\pi(i)}$. Consequently, $\tilde{R}.\bar{F}_i = \tilde{F}'_{\pi(i)}$ and hence, by applying rule R4 (Upshift) to (7.5), Σ derives the XIND

$$(\tilde{S}, [\tilde{F}_{\pi(1)},\ldots,\tilde{F}_{\pi(n)}]) \subseteq (\tilde{S}'.\tilde{R}, [\bar{F}_1,\ldots,\bar{F}_n]) \tag{7.6}$$

Further, since $\tilde{S}'.\tilde{R} = \bar{S}$ per assumption, XIND (7.8) is of the form

$$(\tilde{S}, [\tilde{F}_{\pi(1)},\ldots,\tilde{F}_{\pi(n)}]) \subseteq (\bar{S}, [\bar{F}_1,\ldots,\bar{F}_n]) \tag{7.7}$$

We now show that Σ derives XIND (7.7) also in case that $\tilde{S}' \not\subset \bar{S}$. If $\tilde{S}' \not\subset \bar{S}$ then $\tilde{S}' = \bar{S}$ because of our previous observation that $\tilde{S}' \subseteq \bar{S}$. Combining this with the result that for all $i \in \{1,\ldots,n\}$, $\bar{S}.\bar{F}_i = \tilde{S}'.\tilde{F}'_{\pi(i)}$ we deduce that XIND (7.5) equals XIND (7.7) if $\tilde{S}' = \bar{S}$ and thus Σ derives the XIND (7.7) also in case that $\tilde{S}' \not\subset \bar{S}$.

Now, let $\tilde{v}_1,\ldots,\tilde{v}_k$ be the list of nodes at Line 6 in step s of the chase, i.e. let $\tilde{v}_1,\ldots,\tilde{v}_k$ be the list of nodes which violate σ_s in T_s. Then, from Lemma 5.7,

- $\forall i \in \{1,\ldots,k\}$, $\tilde{v}_i \in \text{nodes}(\tilde{S}.\tilde{F}_i, \mathrm{T}_s)$
- $\forall i,j \in \{1,\ldots,k\}$, $\text{closest}(\tilde{v}_i,\tilde{v}_j) = \text{true}$.

We note that $n < k$ and show next that the claim applies to nodes $\tilde{v}_{\pi(1)},\ldots,\tilde{v}_{\pi(n)}$. Since $\forall i,j \in \{1,\ldots,k\}$, $\text{closest}(\tilde{v}_i,\tilde{v}_j) = \text{true}$, the claim applies to nodes $\tilde{v}_{\pi(1)},\ldots,\tilde{v}_{\pi(n)}$ if $\forall i \in \{1,\ldots,n\}$, $\text{val}(\tilde{v}_{\pi(i)}) = i$. Recall that $\forall i \in \{1,\ldots,n\}$, $\bar{v}_i$ is created in step s per assumption and that $\bar{v}_i \in \text{nodes}(\bar{R}_{\phi(i)}, \mathrm{T}_s)$ where $\bar{R}_{\phi(i)} = \tilde{S}'.\tilde{F}'_{\pi(i)}$. From this and the assumption that the chase removes the violation of σ_s caused by nodes $\tilde{v}_1,\ldots,\tilde{v}_k$ we deduce that $\forall i \in \{1,\ldots,n\}$, $\text{val}(\tilde{v}_{\pi(i)}) = \text{val}(\bar{v}_i)$ at Line 19 in Algorithm 7.1 and therefore $\text{val}(\tilde{v}_{\pi(i)}) = i$ since $\text{val}(\bar{v}_i) = i$ per assumption. Hence, the claim applies to nodes $\tilde{v}_{\pi(1)},\ldots,\tilde{v}_{\pi(n)}$ and thus, from the inductive assumption, Σ derives the XIND

$$(S, [F_1,\ldots,F_n]) \subseteq (\hat{S}, [\hat{F}_{\pi(1)},\ldots,\hat{F}_{\pi(n)}]),\ \text{where} \tag{7.8}$$

- $\forall i \in \{1,\ldots,n\}$, $\hat{S}.\hat{F}_{\pi(i)}$ is the path such that $\tilde{v}_{\pi(i)} \in \text{nodes}(\hat{S}.\hat{F}_{\pi(i)}, \bar{\mathrm{T}}_s)$, and
- $\hat{S} = \hat{S}.\hat{F}_{\pi(1)} \cap \cdots \cap \hat{S}.\hat{F}_{\pi(n)}$.

From the definition of π and the assumption that $\forall i \in \{1,\ldots,k\}$, $\tilde{v}_i \in \text{nodes}(\tilde{S}.\tilde{F}_i, \bar{\mathrm{T}}_s)$ we deduce that $\forall i \in \{1,\ldots,k\}$, $\tilde{v}_{\pi(i)} \in \text{nodes}(\tilde{S}.\tilde{F}_{\pi(i)}, \bar{\mathrm{T}}_s)$. Combining this with the fact that $n < k$ and the assumption that $\forall i \in \{1,\ldots,n\}$, $\tilde{v}_{\pi(i)} \in \text{nodes}(\hat{S}.\hat{F}_{\pi(i)}, \bar{\mathrm{T}}_s)$ we deduce that $\forall i \in \{1,\ldots,n\}$, $\hat{S}.\hat{F}_{\pi(i)} = \tilde{S}.\tilde{F}_{\pi(i)}$. Consequently, $\hat{S}.\hat{F}_{\pi(1)} \cap \cdots \cap \hat{S}.\hat{F}_{\pi(n)} = \tilde{S}.\tilde{F}_{\pi(1)} \cap \cdots \cap \tilde{S}.\tilde{F}_{\pi(n)}$. From this and the assumption that $\hat{S} = \hat{S}.\hat{F}_{\pi(1)} \cap \cdots \cap \hat{S}.\hat{F}_{\pi(n)}$ it follows that $\hat{S} = \tilde{S}.\tilde{F}_{\pi(1)} \cap \cdots \cap \tilde{S}.\tilde{F}_{\pi(n)}$ and therefore $\tilde{S} \subseteq \hat{S}$.

Now, if $\tilde{S} = \hat{S}$ then rule R3 (Transitivity) applies to the XINDs (7.8) and (7.7) since for all $i \in \{1,\ldots,n\}$, $\hat{S}.\hat{F}_{\pi(i)} = \tilde{S}.\tilde{F}_{\pi(i)}$. We observe that the resulting XIND equals $\bar{\sigma}$. Hence,

$\Sigma \vdash \bar{\sigma}$ if $\tilde{S} = \hat{S}$. It therefore remains to show that $\Sigma \vdash \bar{\sigma}$ also in case that $\tilde{S} \subset \hat{S}$. Given that $\tilde{S} \subset \hat{S}$, there exists path $\hat{R}$ such that $\tilde{S}.\hat{R} = \hat{S}$ and by applying rule R5 (Downshift) Σ derives the XIND

$$(S, [F_1, \ldots, F_n]) \subseteq (\tilde{S}, [\hat{R}.\hat{F}_{\pi(1)}, \ldots, \hat{R}.\hat{F}_{\pi(n)}]) \tag{7.9}$$

Since $\tilde{S}.\hat{R} = \hat{S}$ and $\forall i \in \{1, \ldots, n\}$, $\hat{S}.\hat{F}_{\pi(i)} = \tilde{S}.\tilde{F}_{\pi(i)}$, it follows that $\forall i \in \{1, \ldots, n\}$, $\tilde{S}.\hat{R}.\hat{F}_{\pi(i)} = \tilde{S}.\tilde{F}_{\pi(i)}$ and therefore $\hat{R}.\hat{F}_{\pi(i)} = \tilde{F}_{\pi(i)}$. Hence, rule R3 (Transitivity) applies to XINDs (7.9) and (7.7) which yields $\bar{\sigma}$. Thus $\Sigma \vdash \bar{\sigma}$ also in case that $\tilde{S} \subset \hat{S}$.

(C) We assume for the ease of presentation but without loss of generality that μ is the identity function and so that for all $i \in \{1, \ldots, n\}$, $\mu(i) = i$. Then, nodes $\bar{v}_1, \ldots, \bar{v}_m$ are created in step s of the chase and nodes $\bar{v}_{m+1}, \ldots, \bar{v}_n$ already exist in tree T_s.

In order to establish (C) we show first that $\forall i, j \in \{1, \ldots, m\} \times \{m+1, \ldots, n\}$, $\bar{S}.\bar{F}_i \cap \bar{S}.\bar{F}_j = \rho$. Since per assumption $\forall i \in \{1, \ldots, m\}$, node $\bar{v}_i$ is created in step s and $\forall j \in \{m+1, \ldots, n\}$, node $\bar{v}_j$ is a node in T_s, $\forall i, j \in \{1, \ldots, m\} \times \{m+1, \ldots, n\}$, $\{\mathrm{root}(\mathrm{T}_{s+1})\} = \text{anc-or-self}(\bar{v}_i) \cap \text{anc-or-self}(\bar{v}_j)$ from (ii) in Lemma 7.4. Further, since the claim applies to nodes $\bar{v}_1, \ldots, \bar{v}_n$ per assumption, $\forall i, j \in \{1, \ldots, n\}$, $\text{closest}(\bar{v}_i, \bar{v}_j) = \text{true}$. Hence, $\forall i, j \in \{1, \ldots, m\} \times \{m+1, \ldots, n\}$, there exists node $\bar{v}^i_j$ such that

- $\bar{v}^i_j \in \text{anc-or-self}(\bar{v}_i)$
- $\bar{v}^i_j \in \text{anc-or-self}(\bar{v}_j)$
- $\bar{v}^i_j \in \text{nodes}(\bar{S}.\bar{F}_i \cap \bar{S}.\bar{F}_j, \mathrm{T}_{s+1})$.

Combining this with the previous observation that $\forall i, j \in \{1, \ldots, m\} \times \{m+1, \ldots, n\}$, $\{\mathrm{root}(\mathrm{T}_{s+1})\} = \text{anc-or-self}(\bar{v}_i) \cap \text{anc-or-self}(\bar{v}_j)$ we deduce that $\bar{v}^i_j = \mathrm{root}(\mathrm{T}_{s+1})$. Consequently, $\forall i, j \in \{1, \ldots, m\} \times \{m+1, \ldots, n\}$, $\bar{S}.\bar{F}_i \cap \bar{S}.\bar{F}_j = \rho$ since $\bar{v}^i_j \in \text{nodes}(\bar{S}.\bar{F}_i \cap \bar{S}.\bar{F}_j, \mathrm{T}_{s+1})$ per assumption.

We show next that $\bar{S} = \rho$. Since $\forall i, j \in \{1, \ldots, m\} \times \{m+1, \ldots, n\}$, $\bar{S}.\bar{F}_i \cap \bar{S}.\bar{F}_j = \rho$, also $\bar{S}.\bar{F}_1 \cap \cdots \cap \bar{S}.\bar{F}_n = \rho$. Combining this with the fact that $\bar{S} \subseteq \bar{S}.\bar{F}_1 \cap \cdots \cap \bar{S}.\bar{F}_n = \rho$ we deduce that $\bar{S} = \rho$.

Further, since the claim applies to nodes $\bar{v}_1, \ldots, \bar{v}_n$ per assumption and given that nodes $\bar{v}_1, \ldots, \bar{v}_m$ are created in step s, from our result in (B), Σ derives the XIND

$$(S, [F_1, \ldots, F_m]) \subseteq (\tilde{S}, [\tilde{F}_1, \ldots, \tilde{F}_m]), \text{ where} \tag{7.10}$$

- $\forall i \in \{1, \ldots, m\}$, $\tilde{S}.\tilde{F}_i$ is the path such that $\bar{v}_i \in \text{nodes}(\tilde{S}.\tilde{F}_i, \bar{\mathrm{T}}_{s+1})$, and
- $\tilde{S} = \tilde{S}.\tilde{F}_1 \cap \cdots \cap \tilde{S}.\tilde{F}_m$.

Also, since the claim applies to nodes $\bar{v}_1, \ldots, \bar{v}_n$ per assumption and given that nodes $\bar{v}_{m+1}, \ldots, \bar{v}_n$ exist in XML tree T_s, from the inductive assumption, Σ derives the XIND

$$(S, [F_{m+1}, \ldots, F_n]) \subseteq (\hat{S}, [\hat{F}_{m+1}, \ldots, \hat{F}_n]), \text{ where} \tag{7.11}$$

- $\forall i \in \{m+1, \ldots, n\}$, $\hat{S}.\hat{F}_i$ is the path such that $\bar{v}_i \in \text{nodes}(\hat{S}.\hat{F}_i, \bar{\mathrm{T}}_{s+1})$, and
- $\hat{S} = \hat{S}.\hat{F}_{m+1} \cap \cdots \cap \hat{S}.\hat{F}_n$.

Now, if we let $\tilde{R}$ and $\hat{R}$ be the paths such that $\tilde{S} = \rho.\tilde{R}$ and $\hat{S} = \rho.\hat{R}$ then by applying rule R4 (Upshift) to XINDs (7.10) and (7.11) Σ derives the XINDs

$$(S, [F_1, \ldots, F_m]) \subseteq (\rho, [\tilde{R}.\tilde{F}_1, \ldots, \tilde{R}.\tilde{F}_m]) \tag{7.12}$$

$$(S, [F_{m+1}, \ldots, F_n]) \subseteq (\rho, [\hat{R}.\hat{F}_{m+1}, \ldots, \hat{R}.\hat{F}_n]) \tag{7.13}$$

We note that $\forall i \in \{1, \ldots, m\}$, $\tilde{S}.\tilde{F}_i = \bar{S}.\bar{F}_i$ since per assumption $\bar{v}_i \in \text{nodes}(\tilde{S}.\tilde{F}_i, \bar{\mathrm{T}}_{s+1})$ and also $\bar{v}_i \in \text{nodes}(\bar{S}.\bar{F}_i, \bar{\mathrm{T}}_{s+1})$. Likewise, $\forall i \in \{m+1, \ldots, n\}$, $\hat{S}.\hat{F}_i = \bar{S}.\bar{F}_i$, since per assumption $\bar{v}_i \in \text{nodes}(\hat{S}.\hat{F}_i, \bar{\mathrm{T}}_{s+1})$ and again $\bar{v}_i \in \text{nodes}(\bar{S}.\bar{F}_i, \bar{\mathrm{T}}_{s+1})$. Combining this with the previous observation that $\bar{S} = \rho$, we deduce that $\forall i \in \{1, \ldots, m\}$, $\tilde{R}.\tilde{F}_i = \bar{F}_i$ as well as that for all $i \in \{m+1, \ldots, n\}$, $\hat{R}.\hat{F}_i = \bar{F}_i$. Therefore, if rule R6 (Merge) applies to XINDs (7.12) and (7.13) then the resulting XIND equals $\bar{\sigma}$, and hence $\Sigma \vdash \bar{\sigma}$.

We finish the proof by establishing that R6 applies to XINDs (7.12) and (7.13). In particular, rule R6 applies to XINDs (7.12) and (7.13) if $\forall i, j \in \{1, \ldots, m\} \times \{m+1, \ldots, n\}$, $\rho.\tilde{R}.\tilde{F}_i \cap \rho.\hat{R}.\hat{F}_j = \rho$. This follows immediately from our previous result that $\forall i, j \in \{1, \ldots, m\} \times \{m+1, \ldots, n\}$, $\bar{S}.\bar{F}_i \cap \bar{S}.\bar{F}_j = \rho$ since $\bar{S}.\bar{F}_i = \rho.\tilde{R}.\tilde{F}_i$ and $\bar{S}.\bar{F}_j = \rho.\hat{R}.\hat{F}_j$ per assumption. □

The proof of Theorem 7.3 is now straightforward.

Proof (Theorem 7.3) We show that given a set of XINDs Σ and a single XIND σ that conform to a downward-closed set of paths $\boldsymbol{P}$, if $\Sigma \vDash \sigma$ then $\Sigma \vdash \sigma$. For this purpose, let T_σ and $\bar{\mathrm{T}}_\sigma$ be the XML trees such that $\mathrm{T}_\sigma = \text{GIXFC}(\boldsymbol{P}, \sigma)$ and $\bar{\mathrm{T}}_\sigma = \text{CHASE}(\boldsymbol{P}, \mathrm{T}_\sigma, \Sigma)$. Then, $\bar{\mathrm{T}} \vDash \Sigma$ from Lemma 7.1. Hence, if $\bar{\mathrm{T}} \nvDash \sigma$, then this contradicts the assumption that $\Sigma \vDash \sigma$. If instead $\bar{\mathrm{T}} \vDash \sigma$ then $\Sigma \vdash \sigma$ from Lemma 7.7. □

7.4.2 A Decision Procedure for XIND Implication

We finally introduce algorithm DXII, which is a decision procedure for the implication of core XINDs in complete XML trees. The procedure of the algorithm, which is given below, is straightforward and bases on the results established in the previous discussion of the implication of core XINDs.

Algorithm 7.3 DXII - Decide XIND Implication.

```
in:   a set of core XINDs $\Sigma$
      a single core XIND $\sigma$
      a downward-closed set of paths $\boldsymbol{P}$ that $\Sigma \cup \{\sigma\}$ conforms to
out:  true if $\Sigma \vDash \sigma$ and false if $\Sigma \nvDash \sigma$
1: $\mathrm{T}_\sigma \leftarrow \text{GIXFC}(\boldsymbol{P}, \sigma)$
2: $\bar{\mathrm{T}}_\sigma \leftarrow \text{CHASE}(\boldsymbol{P}, \mathrm{T}_\sigma, \Sigma)$
3: return $(\bar{\mathrm{T}}_\sigma \vDash \sigma)$
```

We note that algorithm DXII terminates since both algorithm GIXFC and algorithm CHASE terminate according to our results in Lemma 7.6 and Lemma 7.1, respectively. We then have the following result on the soundness and completeness of algorithm DXII, which also shows that the implication problem for core XINDs in complete XML trees is decidable.

Theorem 7.4 (Soundness and Completeness of Algorithm DXII) *Given a set of core XINDs Σ, a single core XIND σ and a downward-closed set of paths $\boldsymbol{P}$ that $\Sigma \cup \{\sigma\}$ conforms to, then* $\mathrm{DXII}(\Sigma, \sigma, \boldsymbol{P}) = \mathrm{true}$ *iff* $\Sigma \models \sigma$.

Proof (Theorem 7.4) We show first that $\mathrm{DXII}(\Sigma, \sigma, \boldsymbol{P}) = \mathrm{true} \Rightarrow \Sigma \models \sigma$. Then, $\bar{\mathrm{T}}_\sigma \models \sigma$ from Line 3 in Algorithm 7.3. Consequently, $\bar{\mathrm{T}}_\sigma \models \sigma \Rightarrow \Sigma \vdash \sigma$ from Lemma 7.7 and finally $\Sigma \models \sigma$ from Theorem 7.2. Next, $\mathrm{DXII}(\Sigma, \sigma, \boldsymbol{P}) = \mathrm{false} \Rightarrow \Sigma \not\models \sigma$, because then $\bar{\mathrm{T}}_\sigma \not\models \sigma$ from Line 3 in Algorithm 7.3, and thus $\Sigma \not\models \sigma$ since $\bar{\mathrm{T}}_\sigma \models \Sigma$ from Lemma 7.1. $\square$

Chapter 8

Conclusion

Keys and foreign keys for XML are required in order to ensure that XML data is an accurate representation of reality and to preserve original data semantics when XML data is generated from relational data, which frequently precedes the exchange of XML data over the internet. As for keys and foreign keys in any other data model, results on the consistency and implication problems related to XML keys and foreign keys are required so as to lay the foundation for database systems to utilize them.

Existing approaches to value-based XML integrity constraints, the class of XML integrity constraints where XML keys and foreign keys fall into as discussed in Chapter 2, are of limited use for two reasons. First, given that XML integrity constraints are intended to ensure that data is an accurate representation of reality, checking the satisfaction of an XML integrity constraint in an XML document must not yield counter-intuitive results. This is however not always achieved by existing approaches to value-based XML integrity constraints. The actual problem stems from the hierarchical and semi-structured nature of XML data which allows entity nodes[1] to have multiple property nodes[2] for the same entity property, and which also allows entity nodes to have no property node for some entity property at all. In general, checking the satisfaction of a value-based XML integrity constraint in an XML document means to compare entity nodes on the basis of values of specified combinations of property nodes. So, if multiple property nodes occur in checking the satisfaction of a value-based XML integrity constraint, it is necessary to disregard semantically incorrect combinations of property nodes in comparing entity nodes. Otherwise the constraint is violated even though it should be satisfied. If instead absent property nodes occur in checking the satisfaction of a value-based XML integrity constraint, it is necessary to also take into account incomplete combinations of property nodes in comparing entity nodes since otherwise the constraint is satisfied even though it should be violated.

Second, it is frequently required in data exchange scenarios to restructure the information in the original relations prior to the mapping of the relations to XML. Existing approaches to value-based XML integrity constraints often fail in preserving the semantics of original relational integrity constraints when the restructuring of information prior to the mapping leads to changes in the structure of individual tuples.

The enhanced ‘closest node’ approach to XML keys and foreign keys, which was presented in this thesis, adequately handles multiple and also absent property nodes. Moreover, enhanced ‘closest node’ XML keys and foreign keys preserve the semantics of relational keys and foreign keys when a set of relations is mapped to an XML document by first restructur-

[1] An entity node is the representation of a real world entity in XML data.
[2] A property node is the representation of an entity property in XML data.

ing the information in the relations by applying an arbitrary sequence of nesting operations and then mapping the nested relations directly to an XML document.

Table 8.1 compares the enhanced 'closest node' approach to previous approaches to value-based XML integrity constraints[3] with respect to the requirements of (i) adequately handling multiple property nodes, (ii) adequately handling absent property nodes, and (iii) preserving relational data semantics when information is restructured during the transformation process by applying arbitrary sequences of nesting operations to the relations prior to the mapping of the relations to XML. The symbols in Table 8.1 have the following meaning: Symbols $+/-$ denote that an approach meets/does not meet the requirement. Symbol $\sim$ denotes that an approach meets the requirement only with special assistance of the application developer. Symbol $\times$ indicates that the expressivity of XML integrity constraints in an approach is too restricted to meet the requirement.

Requirement / *Approach*	Handle Multiple Property Nodes	Handle Absent Property Nodes	Preserve Relational Semantics
ID and IDREF	$\times$	$\times$	$\times$
Subtree-based approach	$+$	$-$	$+$
Enhanced closest-node approach	$+$	$+$	$+$
Selector/Field Approaches			
With restrictions on fields	$\times$	$\times$	$\times$
With restrictions on field nodes	$-$	$-$	$-$
Unrestricted	$-$	$-$	$-$
Tuple-based Approaches			
Intersection path approach	$-$	$-$	$-$
Tree-tuple approach	$+$	$-$	$+$
Weak closest-node approach	$-$	$-$	$-$
Strong closest-node approach	$+$	$-$	$+$
Hedge-based approach	$+$	$-$	$+$
Formula-based Approaches			
XML template functional dependencies	$\sim$	$-$	$\sim$
XML embedded dependencies	$\sim$	$-$	$\sim$

Table 8.1: Comparison of approaches to value-based XML integrity constraints.

The majority of previous approaches to value-based XML integrity constraints do not adequately handle multiple property nodes nor do they allow to preserve relational data

[3]A detailed evaluation of these approaches is to be found in Chapter 2.

semantics. In fact, all of the previous approaches to value-based XML integrity constraints fail in adequately handling absent property nodes.

We now review the major concepts applied in the design of enhanced 'closest node' XML keys and foreign keys and the major theoretical results established in this thesis. We also give an outlook on possible future work.

Definition of the Syntax: Irrespective of the particular data model used, a foreign key is the combination of a key and an inclusion dependency. The key asserts that certain values identify referenced data entities (entity nodes), and the inclusion dependency asserts a subset relationship between distinguished values of referencing data entities and the identifying values of referenced data entities. To facilitate keys and foreign keys for the XML data model, we defined XML keys (XKeys) and XML inclusion dependencies (XINDs). The syntax of XKeys and XINDs adopts the selector/field framework of XML keys and foreign keys in XML Schema. The selector is used for selecting entity nodes in an XML document, and the fields are used to relate combinations of property nodes to selected entity nodes, which provides application developers with an intuitive manner to specify XKeys and XINDs. In designing XKeys and XINDs we concentrated on the semantics of XKeys and XINDs so as to adequately handle multiple or absent property nodes. We therefore restricted the syntactic expressivity of XKeys and XINDs. In particular, XKeys and XINDs only allow for simple paths as selectors and fields, whereas XML Schema keys and foreign keys allow for a restricted form of XPath expressions. Also, the fields in an XKey or XIND are required to end in attribute or text labels, whereas XML Schema keys and foreign keys allow fields to also end in element labels, which enables the comparison of selected entity nodes on the basis of entire subtrees in an XML document (tree). To revisit the semantics of XKeys and XINDs when these syntactic restrictions are relaxed is left to future work.

Definition of the Semantics: In defining the semantics of XKeys and XINDs we adopted the strong 'closest node' approach originally presented by Vincent et al. when defining an XML functional dependency [20]. In the 'closest node' approach, a combination of property nodes is used for the purpose of value-based comparison of entity nodes if the nodes pairwise satisfy the *closest* property. Intuitively, a pair of nodes satisfy the *closest* property if they cannot be arranged more closely in the XML document with respect to the paths that lead to the nodes, and so XKeys and XINDs disregard combinations of property nodes which are not closely arranged in the XML document. As a consequence, XKeys and XINDs compare entity nodes only on the basis of semantically correct combinations of property nodes since the arrangement of nodes in an XML document reflects the coherence in the information represented by the nodes [4]. Hence, adopting the strong 'closest node' approach allows XKeys and XINDs to adequately handle multiple property nodes.

The enhancement to the strong 'closest node' approach proposed in this thesis regards the manner in which absent property nodes are handled. In the strong 'closest node' approach, strong satisfaction semantics is applied to XML integrity constraints, which means that the constraint is checked on a (virtual) completion of the XML document. To apply strong satisfaction semantics leads to counter-intuitive results in checking the satisfaction of

[4]This rationale is frequently found in work in the area of XML. For example in the approaches to XML keyword search in [43, 44].

an XML integrity constraint in case that the absence of a property does not indicate that some information is missing but merely reflects the fact that there is some heterogeneity in represented real world entities. Also, even if the absence of a nodes indicates missing information, applying strong satisfaction semantics is not appropriate since it is recommended in the XML specification by the W3C to use the *no information* interpretation of missing information, i.e. if a node is absent this merely indicates that some information is not present, for whatever reason. If the *no information* interpretation of missing information is applied, it is clearly inappropriate to assume that there exists a completion of the XML document, which is the basic assumption for applying strong satisfaction semantics to XML integrity constraints. In contrast to the strong satisfaction semantics applied in the original 'closest node' approach, XKeys and XINDs compare entity nodes purely on the basis of existing combinations of property nodes. This novel technique ensures that checking the satisfaction of an XKey or XIND in an XML document yields the expected result also in case of absent property nodes.

Transformation Procedure: To address the fact that XML data is frequently generated from relational data in data exchange scenarios, we presented a procedure for transforming a set of flat relations to an XML document by first applying arbitrary sequences of nesting operations to the individual flat relations and then mapping the nested relations directly to XML documents which are finally added as principal subtrees to the resulting XML tree (document). This procedure extends the transformation procedure presented by Vincent et al. [25] from the transformation of single relations to the transformation of sets of relations, and allows the application developer to govern the restructuring of information during the transformation process by means of specifying the nesting operations to be applied to the initial flat relations. We developed precise algorithms for transforming a set of relations to an XML document according to the procedure just outlined, and we also developed algorithms for deriving XKeys and XINDs in accordance to the original keys and inclusion dependencies. We established the result that if a set of complete flat relations satisfy a set of keys and inclusion dependencies, then the XML document obtained from these relations satisfies the XKeys and XINDs derived from the relational constraints. In establishing this result, we reused the result by Vincent et al. that a set of property nodes in the obtained XML document pairwise satisfy the *closest property* iff the values of these property nodes appear in the same tuple in the initial flat relation.

To investigate the effects on the property of XKeys and XINDs to preserve the semantics of relational keys and inclusion dependencies when allowing for greater flexibility in the restructuring of information during the transformation process is left to future work.

Theoretical Results: To lay the foundation for database systems to utilize XKeys and XINDs, especially in order to accomplish essential database tasks like query optimization or automatic schema design, we studied the consistency and implication problems related to XKeys and XINDs in the class of complete XML documents originally proposed by Vincent et al. [20]. Complete XML documents generalize the notion of complete relations and are a natural subclass in 'data-centric' XML applications. We established the result that every set of XKeys or XINDs is consistent, i.e. that there exists for every set of XKeys or XINDs at least one non-empty and complete XML document which satisfies the XKeys or XINDs.

We presented sets of inference rules for the implication of XKeys and XINDs in complete XML documents, and established the soundness and completeness of these sets of inference rules. We also developed decision procedures for the implication of a single XKey or XIND by a set of XKeys or XINDs. That is, we have shown that it is effectively decidable whether a single XKey or XIND is necessarily satisfied in every complete XML document which satisfies a given set of XKeys or XINDs.

The study of XKeys and XINDs is worth to be continued in two ways. First, the combined implication and consistency problems for XKeys and XINDs in complete XML documents could be investigated. The result on the combined implication problem is however likely to be negative given the well known undecidability result on combined implication problem for functional dependencies and inclusion dependencies in the relational setting. Second, our findings regarding the implication problems related to XKeys and XINDs, especially the sets of inference rules, lend themselves naturally to be extended to the general class of arbitrary XML documents.

List of Figures

List of Tables

List of Definitions

Bibliography

[1] Carlo Batini and Monica Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Data-Centric Systems and Applications. Springer-Verlag, 2006.

[2] Efraim Turban, Dorothy Leidner, Ephraim McLean, and James Wetherbe. *Information Technology for Management: Transforming Organizations in the Digital Economy*. John Wiley & Sons, 6th edition, March 2007.

[3] Wayne Eckerson. Data Quality and the Bottom Line: Achieving Business Success through a Commitment to High Quality Data. Technical report, The Data Warehousing Institute, 2002. `http://download.101com.com/pub/tdwi/Files/DQReport.pdf`.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.

[5] Tim Bray, Jean Paoli, Michael Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible Markup Language (XML) 1.0. Technical report, World Wide Web Consortium (W3C), November 2008. `http://www.w3.org/TR/xml/`.

[6] Andreas Möller and Michael Schwartzbach. *An Introduction to XML and Web Technologies*. Addison-Wesley Publishing Company, 2006.

[7] Athena Vakali, Barbara Catania, and Anna Maddalena. XML Data Stores: Emerging Practices. *IEEE Internet Computing*, 9(2):62–69, 2005.

[8] Ronald Fagin, Phokion G. Kolaitis, and Reneé J. Miller. Data Exchange: Semantics and Query Answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[9] Alon Y. Halevy, Naveen Ashish, Dina Bitton, Michael J. Carey, Denise Draper, Jeff Pollock, Arnon Rosenthal, and Vishal Sikka. Enterprise Information Integration: Successes, Challenges and Controversies. In *SIGMOD Record*, pages 778–787. ACM, 2005.

[10] Wenfei Fan. XML Constraints: Specification, Analysis, and Applications. In *Database and Expert Systems Applications Workshops*, pages 805–809. IEEE Computer Society, 2005.

[11] David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer. Technical report, World Wide Web Consortium (W3C), October 2004. `http://www.w3.org/TR/xmlschema-0/`.

[12] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures. Technical report, World Wide Web Consortium (W3C), October 2004. `http://www.w3.org/TR/xmlschema-1/`.

[13] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes. Technical report, World Wide Web Consortium (W3C), October 2004. `http://www.w3.org/TR/xmlschema-2/`.

[14] Wenfei Fan and Jérôme Siméon. Integrity Constraints for XML. *Computer and System Sciences*, 66(1):254–291, 2003.

[15] Wenfei Fan and Leonid Libkin. On XML Integrity Constraints in the Presence of DTDs. *ACM*, 49(3):368–406, 2002.

[16] Marcelo Arenas, Wenfei Fan, and Leonid Libkin. On Verifying Consistency of XML Specifications. In *Symposium on Principles of Database Systems*, pages 259–270. ACM, 2002.

[17] Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. Keys for XML. *Computer Networks*, 39(5):473–487, 2002.

[18] Wenfei Fan, Gabriel M. Kuper, and Jérôme Siméon. A Unified Constraint Model for XML. *Computer Networks*, 39(5):489–505, 2002.

[19] Lucja Kot and Walker M. White. Characterization of the Interaction of XML Functional Dependencies with DTDs. In *International Conference on Database Theory*, volume 4353 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 2007.

[20] Millist W. Vincent, Jixue Liu, and Mukesh K. Mohania. On the Equivalence between FDs in XML and FDs in Relations. *Acta Informatica*, 44(3-4):207–247, 2007.

[21] Sven Hartmann and Sebastian Link. More Functional Dependencies for XML. In *Advances in Databases and Information Systems*, volume 2798 of *Lecture Notes in Computer Science*, pages 355–369. Springer, 2003.

[22] Sumon Shahriar and Jixue Liu. On Defining Functional Dependency for XML. *International Conference on Semantic Computing*, 0:595–600, 2009.

[23] Marcelo Arenas and Leonid Libkin. A Normal Form for XML Documents. *ACM Transactions on Database Systems*, 29:195–232, 2004.

[24] Kamsuriah Ahmad and Hamidah Ibrahim. Functional Dependencies and Inference Rules for XML. In *International Symposium on Information Technology*, volume 1, pages 1–6. IEEE Computer Society, 2008.

[25] Millist W. Vincent and Jixue Liu. Functional Dependencies for XML. In *Asia-Pacific Web Conference*, volume 2642 of *Lecture Notes in Computer Science*, pages 22–34. Springer, 2003.

[26] Michael J. Carey, Jerry Kiernan, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *International Conference on Very Large Data Bases*, pages 646–648. Morgan Kaufmann, 2000.

[27] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene J. Shekita, Catalina Fan, and John E. Funderburk. Querying XML Views of Relational Data. In *International Conference on Very Large Data Bases*, pages 261–270. Morgan Kaufmann, 2001.

[28] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently Publishing Relational Data as XML Documents. *Very Large Data Bases*, 10(2-3):133–154, 2001.

[29] Mary F. Fernandez, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, and Wang Chiew Tan. SilkRoute: A Framework for Publishing Relational Data in XML. *ACM Transactions on Database Systems*, 27(4):438–493, 2002.

[30] Xiaochun Yang and Guoren Wang. Mapping Referential Integrity Constraints from Relational Databases to XML. In *Advances in Web-Age Information Management*, volume 2118 of *Lecture Notes in Computer Science*, pages 329–342. Springer, 2001.

[31] Jingtao Zhou, Shusheng Zhang, Hongwei Sun, and Mingwei Wang. An XML-based Schema Translation Method for Relational Data Sharing and Exchanging. In *International Conference on Computer Supported Cooperative Work in Design*, volume 1, pages 714–717. IEEE Computer Society, 2004.

[32] Jinhyung Kim, Dongwon Jeong, and Doo-Kwon Baik. A Translation Algorithm for Effective RDB-to-XML Schema Conversion Considering Referential Integrity Information. *Journal of Information Science and Engineering*, 25(1):137–166, 2009.

[33] Michael Benedikt, Chee Yong Chan, Wenfei Fan, Juliana Freire, and Rajeev Rastogi. Capturing both Types and Constraints in Data Integration. In *ACM SIGMOD International Conference on Management of Data*, pages 277–288. ACM, 2003.

[34] Chengfei Liu, Millist W. Vincent, and Jixue Liu. Constraint Preserving Transformation from Relational Schema to XML Schema. *World Wide Web*, 9(1):93–110, 2006.

[35] Joseph Fong and San Kuen Cheung. Translating Relational Schema into XML Schema Definition with Data Semantic Preservation and XSD Graph. *Information & Software Technology*, 47(7):437–462, 2005.

[36] Rui Zhou, Chengfei Liu, and Jianxin Li. Holistic Constraint-Preserving Transformation from Relational Schema into XML Schema. In *International Conference on Database Systems for Advanced Applications*, volume 4947 of *Lecture Notes in Computer Science*, pages 4–18. Springer, 2008.

[37] Wenyue Du, Mong Li Lee, and Tok Wang Ling. XML Structures for Relational Data. In *International Conference on Web Information Systems Engineering*, volume 1, pages 151–160. IEEE Computer Society, 2001.

[38] Dongwon Lee, Murali Mani, Frank Chiu, and Wesley W. Chu. NeT & CoT: Translating Relational Schemas to XML Schemas using Semantic Constraints. In *International Conference on Information and Knowledge Management*, pages 282–291. ACM, 2002.

[39] Dongwon Lee, Murali Mani, and Wesley W. Chu. Schema Conversion Methods between XML and Relational Models. In Borys Omelayenko and Michel C. A. Klein, editors, *Knowledge Transformation for the Semantic Web*, volume 95 of *Frontiers in Artificial Intelligence and Applications*, pages 1–17. IOS Press, 2003.

[40] Teng Lv and Ping Yan. Schema Conversion from Relation to XML with Semantic Constraints. In *International Conference on Fuzzy Systems and Knowledge Discovery*, volume 4, pages 619–623. IEEE Computer Society, 2007.

[41] Michael Karlinger, Millist W. Vincent, and Michael Schrefl. Keys in XML: Capturing Identification and Uniqueness. In *Web Information Systems Engineering*, volume 5802 of *Lecture Notes in Computer Science*, pages 563–571. Springer, 2009.

[42] Michael Karlinger, Millist W. Vincent, and Michael Schrefl. Inclusion Dependencies in XML: Extending Relational Semantics. In *Database and Expert Systems Applications*, volume 5690 of *Lecture Notes in Computer Science*, pages 23–37. Springer, 2009.

[43] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD Conference*, pages 16–27. ACM, 2003.

[44] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. XSEarch: A Semantic Search Engine for XML. In *International Conference on Very Large Data Bases*, pages 45–56. Morgan Kaufmann, 2003.

[45] D. Small. Data management News. `http://searchdatamanagement.techtarget.com/news/article/0,289142,sid91-gci1218078,00.html`, 2006.

[46] James Clark and Murata Makoto. RELAX NG Specification. Technical report, The Organization for the Advancement of Structured Information Standards (OASIS), December 2001. `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`.

[47] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. An Automata-Theoretic Approach to Regular XPath. In *Database Programming Languages*, volume 5708 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2009.

[48] Marta Jacinto, Giovani Rubert Librelotto, José Carlos Ramalho, and Pedro Rangel Henriques. XCSL: XML Constraint Specification Language. *CLEI Electronic Journal*, 6(1), 2003.

[49] Dongwon Lee and Wesley W. Chu. Comparative Analysis of Six XML Schema Languages. *ACM SIGMOD Record*, 29(3):76–87, 2000.

[50] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML Schema Languages Using Formal Language Theory. *ACM Transactions on Internet Technology*, 5(4):660–704, 2005.

[51] Yan Lu, Zhongxiao Hao, and Ran Dai. A Sufficient and Necessary Condition for the Absolute Consistency of XML DTDs. In *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 249–254. IEEE Computer Society, 2007.

[52] Serge Abiteboul and Victor Vianu. Regular Path Queries with Constraints. *Computer and System Sciences*, 58(3), 1999.

[53] Peter Buneman, Wenfei Fan, Jérôme Siméon, and Scott Weinstein. Constraints for Semi-structured Data and XML. *ACM SIGMOD Record*, 30(1):47–45, 2001.

[54] Paul Grosso, Eve Maler, Jonathan Marsh, and Norman Walsh. XPointer Framework. Technical report, World Wide Web Consortium (W3C), March 2003. `http://www.w3.org/TR/xptr-framework/`.

[55] Steve DeRose, Eve Maler, and David Orchard. XML Linking Language (XLink) 1.0. Technical report, World Wide Web Consortium (W3C), June 2001. `http://www.w3.org/TR/xlink/`.

[56] Peter Buneman, Wenfei Fan, and Scott Weinstein. Path Constraints in Semistructured Databases. *Computer and System Sciences*, 61(2):146–193, 2000.

[57] Yves Andre, Anne-Cécile Caron, Denis Debarbieux, Yves Roos, and Sophie Tison. Path constraints in semistructured data. *Theoretical Computer Science*, 385(1-3):11–33, 2007.

[58] Peter Buneman, Wenfei Fan, and Scott Weinstein. Interaction between Path and Type Constraints. *ACM Transactions on Computational Logics*, 4(4):530–577, 2003.

[59] Katerina Opocenska and Michal Kopecky. Incox - A Language for XML Integrity Constraints Description. In *International Workshop on Databases*, volume 330 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[60] Rick Jelliffe. Schematron. Technical Report 19757-3:2006, ISO/IEC, 2006.

[61] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language. Technical report, World Wide Web Consortium (W3C), January 2007. `http://www.w3.org/TR/xpath20/`.

[62] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Járôme Siméon. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium (W3C), January 2007. `http://www.w3.org/TR/xquery/`.

[63] Sven Hartmann, Henning Köhler, Sebastian Link, Thu Trinh, and Jing Wang. On the Notion of an XML Key. In *Semantics in Data and Knowledge Bases*, volume 4925 of *Lecture Notes in Computer Science*, pages 103–112. Springer, 2008.

[64] Teng Lv and Ping Yan. A Survey Study on XML Functional Dependencies. *International Symposium on Data, Privacy, and E-Commerce*, 0:143–145, 2007.

[65] Junhu Wang. A Comparative Study of Functional Dependencies for XML. In *Asia-Pacific Web Conference*, volume 3399 of *Lecture Notes in Computer Science*, pages 308–319. Springer, 2005.

[66] Wenfei Fan and Leonid Libkin. On XML Integrity Constraints in the Presence of DTDs. In *Symposium on Principles of Database Systems.* ACM, 2001.

[67] Marcelo Arenas, Wenfei Fan, and Leonid Libkin. On the Complexity of Verifying Consistency of XML Specifications. *SIAM Journal on Computing*, 38(3):841–880, 2008.

[68] Zijing Tan, Wei Wang, and Baile Shi. Extending Tree Automata to Obtain Consistent Query Answer from Inconsistent XML Document. In *International Multi-Symposium of Computer and Computational Sciences*, pages 488–495. IEEE Computer Society, 2006.

[69] Marcelo Arenas, Wenfei Fan, and Leonid Libkin. What's Hard about XML Schema Constraints? In *Database and Expert Systems Applications*, volume 2453 of *Lecture Notes in Computer Science*, pages 269–278. Springer, 2002.

[70] Zijing Tan. Validating XML Constraints Using Automata. In *International Conference on Computer and Information Science*, pages 1205–1210. IEEE Computer Society, 2009.

[71] Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. Keys for xml. In *World Wide Web Conference*, pages 201–210. ACM, 2001.

[72] Junhu Wang and Rodney W. Topor. Removing XML Data Redundancies Using Functional and Equality-Generating Dependencies. In *Australasian Database Conference*, volume 39 of *CRPIT*, pages 65–74. Australian Computer Society, 2005.

[73] Yi Chen, Susan B. Davidson, and Yifeng Zheng. Validating Constraints in XML. Technical Report MS-CIS-02-03, University of Pennsylvania, Department of Computer & Information Science, 2002.

[74] Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. Reasoning about Keys for XML. *Information Systems*, 28(8):1037–1063, 2003.

[75] Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. Reasoning about Keys for XML. In *Database Programming Languages*, volume 2397 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 2001.

[76] Sven Hartmann and Sebastian Link. Unlocking Keys for XML Trees. In *International Conference on Database Theory*, volume 4353 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2007.

[77] Sven Hartmann and Sebastian Link. Efficient Reasoning about a Robust XML Key Fragment. *ACM Transactions on Database Systems*, 34(2), 2009.

[78] Sven Hartmann and Sebastian Link. Expressive, yet Tractable XML Keys. In *International Conference on Extending Database Technology*, volume 360 of *ACM International Conference Proceeding Series*, pages 357–367. ACM, 2009.

[79] Ping Yan and Teng Lv. Functional Dependencies in XML Documents. In *Workshop on Advanced Web and Network Technologies, and Applications*, volume 3842 of *Lecture Notes in Computer Science*, pages 29–37. Springer, 2006.

[80] Millist W. Vincent, Michael Schrefl, Jixue Liu, Chengfei Liu, and Solen Dogen. Generalized Inclusion Dependencies in XML. In *Asia-Pacific Web Conference*, volume 3007 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 2004.

[81] Cong Yu and H. V. Jagadish. Efficient Discovery of XML Data Redundancies. In *International Conference on Very Large Data Bases*, pages 103–114. ACM, 2006.

[82] Cong Yu and H. V. Jagadish. Xml schema refinement through redundancy detection and normalization. *VLDB*, 17(2):203–223, 2008.

[83] Marcelo Arenas. Normalization Theory for XML. *SIGMOD Record*, 35(4):57–64, 2006.

[84] Millist W. Vincent and Jixue Liu. Multivalued Dependencies in XML. In *British National Conference on Databases*, volume 2712 of *Lecture Notes in Computer Science*, pages 4–18. Springer, 2003.

[85] Millist W. Vincent and Jixue Liu. Checking Functional Dependency Satisfaction in XML. In *International XML Database Symposium*, volume 3671 of *Lecture Notes in Computer Science*, pages 4–17. Springer, 2005.

[86] Millist W. Vincent and Jixue Liu. Completeness and Decidability Properties for Functional Dependencies in XML. *The Computing Research Repository*, cs.DB/0301017, 2003.

[87] Millist W. Vincent, Jixue Liu, and Chengfei Liu. Strong Functional Dependencies and their Application to Normal Forms in XML. *ACM Transactions on Database Systems*, 29(3):445–462, 2004.

[88] Millist Vincent and Jixue Liu. Strong Functional Dependencies and a Redundancy Free Normal Form for XML. In *World Multi-Conference on Systemics, Cybernetics and Informatics*, 2003.

[89] Jixue Liu, Millist W. Vincent, Chengfei Liu, and Mukesh K. Mohania. Checking Multivalued Dependencies in XML. In *Asia-Pacific Web Conference*, volume 3399 of *Lecture Notes in Computer Science*, pages 320–332. Springer, 2005.

[90] Millist W. Vincent and Jixue Liu. Multivalued Dependencies and a 4NF for XML. In *Conference on Advanced Information Systems Engineering*, volume 2681 of *Lecture Notes in Computer Science*, pages 14–29. Springer, 2003.

[91] Millist W. Vincent, Jixue Liu, and Chengfei Liu. A Redundancy Free 4NF for XML. In *International XML Database Symposium*, volume 2824 of *Lecture Notes in Computer Science*, pages 254–266. Springer, 2003.

[92] Jixue Liu, Millist W. Vincent, and Chengfei Liu. Local XML Functional Dependencies. In *Workshop on Web Information and Data Management*, pages 23–28. ACM, 2003.

[93] Fabio Fassetti and Bettina Fazzinga. Approximate Functional Dependencies for XML Data. In *Advances in Databases and Information Systems Research Communications*, volume 325 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[94] Fabio Fassetti and Bettina Fazzinga. FOX: Inference of Approximate Functional Dependencies from XML Data. In *Database and Expert Systems Applications Workshops*, pages 10–14. IEEE Computer Society, 2007.

[95] Sumon Shahriar and Jixue Liu. On Defining Keys for XML. In *International Conference on Computer and Information Technology Workshops*, pages 86–91. IEEE Computer Society, 2008.

[96] Sumon Shahriar and Jixue Liu. On Defining Referential Integrity for XML. *International Symposium on Computer Science and its Applications*, 0:286–291, 2008.

[97] Sumon Shahriar and Jixue Liu. Preserving Functional Dependency in XML Data Transformation. In *Advances in Databases and Information Systems*, volume 5207 of *Lecture Notes in Computer Science*, pages 262–278. Springer, 2008.

[98] Sumon Shahriar and Jixue Liu. Preserving key in XML data transformation. *Acta Informatica*, 46(7):475–507, 2009.

[99] Sumon Shahriar and Jixue Liu. Checking Satisfactions of XML Referential Integrity Constraints. In *Active Media Technology*, volume 5820 of *Lecture Notes in Computer Science*, pages 148–159. Springer, 2009.

[100] Sumon Shahriar and Jixue Liu. On the Performances of Checking XML Key and Functional Dependency Satisfactions. In *On the Move to Meaningful Internet Systems Conferences*, volume 5871 of *Lecture Notes in Computer Science*, pages 1254–1271. Springer, 2009.

[101] Sven Hartmann, Sebastian Link, and Markus Kirchberg. A Subgraph-based Approach Towards Functional Dependencies for XML. In *World-Multiconference on Systemics, Cybernetics and Informatics*, volume 9 of *Computer Science and Engineering II*, pages 200–205, 2003.

[102] Teng Lv and Ping Yan. XML Constraint-tree-based Functional Dependencies. In *International Conference on e-Business Engineering*, pages 224–228. IEEE Computer Society, 2006.

[103] Sven Hartmann and Thu Trinh. Axiomatizing Functional Dependencies for XML with Frequencies. In *International Symposium on Foundations of Information and Knowledge Systems*, pages 159–178, 2006.

[104] Teng Lv and Ping Yan. XML Normal Forms Based on Constraint-Tree-Based Functional Dependencies. In *Advances in Web and Network Technologies, and Information Management Workshops*, volume 4537 of *Lecture Notes in Computer Science*, pages 348–357. Springer, 2007.

[105] Wai Yin Mok. On Utilizing Variables for Specifying FDs in Data-centric XML Documents. *Data & Knowledge Engineering*, 60(3):494–510, 2007.

[106] Alin Deutsch and Val Tannen. Containment and Integrity Constraints for XPath Fragments. In *Knowledge Representation Meets Databases*, volume 45 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2001.

[107] Alin Deutsch and Val Tannen. XML Queries and Constraints, Containment and Reformulation. *Theoretical Computer Science*, 336(1):57–87, 2005.

[108] Alin Deutsch. FOL Modeling of Integrity Constraints (Dependencies). In Ling Liu and Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 1155–1161. Springer, 2009.

[109] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer-Verlag, 3rd edition, 2006.

[110] William Thomas Tutte. *Graph Theory*. Cambridge University Press, 2001.

[111] Arnaud Le Hors, Philippe Le Hgaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document Object Model (DOM) Level 3 Core Specification. Technical report, World Wide Web Consortium (W3C), April 2004. `http://www.w3.org/TR/DOM-Level-3-Core`.

[112] Wai Yin Mok, Yiu-Kai Ng, and David W. Embley. A Normal Form for Precisely Characterizing Redundancy in Nested Relations. *ACM Transactions on Database Systems*, 21(1):77–106, 1996.

[113] Serge Abiteboul and Nicole Bidoit. Non First Normal Form Relations to Represent Hierarchical Organized Data. In *Symposium on Principles of Database Systems*, pages 191–200. ACM, 1984.

[114] Z. Meral Özsoyoglu and Li-Yan Yuan. A Normal Form for Nested Relations. In *Symposium on Principles of Database Systems*, pages 251–260. ACM, March 1985.