1
Introduction

1.1 Documents, schemas, and schema languages

The data format known as extensible mark-up language (XML) describes tree structures based on mark-up texts. The tree structures are formed by inserting, between text fragments, open and end tags that are balanced, like parentheses. A data set thus obtained is often called a *document*. On the surface, XML resembles hypertext mark-up language (HTML), the most popular display format for the Web. The essential difference, however, is that in XML the structure permitted to documents, including the set of tag names and their usage conventions, is not fixed *a priori*.

More precisely, XML allows users to define their own *schemas*; a schema determines the permitted structure of a document. In this sense, it is often said that a schema defines a "subset of XML" and thus XML is a "format for data formats." With the support of schemas each individual application can define its own data format, while virtually all applications can share generic software tools for manipulating XML documents. This genericity is a prominent strength of XML in comparison with other existing formats. Indeed, XML has been adopted with unprecedented speed and range: an enormous number of XML schemas have been defined and used in practice. To raise a few examples, extensible HTML (XHTML) is the XML version of HTML, simple object access protocol (SOAP) is an XML message format for remote procedure calls, scalable vector graphics (SVG) is a vector graphics format in XML, and MathML is an XML format for mathematical formulas.

One naturally asks: what constraints can schemas describe? The answer is: such constraints are defined by a *schema language*. In other words, there is not one schema language but many, each having different constraint mechanisms and thus different expressiveness. To give only a few of the most important, document type definition (DTD), XML Schema, as defined by the World Wide Web Consortium

(W3C), and regular language description for XML (RELAX NG), defined by the Organization for the Advancement of Structured Information Standards (OASIS) and the International Standards Organization (ISO), are actively used in various applications. This lack of a single schema language certainly confuses application programmers, since they need to decide which schema language to choose, and it also troubles tool implementers since they need to support multiple schema languages. However, this situation can also be seen as a natural consequence of XML's strength; such genericity and usability had never been provided by any previous format and thus it has attracted a huge number of developers, though their requirements have turned out to be vastly different.

1.2 Brief history

The predecessor of XML was the standard generalized markup language (SGML). It was officially defined in 1986 (by ISO) but had been used unofficially since the 1960s. The range of users had been rather limited, however, mainly because of the complexity of its specification. Nonetheless it was in this period that several important data formats such as HTML and DocBook (which are now revised as XML formats) were invented for SGML.

In 1998 XML was standardized by W3C. The creators of XML made a drastic simplification of SGML, dropping a number of features that had made the use of SGML difficult. The tremendous success of XML was due to its simplicity, together with its timely fit to the high demand for a standard, non-proprietary, data format.

At first DTD was adopted as a standard schema language for XML. This was a direct adaptation of the version of DTD used for SGML, made in view of the compatibility of XML and SGML. A number of software tools for SGML were already available, and therefore exploiting these was highly desirable at that time. However, the lack of certain kinds of expressiveness that were critical for some applications had by then already been recognized.

Motivated by a new, highly expressive, schema language, the standardization activity for XML Schema started in around 1998. However, it turned out to be an extremely difficult task. One of the biggest difficulties was that the committee was attempting to mix two completely different notions: regular expressions and object orientation. Regular expressions, the most popular notation for string patterns, was the more traditional notion and had been used in DTD since the era of SGML. The reason for using this notion was "internal" since XML (and SGML) documents are based on *texts*, using regular expressions is a very natural way to represent constraints on such ordered data. However, the demand for object orientation was "external." It arose from the coincident popularity of the Java programming language. Java had a rich library support for network programming, and therefore developers naturally wanted an object serialization format for the exchange of this data between applications on the Internet. These two concepts, one based on automata theory and the other based on a hierarchical model, cannot be integrated in a smooth manner. The result after four years' efforts was inevitably a complex gigantic specification (finalized in 2001). Nonetheless, a number of software programmers nowadays attempt to cope with this complexity.

In parallel with XML Schema, there were several other efforts towards the standardization of expressive schema languages. Among others, the aim with RELAX was to yield a simple and clean schema language in the same tradition as DTD but based on the more expressive regular tree languages rather than the more conventional regular string languages. The design was so simple that the standardization went very rapidly (it was released by ISO in 2000) and became a candidate for a quick substitute for DTD. Later, a refinement was made and released as RELAX NG (by OASIS and ISO in 2001). Although these schema languages are not yet widely used in comparison with DTD or XML Schema, the number of users exhibits a gradual increase.

Meanwhile, what has happened in the academic community? After the emergence of XML, at first most academicians were skeptical since XML appeared to be merely a trivial tree structure, which they thought they completely understood. Soon, however, researchers began to notice that this area was full of treasures. In particular, the notion of schemas, which looked like the ordinary *types* found in traditional programming languages, in fact had a very different mathematical structure. This meant that, in order to redo what had been done for conventional types, such as *static typechecking*, a completely new development was needed. Since then a huge amount of research has been undertaken. Scientists now agree that the most central concept relevant to XML's schemas is *tree automata*. The aim of this book is to introduce the theory and practice arising from this direction of research on the foundations of XML processing, that is, the tree-automata approach.

1.3 Overview of the book

Here we summarize the topics covered in this book and explain how they are organized into parts and chapters.

1.3.1 Topics

Schemas and tree automata

With regard to schemas, we first need to discuss the constraint mechanisms that should be provided by a schema and how these can be checked algorithmically.

In Chapter 3 we define an idealized schema language called the *schema model* and, using this, we compare three important schema languages, namely, DTD, XML Schema, and RELAX NG. Then, in Chapter 4, we introduce *tree automata*, which provide a finite acceptor model for trees that has an exact correspondence with schemas. Using the rich mathematical properties of tree automata, we can not only efficiently check the validity of a given document with respect to a given schema but also solve other important problems related to schemas, such as static typechecking.

Chapters 3 and 4 exemplify a pattern of organization used repeatedly in this book. That is, for each topic, we usually first define a specification language that is helpful for the user and then give a corresponding automata formalism that is useful for the implementer. In these two chapters, we introduce schema, primarily for document constraints, and then tree automata for validation and other algorithms. Specification and algorithmics are usually interrelated, since as schemas become more expressive, the algorithmics becomes more difficult. The chapter organization is intended to help the reader to appreciate such trade-offs.

The last paragraph might suggest that if efficiency is important then we should restrict expressiveness. However, if there is a way to overcome inefficiency then a design choice that allows for full expressiveness becomes sensible. In Chapter 8 we present one such case. This chapter gives a series of efficient algorithms for important problems related to tree automata, where all the algorithms are designed in a single paradigm, the *on-the-fly* technique. This technique has been extremely successful in achieving efficiency without compromising expressiveness. The core idea lies in the observation that we can often obtain a final result by exploring only a small part of the entire state space of an automaton.

We will also cover several other advanced techniques related to schemas. In Chapter 9 we will extend schemas with *intersection types* and their corresponding *alternating tree automata*. The latter are not only useful for algorithmics but also enable a clear formalization of certain theoretical analyses presented in later chapters. In Chapter 14 we discuss the *ambiguity* properties of schemas and automata. Such notions are often valuable in practical systems designed to discover typical mistakes made by users. In Chapter 15 we turn our attention to schema mechanisms for the description of *unordered* document structures. This chapter is unique in the sense that all the prior chapters treat ordered data. Unorderedness arises in certain important parts of XML documents, and in certain important kinds of application, and therefore cannot be ignored. However, treating it is technically rather tricky since it does not fit well into the framework of tree automata. Chapter 15 gives an overview of some recent attempts to solve the problem.

Subtree extraction

Once we know how to constrain documents, our next interest is in how to process them, when the most important aim is to extract information from an input XML tree. In Chapter 5 we introduce the notion of *patterns*, which are a straightforward extension of the schema model that allows *variable bindings* to be associated with subtrees. Corresponding to patterns are *marking tree automata*, introduced in Chapter 6. These automata not only accept a tree but also put marks on some of its internal nodes. In these chapters we will discuss various design choices involved in pattern matching and marking tree automata.

As alternatives to patterns, we present *path expressions* and *logic-based approaches*. Path expressions, discussed in Chapter 12, are a specification for the navigation required to reach a target node and are widely used in the notation called *XPath* (standardized by the World Wide Web Consortium, W3C). In the same chapter we define a framework for the corresponding automata, called *tree-walking automata*, which use finite states to navigate in a given tree. We will compare their expressiveness with that of normal tree automata. Chapter 13 gives a logic-based approach to subtree extraction; here we introduce *first-order (predicate) logic* and its extension, *monadic second-order (MSO) logic*, and explain how these can be useful for the concise specification of subtree extraction. Then, in the same chapter we detail the relationship between MSO logic and tree automata, thus linking this chapter to the rest of the book.

Tree transformation and typechecking

In subtree extraction we consider the analysis and decomposition of an input tree; tree transformation combines this with the production of an output tree. In this book we will not go into the details of complex tree transformation languages but, rather, will present several very small transformation languages with different expressivenesses. The reason is that our purpose is to introduce the *typechecking* technique, the book's highlight.

Typechecking is adopted by many popular programming languages in order to statically analyze a given program and to guarantee that certain kinds of error never occur. Since schemas for XML are just like types in such programming languages, we might imagine that there could be a similar analysis for XML transformations. However, vastly different techniques are needed for this since schemas are based on regular expressions (see Section 2.1), which are not standard in the usual programming languages. This book presents a particularly successful approach based on tree automata.

In XML typechecking there are two very different methodologies, namely, *exact typechecking* and *approximate typechecking*. Exact typechecking is the ideal static analyzer; it signals an error if and only if a given program is incorrect. However, such a typechecker cannot deal with a *general* transformation language, that is, one as expressive as Turing machines. This is a direct consequence of the established notion that the behavior of a Turing machine cannot be predicted precisely. Therefore an exact typechecker is necessarily targeted to a *restricted* transformation language. Approximate typechecking, however, has no such limitation since it can give a false-negative answer, that is, it may reject some correct programs. The question is how can we design a "type system" that yields a reasonable set of accepted programs while ensuring tractability? In Chapter 7, we define a small but general transformation language called μXDuce – a subset of XDuce, the first practical XML processing language with static typechecking – and describe an approximate typechecking algorithm for this language. Later in Chapter 10, we introduce a family of *tree transducers*, which are finite-state machine models for tree transformations. These models are sufficiently restricted to perform exact typechecking, and Chapter 11 describes one such algorithm. This algorithm treats only the simplest case, that of *top-down tree transducers*, but even so it includes important ideas in exact typechecking such as *backward inference*.

1.3.2 Organization

Immediately following the introduction is a chapter giving the mathematical preliminaries including regular expressions and string automata. The remaining chapters are divided into two parts. Part I focuses on basic topics that need to be understood by all who are interested in the tree-automata approach to XML processing. This part starts with basic concepts, namely, schemas (Chapter 3), tree automata (Chapter 4), patterns (Chapter 5), and marking automata (Chapter 6). After this, we introduce the XML processing language, μXDuce, and a typechecking algorithm for it (Chapter 7), integrating the preceding basics.

Part II features more advanced topics and collects various notions that are often used in frontier research articles in this area. Therefore this part would be suitable reading for researchers who need a quick introduction to the background or starting points for new contributions. The topics presented are on-the-fly algorithms (Chapter 8), intersection types and alternating tree automata (Chapter 9), tree transducers (Chapter 10), exact typechecking (Chapter 11), path expressions and tree-walking automata (Chapter 12), logic-based queries (Chapter 13), ambiguity (Chapter 14), and unorderedness (Chapter 15). Figure 1.1 depicts the logical dependencies between the chapters in the two parts.

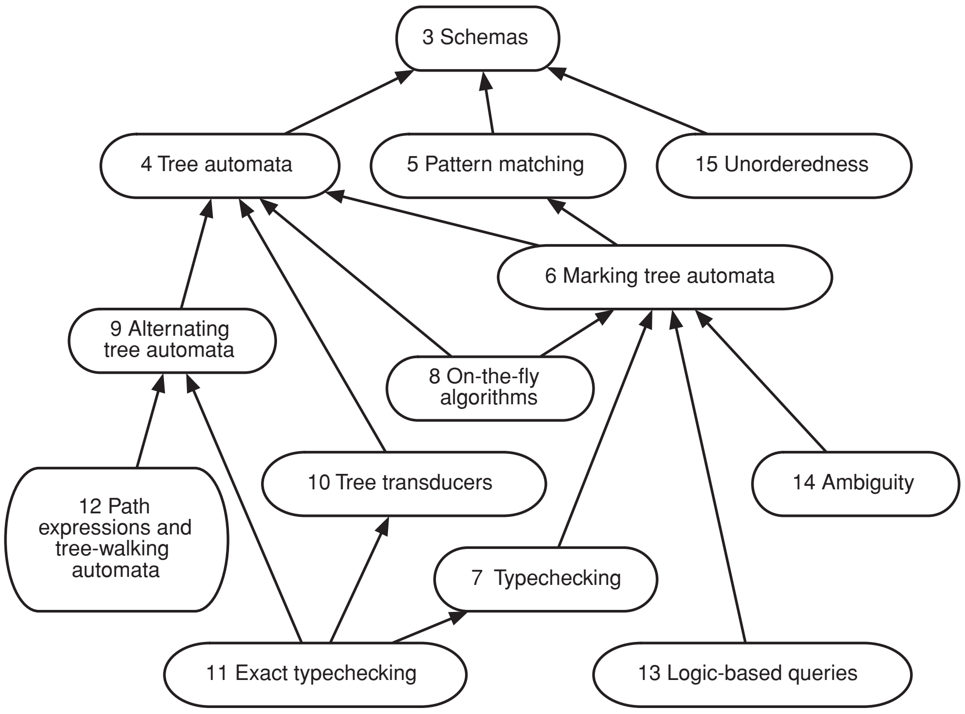

Figure 1.1 Chapter dependencies.

In order to assist understanding, exercises are given in most chapters. Each exercise is marked ⋆ (easy), ⋆⋆ (intermediate), or ⋆⋆⋆ (hard). The reader is encouraged to do all the exercises. Solutions to many of them are given in an appendix at the end of the book.

In an example or an exercise, we sometimes associate its number with concrete instances defined there, such as trees or automata. For example, in Example 4.1.1 we define a tree $t_{4.1.1}$. Such instances may be used several times in later examples, and the reader can use the number to turn back and refer to the original definitions.

Understanding this book requires an adequate background in elementary computer science, including basic set theory, algorithms, and data structures, complexity theory, and formal language theory. In particular, formal language theory is the basis of all the theories described in this book and this is why the summary in Chapter 2 is provided. Readers not familiar with this area are encouraged to study an introductory text first. For this, *Introduction to Automata Theory, Languages, and Computation* (Hopcroft and Ullman, 1979), is recommended. In addition, some background in programming language theory and in XML technologies would help, though these topics are not prerequisites; excellent references are *Types and*

Programming Languages (Pierce, 2002) and *An Introduction to XML and Web Technologies* (Møller and Schwartzbach, 2006). Also, readers who wish to refer to the actual specifications of various standards should look at Bray *et al.* (2000) for XML and DTD, Fallside (2001) for XML Schema, Clark and Murata (2001) for RELAX NG (Murata (2001b) for its predecessor RELAX), and Sperberg-McQueen and Burnard (1994) for a good introduction to SGML.

2

Preliminaries

This chapter introduces some notions used throughout the book, including the basic theory of regular expressions and finite string automata as well as the notational conventions of grammars and inference rules. Readers already familiar with them can skip this chapter.

2.1 Regular expressions

In this book, we often consider sequences of various kinds. For a given set $\mathcal{S}$, we write a *sequence* of elements from $\mathcal{S}$ by simply listing them (sometimes separated with commas); in particular, we write the sequence of length 0 by ϵ and call it the *empty sequence*. The *concatenation* of two sequences s and t is written st (or s, t when commas are used as separators); when a sequence s can be written as tu, we say that t and u are a *prefix* and a *suffix* of s, respectively. The length of a sequence s is written as $|s|$.

The set of all sequences of elements from $\mathcal{S}$ is denoted $\mathcal{S}^*$. For example, $\{a, b\}^*$ contains the sequences $\epsilon, a, aa, ab, abab, \ldots$ When a strict total order $<$ is defined on $\mathcal{S}$, the *lexicographic order* $\preceq$ on $\mathcal{S}^*$ can be defined: $s \preceq t$ if either $t = ss'$ for some s' (i.e., s is a prefix of t) or else $s = uas'$ and $t = ubt'$ for some sequences u, s', t' and some elements a, b with $a < b$ (i.e., s and t have a common prefix u, immediately after which t has a strictly larger element). For example, for sequences in $\{1, 2\}^*$ (with the order $1 < 2$) we have $11 \preceq 112$ and $112 \preceq 121$. Note that $\preceq$ is reflexive since we always have $s = ss'$ with $s' = \epsilon$ (i.e., s is a trivial prefix of s itself).

Let us assume a finite set Σ of *labels* $a, b, \ldots$ We call a sequence from Σ^* a *string*. The set of *regular expressions* over Σ, ranged over by r, is defined by the following grammar:

$$
\begin{array}{rcl}
r & ::= & \epsilon \\
 & & a \\
 & & r_1 r_2 \\
 & & r_1 \mid r_2 \\
 & & r^*
\end{array}
$$

That is, ϵ and any element a of Σ are regular expressions; when r_1 and r_2 are regular expressions, so are $r_1 r_2$ and $r_1 \mid r_2$; when r is a regular expression, so is r^*. For example, we have regular expressions over $\Sigma = \{a, b\}$ such as $(a \mid \epsilon)b$ and $(ab)^*$.

Since this is the first time we are using the grammar notation, let us explain the general convention. A *grammar* of the form

$$
\begin{array}{rcl}
A & ::= & f_1[A_1, \ldots, A_n] \\
 & & \vdots \\
 & & f_k[A_1, \ldots, A_n]
\end{array}
$$

(the form $f[A_1, \ldots, A_n]$ is an expression containing metavariables $A_1, \ldots, A_n$ as subexpressions) inductively defines a set $\mathcal{S}$, ranged over by the metavariable A, such that, if $A_1, \ldots, A_n$ are all in the set $\mathcal{S}$ then $f_i[A_1, \ldots, A_n]$ is also in the set $\mathcal{S}$ for each $i = 1, \ldots, k$. Also, we sometimes use a mutually defined grammar in the form

$$
\begin{array}{rcl}
A & ::= & f_1[A_1, \ldots, A_n, B_1, \ldots, B_m] \\
 & & \vdots \\
 & & f_k[A_1, \ldots, A_n, B_1, \ldots, B_m] \\
 & & \\
B & ::= & g_1[A_1, \ldots, A_n, B_1, \ldots, B_m] \\
 & & \vdots \\
 & & g_l[A_1, \ldots, A_n, B_1, \ldots, B_m]
\end{array}
$$

which inductively defines two sets $\mathcal{S}$ and $\mathcal{T}$, each ranged over by A and B, such that if $A_1, \ldots, A_n$ are all in the set $\mathcal{S}$ and $B_1, \ldots, B_m$ are all in the set $\mathcal{T}$ then each $f_i[A_1, \ldots, A_n, B_1, \ldots, B_m]$ is also in the set $\mathcal{S}$ and each $g_j[A_1, \ldots, A_n, B_1, \ldots, B_m]$ is also in the set $\mathcal{T}$. The notation can be generalized to an arbitrary number of sets.

Returning to regular expressions over Σ, their *semantics* is usually defined by interpreting a regular expression as a set of strings from Σ^*. Formally, the