Connecting XML Processing and Term Rewriting with Tree Grammars

Martin Bravenboer
martin@cs.uu.nl

MSc Thesis
INF/SCR-04-08

Center for Software Technology,
Institute of Information and Computing Sciences,
Utrecht University,
P.O. Box 80089, 3508 TB,
Utrecht, The Netherlands.

20 November 2003

Preface

This master thesis has evolved from quite a weird way of finishing my master study in computer science. I started this study as a Medical Technical Computer Science student. Only when starting the so called 'specialization phase' (comparable to a master study) I found out that I am a geek. In fact I have always known that I love software technology and so I decided to switch to the ordinary Computer Science study. That was a mistake because I had to do the Software Generation course, which was taught by Eelco Visser. At that time I still believed that object orientation is the solution to all your problems and therefore I implemented my project in Java and not in the Stratego language, which has been designed by Eelco Visser. This was not a mistake because I learned from this that meta programming in an object oriented language like Java is horrible. Also this project resulted in an article on the separation of navigation from computation using guided visitors [BV01].

Of course I had to do the 'High Performance Compilers' course, lecturer Eelco Visser, now I had found out that I am a geek. Doing a course on high performance compilers sounds like the sum of all geekness, but actually I really enjoyed it. It was however my second mistake: for this course I had to learn Stratego. Following this course resulted in yet another paper, this time on implementing dynamic instruction selection with rewriting strategies [BV02]. During this course I found out that Stratego is a beautiful language for meta programming and therefore I quited programming this kind of programs in Java.

But still I could not get rid of the past, in which I have implemented a lot of XML processing code in Java and XSLT. Because of this experience I really wanted to help all those Java, XML, and XSLT programmers. You have just started reading the result of this. The main goal of this work is to connect Stratego to the outside, mainstream world where XML is used to exchange data between software tools.

Unfortunately this thesis just covers the core parts of the software that has been written to make Stratego programs more accessible and to make the outside world available to Stratego programs. A lot of software packages, of which some are already used in the Stratego/XT project, are not presented in this thesis. Examples include: java-front, stratego-net, stratego-shell, parse-unit, stratego-box and the aterm database interface. Also a lot of ideas for applying techniques used in the Stratego/XT project to mainstream languages are missing. I hope I will be able to work and publish on this in the future.

Acknowledgments

Many people contributed directly or indirectly to this thesis. First of all I must mention Eelco Visser, my master thesis advisor. His attitude towards my work, and I believe students in general, has been stimulating. On his proposal we have worked together on papers related to the work I have done in his courses. This cooperation has been very instructive. His confidence in allowing us all to work directly on essential parts of his Stratego/XT project is very motivating. Finally I would like to thank him for giving me the freedom to explore interesting areas related to Stratego/XT. Following a strict path towards a master thesis would have cut off a lot of interesting developments and in fact would have resulted in a master thesis on a completely different topic.

Next I would like to thank Rob Vermaas, Niels Janssen, and Merijn de Jonge. They have been the first to apply the tools that I have developed in this project. They, especially Rob, have suffered a lot from being the first users of xml-tools. Arthur van Dam has developed the beautiful LaTeX style that is used in this thesis. His LaTeX expertise and book has been very helpful.

It is quite odd to thank persons who do not know me, but many of the ideas of this thesis have been inspired by lurking at the xml-dev and www-tag mailing lists. Especially the contributions of Simon St. Laurent , Elliotte Rusty Harold, Tim Bray, and James Clark have improved my understanding on various topics in this thesis.

Last but not least, and I know this sounds very boring, I must thank my parents. Believe me, there is reason for this. Tijdens mijn werk aan deze master thesis zijn zij buitengewoon goed en stimulerend voor me geweest. Halverwege mijn werk aan deze thesis hebben ze op overtuigende wijze laten zien dat ze inderdaad de enige zijn op wie ik onder alle omstandigheden kan terugvallen als ik in de ellende zit, iets waar ik in het verleden nog weleens smalend om gelachen heb.

Thesis Techniques

All examples in this thesis are automatically generated by applying real tools in the packages stratego-regular, xml-tools, java-front, and java-xml.

Statistics

In case you do not like XML I have to warn you. This thesis contains the word XML 1338 times. The word regular is used 264 times and Stratego occurs 306 times.

Contents

Chapter 1

Introduction

Exchange of data is a way of making software tools work together. This method is applied in many different settings. In the Unix philosophy programs are designed to work together by expecting the output of every program to become the input to another [MET78]. The exchange of data between software tools is also applied when exchanging data in the XML language between software tools, especially by invoking services on the Internet. In Stratego/XT program transformation systems the exchange of data in the ATerm format is used to compose small transformation components into big program transformation systems.

In the case of XML services, data is exchanged because a software system needs separately implemented and deployed software tools. Services are often provided by an other party and are then running on different machines in the network. By using a language like XML for the exchange of data the software tools are highly independent of each other. The tools can and will be implemented in different programming languages, will be running on different platforms, and will be running in very different cultures.

In Stratego/XT program transformation systems [Vis03] the data is maily exchanged to allow the development of independent transformation tools that all perform some clearly defined transformation on a program representation. The tools can be composed at will into complete program transformation pipelines. By separating a complex transformation system, like for example a compiler, in small tools that do just one thing, the complexity of program transformation systems is controlled. By exchanging structured data the components can be implemented in different languages and can be used dynamically in new compositions, even in an end-user interface like a command shell.

In this thesis a tool kit is presented that clears the distinction between the world of software tools exchanging XML over a network and the world of Stratego/XT transformation systems. Stratego/XT transformation systems must be able to work together with tools in the outside world. Tools in the outside world must be able to connect with Stratego/XT transformation tools, without having to adopt the ATerm format for data exchange, which software tools developed on top of Stratego/XT are using to exchange data. Connecting the Stratego/XT tools to the world of XML and vice versa serves several purposes:

- Stratego/XT systems can invoke external software tools and services based on XML . This opens an area of new applications of Stratego/XT. Stratego/XT has until now been used to implement complex program transformation systems like compilers, optimizers and interpreters. If Stratego/XT is able to connect to the outside world, data

manipulation systems in general can be implemented in Stratego. In such systems being able to connect to existing data sources and software tools is important.

- Stratego/XT tools can easily be invoked from XML based systems. XML based solutions can thus profit from the large set of tools that has been developed in the Stratego/XT project. More complex XML transformations can be implemented in the powerful Stratego language and existing tools can be invoked as if they have always communicated with each other in the XML language. An illustrative example that not directly comes to mind, because it has been establised to an amazing extent, is the cooperation of server-side applications and a web browser like Mozilla and Internet Explorer. A web browser is just another software tool that works together with software tools on the server-side by exchanging XHTML and other XML based languages. If Stratego/XT software tools can communicate in XML then a web browser can work together with them.
- Scannerless Generalized-LR Parsing (SGLR) [Vis97a] technology, which produces parse trees in the ATerm format, can be used naturally from within XML centered environments. Concrete syntax can be defined in the SDF2 syntax definition formalism after which SGLR can be used to parse concrete syntax to an abstract syntax tree in XML .

All purposes are in fact examples of the more general contribution of the tool kit that has been developed: XML can be converted *naturally* into an ATerm and vice versa. Software tools developed for one of these exchange formats can now work together with tools developed for the other. The tool kit has been developed with some goals in mind, which are based on observations of typical XML and ATerm processing applications:

1. Different XML processing applications have different needs concerning the representation of an XML document. These needs must be satisfied by providing different representations of an XML document and offering a set of tools that process XML documents to the appropriate representation.
2. Applications that operate on data in the ATerm format expect the data to be more explicitly structured than in typical XML applications. Expecting XML applications to turn over to a more explicit structure in XML documents is not the solution. Instead the structure is implicit in the XML document when a definition of the XML based language is available.
3. The implementation of XML transformations and generators must be done using a convenient, XML like, syntax, not in a verbose exploded representation of the XML document.

In chapters 5 and 6 the tools and languages related to the first issue are presented. The second point is actually related to the first point. Chapter 6 is fully dedicated to this topic and forms the substantial part of our tool kit. In chapter 7 solutions to the third issue are discussed. The chapters 2, 3, and 4 are preliminaries to our work. In chapter 2 we discuss the formal language theory of word, tree, and hedge languages. The tools presented in chapter 6 are based on this work. In chapter 3 we give a short overview of XML and relate XML to the formalism that have been discussed in the previous chapter. In chapter 4 we do the same thing for the ATerm format, but we discuss the ATerm format in more detail because the reader is probably not familiar with this format.

1.1 Implementation

The tools presented in this thesis are available in two packages:

stratego-regular [Brae]
: The stratego-regular package contains all formal language theory related tools that are not directly related to XML . Lines of code: 7060 of which 4711 are written in Stratego .

xml-tools [Braf]
: The xml-tools package requires the stratego-regular package. It applies the tools in stratego-regular to XML . Lines of code: 7212 of which 2537 are written in Stratego .

The packages are currently available as add-on packages of Stratego/XT. They are however expected to become an integral part of Stratego/XT in the near future. All sections contain an overview of the tools in these packages that are relevant to the subject. Having real implementations for formalisms and languages that are used to reason about structured data is a major point of this thesis. Examples will illustrate the effect of invoking tools. Some other packages that have been written in the time this thesis has been written, are not directly relevant to the subject, but are used by applications of xml-tools and stratego-regular. Some examples are taken from theses packages.

java-front [Brab]
: Support for writing program transformation systems for the Java Language. The package consists of a hand crafted SDF2 syntax definition for Java and Generic Java and a hand crafted pretty printer.
Lines of code: 3979

stratego-net [Brad]
: Support for using ATerm and XML services in the Stratego Language.
Lines of code: 2070

samples-net-xml [Brac]
: Illustrates the application of the xml-tools and stratego-net packages.
Lines of code: 972

aterm-dbi [Braa]
: The ATerm Database Interface: a databases independent (because of using JDBC) ATerm service for accessing relational databases.

Part I

Preliminaries

Chapter 2

Word, Tree, and Hedge Language Theory

2.1 Introduction

This chapter surveys the basic results of word, tree, and hedge language theory. The areas of word, and to a less extent, tree languages has been covered extensively in literature [Cho56, CDG+97]. Word languages and parser algorithms are discussed in almost every computer science program. Tree and hedge languages and automata are not that well-known. With the overwhelming acceptance of the XML language [BPSM00] for exchanging tree-like structured data, this area is getting more interest these days. Tree language theory for example has been used for the formal analysis of XML schema languages [LMM00, MLM01] and the design of new XML schema languages [CM01].

First the formal word language theory is discussed. This area is well-known, nevertheless we repeat the grammar related definitions because the formal tree language theory is introduced by comparing the formalisms to the formalisms in word language theory. Finally we discuss hedge language theory, which is applied in reasoning about XML languages.

2.2 Word Languages and Grammars

Written languages are compositions of symbols. By composing symbols words and sentences are formed.

Definition 1: Alphabet
An *alphabet* is a finite set of symbols. □

This kind of alphabet is also called an *unranked alphabet* in a context where ranked alphabets, which will be introduced next, are used as well. When discussing alphabets the uppercase letter $\mathcal{A}$ is a variable that refers to an alphabet. The variable a denotes a symbol that is an element of an alphabet.

Definition 2: Sequence [JS01]
A *sequence* over an alphabet $\mathcal{A}$, denoted by $\mathcal{A}^*$ is defined as:

- [] is a sequence.
- if as is a sequence and $a \in \mathcal{A}$ then $a : as$ is a sequence.

□

The order of symbols in a sequence is thus not in any way restricted. Any list of symbols over some alphabet is a sequence. Formal word languages restrict the allowed sequences.

Definition 3: Formal Word Language [JS01, Wikc]
A *formal word language* is a subset of $\mathcal{A}^*$, for some unranked alphabet $\mathcal{A}$. A sequence in a formal word language is called a *sentence* of this language. □

Formal word grammars are a way to define word languages. A formal word grammar thus defines a set of sequences of over symbols in some alphabet. The idea of a formal word grammar is to have a start symbol and a set of production rules that rewrite symbols to other symbols. Using these production rules the sentences of the word language defined by the word grammar can be generated. Word grammars are not the only way to define a word language. Regular expressions are an alternative way to define a (regular) word language. They will be discussed later.

Definition 4: Formal Word Grammar [JS01, Wikb]
A *formal word grammar* is a tuple $(\mathcal{S}, \mathcal{N}, \mathcal{A}, P)$ where

- $\mathcal{S}$ is a set of start symbols $\mathcal{S} \subset \mathcal{N}$.
- $\mathcal{N}$ an unranked alphabet called non-terminal symbols.
- $\mathcal{A}$ is an unranked alphabet, called terminal symbols.
- P is a set of production rules of the form $\alpha \to \beta$, where $\alpha, \beta \in (\mathcal{N} \cup \mathcal{A})^*$

□

The language of formal word grammars is extremely general. The Chomsky hierarchy defines a set of three more restricted classes of formal word grammars. The more restrictive types of word grammars defined by Noam Chomsky impose certain constraints on the production rules of a grammar. The more restrictive these constraints for a certain class are, the less formal word languages can be described by the class.

Definition 5: Chomsky Hierarchy [Cho56, Wika]
The Chomsky hierarchy consists of four levels: Type-0, Type-1, Type-2, and Type-3. The levels 1, 2 and 3 impose constraints on the production rules of a formal word grammar $(\mathcal{S}, \mathcal{N}, \mathcal{A}, P)$

Type-0: Unrestricted
$\alpha \to \beta$, where α and $\beta \in (\mathcal{N} \cup \mathcal{A})^*$

Type-1: Context-sensitive Grammars
$\alpha \to \beta$, where α contains at least one non-terminal and β contains at least one terminal or non-terminal.

Type-2: Context-free Grammars
$A \to \beta$, where $A \in \mathcal{N}$

Type-3: Regular Grammars
$A \rightarrow a$ or $A \rightarrow aB$, where A and $B \in \mathcal{N}$ and $a \in \mathcal{A}$.

□

Of this set of formal word grammar types the context-free languages and regular languages are widely used in computer science because sentences in the languages generated by these grammars are efficiently recognizable. Unrestricted word grammars generally require the computational completeness of a Turing machine to recognize sentences in the languages generated by them.

Regular word grammars (Type-3) are equivalent to regular word expressions. Regular word expressions are extensively used in our work, so they are discussed in more detail. Regular word expressions define, like word grammars, a language over an unranked alphabet.

Definition 6: Regular Word Expression [JS01]
$RegExp_w(\mathcal{A})$ is the set of regular word expressions over the unranked alphabet $\mathcal{A}$. For R, R_1, and $R_2 \in RegExp_w(\mathcal{A})$ and $a \in \mathcal{A}$, $RegExp_w(\mathcal{A})$ is defined by induction as:

$$\begin{aligned} \emptyset &\in RegExp_w(\mathcal{A}) \\ \epsilon &\in RegExp_w(\mathcal{A}) \\ a &\in RegExp_w(\mathcal{A}) \\ R_1R_2 &\in RegExp_w(\mathcal{A}) \\ R_1|R_2 &\in RegExp_w(\mathcal{A}) \\ R* &\in RegExp_w(\mathcal{A}) \\ R+ &\in RegExp_w(\mathcal{A}) \\ R? &\in RegExp_w(\mathcal{A}) \\ (R) &\in RegExp_w(\mathcal{A}) \end{aligned}$$

□

The word language defined by such a regular word expressions is defined in the next definition. This is the semantics of regular word expressions.

Definition 7: Language of a Regular Word Expression [JS01]
The language of a regular word expression $RegExp_w(\mathcal{A})$ is denoted by $[\![RegExp_w(\mathcal{A})]\!]$. $[\![RegExp_w(\mathcal{A})]\!]$ is defined by induction as:

$$\begin{aligned} [\![\emptyset]\!] &= \emptyset \\ [\![\epsilon]\!] &= \{[]\} \\ [\![a]\!] &= \{a\} \ , \ where\ a \in \mathcal{A} \\ [\![R_1R_2]\!] &= [\![R_1]\!][\![R_2]\!] \\ [\![R_1|R_2]\!] &= [\![R_1]\!] \cup [\![R_2]\!] \\ [\![R*]\!] &= [\![R]\!]^* \\ [\![R+]\!] &= [\![RR*]\!] \\ [\![R?]\!] &= [\![\epsilon]\!] \cup [\![R]\!] \\ [\![(R)]\!] &= [\![R]\!] \end{aligned}$$

For R, R_1, and $R_2 \in RegExp_w(\mathcal{A})$ and $a \in \mathcal{A}$ □

2.3 Tree Languages and Grammars

Formal word language theory cannot be applied to reason about tree languages. In word languages finite automata and formal word grammars are used for recognizing and defining word languages. For tree languages highly related formalisms exist. In the tool kit that will be developed in this thesis, these formalisms play a central role. This section reviews the basic terminology and formalisms in this area.

Tree language theory is not based on the alphabets used for languages. Instead ranked alphabets are used. Symbols in a ranked alphabets have an arity.

Definition 8: Ranked Alphabet [CDG+97]
A ranked alphabet is tuple ($\mathcal{A}$, $Arity$) where

- $\mathcal{A}$ is an alphabet.
- $Arity$ is a relation $\mathcal{A} \rightarrow \mathbb{N}$

□

A ranked alphabet is denoted by a variable $\mathcal{F}$, or a term $\mathcal{F}(\mathcal{A}, Arity)$, where $\mathcal{A}$ denotes the alphabet of symbols and $Arity$ is the arity relation. A symbol in a ranked alphabet is denoted by a variable f. This notation does not specify the actual arity of the symbol. A symbol f in a ranked alphabet $\mathcal{F}$ can have more then one arity. We use the notation f/n to refer to a symbol and its rank in a ranked alphabet. It denotes that f has arity n in an ranked alphabet. This is not necessarily the only arity because f can have more than one arity.

The right image $f.Arity$ maps the symbol f in $\mathcal{A}$ to a set of arities. The left image $Arity.n$ maps the arity n to the set of symbols of this arity. It is also denoted as $\mathcal{F}/n$. $f/n \in \mathcal{F}$ means that $f \in \mathcal{A} \wedge n \in f.Arity$. Symbols of arity 0,1, ... n are respectively called a constants, unary, binary, ... n-ary symbols.

In examples ranked alphabets the alphabet $\mathcal{A}$ is not explictly mentioned because it is equivalent to the domain of the relation $Arity$. Also we use the f/n notation for specifying the arity relation.

Example 1: Ranked Alphabet
The ranked alphabet $\mathcal{F}_1(\mathcal{A}, Arity)$ where

$$\mathcal{A} = \{plus, mul, int, succ, zero\}$$

$$Arity = \{(plus, 2), (mul, 2), (int, 1), (succ, 1), (zero, 0)\})$$

is denoted as:

$$\mathcal{F}_1 = \{plus/2, mul/2, int/1, succ/1, zero/0\}$$

In word languages theory a sequence is a list over symbols some alphabet. Terms are the sequences of the tree language theory. Terms play a central role in this thesis. In literature

[CDG+97] terms are defined as a set over a ranked alphabet and a set of variables. The set of terms without variables is called ground terms. In our work terms with variables play no role, but we use the term ground terms anyway.

Definition 9: Terms [CDG+97]
The set of terms over the ranked alphabet $\mathcal{F}$ and the set of variables $\mathcal{X}$, denoted by $Term(\mathcal{F}, \mathcal{X})$, is defined as:

$$\frac{f/n \in \mathcal{F} \quad \wedge \quad t_1 \dots t_n \in Term(\mathcal{F}, \mathcal{X})}{f(t_1 \dots t_n) \in Term(\mathcal{F}, \mathcal{X})}$$

$$\frac{x \in \mathcal{X}}{x \in Term(\mathcal{F}, \mathcal{X})}$$

□

Definition 10: Ground Terms [CDG+97]
The set of terms over the ranked alphabet $\mathcal{F}$ and an empty set of variables, that is $Term(\mathcal{F}, \emptyset)$, is called a set of ground terms. A term is ground if there are no variables in it. This set is also denoted by $GroundTerm(\mathcal{F})$. □

Example 2: Ground Terms

- $zero()$,
- $succ(zero())$, and
- $mul(int(succ(zero())), int(zero()))$

are ground terms over the ranked alphabet $\mathcal{F}_1$ in example 1.

Like in the case of sequences over an unranked alphabet in word language theory, there is no restriction whatsoever on the structure of ground terms. Tree languages introduce such restrictions, like word languages do for sequences over an unranked alphabet.

Definition 11: Tree Language [CDG+97]
A tree language is a subset of $GroundTerm(\mathcal{F})$, for some ranked alphabet $\mathcal{F}$. □

Tree grammars are a way to define tree languages, like word grammars define word languages. A tree grammar differs from a word grammar in the left and right hand side of production rules. In the case of word grammars they both consist of a sequence of symbols. Naturally in the case of tree grammars these sides are ground terms over the terminals and non-terminals of the grammar. Tree grammars are again not the only way to define a tree language. Regular tree expressions can be used as well for example, but they are less expressive: they can only be used to define regular tree languages.

Definition 12: Tree Grammar [CDG+97]
A *tree grammar* is a tuple $(\mathcal{S}, \mathcal{N}, \mathcal{F}, P)$ where

- $\mathcal{N}$ is a ranked alphabet, called non-terminal symbols

- $\mathcal{S}$ is a set of start symbols, where
  - $\mathcal{S} \subseteq \mathcal{N}$, and
  - $\forall_{f/n \in \mathcal{S}} n = 0$
- $\mathcal{F}$ is a ranked alphabet, called terminal symbols
- P is a set of production rules of the form $\alpha \to \beta$, where
  - α and $\beta \in Term(\mathcal{N} \cup \mathcal{F} \cup \mathcal{X})$, where $\mathcal{X}$ is a set of variables.
  - α contains at least one non-terminal (TODO: weird restriction?).

□

This language of tree grammars is very general, which is probably not surprising in this comparison to word languages. For tree languages there is however not a really well-known hierarchy of languages defined. Therefore, we just define regular tree grammars, which is a widely used class of tree grammars. A more general class of tree grammars are the context-free tree grammars. They are for example discussed in [CDG+97].

A regular tree grammar is a specific form of tree grammar where all non-terminal symbols are constants (have arity 0) and the right hand side of a production rule is just a non-terminal. For the sake of clarity we define regular tree grammars not in terms of tree grammars, but afresh.

Definition 13: Regular Tree Grammar [CDG+97]
A *regular tree grammar* is a tuple $(\mathcal{S}, \mathcal{N}, \mathcal{F}, P)$ where

- $\mathcal{S}$ is a set of start symbols, where $\mathcal{S} \subseteq \mathcal{N}$
- $\mathcal{N}$ is an *unranked* alphabet called non-terminal symbols.
- $\mathcal{F}$ is a ranked alphabet, called terminal symbols
- P is a set of production rules of the form $a \to \beta$, where
  - $a \in \mathcal{N}$ and
  - $\beta \in GroundTerm(\mathcal{N} \cup \mathcal{F})$

□

Example 3: Regular Tree Grammar

$$G_1 = (\{Expr\}, \{Expr, IntConst\}, \{plus/2, mul/2, int/1, succ/1, zero/0\}, \{$$
$$\begin{aligned} Expr &\to plus(Expr, Expr) \\ Expr &\to mul(Expr, Expr) \\ Expr &\to int(IntConst) \\ IntConst &\to succ(IntConst) \\ IntConst &\to zero \end{aligned}$$
$$\})$$

2.4 Hedge Languages and Grammars

We want to apply formal language theory in a tool kit for various languages for exchanging tree-like structured data. The best known languages for this is the XML language. The tool kit will apply formal language theory for the definition of XML based languages. At first sight the tree language theory of the last section might seem suitable for this. Indeed the first publications that applied formal language theory in the context of XML schema languages [LMM00, MLM01] confusingly referred to regular tree grammars. Regular tree grammars as defined in the last section are however not suitable for the formalization and reasoning about XML schema languages. Tree grammars are namely based on ranked alphabets, where in XML elements typically not have a fixed number of children. This is a pity because the formalisms for word and tree languages we have discussed until now are well-known and have been studied deeply.

In the formal XML community a new term has been introduced for a formalism that is suitable for reasoning about XML languages. The term 'forest' was not found suitable for this because this usually refers to a set of trees. In a tutorial at MetaStructures 99 conference Paul Prescod introduced the term *hedge* in the context of XML schema formalisms. Originally the term hedge was proposed by Courcelle [Cou89]. A hedge is thus basically a short name for a sequence of trees. From this time the term regular hedge grammars seems to be preferred over regular tree grammars by the XML community when discussing formalizations of XML schema languages.

In this section hedges, hedge languages, hedge grammars and regular hedge grammars are discussed. We split the definition of an hedge [Mur00, BMW] in two parts: the terms that form a hedge and the hedge itself.

Definition 14: Hedge and Hedge Term
The set of hedge terms over an unranked alphabet $\mathcal{A}$ is denoted by $HedgeTerm(\mathcal{A})$. It is defined as:

$$\frac{a \in \mathcal{A} \quad \wedge \quad H \in HedgeTerm(()\mathcal{A})}{a\langle H \rangle \in HedgeTerm(()\mathcal{A})}$$

The set of hedges over a unranked alphabet $\mathcal{A}$, denoted by $Hedge(\mathcal{A})$ is defined as a sequence of hedge terms, that is:

$$Hedge(\mathcal{A}) = HedgeTerm(\mathcal{A})^*$$

□

Like a word language is a subset of a sequence over an alphabet, and a tree language is a subset of ground terms over a ranked alphabet, a hedge language is a subset of a set of hedges over an unranked alphabet.

Definition 15: Hedge Language
A hedge language is a subset of $Hedge(\mathcal{A})$, for some unranked alphabet $\mathcal{A}$. □

Hedge languages need to defined. Like word and tree grammars are used for word and tree languages, hedge grammars can be used for this. They are again not the only way of specifying hedge languages. A hedge grammar can however define every hedge language.

Definition 16: Hedge Grammar
A hedge grammar is a tuple $(s, \mathcal{N}, \mathcal{A}, P)$ where

- $s \in \mathcal{N}$
- $\mathcal{N}$ is an unranked alphabet, called non terminal symbols.
- $\mathcal{A}$ is an unranked alphabet, called terminal symbols.
- P is a set of production rules of the form $h_1 \rightarrow h_2$ where h_1 and $h_2 \in Hedge(\mathcal{N} \cup \mathcal{A})$.

□

As usual this grammar definition language is extremely general. Restricted forms of hedge grammar and languages are always used in the context of tree and hedge languages. For example schema languages for XML are restricted to the definition of *regular* hedge languages. A regular hedge grammar is a specific form of a hedge grammar that limits the structure of a hedge to a regular language. This is not the only way to define a regular hedge language, as will be illustrated next.

Definition 17: Regular Hedge Grammar
A regular hedge grammar is a tuple $(s, \mathcal{N}, \mathcal{A}, P)$ where

- $s \in \mathcal{N}$. s defines the structure of the top level hedge.
- $\mathcal{N}$ a ranked alphabet, called non terminal symbols, where $\forall_{f/n \in \mathcal{N}} n = 0$
- $\mathcal{A}$ an unranked alphabet, called terminal symbols.
- P is a set of production rules of the form
  - $X_1 \rightarrow a\langle X_2 \rangle$, or
  - $X_1 \rightarrow a\langle X_2 \rangle\ X_3$

  where X_1, X_2, and $X_3 \in \mathcal{N}$, $a \in \mathcal{A}$

□

This definition ensures that the hedges generated by this definition are regular hedge language. In XML schema languages that must be used by humans such a rigid constraint on production rules will of course not work. XML schema languages like W3C XML Schema and RELAX NG use additional constraints beyond the language itself to enforce regularity of the defined language. In RELAX NG simple syntax [CM01] regularity is enforced by requiring that every `element` element is the child of a `define` element, and that the child of every `define` element is an `element` element. In the full syntax RELAX NG disallows recursion when expanding `ref` elements that refer to a `define` element whose child a not an `element` element.

Regular expressions are useful in the definition of a regular hedge language. They provide a clear way of specifying the regular structure of a hedge. Regular hedge grammars are in the context of XML schema languages usually even immediately defined in the variant with regular expressions. This variant is then just called a regular hedge grammar as well.

Definition 18: Regular Hedge Grammar (using Regular Expressions)
A regular hedge grammar is a tuple $(R_s, \mathcal{N}, \mathcal{A}, P)$ where

- $R_s \in RegExp_n(\mathcal{N})$
- $\mathcal{N}$ a ranked alphabet, called non-terminal symbols, where $\forall_{f/n \in \mathcal{N}} n = 0$
- $\mathcal{A}$ an unranked alphabet, called terminal symbols.
- P is a set of production rules of the form $X \rightarrow a\langle R \rangle$, where $X \in \mathcal{N}$, $a \in \mathcal{A}$, and $R \in RegExp_n(\mathcal{N})$.

□

Example 4: Regular Hedge Grammar

$$G_1 = (Expr, \{Expr/0, IntConst/0\}, \{plus, mul, int, succ, zero\}, \{$$

$$\begin{array}{rcll} Expr & \rightarrow & plus & \langle Expr\ Expr \rangle \\ Expr & \rightarrow & mul & \langle Expr\ Expr \rangle \\ Expr & \rightarrow & int & \langle IntConst \rangle \\ IntConst & \rightarrow & succ & \langle IntConst \rangle \\ IntConst & \rightarrow & zero & \langle \epsilon \rangle \end{array}$$

$$\})$$

For word grammars and tree grammars we have not defined the language generated by them. For regular hedge grammars we do not omit this, because they play a central role in this thesis.

Definition 19: Language of a Regular Hedge Grammar
The hedge language generated by some regular hedge grammar (using regular expressions) $G = (R_s, \mathcal{N}, \mathcal{A}, P)$, is denote by $[\![\ G\ ]\!]$.

$$[\![\ G\ ]\!] = [\![\ R_s\ ]\!]_G$$

where $[\![\ R_s\ ]\!]_G$ is the hedge language generated by R_s given the regular hedge grammar G. The language of a regular expression over non-terminals in a regular hedge grammar, that is $R \in RegExp_n(\mathcal{N})$, is defined as

$$
\begin{aligned}
[\![\emptyset]\!]_G &= \emptyset \\
[\![\epsilon]\!]_G &= \{[]\} \\
[\![X]\!]_G &= \{a\langle H\rangle \mid X \to a\langle R\rangle \in P, H \in [\![R]\!]_G\} \\
[\![R_1R_2]\!]_G &= [\![R_1]\!]_G [\![R_2]\!]_G \\
[\![R_1|R_2]\!]_G &= [\![R_1]\!]_G \cup [\![R_2]\!]_G \\
[\![R*]\!]_G &= [\![R]\!]_G^* \\
[\![R+]\!]_G &= [\![RR*]\!]_G \\
[\![R?]\!]_G &= [\![\epsilon]\!]_G \cup [\![R]\!]_G \\
[\![(R)]\!]_G &= [\![R]\!]_G
\end{aligned}
$$

For R, R_1, and $R_2 \in RegExp_n(\mathcal{N}), a \in \mathcal{A}$, and $X \in \mathcal{N}$. □

Chapter 3

Extensible Markup Language

3.1 Introduction

XML is a markup language for exchanging data between software components. These XML processing software components might reside on different computers, different operating systems, different platforms and in different cultures. XML is not the only language for exchanging data: there have been exchange formats around for many years. The internationalization features of XML are however rather unique. By not limiting the character set to the ASCII character set and not even defining a fixed character encoding, XML goes beyond the western world. XML has thus been designed to cope with this diversity of environment.

In this chapter we will give a short overview of the current state of XML as far as relevant to our work. The purpose is not to discuss the XML related languages in depth, but to give a reader unfamiliar with XML an overview of what is going on in the areas of XML schema languages and data models for XML .

3.2 Schema Languages for XML

A schema [1] for a language with an XML syntax describes the structure of the language. Such a a schema is comparable to a word grammar for a word language, a tree grammar for a tree language and a hedge grammar for a hedge language. XML is exchanged between software tools, but XML is not the magic bullet for representing data in a way that everybody can understand the data. There has to be some agreement on the structure of the XML documents that are exchanged. Schema languages for XML have been designed to allow the specification of this structure.

Schemas are used to

- define an XML language,
- validate XML documents that claim to adhere to a schema,

[1] Unfortunately the W3C decided to call their schema language for XML 'XML Schema'. This name suggests that this language is *the* schema language for XML . The name 'schema' in this thesis however refers to a schema in any schema language for XML (e.g. RELAX NG or DTD). W3C XML Schema is used to denote the XML Schema language of the W3C.

- validate programs that claim to consume and produce XML documents that adhere to a schema,
- generate software components and libraries with specific schema knowledge,

Because different language definitions and applications have different needs, there exist several schema languages. These schema languages differ in the level and the kind of abstraction they use to describe XML languages. Well-known schema languages defined by the W3C are DTD, defined in the XML recommendation [BPSM00] itself, and W3C XML Schema [Tho01]. Less well-known, but not at all less attractive, are RELAX NG [CM01], created by James Clark, and Schematron [Jel], created by Rick Jelliffe. The first three schema languages all define the complete structure of a language. Schematron [Jel] is rather different because its main purpose is not to define a language, but to validate a document against a set of rules.

3.2.1 Document Type Definition (DTD)

DTD is part of the original XML recommendation [BPSM00]. XML has inherited DTD from SGML. As shown in [LMM00, MLM01] the DTD language allows the definition of local regular hedge grammars. Besides this lack of expressive power DTD has very poor abstraction capabilities, there is no support for XML namespaces, which nearly every XML related language is using. DTDs are however still used because there are many validating XML processors around that only support the DTD schema language. Also XML schema languages like W3C XML Schema, which is assumed to replace DTD, do not support entities. Recently there have been some proposals to separate the definition of entities from a schema language [Cla03].

3.2.2 RELAX NG

RELAX NG [Clab, CM01] is a schema language developed by James Clark and Murata Makoto. It is standardized by OASIS, but the first specification has not been developed by a committee. RELAX NG is based on regular hedge grammar theory. It allows the full class of regular hedge grammars and therefore it also allows ambiguities. RELAX NG is a small schema language compared to W3C XML Schema. A simplification of the full RELAX NG syntax to a simple syntax is defined in the RELAX NG specification and the semantics of the simple syntax of RELAX NG is well defined. There is also a compact (not XML) syntax for RELAX NG [Cla02].

3.2.3 W3C XML Schema

To attack the deficiencies of DTD, the W3C has created XML Schema [Tho01, SW03], which we will call W3C XML Schema, as new recommendation for defining schemas. W3C XML Schema is itself an XML based language and there is no concrete syntax for it. W3C XML Schema has been designed for applications that need a typed representation of an XML document. The language supports the declaration of primitive data types like integers and it supports abstractions like subtyping. Unfortunately these typing mechanisms do not map very well to existing object oriented languages [MS03a, Oba03]. A disadvantage of W3C XML Schema is that despite its XML syntax, which should be an advantage for software tools, it is very difficult to write proper tooling for it. It is quite a large language

with a complex semantics. Therefore many implementations that claim to support W3C XML Schema actually just support a subset of it.

3.3 Data Models for XML

Data models for XML are abstractions of the XML syntax. Data models differ in how close they are to the original XML document. They all preserve the original XML document to a certain level of detail and regard the details that are lost in this data model to be irrelevant. Several abstractions from the XML syntax exist. They are used in specifications of XML related standards and languages and for the design of APIs that provide the content of an XML document to an application in a certain level of detail.

XML Information Set

The XML Information Set (usually called XML Infoset) recommendation [CT01], created by the W3C XML Core Working Group, introduces a shared terminology for referring to the information in an XML document in XML related specifications. Because the XML Infoset defines terminology for referring to *information* in an XML document, the XML Infoset is often regarded as a definition of what the *information* of an XML document actually consists of. In this way the XML Infoset introduces an abstraction from the XML syntax. The XML Infoset only contains constructs for what it considers to be the important properties of an XML document. Recent W3C specifications like XML Schema and XML Query define their data models in terms of the XML Infoset.

In the XML community the XML Infoset has been under debate from the very beginning, especially at the xml-dev mailing list. The xml.com article Investigating the Infoset [Dod00] provides an interesting overview of the early opinions on the XML Infoset. The main argument against considering the XML Infoset as the data model for an XML document on top of which applications and processors should be build, is that what is important in an XML document differs from application to application. The nature of an application decides what is relevant in an XML document.

Recently the debate has escalated because more and more proposals appear for the binary interchange of the XML Information Item Set. These proposals are based on the assumption that such a binary format is much more efficient and does not fundamentally reduce the interoperability of applications that use such a format. This is an interesting discussion, especially if related to the call for being textual in the Unix philosophy [MET78] of creating small programs that do one thing well and can be composed by making the output of a tool the input of an other tool. By exchanging textual data the tools become transparant for humans.

Canonical XML

The Canonical XML recommendation [Boy01] is developed by the W3C XML Signature Working Group. It does not really define a data model, but is concerned with the relevance of XML language constructs. It might seem odd that a canonicalization of XML has been specified by the XML Signature Working Group. This Working Group was however concerned with when an XML document is different from an other XML document. For this

purpose Canonical XML has been created. This is yet another recommendation that tries to figure out what is relevant in an XML document. After applying the canonicalization defined in this specification to XML documents, they will only be the same if the are 'logically equivalent within an application context'. If the original XML documents were at first not equivalent then they vary only in syntactical details as permitted by XML and XML Namespaces.

RELAX NG Data Model

The RELAX NG data model for XML is used in the semantics of for RELAX NG as defined by the the RELAX NG specification [CM01, Clab]. This model is more abstract and compact then the XML information set. The construction of an instance of this data model from an instance of the XML Infoset is defined in the specification. The RELAX NG data model is very close to a data model we will use later in this thesis. The data model consists of a few constructs: string, name, context, attribute and element. An XML document is represented by an element.

- string, which is a sequence of zero or more characters
- name, which consists of
  - string representing the namespace URI.
  - string representing the local name
- context, which consists of
  - base URI
  - default namespace URI
  - namespace map, which maps namespace prefixes to namespace URIs
- attribute, which consists of
  - name
  - string representing the value
- element, which consists of
  - name
  - context
  - set of attributes
  - sequence of zero or more children. A child is either an element or a non-empty string.

Post Schema Validation Infoset

The Post Schema Validation Infoset (PSVI) is the augmented infoset that results from XML Schema Validation. The PSVI representation includes the *types* of the content in an XML document. The PSVI is strongly related to the W3C XML Schema language. This schema language for XML is targeted at applications that need a typed view on

XML documents. The main purpose of the PSVI is thus to add type information to an XML document as declared in a schema. The main critique on this representation is that the relation to W3C XML Schema is too strong and that W3C XML Schema becomes the essence of XML [SW03]. The more general term for adding type information to an instance document is type-augmented infoset (TAI). Important requirements expressed by the XML community are that such a TAI must not require validation against a schema and that it should not be bound to a specific schema language for XML .

In our work we are also using some kind of type-augmented infoset, although it is not really an augmentation of the instance. In chapter 6 we will present the IRHG language, which describes how a hedge is an instance of a regular hedge grammar. This representation contains even more information than the PSVI. The IRHG completely describes how an XML document is an element of the language generated by a regular hedge grammar.

A major problem with the PSVI is that there is no standard or agreement on a serialization of this representation. This leads to a situation where every single tool must be able to perform schema validation or even worse, it might prevent a separation into reusable components. A schema for serialized PSVI has been proposed [TT01], but for some reason there has not been paid much attention to it.

Chapter 4

Annotated Term Format

4.1 Introduction

The Annotated Term (ATerm) data exchange format [dBdJKO00] is an abstract data type with efficient encodings for the exchange of tree-like structured data between software components. The Annotated Term Format has been developed at the University of Amsterdam and is now maintained and further developed at the CWI. [1] in the Netherlands. The ATerm format is supported by libraries for C, Java, and Haskell. The ATerm format itself is highly related to the ground terms of tree languages (see definition 10).

The ATerm format and libraries are used in several projects related to program transformation. Stratego [Vis01b, VBT98, www] is a transformation language developed at Utrecht University that uses the ATerm format and the ATerm C library. In Stratego, transformations are defined over ATerms. In the ASF+SDF Meta-Environment [dBHdJ+01] programs are represented in ATerms as well. The ATerm format is currently mostly used for program representations. It is however also suitable for exchanging data in general and even more document-like data can be represented concisely in the ATerm format. In this chapter the ATerm format is described and some of the problems of the ATerm format are discussed.

The general idea behind the ATerm format is comparable to the ideas of the XML language. Although the languages are influenced by different design decisions, they are both designed for the exchange of data between programs. The languages will be compared to each other in the last section.

4.2 ATerm Format

The ATerm format is based on ground terms over ranked alphabets. The symbols of an alphabet are usually called function symbols in the ATerm format. A function symbol has a rank, which must correspond to the number of children in an application of the function symbol. The ATerm format defines several more specific constructs to make the idea of ground terms applicable in practice. This complicates matters because more constructs have

[1] Centrum voor Wiskunde en Informatica, National Research Institute for Mathematics and Computer Science in the Netherlands. Many tools we are using are maintained at the CWI: ATerm format, ATerm library, SDF2 , and SGLR

to be handled by all tools and definitions over ATerms, yet having a plain term language, for example without lists, characters, strings, and integers, is not acceptable in practice.

In [dBdJKO00] the ATerm data type is defined. The data type consists of the following constructs:

Int	32-bits integer
Float	64-bits IEEE floating point number
List	list of zero or more ATerms
Blob	list of zero or more bytes
Application	function symbol and a list of zero or more ATerms

Also every ATerm can have a list of ATerm (label, annotation) tuples associated with it. A tuple is an application of a special (empty) function symbol. A string is an application of a double quoted function symbol.

There is not just one encoding of the ATerm format. Currently there are three different formats: unshared textual format (text), textual ATerm format (taf), and binary ATerm format (baf). The taf and baf formats preserve the maximal sharing of an ATerm, which is used to reduce memory usage in the ATerm libraries. All ATerm examples in this thesis use the unshared textual format.

4.3 Examples

The ATerm format is best illustrated by some concrete examples.

- `13`
  is an integer term.
- `"foo"`
  is a string term.
- `name("Faramir")`
  is an application of the function symbol 'name' to the string term `"Faramir"`.
- `address(street("45␣Usura␣Place"),city("Hailey"),state("ID"))`
  is an application of the function symbol address to three ATerm applications.
- `[Scalar(Int(1)),Scalar(Int(2))]`
  is a list of two ATerm applications.
- `Call(Var("f"),[Int(1),Int(2)])`
  is an application of the function symbol 'Call' to an application term and a list term.

4.4 ATerms Defined

Unfortunately the ATerm data type has some properties that make it unattractive for formal reasoning. Therefore we define it in a more abstract way, ignoring practical details like the size of an integer and the allowed characters. Also we represent an ATerm in its *exploded* form, that is, a representation of the ATerm in an ATerm. This representation of an ATerm will be used in all ATerm related definitions, but of course real ATerms examples will use the imploded ATerm.

Definition 20: Annotated Term
The set of ATerms over a ranked alphabet $\mathcal{F}$, an unranked alphabet $\mathcal{A}$ of annotation labels and an unranked alphabet of characters $\mathcal{C}$, denoted by $ATerm(\mathcal{F}, \mathcal{A}, \mathcal{C})$, is defined as:

integer term

$$\frac{x \in \mathbb{Z}}{\texttt{int}(x) \in Term(\mathcal{F}, \mathcal{A}, \mathcal{C})}$$

real term

$$\frac{x \in \mathbb{R}}{\texttt{real}(x) \in Term(\mathcal{F}, \mathcal{A}, \mathcal{C})}$$

character string term

$$\frac{c_0, c_1, \ldots, c_n \in \mathcal{C}}{\texttt{string}([c_0, c_1, \ldots, c_n]) \in Term(\mathcal{F}, \mathcal{A}, \mathcal{C})}$$

application term

$$\frac{f/n \in \mathcal{F} \qquad t_0, t_1, \ldots, t_n \in ATerm(\mathcal{F}, \mathcal{A}, \mathcal{C})}{\texttt{appl}(\texttt{fun}(f),\ [t_0, t_1, \ldots, t_n]) \in Term(\mathcal{F}, \mathcal{C})}$$

list term

$$\frac{t_0, t_1, \ldots, t_n \in ATerm(\mathcal{F}, \mathcal{A}, \mathcal{C})}{\texttt{list}([t_0, t_1, \ldots, t_n]) \in Term(\mathcal{F}, \mathcal{A}, \mathcal{C})}$$

tuple term

$$\frac{t_0, t_1, \ldots, t_n \in ATerm(\mathcal{F}, \mathcal{A}, \mathcal{C})}{\texttt{tuple}([t_0, t_1, \ldots, t_n]) \in Term(\mathcal{F}, \mathcal{A}, \mathcal{C})}$$

none term

$$\frac{}{\texttt{none}() \in Term(\mathcal{F}, \mathcal{A}, \mathcal{C})}$$

some term

$$\frac{t \in ATerm(\mathcal{F}, \mathcal{C})}{\texttt{some}(t) \in Term(\mathcal{F}, \mathcal{A}, \mathcal{C})}$$

annotation

$$\frac{t \in ATerm(\mathcal{F}, \mathcal{A}, \mathcal{C}) \quad \wedge \quad l \in \mathcal{A}}{\texttt{anno}(l,\ t) \in Anno(\mathcal{F}, \mathcal{A}, \mathcal{C})}$$

annotated term

$$\frac{a_0, a_1, \ldots, a_n \in Anno(\mathcal{F}, \mathcal{A}, \mathcal{C}) \quad t \in Term(\mathcal{F}, \mathcal{A}, \mathcal{C})}{\texttt{annotated}(t,\, [a_0, a_1, \ldots, a_n]) \in ATerm(\mathcal{F}, \mathcal{A}, \mathcal{C})}$$

not annotated term

$$\frac{t \in Term(\mathcal{F}, \mathcal{A}, \mathcal{C})}{t \in ATerm(\mathcal{F}, \mathcal{A}, \mathcal{C})}$$

□

This definition ignores real life facts and deficiencies in the ATerm format. As we already mentioned integers are limited to 32 bits and reals are in fact 8 byte IEEE floats. Tuples are in the ATerm format represented by application to empty function symbols and are not a separate construct. String terms are in the ATerm format represented by applications of quoted function symbols to zero term arguments. $\texttt{some}(t)$ and $\texttt{none}()$ have no special construct in the ATerm format. They are represented by `None()` and `Some(t)` applications terms. These applications will also be used in concrete ATerm examples and program code.

A major limitation of the ATerm format is that characters in function symbols, and thus also in what we call string terms, are limited to the Extended ASCII Character Set. Because of this the ATerm format is no match for XML if an application domain requires internationalization capabilities. This problem will discussed in more detail in the future work section of the next chapter.

Part II

Data Exchange

Chapter 5

Modular and Reusable XML Parsing

5.1 Introduction

The W3C XML recommendation [BPSM00] defines the syntax of well-formed XML documents [1]. The W3C XML recommendation recommends no level of abstraction at which the XML document is to be processed. There is no description of how XML documents should be regarded beyond its well-formed XML syntax. There exist higher level abstractions of XML documents. Examples include the XML Information Set, the Post Schema Validation Information Set and the RELAX NG data model. These abstractions are often called a data model for XML (see section 3.3). The XML syntax has been designed to be exchangeable and interoperable. Data models are designed to process the XML document in an application.

5.1.1 Need for Different Representations

The XML recommendation specifies exactly what characters and constructs XML processors should pass to an application. Although such a specification is useful in general, applications have specific needs [Lau03]. An application must be able to control what representation it wants of an XML document.

Most applications processing an XML based language reduce the XML syntax to a very simple model: a tree of XML content. XML elements are the nodes of this tree. Nodes have a name, a list of child nodes and a set of attributes. The child nodes are text nodes, or elements. Attributes have a name and a single text value. Most data models are variations on this theme, usually added with some more context information of the original document. The RELAX NG data model is based on such an abstraction and provides a context, which includes namespace prefixes, default namespaces and a base URI. These abstractions and applications that manipulate XML at this level of abstraction ignore the syntactical details of an XML document. Indeed most XML processing applications are not interested in the syntactical details of an XML document and just want a high level representation of the data in the XML document. For these applications parsing XML is sometimes [SW03] regarded unnecessarily difficult for a data exchange format.

[1] Contrary to the way the term well-formed is often used, there is no XML document that is not well-formed because these textual files are not XML documents

Some applications, notably meta XML applications, care about the syntactical details in an XML document. They require a more detailed representation of the actual syntax used in an XML document. Allowing layout preserving or generating transformations is a common example of the needs of these applications. Different XML processing applications require different representations of an XML document. Some want a syntactical representation, some want a representation of an XML document without syntactical details, some want a representation of the data in an XML document that feels natural in a certain programming language, some applications even want a typed representation of the XML document.

Most XML processors can be configured to some extent. For example SAX2 [BM] defines a set of features and properties that can be set to change the behaviour of the parser. XML readers are not required to support these features, except for two XML Namespaces related ones. Setting such properties is currently the typical way of configuring an XML processor.

5.1.2 Need for Modularization

What is currently considered to be an XML processor has two main purposes:

1. Process an XML document into a certain representation.
2. Offer this representation in some natural way to the application.

The second purpose of an XML parser is a highly implementation language specific task. The first purpose however is not at all language specific. A tool kit for the processing of XML is required to be flexible because there is need for different representations of an XML document. To meet the specific needs of an application, applications must be able to compose their own XML processing pipeline from modular, reusable, and well-defined XML processing components in such a tool kit. In this way the application can control itself what kind of XML representation it gets. This component-based approach to XML processing is inspired by the proposal for a layered model for XML processing by Simon St. Laurent [Lau99]. In current XML processors the control on the processing of XML is limited to some configuration interface, for example the features and properties of SAX2.

A tool kit for the processing of XML to a certain representation is complex because XML is not a trivial language. The interoperability approach of XML has its cost. Developing the whole bunch of code required for XML processing takes quite some amount of code and development time. A tool kit for XML processing has to handle different encodings, resolve entities, resolve namespaces, apply whitespace rules, and implement validation against DTD schemas and maybe even other XML schema languages like W3C XML Schema and RELAX NG.

The *complexity* and required *flexibility* of the first task of an XML processor demands for modularization. Implementing XML processing in a single piece of software breaks the rule of modularity [Ray03]. Controlling complexity is the essence of computer programming [KP76]. Complexity is to be controlled by modularity: do just one thing, do it well. For XML processing this means separating monolithic XML processors into a tool kit of XML processing components. Tool kit is to be taken very literally as a set of tools that can be executed independent of each other. Predefined XML processing compositions can meet the requirements for being a well-formed or validating XML processor as defined in the XML recommendation or provide different useful processing pipelines. Obviously modularity

is useful for validation against schemas defined different XML schema languages, such as RELAX NG, but data binding and XML processing itself require modularization as well.

5.1.3 Need for Reuse

Many languages and platforms have their own XML parser completely implemented specifically for this language or platform. In the best case implementations in C or C++ are reused, like Expat in Python and the GNOME XML library (libxml) in PHP. In both cases the influence on the representation of the XML document is limited. This leads to many different implementations and code duplication. Most existing XML processors are rather monolithic and thus their components cannot be reused in different settings. They never allow the exchange of data between components of the XML processor, which is essential for making the components of an XML processor reusable. Because processing XML to a certain representation is not language dependent, the tool kit required for this can be reused across all languages and platforms. XML processor components must be reusable beyond the boundaries of a programming language. This prevents the reimplementation of the full XML processing tool kit for each language or platform.

5.1.4 Structure

In this chapter we discuss the basic tools in our tool kit for XML processing. We do not cover more domain specific tools and representations of an XML document. Chapter 6 discusses the tools in our tool kit that are more related to XML data binding [Bou], where a natural representation in native data structures of a certain programming language is to be found for an XML document.

In the next section we discuss the overall architecture of the basic XML processing tools in our tool kit. In section 5.3 the xml-doc representation of an XML document and the tools related to this representation are presented. In section 5.4 the xml-info language is presented, which is a more abstract representation of an XML document. Finally we discuss some related and future work.

5.2 Architecture

The tools in the XML processing tool kit must be hooked together in some way. Hooking tools together requires two things. First tools must be able to communicate. Second they must be actually hooked together by defining a composition of tools. For communication between the tools in our tool kit we are using the ATerm format [dBdJKO00]. To allow the tools to be composed into pipeline we follow the Unix principle of pipes and filters.

5.2.1 Compositionality

We have thus implemented the tools in our tool kit as independent executable programs, reading input data, performing an operation on the data, and producing output data. These tools accept data on their standard input and produce data on their standard output. Additional input and configuration is provided by using command line arguments. The tools are to be composed into a pipeline by passing the output of a tool to the standard input of the next tool. This architecture is based on the successful pipe and filters philosophy

of connecting programs in Unix [MET78, Rit84]. By following this idea, compositions can be defined in shell scripts, at the console, and in almost any general purpose programming language.

5.2.2 Exchange of Structured Representations

For the exchange of data between the programs we are using the ATerm format [dBdJKO00]. The ATerm format has a textual and binary encoding. The first reason for choosing the ATerm format is related to the original goal of the project. The project started with the question how the Stratego transformation language could be applied for the transformation of XML. The Stratego transformation language transforms structured data in the ATerm format. We want to implement the tools required to support XML in Stratego itself, becauseStratego has some interesting features for implementing XML transformations, such as generic traversals and scoped-dynamic rewrite rules. In a sense XML processing itself is all about generic XML transformations, so implementing the tool kit in Stratego is a useful experience. Second, and this probably a better reason, the ATerm format is a more concise format. The ATerm format separates character data by double quotes, has a list construct, and allows attributes to be structured. Especially the quotation of character data is useful for document representations. Third, the XML language is designed for interoperability of software tools on different machines, on different platforms, in different cultures. The tools in our tool kit are *local*: they are all applied at the same machine. This local machine and the tools in our tool kit are under our control. The interoperability facilities of XML are less useful in this setting. The data these tools exchange should *not* be exchanged for other purposes then XML processing. This means that they should in general never leave the machine on which the XML document was first parsed to a representation in the ATerm format.

We will not hide the fact that there is one enormous disadvantage of using the ATerm format for representing XML documents. Characters in the ATerm format are namely restricted to the ASCII range [2]. This disadvantage reduces the general applicability of our tool kit in XML processing to nothing more then a proof of concept of the design of an XML processing tool kit. By representing characters of an XML document as ATerm characters we are throwing away all the the localization and internalization features of XML . Thus, in our tool kit all characters in an XML document must stay in the ASCII range or must be escaped. We hope that we can in the near future persuade the ATerm developers that the ATerm format should be improved by using Unicode code points in an encoding like UTF-16. We could have worked around this issue by using lists and integers to represent character strings. We have decided not to do this for performance and customary related reasons.

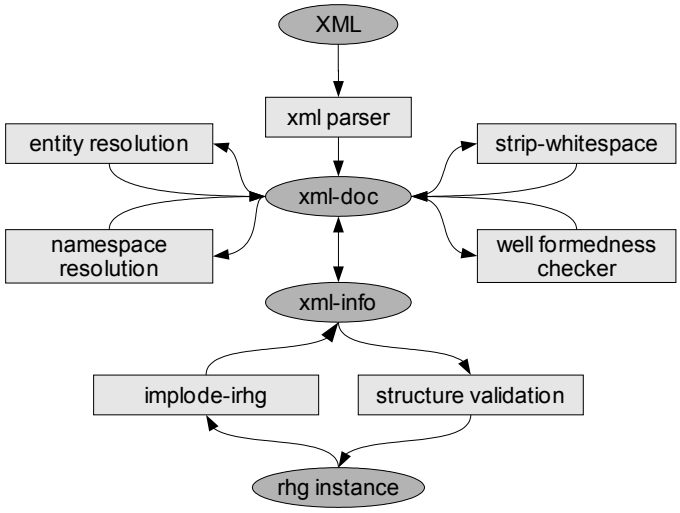
5.2.3 Different Representations

The tools exchange representations of XML documents in the ATerm format. In this chapter we introduce two representations: xml-doc and xml-info [3]. As noted before these represen-

[2] See http://www.joelonsoftware.com/articles/Unicode.html for an interesting read on the importance of supporting Unicode.

[3] We have carefully considered the use of the name XML in the ATerm representations we will discuss in this chapter. Earlier versions even used 'not-xml', referring to all the fuss about being XML or not. Yet, choosing different names would be rather confusing so we decided to use the name XML anyway.

Figure 5.1: Visualization of the tools and XML representations discussed in this chapter. The ellipses are the representations of an XML document, the rectangles are the tools that can be applied to the representations they are connected with.



tations are *not* to be exchanged beyond the scope of XML processing itself.

The xml-doc representation is used if a detailed description of the original document is required. An xml-doc term almost completely describes the actual syntax used in an XML document. This representation for example contains information on the use of double quotes or single quotes for attribute values, all whitespace in elements, all unresolved entities, and the actual XML namespace notation that is used.

The xml-info representation is a more abstract representation of an XML document. In the xml-info representation many constructs are desugared to a more canonical form. In the xml-info representation there are no entities, no namespace attributes and all names of attributes and elements contain their full namespace URI.

The tools in our toolkit take a representation of an XML document as an input and produce a representation of an XML document. These representations are not necessarily the same. Conversion tools from one representation to the other can be used if a tool requires a different representation then currently is available. Figure 5.1 shows the XML representations and some of the tools that operate on these representations.

5.3 Representation in xml-doc

The xml-doc language is an almost complete representation of the actual syntax used in an XML document. This representation of an XML document is suitable for applications that care about the actual syntactical constructs used in an XML document. Textually different XML documents almost never result in the same xml-doc term. In more abstract data models, like data models based on the XML infoset, this is the case because some constructs are considered to be equivalent.

Example 5: Character data in xml-doc
All character-like data of XML is represented in a Text term. The Text term has one child, which is a list of character-like constructs. CharData is in a Literal term.

```
Asterix and Obelix
```

```
Text([Literal("Asterix␣and␣Obelix")])
```

Character references are a special construct in the list of character-like data.

```
Asterix &#x26; Obelix
```

```
Text([Literal("Asterix␣"), HexCharRef("26"), Literal("␣Obelix")])
```

Entity references are considered to be character-like data. This is the usual application of entities, yet this is somewhat confusing because it is possible that an entity refers to more then just characters.

```
Asterix &amp; Obelix
```

```
Text([Literal("Asterix␣"), EntityRef("amp"), Literal("␣Obelix")])
```

CDATA sections are also character like data.

```
Asterix <![CDATA[&]]> Obelix
```

```
Text([Literal("Asterix␣"), CDATASection("&"), Literal("␣Obelix")])
```

Example 6: Attributes in xml-doc
The kind of quote character used for an attribute value is also represented in xml-doc. The representations for `foo="bar"` and `foo='bar'` are for example:

```
Attribute(QName(None, "foo"), DoubleQuoted([Literal("bar")]))
```

```
Attribute(QName(None, "foo"), SingleQuoted([Literal("bar")]))
```

The representation of character-like data in attribute values is comparable to the representation in elements.

```
title="Eowyn␣&amp;␣Eomer"
```

```
Attribute(
  QName(None, "title")
, DoubleQuoted([Literal("Eowyn␣"), EntityRef("amp"), Literal("␣Eomer")])
)
```

The xml-doc representation is not a complete representation of the original XML document. Whitespace inside element and attribute tags and outside the document element is not represented in xml-doc. Tools that parse an XML document to an xml-doc term must however preserve all whitespace in XML content. In an ATerm this whitespace is contained

in a quoted string term. Concerning XML Namespaces xml-doc knows only about the existence of namespace prefixes. The prefix is separated from the local part in a QName construct.

Example 7: Elements in xml-doc
The following XML content illustrates how XML elements are represented in xml-doc. Note that there is a separate EmptyElement construct and that whitespace is preserved.

```
<foo><barney:fred/><bar>  </bar></foo>

Element(
  QName(None, "foo")
, []
, [ EmptyElement(QName(Some(Prefix("barney")), "fred"), [])
  , Element(
      QName(None, "bar")
    , []
    , [Text([Literal("␣␣")])]
    , QName(None, "bar")
    )
  ]
, QName(None, "foo")
)
```

The xml-doc language is defined in an abstract syntax definition, but it is not included in this thesis because the definition is rather large. It is available in the xml-tools package [Braf]. The abstract syntax is defined in the RTG language, which we will introduce in the next chapter. The XML language language itself can be considered to be a concrete syntax for the xml-doc language. The abstract syntax definition for the xml-doc has in fact even been generated from a concrete syntax definition for XML in SDF2 . This SDF2 concrete syntax definition for XML is annotated with the terminal names of the productions of xml-doc. This annotated syntax definition is transformed into the RTG abstract syntax definition of xml-doc using tools that will be introduced in the next chapter as well (see chapter 6 on page 49).

An xml-doc term does not always represent a (well formed) XML document. The most obvious problem is the separate construct for start and end tags. These tags are allowed to be different because their name is not restricted by the xml-doc language. Also the XML document that would be represented by an xml-doc term does not always conform to the so called 'namespace constraints' defined in the XML namespaces recommendation [BHL99]. The namespace constraint *prefix declared* is for example not enforced by the xml-doc language.

5.3.1 Tool Support

Although the xml-doc representation has been discussed first, the tools that operate on xml-doc are the heart of the matter. The xml-doc representation of an XML document is to be exchanged between XML processing tools. These tools implement much smaller operations than the XML tools that are already used in practice, like XML well-formedness checking, validation, XInclude processing, applying XSLT trasformations and so on. The

tools are really about processing an XML document to an appropiate representation. It is up to the application to decide what is an appropiate representation. Next we list the tools related to xml-doc in the xml-tools package [Braf].

parse-xml-doc
: Parses an XML document to an xml-doc term. The implementation is based on *sglr* [Vis97a]. A parse table for *sglr* is generated from the SDF2 syntax definition for xml by the *sdf2table* parse table generator in the pgen package [vdB03].

  Different, optimized, implementations are possible as soon as applications using *parse-xml-doc* experience performance problems because of the use of scannerless generalized-LR parsing in this tool. Especially we are interested in using Gorille Ripper [Lau, Lau03] to parse XML documents to xml-doc. Using standard XML parser interfaces like the W3C DOM or SAX2 is not possible because they do not provide the required level of syntactical detail to the application.

pp-xml-doc
: Pretty prints an xml-doc term to an XML document. The tool *pp-xml-doc* does not add any whitespace to the XML document, except of course in the element tags themselves for separation of element name and attributes. The result is thus 'pretty' if and only if the xml-doc term contains the whitespace that is required to make the result pretty.

xml-doc-add-layout
: Adds whitespace to an xml-doc term in such a way that it will be pretty printed in an attractive way if pretty printed by *pp-xml-doc*.

xml-doc-strip-whitespace
: Removes whitespace in an xml-doc term without schema knowledge. All literals that just contain whitespace characters are removed. This strategy, although simple, is quite useful in practice. Schema aware removal of whitespace has not been implemented yet, but it can easily be added as yet another component that transforms xml-doc to xml-doc.

xml-doc-resolve-entities
: Performs entity resolution and rewrites character references to the characters they refer to.

xml-doc-well-formed
: Checks whether an xml-doc term corresponds to a well formed XML document. This tool does not rewrite the xml-doc term. It is thus an identity transformation if it succeeds.

5.4 Representation in xml-info

The xml-doc representation of an XML document is very close to the actual document. Thus their are many syntactical details in this representation. More abstract data models like the XML Infoset and the RELAX NG data model abstract from the more syntactical constructs (see section 3.3 on page 25). If these constructs are irrelevant details to a certain XML processing application, then the application should use a more abstract representation of the document. The xml-info representation is a language, defined in an abstract syntax, for such an abstract representation. By providing such an abstraction, the xml-info representation is based on a certain idea of what is relevant in an XML document for a

```
regular hedge grammar
  start Document

  productions
    Document  -> Document (Element)

    Element   -> Element (Name Attribute* Content*)
    Content   -> Element
    Content   -> Text (<string>)

    Attribute -> Attribute (Name <string>)

    Name      -> Name (Namespace? <string>)
    Namespace -> Namespace (<string>)
```

Figure 5.2: Definition of the abstract syntax of xml-info in RHG

certain range of applications. It is not *the* definition of wat is the information provided by an XML document.

5.4.1 Language

The xml-info language is much smaller than xml-doc language. It has just a few constructs: Element, Text, Attribute, Name, Namespace and Document. An Element has a Name, a list of Attributes, and a list of Content. A Name has an optional Namespace and a local name, which is a string. An Attribute has a Name and a String value. Content is Text or an Element, where Text just wraps a String. Note that the xml-info does not contain constructs for processing instructions and comments. The abstract syntax of xml-info is defined by the RHG in figure 5.2. The RHG language will be introduced in the next chapter, but the definition will be clear without knowing the exact details of the RHG language.

Example 8: Character data in xml-info
This example shows how the character data in example 5 on page 40 is represented in xml-info.

```
Asterix and Obelix
```

```
Text("Asterix␣and␣Obelix")
```

The character data examples

```
Asterix &#x26; Obelix
```

```
Asterix &amp; Obelix
```

```
Asterix <![CDATA[&]]> Obelix
```

are all represented by the same xml-info term

```
Text("Asterix␣&␣Obelix")
```

Example 9: Attributes in xml-info
The kind of quote character used for an attribute value is no longer represented in xml-info (see example 6 on page 40 for the representations in xml-doc). The xml-info representation of `foo="bar"` and `foo='bar'` is thus:

```
Attribute(Name(None,"foo"),"bar")
```

Entities and character references are no longer available in the value of an attribute. The value is now just a string.

```
title="Eowyn &amp; Eomer"

Attribute(Name(None,"title"),"Eowyn & Eomer")
```

Example 10: Elements in xml-info
Elements in xml-info no longer contain the names used in both tags. All element names contain their full namespace URI and there is no distinction between the compact notation for an element and a normal element without children.

```
<foo xmlns:barney="fred">
  <barney:bar/>
  <barney:bar></barney:bar>
</foo>

Element(
  Name(None,"foo")
, []
, [ Element(Name(Some("fred"),"bar"),[],[])
  , Element(Name(Some("fred"),"bar"),[],[])
  ]
)
```

5.4.2 Level of Abstraction

It is difficult to choose the right level of abstraction an application should use. The xml-info representation is more abstract then most abstract data models for XML. It contains less information about the original XML document and if this information is relevant to an application then using xml-info will cause trouble. The RELAX NG data model [CM01, Clab] (see section 3.3) contains for example more information on the context of an element. The xml-info representation is in fact equivalent to this data model minus the context of an element. In the RELAX NG data model the context of an element consists of a base URI [Mar01] (used to resolve relative URI references) and a namespace map of a default namespace and the declared namespace prefixes. This information is missing in the xml-info language and there is no way to retrieve it from the information in the term itself.

In composing pipelines of XML processing tools, the programmer should be careful at deciding when to drop what information. Also XML processing tools must be careful at deciding on what representation they should operate. If a tool T operates on an xml-info

representation, then available information will be dropped at this point in the pipeline. This might limit the possible pipelines that can be composed, because tools that operate on for example the RELAX NG data model cannot be applied in the pipeline after this tool T. In some circumstances 'fake' data might help out. In fact the conversion from xml-info to xml-doc introduces such fake data.

5.4.3 Tool Support

The xml-info language definition in the xml-tools package comes with some tools to convert XML and xml-doc to xml-info. There are however no other tools that actually operate on xml-info itself. In applications of xml-tools the conversion to xml-info is typically done at the end of a pipeline of XML processing tools. Just before domain specific tools the more detailed representation of an XML document is converted to xml-info.

xml-doc2xml-info
: Transforms xml-doc to xml-info. Entities and character references should already be removed. The main issue left is the resolution of namespaces. The implementation uses the scoped dynamic rules of Stratego [Vis01a] to resolve namespace prefixes to the full URI.

xml-info2xml-doc
: Transforms xml-info to xml-doc. The translation scheme for XML namespaces in xml-info to xml-doc is currently very verbose: every element will have a default xmlns attribute.

parse-xml-info
: Composes parse-xml-doc and xml-doc2xml-info into one tool that directly processes an XML document to xml-info.

parse-xml-info –parser jaxp
: Processes an XML document to xml-info. When using this flag the aterm-xml bridge is used, which parses an XML document and and outputs a representation of this document in the xml-info language. The aterm-xml bridge is implemented in Java, using the default XML parser of the Java 2 runtime environment. It uses the ATerm library for Java and the SAX interface to the XML parser.

xml-info2data
: This tool will be discussed in section 6.1.2 on page 52.

xml-info2irhg
: This tool will be discussed in section 6.2.6 on page 72.

5.5 Related Work

5.5.1 XML representation

XSet [Bor00] is comparable to the xml-doc language. XSet explodes an XML document to a detailed representation of the document in XML . Such a detailed representation is also called a 'full fidelity information set representation' in this context. As far as we know there is no implementation available that parses an XML document to an XSet document. An

example published on the web [4] illustrates the XSet language. The xml-doc representation has essentially the same purpose. The main difference is that XSet is closer to the actual productions used in in the W3C XML recommendation and that the XSet representation is in XML itself instead of ATerm . This is not fundamental difference as we will discuss in the next chapter.

Gorille Ripper [Lau, Lau03], which was already mentioned in the description of the parse-xml-doc tool, and the proposal for a layered model for XML [Lau99] are based on the same ideas as modular tool kit for XML processing. Gorille Ripper does not result in some structured output, but expects the implementation of an interface (in Java). The implementation will be notified of every single character, which is a more detailed representation than xml-doc. Whitespace inside tags is for example reported by Gorille Ripper. The Gorille Ripper representation is at this point also more detailed than the XSet representation.

ATerm-XML [Hen02] aims at the conversion from XML to the ATerm format and vice versa. The approach of the ATerm-XML tools is more related to the topics discussed in the next chapter, so we postpone the comparison to this work until the next chapter (see section 6.7.2).

5.6 Conclusion and Future Work

Having different representations for an XML document is the solution to the everlasting discussion on the appropriate representation. The discussion can then be moved to a more useful location: what representation should a specific application work on. Having separate tools for processing XML documents to an appropriate representation clears much of the mystique of XML processing. The tools are reusable by design and the modular design of the XML processing tool kit allows easy extension of the tool kit with tools offering new features or alternative implementations of already available XML processing tools. Applications control by themselves the representation of the XML document by composing the required tools into an XML processing pipelines. A lot of work is still to be done however to make this tool kit usable in practice.

5.6.1 More Representations, More Tools

The XML processing tool kit is currently still quite limited. Only the tools that are needed for the topics discussed in the next two chapters have been implemented and the tool kit is thus far from complete. Some essential tools are missing to be able to compose a validating XML processor as defined in the W3C XML recommendation from our XML processing tools. For example a tool that distinguishes whitespace in element content whitespace [5] and whitespace that might constitute character data must be implemented. This tool must also handle the `xml:space` attribute.

Another schema aware tool that must be implemented is the addition of default attributes. If a default attribute is declared in a DTD then a validating XML processor must report

[4] The XSet representation of http://www.openhealth.org/XSet/ericvdvexample.xml is: http://www.openhealth.org/XSet/ericvdvxset.xml

[5] An XML element type is said to consist of element content when only elements are allowed as children. The elements are allowed to be separated by whitespace.

the default attribute as if it is present in the document if its omitted in the XML document. This is an excellent task for an independent XML processing tool. It can be applied in an XML processing pipeline if an application needs this feature. In a default validating XML processing pipeline it must be applied to satisfy the requirements stated in the XML recommendation.

The xml-info2xml-doc currently uses a very verbose way to define the namespaces of element and attributes names. Although this translation is quite entertaining, a more compact translation scheme is desirable. Maybe this implementation should involve a specification of the preferred namespace prefixes for a certain namespace URI.

Also we would like to extend our set of representations in the future. Especially we are looking forward to creating a representation for the full XML Information Set, and the RELAX NG Data Model. These representation are in level of abstraction somewhere between the xml-doc and xml-info representations. The Post Schema Validation Information Set (PSVI) representation is a different issue because validation and typing is involved. In the XML processing tool kit there are still no representations of XML documents that include type information or tools that apply schemas. This is the subject of the next chapter.

5.6.2 ATerm, SDF2, and SGLR: The Importance of Supporting Unicode

The limitation of ATerm characters to the ASCII character set is a big objection to the the practical application of our tools. The ATerm format should be extended to a larger character set in the future, preferably Unicode codepoints in some encoding. If this is not going to happen we must consider to represent characters as lists of Unicode codepoints.

The SDF2 syntax definition formalism should be extended to a larger character set as well. Characters in an SDF2 syntax definition must not refer to a certain encoding of a character in ASCII or a larger character set. Characters must refer to Unicode codepoints to abstract from a specific encoding. Tools that apply the SDF2 syntax definition in some way must handle different encodings by doing a semantic comparison in terms of the Unicode codepoints of characters, not the way the characters are encoded. The tools that are directly influenced by this are PGEN, the parse table generator, and SGLR , the parser. In this new setting SGLR must accept an encoding argument, or the file must be converted to a certain agreed upon encoding before passing it to SGLR .

Again, we must consider workarounds if this is not going to happen. The parse-xml-doc can be implemented in a different language without much trouble. Still the SDF2 syntax definition is an essential component of the tools discussed in this thesis because in the next chapters it will be used to embed XML in programming languages. In this case we must consider to escape all characters that are not in the ASCII character set.

Chapter 6

Tree Grammar Tools for Data Exchange

XML is a language for exchanging data between software tools. In this market of data exchange formats XML is not the only competitor [HK00]. The XML language however has some properties that make it applicable for data exchange in heterogeneous networks, especially over the Internet. The software components that exchange data in XML might reside on different computers, different operating systems, different platforms and different locations. XML has been designed to cope with this diversity of environment.

The application environment of rewriting systems, such as the ASF+SDF Meta Environment [dBHdJ+01] and Stratego [Vis01b, VBT98, www], has been more homogenous until now. Both systems are usually applied for the implementation of program transformations, such as optimizers, compilers, interpreters, design improvers, and visualizers. In both systems the ATerm [dBdJKO00] format is used for the representation of programs in the system itself and for the exchange of representations between transformation components.

Why not just use XML? The ATerm format has some disadvantages and some advantages compared to XML. The most important advantage of the ATerm format is the more explicit structure of terms. This structure is provided by some built-in constructs of the ATerm format. The ATerm format has, besides function applications (elements), constructs for lists, tuples, integers, strings (character data), and structured annotations (attributes). The most important disadvantage of the ATerm format is its poor internationalization support. Characters in ATerm strings and function symbols are limited to the ASCII range. This limits the possible characters to un-accented English letters, which is getting problematic now recent programming languages such as Java and C# allow characters in the Unicode character set. In general that is a problem when the ATerm format is used for exchange of data in more heterogenous environments like the Internet. If we postpone this serious limitation to future work then XML and ATerm are not that different.

Many interesting ATerm and XML tools have been developed.This raises the question how we can make these tools work together in such a way that without much trouble an ATerm tool can exchange data with an XML tool and vice versa. An interesting side effect of such a facility is that existing programming language research and implementations for the manipulation of ATerms can be used for the manipulation of XML documents as well. Although some languages for manipulating XML have already been developed, this might be an interesting contribution to the area of XML processing.

If we include this last goal then there are two challenges:

1. XML and ATerm are different exchange formats. In particular XML documents have a less explicit structure.
2. XML and ATerm are used for different types of data. ATerms are usually program representations. XML is used for documents and data.

In the last chapter an XML processing tool kit has been presented. The tools of this tool kit work together by exchanging representations of the XML document in the ATerm format. Thus we have already implemented some conversion from XML to the ATerm format. Processing XML documents in this way is however not a solution to the interoperability problem. XML tools and ATerm tools must be connected without being designed to handle the other data exchange format. Thus the data of an XML document should be converted to a natural representation in an ATerm and vice versa.

The xml-doc and xml-info representations in the ATerm format are natural representations of the *XML document*, but not of the *data* in an XML document. Both representations are too specific to the representation of the data in XML. The ATerm format is a data exchange format as well, so these representations in fact represent an exchange format in an exchange format.

In this chapter we discuss how we have tackled the first challenge by creating reusable tools for the conversion of XML to ATerms and vice versa. The goal of these tools is to add an explicit structure to an XML document. This structure is added by applying the implicit structural information of a schema to the instance document. The tools are based on the formal language theory of regular tree languages and regular hedge languages.

6.1 An Introduction to Structure

6.1.1 Running Examples

Two running examples with different kinds of data will illustrate the effect of applying the tools discussed in this chapter. The first running example is a typical XML document. The second example is more typical for program transformation systems.

Example 11: Running Document and Data Example
Our first example has been taken from an article at xml.com that compares XML data binding solutions for Python [Ogb03]. The XML document

```
<?xml version="1.0" encoding="iso-8859-1"?>
<labels>
  <label>
    <quote>
      <!-- Mixed content -->
      <emph>Midwinter Spring</emph> is its own season&#33;
    </quote>
    <name>Thomas Eliot</name>
    <address>
      <street>3 Prufrock Lane</street>
      <city>Stamford</city>
      <state>CT</state>
    </address>
  </label>
  <label>
    <name>Ezra Pound</name>
    <address>
      <street>45 Usura Place</street>
      <city>Hailey</city>
      <state>ID</state>
    </address>
  </label>
</labels>
```

is almost exactly the document of that article. Only the character reference has, for reasons mentioned in the introduction, been replaced by a character in the (non-extended) ASCII range. Note that our tools will thus not pass this test for proper character support. This XML document demonstrates some different flavors of XML : data flavor with records like constructs, and document flavor with mixed content [a].

[a] An element type has mixed content when elements of that type may contain character data, optionally interspersed with child elements [BPSM00].

Example 12: Running Program Example
Our second running example is a typical piece of data exchanged in Stratego/XT program transformation systems. This example will show that program representations are not that different from the data that is usually exchanged in XML. Also it shows how Stratego/XT program transformation systems will be able to connect to the outside world and vice versa using our tools.

The example program is an almost trivial Java class, yet there are some interesting problems involved, which will be discussed later.

```
public class HelloWorld {
  void hello() { }
}
```

An XML representation of this Java program looks like

```
<?xml version="1.0" ?>

<CompilationUnit>
  <ClassDec>
    <Public/>
    <Id>HelloWorld</Id>
    <ClassBody>
      <MethodDec>
        <Head>
          <Void/>
          <Id>hello</Id>
        </Head>
        <Block>
        </Block>
      </MethodDec>
    </ClassBody>
  </ClassDec>
</CompilationUnit>
```

when it is parsed using the syntax definition for Java in the java-front package [Brab].

6.1.2 Unnatural XML ⇔ ATerm

Ignoring all ATerm customs, XML documents can be translated directly into an ATerm. This translation is defined in the following definition. The definition uses the exploded representation of an ATerm (see section 4.4) and the xml-info representation of an XML document (see section 5.4).

Definition 21: Unnatural XML ⇔ ATerm
The following set of equation defines an unnatural mapping from XML to ATerm and vice

```
module xml-info2data
imports xml-info options simple-traversal string

strategies

  main-xml-info2data =
    iowrap( xml-info2data )

  xml-info2data =
      ?Document(<id>)
    ; topdown-wannos( try(Implode) )

rules

  Implode:
    Element(Name(_, s), [atts*], children) -> s#(children){atts*}

  Implode:
    Text(s) -> s

  Implode:
    Attribute(Name(_, s), value) -> (s, value)

strategies

  topdown-wannos(s) =
    rec x(
      topdown(s; id{map(x)} )
    )
```

Figure 6.1: Implementation of the unnatural mapping from XML to ATerm in Stratego. Note the clear relation with the equations of definition 21 on the preceding page

versa.

$$\begin{aligned}
\texttt{Element}(n, a*, c*) &\Leftrightarrow \texttt{annotated}(\texttt{appl}(n, c*), a*) \\
\texttt{Attribute}(n, s) &\Leftrightarrow \texttt{anno}(n, \texttt{string}(s)) \\
\texttt{Text}(s) &\Leftrightarrow \texttt{string}(s) \\
\texttt{QName}(\texttt{Some}(\texttt{Namespace}(s_1)), s_2) &\Leftrightarrow \texttt{fun}(\{s_1\}s_2) \\
\texttt{QName}(\texttt{none}(), s) &\Leftrightarrow \texttt{fun}(s)
\end{aligned}$$

□

In other words, an XML element is translated into an ATerm application of a function symbol. Attributes are mapped to annotations and XML strings are mapped to ATerm strings. This unnatural mapping from XML to ATerm is implemented in the Stratego module of figure 6.1. This is a transformation from xml-info to ATerm. The following examples illustrate how this mapping works if applied on our running examples.

Example 13: Unnatural labels
The result of the application of xml-info2data to an xml-info representation of the XML document in example 11 is:

```
labels(
  label(
    quote(emph("Midwinter␣Spring"), "␣is␣its␣own␣season!")
  , name("Thomas␣Eliot")
  , address(street("3␣Prufrock␣Lane"), city("Stamford"), state("CT"))
  )
, label(
    name("Ezra␣Pound")
  , address(street("45␣Usura␣Place"), city("Hailey"), state("ID"))
  )
)
```

Note that if the input contains more labels, the arity of the labels symbol increases.

Example 14: Unnatural Java
The unnatural ATerm representation of the XML document in example 12 is

```
CompilationUnit(
  ClassDec(
    Public
  , Id("HelloWorld")
  , ClassBody(MethodDec(Head(Void, Id("hello")), Block))
  )
)
```

To someone not familiar with other data exchange formats than XML these 'unnatural' representations might no look that bad. The arity of the function symbols in these examples are however determined by the number of children that an XML element has. This causes an explosion of the number of function symbols in the ATerm because every function symbol has an associated arity. For every combination of an element name and the number of children this element, a different function symbol is used. Just one production in a regular hedge grammar for the XML language might in this case result in a more than one, and usually even infinite, productions in the regular tree grammar for the ATerm language. In typical ATerm languages this is different: conceptually equivalent terms are described by just one production with a fixed arity for the terminal (function symbol). Applications of the ATerm format make use of its additional structure capabilities to make the structure more explicit in the term itself.

The rewriting of definition 21 is thus naive, but an important advantage is that it can be implemented generically for all XML documents. No knowledge of the structure of the XML language is required. The tools do not require any domain specific information like a schema for the language of the XML document being transformed. However this generic conversion, implemented by *xml-info2data*, is unfortunately not a solution to the problem of connecting the world of XML and ATerm tools.

The other way around, from ATerm to XML there is however no problem. In this rewriting

the more explicit structure of an ATerm may be lost. This rewriting can be implemented generically as well and can be used in practice to invoke ATerm tools as if they were XML tools. Notice that this already solves a big part of one way of the communication that is required to make ATerm and XML tools work together. However, we need a few more rewrite rules to handle the additional ATerm constructs that are not directly related to a construct in XML. We do however not define how an ATerm annotation that does not have a string value must be mapped to an XML attribute. This problem cannot be solved in a clear and generic way because XML attributes are not be structured.

Definition 22: ATerm ⇒ XML
The following set of rules defines a mapping from the ATerm specific constructs to XML. ATerm to XML .

$$\begin{aligned} \texttt{int}(x) &\Rightarrow \texttt{Text}(x) \\ \texttt{real}(x) &\Rightarrow \texttt{Text}(x) \\ \texttt{list}([t_0, t_1, \ldots, t_n]) &\Rightarrow [t_0, t_1, \ldots, t_n] \\ \texttt{tuple}([t_0, t_1, \ldots, t_n]) &\Rightarrow [t_0, t_1, \ldots, t_n] \\ \texttt{none}() &\Rightarrow [] \\ \texttt{some}(t) &\Rightarrow t \end{aligned}$$

□

The mappings that are defined in this section are implemented by the following tools in the xml-tools package:

xml-info2data
: Implements the generic rewriting of xml-info to an unnatural ATerm .

data2xml-info
: Implements the generic rewriting of an ATerm to xml-info, including the additional mapping rules of definition 22.

6.1.3 Structured ATerm Representation

But what then is a natural ATerm representation of the data in an XML document? 'Natural' is usually quite a subjective term, but in this theses we will apply formal language theory to determine the natural representation of an XML document in the ATerm format. The distinction between ATerms and XML follows from the formalisms that are used to define XML and ATerm languages.

Elements in an XML document have a list of children. These children are elements or character data. This structure of elements in an XML document is comparable to a *hedge*. The structure of the hedges in almost every XML schema language defined by a regular expression. XML languages are thus described by regular hedge grammars and the language is a regular hedge language.

ATerm function applications, which are to be compared to XML elements, also have a list of children. The number of children is however more important in ATerm languages. This number is called the arity and it is associated with the function symbol of the application.

The function symbols in an ATerm language are thus part of a *ranked* alphabet. Therefore ATerms are comparable to *ground terms* over a ranked alphabet. ATerm languages are usually defined by a regular tree grammar and are thus regular tree languages.

Before discussing in detail how this can be fixed, the next examples show the natural representations of the running examples and discuss how they are different from the unnatural ATerm representation.

Example 15: Structured labels ATerm
The structured ATerm representation of the XML document in the labels example cannot just be inferred from the document. The XML document has an *implicit* structure and we need more information to make the structure *explicit* in an ATerm . The structure follows from a definition of the *labels language*. The original article [Ogb03] uses a W3C XML schema for this purpose. For now we will describe the language in informal language. A formal definition of the label language will given later in the RHG language (see example 20). A `labels` elements zero or more `label` children. A `label` element has an optional `quote`, a `name` and an `address`. A `quote` is mixed content with one `emph` element. This is equivalent to an optional string, followed by an `emph`, followed by an optional string. An `address` just takes a `street`, a `city`, and a `state`. The other elements have character data content. The structured ATerm representation, based on this informal language definition, is

```
labels(
  [ label(
      Some(
        quote(
          Some("")
        , emph("Midwinter␣Spring")
        , Some("␣is␣its␣own␣season!")
        )
      )
    , name("Thomas␣Eliot")
    , address(street("3␣Prufrock␣Lane"), city("Stamford"), state("CT"))
    )
  , label(
      None
    , name("Ezra␣Pound")
    , address(street("45␣Usura␣Place"), city("Hailey"), state("ID"))
    )
  ]
)
```

At several places there is a more explicit structure. First of all the `label` elements are now in a list. Because of this the `labels` term has just one child. The function symbol labels has thus a rank of 1. In the second label there is no quote. This is represented by `None`. In the first `label` there is a `quote` and because it is optional this quote is wrapped in a `Some` term. Both `label` terms now have three children and the rank of the label function symbol is thus 3.

In the mixed content of the quote term there is actually represented in a strange way. The optional text is represented by two `Some` terms, but before the `emph` element there was no text at all. This is a typical ambiguity. This structured representation has not been hand-written. It is produced by the tools that are presented later in this chapter. In this case their is an ambiguity because no Text node in xml-info also matches against a string in the language definition. This produces the empty string. A different interpretation would be that there is no text. Our current implementation just chooses one of the options and does not apply any disambiguation filters. See the section 6.8.1 in the future work for a discussion on this issue in general.

Example 16: Structured Java ATerm
In the case of the program example, the difference between the unnatural ATerm representation and the structured ATerm is even larger. Based on the abstract syntax used in the java-front package [Brab], the structured ATerm representation should be

```
CompilationUnit(
  None
, []
, [ ClassDec(
      [Public]
    , Id("HelloWorld")
    , None
    , None
    , ClassBody(
        [ MethodDec(
            Head([], Void, Id("hello"), [], None)
          , Block([])
          )
        ]
      )
    )
  ]
)
```

In this case too the structured representation has been produced by applying the techniques and tools that will be discussed in this chapter. To explain the explicit structure we really need the production rules in the abstract syntax definition for Java. The productions of ClassDec and ClassBody are for instance:

```
ClassDec  -> ClassDec  (ClassMod* Id Super? Interfaces? ClassBody)
ClassBody -> ClassBody (ClassBodyDec*)
```

The production for a ClassBody takes a number of class body declarations. In a structured ATerm this is represented by a ClassBody with just one child term, which is a list of ClassBodyDec. The production for a ClassDec takes 5 arguments:

`ClassMod*`	List of class modifiers like for example abstract and public
`Id`	Name of the class
`Super?`	Optional super class, which this class extends
`Interfaces?`	Optional declaration of the interfaces this class implements
`ClassBody`	Body of class for constructs like fields, constructors and methods

The arity of the ClassDec function symbol is thus five. A ClassDec term must therefore have five children. This is the case in the structure representation. Also the Public term is in a list, because the first child must be a list of class modifiers. The Super and Interfaces children are represented by None because the example has no super class and implements no interfaces. The ClassBody production takes a list of ClassBodyDecs and therefore the MethodDec is now in a list as well. Also note the structure of the Head term and compare this to the unnatural term. The production for this application is

```
MethodHead -> Head (MethodMod* ResultType Id FormalParam* Throws?)
```

6.1.4 Applying Schema Knowledge

The question now is how this structure is to be added to the ATerm. In the first experiments [Bra02] this structuring, then called atermfication, has been implemented by hand for a specific language. This is however error-prone and tiresome work. It requires hand coding for every XML language that is to be represented in a structured ATerm.

The structuring is in fact based on the definition of a language as has been illustrated in the examples. This definition, called a schema in the XML world, contains the essential information of what a structured ATerm in this language must look like. The schema thus provides the structure that was implicit in the XML document itself. In [MLM01, LMM00] it is shown how the several schema languages for XML can be analyzed using the theory of regular hedge grammars [1]. The authors use the term *interpretation*, which plays a central role in this chapter. We repeat their definition in our notation and terminology.

Definition 23: Regular Hedge Interpretation [MLM01]
An interpretation I of a hedge $t_0, t_1, \ldots, t_i$ against a regular hedge grammar $G = (s, \mathcal{N}, \mathcal{A}, P)$ is a mapping from each hedge term t in $t_0, t_1, \ldots, t_i$ to a non-terminal in $\mathcal{N}$, denoted by $I(t)$, such that:

- $I(h_0), I(h_1), \ldots, I(h_i)$ matches s
- for each hedge term t with symbol a and hedge $h'_0, h'_1, \ldots, h'_i$ there exists a production rule $X \rightarrow a\langle R \rangle \in P$ such that
  - $I(t)$ is X
  - $I(h'_0), I(h'_1), \ldots, I(h'_i)$ matches R.

□

For the structuring of an ATerm based on a regular hedge grammar more information is needed. In this definition of interpretation to every XML element a non-terminal will be assigned. Besides the non-terminal assignment, it is crucial to the structuring *how* the children of elements have been matched against the regular expressions of the production rules. This information can be used to put terms in lists, tuples, or optional constructs. The definition for interpretation in our tools is thus an extended version of this definition of interpretation. Besides a mapping from nodes to non terminals, an interpretation also involves a mapping from the lists of children against the regular expressions of production rules in the regular hedge grammar.

Definition 24: Regular Hedge Interpreter
A regular hedge interpreter determines a regular hedge interpretation or the set of regular hedge interpretations of a hedge against a regular hedge grammar. □

In our application area many different languages for schemas (hedge, tree, and word grammars) are involved: various schema languages for XML , concrete syntax definition languages, such as SDF2 , and abstract syntax definition languages, such as Stratego signatures and AbstractSDF. Implementing a powerful interpreter for a specific language like

[1] In the articles the grammars are actually called regular tree grammars. The term 'regular hedge grammar' (see section 2.4) has been rediscovered later and is nowadays used for what the authors then called regular tree grammars.

SDF2 is therefore a waste of time. In this settings their is a need for a more basic grammar definition language. The regular hedge grammar [Mur00, BMW] and regular tree grammar [CDG+97] formalisms are excellent candidates.

For this purpose a set of tools and languages has been developed for working with regular expressions, regular hedge grammars, and regular tree grammars. All domain specific schema languages must be translated into these core grammar formalisms. The extensions to the XML processing tool kit are thus real tools and languages based on formal tree and hedge language theory. The tool kit applies the formal language theory not just in formalizations, discussions, proofs, and the design of new languages, but also in practice as real implementations. As such we have developed a framework for tools based on the formal language theory of regular hedge and tree grammars. This tool kit is not just useful for working with XML , but it will also improve the way abstract syntax is defined in Stratego/XT transformation systems.

The choice for *regular* tree and hedge grammars limits the set of tree languages the tools can handle. This is a limitation. For example the hedge $a^n b^n$ cannot be described by a regular expression. In practice this is not a real limitation because most schema languages for XML are limited as well to regular hedge grammars or subsets thereof. Cause or effect, most XML languages are regular. Also the tree language of all the abstract syntax trees resulting from parsing against a context-free word grammar is a regular tree language. Because of this property there is not a serious need for a more general tree language in typical program transformation systems. But still the tool kit might be extended in the future to handle more expressive, first of all context-free, tree and hedge languages.

6.2 Regular Hedge Grammars

6.2.1 Reusable Language for Regular Expressions

In section 2.4 the definition (18) of regular hedge grammars is based on a separate definition of a regular expression language. Also the definition of the language generated by a regular hedge grammar is almost competely based on the language of a regular expression (see definition 19). The RHG language, which we are going to introduce in section 6.2.4, also reuses a separate language for regular expressions, simply called regexp. The regexp language is defined in a concrete and an abstract syntax and is independent of RHG. It can thus be reused in other languages.

A concrete syntax for the regexp language is defined in the SDF2 module of figure 6.2. The SDF2 module is parameterized with the non-terminal of the alphabet over which an regexp application defines a list of symbols. This module can be imported in a language definition by providing an actual non-terminal over which the regular expressions are defined. The actual symbol can for example be a character or a non-terminal identifier.

The abstract syntax of the regexp language is defined in the Stratego signature of figure 6.3. The RTG and RHG languages, which we use for the specification of abstract syntax in this thesis, do not allow parameterized non-terminals. This parameterization is a useful feature in the definition of such an abstract language, otherwise the abstract syntax definition must be repeated for every actual symbol. The RTG and RHG languages can thus not be used to define the abstract syntax of regexp without knowing an actual symbol. The abstract syntax follows directly from the concrete syntax definition in SDF2. The production rules

```
module regexp[Symbol]

exports
  sorts RegExp

  context-free syntax
    "/"        -> RegExp {cons("empty")}
    Symbol     -> RegExp {cons("sym")}
    RegExp "*" -> RegExp {cons("star")}
    RegExp "+" -> RegExp {cons("plus")}
    RegExp "?" -> RegExp {cons("opt")}

    RegExp "|" RegExp -> RegExp {cons("choice"), right}
    RegExp RegExp     -> RegExp {cons("seq"), right}

    "(" RegExp ")" -> RegExp {bracket}

  context-free priorities
    {left:
      RegExp "*" -> RegExp
      RegExp "+" -> RegExp
      RegExp "?" -> RegExp
    }
  > RegExp     RegExp -> RegExp
  > RegExp "|" RegExp -> RegExp
```

Figure 6.2: Concrete Syntax Definition of regexp

in the syntax definition are annotated with constructor attributes, which are the terminal symbols of the abstract syntax. Parsing a regexp application with the SGLR parser and a parse table generated from this syntax definition by PGEN, will result in a term that is an instance of this abstract syntax.

```
module regexp

signature
  sorts RegExp(a)
  constructors
    empty  : RegExp(a)
    sym    : a -> RegExp(a)
    star   : RegExp(a) -> RegExp(a)
    plus   : RegExp(a) -> RegExp(a)
    seq    : RegExp(a) * RegExp(a) -> RegExp(a)
    choice : RegExp(a) * RegExp(a) -> RegExp(a)
    opt    : RegExp(a) -> RegExp(a)
```

Figure 6.3: Stratego Signature of the regexp language

Example 17: regexp language
To illustrate how the regexp language can be used, we define a tiny language for regular expressions over characters in the following SDF2 module.

```
module regwexp
imports regexp[Char]
exports
  lexical syntax
    [a-zA-Z0-9] -> Char
```

A tiny regular expression in this language of regular expressions over characters is:

```
a+(b|c)*d?
```

The abstract syntax representation of this example regular expression is:

```
seq(
  plus(sym("a"))
, seq(
    star(choice(sym("b"), sym("c")))
  , opt(sym("d"))
  )
)
```

6.2.2 Language for Regular Expression Instances

The definition of the interpretation of an XML document against a RHG definition is based on the interpretation sequences against these regular expressions, like the RHG language is based on regular expressions. Before we introduce the RHG language and its semantics, we therefore first discuss the semantics of the regexp language in the next section.

The semantics of the regexp language concerns the question which sequences of symbols are elements of the language generated by a regexp. The language generated from a regular expression has only been defined for regular word expressions (Definition 7) and in a regular hedge grammar (Definition 19). In both cases the language defined by a regular expressions is denoted by $[\![R]\!]$. Sequence that are in $[\![R]\!]$ are called *instances*.

For the representation of instances the iregexp language has been developed. For each regexp construct there is a construct in iregexp, which describes how a list of symbols is an instance of the construct in the regexp language. The abstract syntax of iregexp language is defined in the following Stratego signature.

```
module iregexp
imports option list-cons

signature
  constructors
    iempty   : IRegExp(b)
    isym     : b -> IRegExp(b)
    istar    : List(IRegExp(b)) -> IRegExp(b)
    iplus    : List(IRegExp(b)) -> IRegExp(b)
    iseq     : IRegExp(b) * IRegExp(b) -> IRegExp(b)
    ichoice  : RegExpChoice * IRegExp(b) -> IRegExp(b)
    iopt     : Option(IRegExp(b)) -> IRegExp(b)

    leftchoice  : RegExpChoice
    rightchoice : RegExpChoice
```

Example 18: iregexp Language
Assuming the usual semantics of regular expressions, the list of characters aaabc is an instance of the regular expression in example 17. The representation of this instance in iregexp is:

```
iseq(
  iplus([isym("a"), isym("a"), isym("a")])
, iseq(
    istar([isym("b"), isym("c")])
  , iopt(None)
  )
)
```

6.2.3 Instance of Regular Expression

In Definition 7 on page 15 the language generated by a regular word expression has been defined. The language of a regular expression over non-terminals in a regular hedge grammar has been defined in Definition 19 on page 21. The semantics of the regexp language is based on the common aspects of these definitions. Both definitions are only different for a symbol in the alphabet over which the regular expression is defined. This section provides an abstract definition of what lists of symbols are in the language generated by a regular expression in regexp . The definition is abstract because it does not define how to handle the sym regular expression, which is domain specific.

The following definition does not just give the semantics of the regexp language, but also defines how an iregexp representation of an instance is to be constructed. The first column contains the inference rules that define when a list of symbols is an instance of a regexp . The inference rules specify how to deduce new facts from known facts. Every application of an inference rule, called a deduction, results in one or more new facts. The right column

specifies how an iregexp term should be constructed when applying the inference rule in the left column.

The definition uses some new notation:

seq	refers to a sequence
$seq_1 +\!\!+ seq_2$	concatenation of the sequences seq_1 and seq_2
$seq \Rightarrow r$	the fact that $seq \in [\![r]\!]$
$(\!\| fact \|\!)$	the iregexp representation of the deduction that has established $fact$

Definition 25: Interpretation of Regular Expression
An interpretation of a sequence of symbols seqagainst a regular hedge expression in regexp is an iregexp term $(\!| \; seq \; |\!)$ that can be deduced using the following inference rules, and an additional set of inference rules for the `sym(_)` regular expression.

Inference Rule	iregexp Representation
$\dfrac{}{[] \Rightarrow \texttt{empty}}$	$\texttt{iempty}()$
$\dfrac{seq \Rightarrow r_1}{seq \Rightarrow \texttt{choice}(r_1, r_2)}$	$\texttt{ichoice}(\texttt{leftchoice}, (\!| \; seq \Rightarrow r_1 \; |\!))$
$\dfrac{seq \Rightarrow r_2}{seq \Rightarrow \texttt{choice}(r_1, r_2)}$	$\texttt{ichoice}(\texttt{rightchoice}, (\!| \; seq \Rightarrow r_2 \; |\!))$
$\dfrac{seq \Rightarrow \texttt{empty}}{seq \Rightarrow \texttt{opt}(r)}$	$\texttt{iopt}(\texttt{None}())$
$\dfrac{seq \Rightarrow r}{seq \Rightarrow \texttt{opt}(r)}$	$\texttt{iopt}(\texttt{Some}((\!| \; seq \Rightarrow r \; |\!)))$
$\dfrac{seq \Rightarrow \texttt{empty}}{seq \Rightarrow \texttt{star}(r)}$	$\texttt{istar}([])$
$\dfrac{seq \Rightarrow \texttt{plus}(r)}{seq \Rightarrow \texttt{star}(r)}$	$\dfrac{(\!| \; seq \Rightarrow \texttt{plus}(r) \; |\!) = \texttt{iplus}(rs)}{\texttt{istar}((\!| \; rs \; |\!))}$
$\dfrac{seq \Rightarrow r}{seq \Rightarrow \texttt{plus}(r)}$	$\texttt{iplus}([(\!| \; r \; |\!)])$
$\dfrac{seq \Rightarrow r \wedge seq \Rightarrow \texttt{plus}(r)}{seq \mathbin{++} seq \Rightarrow \texttt{plus}(r)}$	$\dfrac{(\!| \; seq \Rightarrow \texttt{plus}(r) \; |\!) = \texttt{iplus}(rs)}{\texttt{iplus}([(\!| \; seq \Rightarrow r \; |\!)] \mathbin{++} (\!| \; rs \; |\!))}$
$\dfrac{seq_1 \Rightarrow r_1 \wedge seq_2 \Rightarrow r_2}{seq_1 \mathbin{++} seq_2 \Rightarrow \texttt{seq}(r_1, r_2)}$	$\texttt{iseq}((\!| \; seq_1 \Rightarrow r_1 \; |\!), (\!| \; seq_2 \Rightarrow r_2 \; |\!))$

□

Note that this definition is thus not complete: their is no specification of the semantics of the `sym` construct and how this should be mapped to an `isym` term. These inference rules must be added in the definitions of applications of the regexp language. The inference rules for symbols are not defined in this definition because they are domain specific. The inference rules for deducing facts from symbols are often not just based on equivalence and even equivalence is domain specific.

Definition 26: Instance of Regular Expression
If there is an interpretation of a list of symbols against a regexp then the list of symbols is called an instance of the regexp . □

If a regexp is ambiguous then there can be more than one iregexp representation for an instance of the regexp . We do not define what representation is preferred, how they could be disambiguated, nor is there any restriction on regexp definitions.

Example 19: Ambiguous regexp leads to more than one iregexp
Take this ambiguous regular word expression:

```
a*b?a?
```

This regular expression is parsed to the following abstract syntax representation:

```
seq(
  star(sym("a"))
, seq(opt(sym("b")), opt(sym("a")))
)
```

The string of symbols aaa has two different iregexp representations:

```
iseq(
  istar([isym("a"), isym("a"), isym("a")])
, iseq(iopt(None), iopt(None))
)

iseq(
  istar([isym("a"), isym("a")])
, iseq(
    iopt(None)
  , iopt(Some(isym("a")))
  )
)
```

6.2.4 Language for Regular Hedge Grammars

The RHG language is a language for regular hedge grammars. Although the RHG language is now defined in an abstract syntax and a concrete syntax, the concrete syntax has been created as an afterthought. The RHG language is designed to be used as a kind of core language for abstract syntax definitions. In the RHG language there is one major difference with the basic formalism of regular hedge grammars. To meet the requirements of both XML and the ATerm format, there must be some way to specify the structure of attributes and annotations, as they are called in the ATerm format. In basic regular hedge grammars a production rule only specifies the structure of a single hedge. We extend this to a set of hedges. One of these hedges is the main hedge. The other hedges are labeled with a terminal symbol. They specify the possible annotations and their structure. The ATerm format allows any ATerm as an annotation. The values of XML attributes however are not hedges but just a single string of characters. RHG allows a hedge declaration in an attribute to keep RHG language clear and as format independent as possible. Note that there are because

```
module rhg
imports regexp
signature
  constructors
    rhg       : Start * ProdRules -> RHG
    start     : List(NonTerm) -> Start
    prodrules : List(ProdRule) -> ProdRules

    prodrule  : NonTerm * List(ProdRuleRHS) -> ProdRule
    rhs       : Term * Content * List(Labelled) -> ProdRuleRHS
    ref       : NonTerm -> ProdRuleRHS

    opt-labelled: Term * Content -> Labelled
    labelled:     Term * Content -> Labelled

    content   : RegExp(NonTerm) -> Content

    string    : NonTerm
    int       : NonTerm
    nonterm   : NonTermId -> NonTerm

    term      : TermId -> Term

    quoted    : String -> NonTermId
    quoted    : String ->    TermId
    plain     : String -> NonTermId
    plain     : String ->    TermId
```

Figure 6.4: Abstract syntax definition for RHG in Stratego signature

of this RHG abstract syntax definitions for which there is no XML document that is an element of this language.

The concrete syntax definition of RHG in SDF2 is shown in figure 6.5 on the following page. The production rules of the concrete syntax definition are annotated with the constructor names of the abstract syntax for RHG, which is defined by the Stratego signature in figure 6.4. A RHG term takes two arguments: a top level regular expression in a `start` term and a list of production rules in a `prodrules` term. A production rule has a list of right hand sides, which is just an abbreviation for separate production rules. The right hand side is a reference to another non-terminal or a terminal followed by a regular expression over non-terminals. Cyclic references are not allowed. In RHG there are three non-terminal constructs. Two of them are built in non terminals: `string` and `int`. The third, `nonterm` refers to a user defined non terminal in the regular hedge grammar. The terminal symbols of RHG are produced in the same way as the user defined non-terminal. There are no built-in terminal symbols.

```
module rhg
imports rg-term rg-nonterm rg-identifier rg-layout
        regexp [NonTerm][ RegExp => RegExpNonTerm ]
exports
  sorts RHG
  context-free syntax
    "regular" "hedge" "grammar" Start ProdRules
      -> RHG {cons("rhg")}

    "start"       NonTerm+   -> Start     {cons("start")}
    "productions" ProdRule+  -> ProdRules {cons("prodrules")}

    NonTerm "->" {ProdRuleRHS "|"}+
      -> ProdRule {cons("prodrule")}

    Term Content Labelled* -> ProdRuleRHS {cons("rhs")}
    NonTerm                -> ProdRuleRHS {cons("ref")}

    "." TermId "?" Content -> Labelled {cons("opt-labelled")}
    "." TermId Content     -> Labelled {cons("labelled")}

    "(" RegExpNonTerm ")"  -> Content {cons("content")}
```

Figure 6.5: Concrete syntax definition for RHG in SDF2

Example 20: rhg

The RHG abstract syntax definition for the running document and data example is

```
regular hedge grammar
  start Labels
  productions
    Labels  -> labels  (Label*)
    Label   -> label   (Quote? Name Address)
    Quote   -> quote   (<string>? Emph <string>?)
    Name    -> name    (<string>)
    Address -> address (Street City State)
    Emph    -> emph    (<string>)
    Street  -> street  (<string>)
    City    -> city    (<string>)
    State   -> state   (<string>)
```

Tool Support

parse-rhg
: Parses an RHG definition in the concrete syntax to the abstract syntax.

pp-rhg
: Pretty prints an RHG definition in the abstract syntax to the concrete syntax.

rhg-group
: For each non-terminal groups the production rules for that non-terminal in one production rule.

rhg-reduce
: Removes all production rules of non-terminals that are not productive or reachable. A non-terminal is reachable if there is a derivation from the start regular expression containing this non-terminal. A non-terminal is productive if the language generated by this grammar with the non-terminal as start symbol is not empty.

rhg-normalize
: removes injections of user defined non-terminals.

rhg2rtg and rtg2rhg
: Rewrites an RHG definition to an RTG definition and vice versa. The RTG language will be discussed in section 6.3.1 and in section the functionality of these conversion tools will be discussed.

6.2.5 Language for Regular Hedge Grammar Instances

A regular hedge grammar G generates a regular hedge language $[\![G]\!]$. The semantics of RHG concerns the definition of this language $[\![G]\!]$. We cannot just use the definitions for formal regular hedge grammars because we have extended the formalism with a few constructs: labeled content, and the built-in non-terminals `<string>` and `<int>`. In our application definitions for the specific exchange formats like XML are required as well. These definitions are provided in the next section.

If a hedge in some exchange format is an element of $[\![G]\!]$ then we call it an instance. Instances are again represented by an instance language, called IRHG . The IRHG language is very small because it reuses the language for regexp instances: iregexp . The IRHG language is defined in the following Stratego signature.

```
module irhg
imports rhg iregexp
signature
  constructors
    irhg     : IRegExp(RegLangNode) -> IRHG
    appl     : NonTerm * Term * IRegExp(RegLangNode)
               * List(LabelledContent) -> RegLangNode

    lcontent : String * IRegExp(RegLangNode) -> LabelledContent

    string   : String -> RegLangNode
    int      : Int    -> RegLangNode
```

Tool Support

implode-irhg2xml-info
: Implodes an irhg to xml-info.

irhg2irtg and irtg2irhg
: Rewrites an IRHG term to IRTG and vice versa. The IRTG language and this rewriting will be discussed later in section 6.4.

implode-irhg2aterm
: Composes irhg2irtg and implode-irtg into a tool that implodes an IRHG term to its ATerm representation.

Example 21: irhg
IRHG terms are usually very large. Take for example the tiny Java expression 3 . In our Java abstract syntax this expression is represented by the following abstract syntax tree:

```
<?xml version="1.0" ?>

<Lit><Deci>3</Deci></Lit>
```

Interpreting this tiny XML document against the RHG for the Java language [a] already results in quite a large term:

```
irhg(
  isym(
    appl(
      nonterm("Expr")
    , term("Lit")
    , isym(
        appl(
          nonterm("IntLiteral")
        , term("Deci")
        , isym(string("3"))
        , []
        )
      )
    , []
    )
  )
)
```

[a] The Java abstract syntax and concrete syntax definition we are using here are part the java-front package.

6.2.6 Instance of Regular Hedge Grammar

All the tools and languages that have been presented until now have been developed for one reason: interpreting an XML document against a regular hedge grammar. Now all these tools are at our disposal we can finally define the interpretation of XML documents against a regular hedge grammar. An XML document is an instance of a regular hedge grammar in the RHG language if and only if there is a full interpretation of this XML document against the grammar. The full interpretation is an IRHG term. The next definition provides the semantics of the RHG language in the context of the XML format. It shows how an XML document is to be fully interpreted against a regular hedge grammar, producing an IRHG term.

Definition 27: XML Interpretation against Regular Hedge Grammar
An interpretation of an xml-info term $\texttt{Document}(e)$ against a regular hedge grammar $\texttt{rhg}(r_s,\, P)$ is an IRHG term $(\!|\, \texttt{Document}(e) \Rightarrow \texttt{rhg}(r_s,\, P) \,|\!)$ that can be deduced using the following inference rules and the inference rules of definition 25 on page 65.

Inference Rule	IRHG Representation
$\dfrac{e \Rightarrow r_s}{\texttt{Document}(e) \Rightarrow \texttt{rhg}(r_s,\, P)}$	$\texttt{irhg}((\!|\, e \Rightarrow r \,|\!))$
$\dfrac{qn \Rightarrow_n t \wedge s \Rightarrow r}{\texttt{Attribute}(qn,\, s) \Rightarrow_l \texttt{lcontent}(t,\, r)}$	$\texttt{lcontent}(t,\, (\!|\, s \Rightarrow r \,|\!))$
$\dfrac{\begin{array}{c}\texttt{prodrule}(nt,\, t,\, r,\, lc*\,) \in P \\ \forall a \in a* : \exists lc \in lc* : a \Rightarrow_l lc \\ \forall lc \in lc* : \exists a \in a* : a \Rightarrow_l lc \\ qn \Rightarrow_n t \wedge c* \Rightarrow r\end{array}}{\texttt{Element}(qn,\, a*\,,\, c*\,) \Rightarrow \texttt{sym}(nt)}$	$\texttt{appl}(nt,\, t,\, (\!|\, c* \Rightarrow r \,|\!),\, \forall a \in a* : (\!|\, a \Rightarrow_l lc \,|\!))$
$\dfrac{seq \Rightarrow \texttt{empty}}{seq \Rightarrow \texttt{sym}(\texttt{string})}$	$\texttt{string}(\text{""})$
$\dfrac{}{\texttt{Text}(s) \Rightarrow \texttt{sym}(\texttt{string})}$	$\texttt{string}(s)$
$\dfrac{\texttt{Text}(s) \Rightarrow_{\mathbb{Z}} x}{\texttt{Text}(s) \Rightarrow \texttt{sym}(\texttt{int})}$	$\texttt{int}(x)$
$\dfrac{}{\texttt{QName}(\texttt{none}(),\, s) \Rightarrow_n \texttt{term}(s)}$	
$\dfrac{}{\texttt{QName}(\texttt{Some}(\texttt{Namespace}(s_1)),\, s_2) \Rightarrow_n \texttt{term}(\{s_1\}s_2)}$	

□

For the conversion of XML to an ATerm the tools are interested in the full interpretation of an XML document against a regular hedge grammar. Many other software tools are not at all interested in the exact details of an interpretation. In most XML applications a schema for the XML language being processed is just used to *validate* an XML document. Most

validity checking is performed on a structural schema in an XML schema language, such as RELAX NG, W3C XML Schema, and DTD. Checking a document against different kinds of schema languages, such as Schematron, is however also called as validation. Structural validity checking answers the question whether a document can be generated by a grammar, or in other words: if it is an element of the language generated by the grammar.

Definition 28: XML Instance of Regular Hedge Grammar
An xml-info term is an instance of an RHG if there is an interpretation of the xml-info term against the RHG . □

Note that there can be more than one interpretation of the xml-info term against the RHG grammar because more than one production rule might be applicable and the regular expressions are ambiguous as well if multiple interpretations are possible. These differences yield different interpretations. The set of interpretations of an xml-info term against a regular hedge grammar is the set of all possible IRHG terms that can be deduced using the inference rules of definition 27.

Tool Support

A prototype of the interpretation as defined in Definition 27 has been implemented and is available as part of the xml-tools package. This tool has been used to generate all the structured ATerm examples in this thesis, which illustrates how effective this tool already is. Because of the highly component-based design of XML interpretation the complexity of XML interpretation is fully under control.

xml-info2irhg
: Produces an interpretation of an xml-info representation of an XML document against a regular hedge grammar in the RHG language. This is the core tool for interpreting an XML document against a schema.

6.3 Regular Tree Grammars

6.3.1 Language for Regular Tree Grammars

Regular hedge grammars are suitable as a grammar formalism for XML , but not for the ATerm format. The set of terminal symbols in a regular hedge grammar is an unranked alphabet. A hedge term consists of a terminal symbol and a hedge, which is a sequence of hedge terms. In regular hedge grammars the structure of this sequence is specified with a regular expression over the non-terminal symbols of the regular hedge grammar. Regular expressions are suitable for this because they define permissable sequences of symbols and a hedge is a list of symbols, where the symbols are hedge terms in this case. The language of a regular expression, which is a set of sequences, usually contains sequences with a different number of symbols.

In the ATerm format the set of function symbols in a certain ATerm language is a *ranked* alphabet. All application terms of a particular function symbol f/n have the same number of children, which is the arity of the function symbol. Regular sequence expressions are not suitable for defining the allowed arguments of a function symbol because this is a fixed

```
regular hedge grammar
  start RTG
  productions
    RTG        -> rtg (Start ProdRules)
    Start      -> start (NonTerm+)
    ProdRules  -> prodrules (ProdRule+)
    ProdRule   -> prodrule (NonTerm AnnoTreeFN+)
    AnnoTreeFN -> TreeFN
    AnnoTreeFN -> annotated (TreeFN Labelled+)
    TreeFN     -> appl (Term AnnoTreeFN*)
    TreeFN     -> tuple (AnnoTreeFN*)
    TreeFN     -> list (AnnoTreeFN)
    TreeFN     -> opt (AnnoTreeFN)
    TreeFN     -> ref (NonTerm)
    Labelled   -> labelled (TermId AnnoTreeFN)
```

Figure 6.6: Abstract syntax definition for RTG in RHG

number of arguments. The ATerm format is thus much closer to regular tree languages over ranked alphabets as defined in definition 13.

It is possible to give a certain meaning to a regular sequence expression in order to map it to a definition for the arguments of a function symbol with a fixed arity. In fact in a few sections such a mapping will be defined to be able to covert regular hedges to regular trees and vice versa. Yet, in our tool kit there is a need for a specific language defining regular tree grammars. This clearly separates the world of regular hedges and regular trees. Tools operate on one of these and there is no need to derive some regular tree definition from the regular expression. This conversion is however defined in separate definitions and implemented in separate tools.

For the definition of regular tree languages two formalism are used in literature: regular tree expressions and regular tree grammars [CDG+97]. Both formalisms are interesting to have in our tool kit, yet the regular tree grammars have been chosen as the primary specification language for regular tree languages. The RTG language is a language for regular tree grammars. As usual in our tool kit it is defined in a concrete and an abstract syntax, where the concrete syntax again is not intended for heavy use by humans. The RTG language is like the RHG language designed to be used by program, not by humans. Like in the RHG language, the RTG language extends the formalism of regular tree grammars with labeled content. In the RHG language the regular expression over non-terminals defines the permissable hedges for the default hedge and the labeled hedges. An application of a function symbol in the trees described by the RTG language can have a number of labeled trees in addition to the arguments of the function symbols. These labeled trees represent the attributes of XML and the annotation of the ATerm format.

In the RTG language, which is defined in figure 6.6 and 6.7, there are several non-terminal constructs. Besides the user-defined non-terminals the RTG language has the built-in primitive non-terminals `<int>`, `<string>` and the parameterized non-terminals `[ ...]` for lists and `( ...)` for tuples. The list non-terminal takes one non-terminal as an argument. The tuple non-terminal takes zero or more non-terminal arguments. Terminal symbols in an RTG grammar can be quoted in case of special characters in the same as in RHG. The left hand side of a production rule is a non-terminal. Built-in non-terminals are not allowed

```
module rtg
imports rg-term rg-nonterm rg-identifier rg-layout
        regexp [NonTerm][ RegExp => RegExpNonTerm ]
exports
  sorts RTG
  context-free syntax
    "regular" "tree" "grammar" Start ProdRules
      -> RTG {cons("rtg")}

    "start"       NonTerm+  -> Start     {cons("start")}
    "productions" ProdRule+ -> ProdRules {cons("prodrules")}

    NonTerm "->" {AnnoTreeFN "|"}+ -> ProdRule {cons("prodrule")}
    TreeFN           -> AnnoTreeFN
    TreeFN Labelled+ -> AnnoTreeFN    {cons("annotated")}

    Term "(" {AnnoTreeFN ","}* ")" -> TreeFN {cons("appl")}
         "(" {AnnoTreeFN ","}* ")" -> TreeFN {cons("tuple")}
         "[" AnnoTreeFN "]"        -> TreeFN {cons("list")}
             AnnoTreeFN "?"        -> TreeFN {cons("opt")}
    NonTerm                        -> TreeFN {cons("ref")}

    "." TermId AnnoTreeFN -> Labelled {cons("labelled")}
```

Figure 6.7: Concrete syntax definition for RTG in SDF2

at the left hand side of a production rule. The right hand side is a tree of terminals and non-terminals. This means that more than one terminal application can occur in this tree.

Definition 29: Normalized RTG
A normalized RTG is an RTG where all production rules are of one of the following forms:

`prodrule(nonterm(_), appl(term(_), nts)`
: where nts is a list on non-terminals. If in the left hand side parameterized, built-in non-terminals occur, then all the arguments of the non-terminal must be non-terminals.

`prodrule(nonterm(_), ref(nt))`
: where nt is one of the built-in non-terminals.

□

The definition of a normalized regular tree grammar in [CDG+97] does not allow any production rule of the form $A_1 \rightarrow A_2$ where A_1 and A_2 are non-terminals. The productions of this form are also known as injections and we denote them by $I = \{P|(A_1 \rightarrow A_2)\}$. Injections can be removed by replacing them with the production rules in $P \setminus I$ that are reachable from A_1 by using just the production rules in I. In the RTG language it is however not possible to remove all injections in this way because there are no production rules for the built-in non-terminals. Therefore this kind of injection is allowed in a normalized RTG . In section 6.8.4 of the future work we discuss a cleaner approach to the built-in non-terminals where the built-in non-terminals refer to built-in production rules. If the RTG language is extended with these built-in production rules then there is no need for the second form of a production rule.

Tool Support

parse-rtg
: Parses an RTG definition in the concrete syntax to the RTG abstract syntax.

pp-rtg
: Pretty prints an RTG definition in the abstract syntax to the concrete syntax.

rtg-group
: Groups the production rules of the same non-terminal in one production rule with several alternatives.

rtg-reduce
: Removes all production rules of non-terminals that are not productive or reachable. The definitions of reachable and productive non-terminals are equal to the definitions in [CDG+97] and are also used for the rhg-reduce tool.

rtg-normalize
: Rewrites an RTG to a normalized RTG as defined by definition 29.

rhg2rtg and rtg2rhg
: Rewrites an RHG definition to a RTG definition and vice versa. This conversion will be presented in section 6.4.

6.3.2 Language for Regular Tree Grammar Instances

The IRTG language is a language for the representation of regular tree grammar instances, that is: regular trees. A term in the IRTG language exactly describes how a regular tree is an instance of an RTG . In our tool kit the regular trees are currently always ATerms, but other exchange formats might be added in the future. Currently the IRTG language is thus used to describe how an ATerm is an instance of an RTG . IRTG is like IRHG exclusively intended for use by programs, thus there is no concrete syntax for IRTG . In the Stratego signature of figure 6.8 the abstract syntax of IRTG is defined. Note that the IRTG is very close to the exploded representation of an ATerm , the main difference is the presence of non-terminal information in IRTG .

Tool Support

implode-irtg
: Implodes an IRTG representation of a regular tree to an ATerm . The implementation is shown in figure 6.9.

irhg2irtg and irtg2irhg
: Rewrites an IRHG term to IRTG and vice versa. As mentioned several times before this rewriting is the topic of section 6.4.

6.3.3 Instance of Regular Tree Grammar

The RTG language is independent of a specific exchange format for tree-like structured data as long as the format is similar to the set of ground terms over a ranked alphabet. The semantics for the RTG language has to be defined separately for these exchange formats. This section defines when an ATerm is an instance of an RTG . The definition also specifies how to construct an instance representation in IRTG .

```
module irtg
signature
  constructors
    irtg      : ARegTree -> IRTG

              : RegTree -> ARegTree
    annotated : RegTree * List(Anno) -> ARegTree
    anno      : Term * ARegTree -> Anno

    appl      : NonTerm * Term * List(ARegTree) -> RegTree
    anon-appl : Term * List(ARegTree) -> RegTree
    list      : List(ARegTree) -> RegTree
    tuple     : List(ARegTree) -> RegTree
    opt       : Option(ARegTree) -> RegTree
    string    : String -> RegTree
    int       : Int    -> RegTree

    nonterm   : String -> NonTerm
    term      : String -> Term
```

Figure 6.8: Abstract syntax definition for IRTG in Stratego signature

Definition 30: ATerm Instance of Regular Tree Grammar
The instance representation of an ATerm t against a regular tree grammar $\texttt{rtg}(\texttt{start}(\mathcal{S}), \texttt{prodrules}(P))$ is an IRTG term $(\!| t \Rightarrow \texttt{rtg}(\texttt{start}(\mathcal{S}), \texttt{prodrules}(P)) |\!)$ that can be deduced using the following inference rules.

```
module implode-irtg
imports rtg-util options

strategies

  main-implode-irtg =
    io-wrap(implode-irtg)

  implode-irtg =
    bottomup-irtg(Ignore <+ Implode)

rules

  Implode : irtg(ch) -> ch

  Implode : annotated(ch, anno*) -> ch{anno*}

  Implode : anno(t, v) -> (t#([]), v)

  Implode : appl(nt, t, chs) -> t#(chs)

  Implode : anon-appl(t, chs) -> t#(chs)

  Implode : opt(ch) -> ch

  Implode : list(chs) -> chs

  Implode : tuple(chs) -> ""#(chs)

  Implode : int(x) -> x

  Implode : string(s) -> s

  Implode : term(s) -> s

strategies

  Ignore = ?nonterm(_)
```

Figure 6.9: Implosion of IRTG to an ATerm

Inference Rule	IRTG Representation
$\dfrac{t \Rightarrow \texttt{ref}(A) \wedge A \in \mathcal{S}}{t \Rightarrow \texttt{rtg}(\texttt{start}(\mathcal{S}), \texttt{prodrules}(P))}$	$\texttt{irtg}(\langle\!\vert\ t \Rightarrow \texttt{ref}(A)\ \vert\!\rangle)$
$\dfrac{}{\texttt{int}(x) \Rightarrow \texttt{int}(x)}$	$\texttt{int}(x)$
$\dfrac{}{\texttt{string}([c_0, c_1, \ldots, c_n]) \Rightarrow \texttt{string}()}$	$\texttt{string}([c_0, c_1, \ldots, c_n])$
$\dfrac{t_0 \Rightarrow T_0 \wedge t_1 \Rightarrow T_1 \wedge \ldots \wedge t_n \Rightarrow T_n}{\texttt{tuple}([t_0, t_1, \ldots, t_n]) \Rightarrow \texttt{tuple}([T_0, T_1, \ldots, T_n])}$	$\texttt{tuple}([\langle\!\vert\ t_0 \Rightarrow T_0\ \vert\!\rangle, \langle\!\vert\ t_1 \Rightarrow T_1\ \vert\!\rangle, \ldots, \langle\!\vert\ t_n \Rightarrow T_n\ \vert\!\rangle])$
$\dfrac{t_0 \Rightarrow T, t_1 \Rightarrow T, \ldots, t_n \Rightarrow T}{\texttt{list}([t_0, t_1, \ldots, t_n]) \Rightarrow \texttt{list}(T)}$	$\texttt{list}([\langle\!\vert\ t_0 \Rightarrow T\ \vert\!\rangle, \langle\!\vert\ t_1 \Rightarrow T\ \vert\!\rangle, \ldots, \langle\!\vert\ t_n \Rightarrow T\ \vert\!\rangle])$
$\dfrac{}{\texttt{none}() \Rightarrow \texttt{opt}(T)}$	$\texttt{opt}(\texttt{None}())$
$\dfrac{t \Rightarrow T}{\texttt{some}(t) \Rightarrow \texttt{opt}(T)}$	$\texttt{opt}(\texttt{Some}(\langle\!\vert\ t \Rightarrow T\ \vert\!\rangle))$
$\dfrac{t_0 \Rightarrow T_0 \wedge t_1 \Rightarrow T_1 \wedge \ldots \wedge t_n \Rightarrow T_n}{\texttt{appl}(\texttt{fun}(s), [t_0, t_1, \ldots, t_n]) \Rightarrow \texttt{appl}(\texttt{term}(s), [T_0, T_1, \ldots, T_n])}$	$\texttt{appl}(\texttt{term}(s), [\ \langle\!\vert\ t_0 \Rightarrow T_0\ \vert\!\rangle, \langle\!\vert\ t_1 \Rightarrow T_1\ \vert\!\rangle, \ldots, \langle\!\vert\ t_n \Rightarrow T_n\ \vert\!\rangle\ ])$
$\dfrac{t \Rightarrow T}{\texttt{anno}(\texttt{fun}(s), t) \Rightarrow \texttt{anno}(\texttt{term}(s), T)}$	$\texttt{anno}(\texttt{term}(s), \langle\!\vert\ t \Rightarrow T\ \vert\!\rangle)$
$\dfrac{t_0 \Rightarrow T_0 \wedge t_1 \Rightarrow T_1 \wedge \ldots \wedge t_n \Rightarrow T_n \wedge t \Rightarrow T}{\texttt{annotated}(t, [t_0, t_1, \ldots, t_n]) \Rightarrow \texttt{annotated}(T, [T_0, T_1, \ldots, T_n])}$	$\texttt{annotated}(\langle\!\vert\ t \Rightarrow T\ \vert\!\rangle, [\ \langle\!\vert\ t_0 \Rightarrow T_0\ \vert\!\rangle, \langle\!\vert\ t_1 \Rightarrow T_1\ \vert\!\rangle, \ldots, \langle\!\vert\ t_n \Rightarrow T_n\ \vert\!\rangle\ ])$
$\dfrac{\texttt{prodrule}(A, T) \in P \wedge t \Rightarrow T}{t \Rightarrow \texttt{ref}(A)}$	$\texttt{iref}(A, \langle\!\vert\ t \Rightarrow T\ \vert\!\rangle)$

□

Tool Support

aterm2irtg
: Interprets an ATerm against a regular tree grammar and produces an IRTG term.

wf-checker
: Checks whether an ATerm is an instance of an RTG by applying *aterm2irtg*.

6.4 Mapping Regular Hedge and Tree Instances

The RTG and RHG languages, which are used to define hedge (XML) and tree (ATerm) languages, and their instance representations in IRTG and IRHG have now been defined. Now there is one issue left to complete the conversion from XML to ATerm: convert an RHG term to an IRTG term. This section defines a default, natural, mapping for this.

For many constructs in IRHG there is an obvious mapping into RTG . For example `istar` and `iplus` can be represented by a list, `string` and `int` can even be represented in exactly the same way. The problem is the `iseq` construct: IRHG terms describe how *sequences* are an instance of a regular expression. Sequences can occur anywhere. The mapping we have chosen for this is to put a sequence in a tuple at places one term is required in RTG . For example an RTG `opt` term has one child. If in the corresponding construct in an RHG term the children of correspond to a sequence of more than one term, then a tuple is inserted to turn the sequence in a single term. The only construct in RHG that produces more than one term in is the `iseq`.

Example 22: XML to IRHG to IRTG to ATerm
As an example of how this mapping works consider this hedge of XML elements:

```
<a/><b/><c/><c/><c/><c/><d/>
```

The structure of this hedge is specified by a regular expression over non-terminals:

```
(a b)? (c c)+ d*
```

Applying the *parse-xml-info* | *xml-info2irhg* tools results in the following RHG fragment [a]:

```
iseq(
  iopt(Some(iseq(isym("a"), isym("b"))))
, iseq(
    iplus(
      [ iseq(isym("c"), isym("c"))
      , iseq(isym("c"), isym("c"))])
  , istar(
      [isym("d")])))
```

In the conversion to IRTG by applying *irhg2irtg* several tuples are created in the instance representation [b]:

```
tuple(
  [ opt(
      Some(
        tuple([appl("a"), appl("b")])))
  , list(
      [ tuple([appl("c"), appl("c")])
      , tuple([appl("c"), appl("c")])
      ])
  , list([appl("d")])
  ])
```

Applying *implode-irtg* results in the ATerm

```
(Some((a,b)),[(c,c),(c,c)],[d])
```

[a] In this example `appl("a")` stands for `appl(nonterm("a"),term("a"),[])`
[b] In this example `appl("a")` stands for `ref("a",appl(term("a"),[]))`

Tool Support

irhg2irtg and irtg2irhg
: Converts an IRHG into an IRTG and vice versa. The implementation of *irhg2irtg* is shown in figure 6.10

rhg2rtg and rtg2rhg
: Converts an RHG to an RTG and vice versa according to the mapping defined in this section.

```
module irhg2irtg
imports irhg irtg options rtg-util

strategies
  main-irhg2irtg =
    io-wrap(irhg2irtg)

strategies
  irhg2irtg =
    \ irhg(i) -> irtg(i) \; traversal

  rewrite-one =
       ?isym(<rewrite-one>)
     + Rewrite-1; traversal
    <+ !tuple(<rewrite-seq>)

  rewrite-seq =
       \ iseq(s1, s2) -> <conc> (<rewrite-seq> s1, <rewrite-seq> s2) \
     + \ iempty() -> [] \
     + is-list; map(rewrite-one)
    <+ ![<rewrite-one>]

  traversal =
      irtg(rewrite-one)
    + annotated(rewrite-one, map(rewrite-one))
    + anno(id, rewrite-one)
    + appl(id, id, rewrite-seq)
    + anon-appl(id, rewrite-seq)
    + list(rewrite-seq)
    + tuple(rewrite-seq)
    + opt(Some(rewrite-one))
    + opt(None())
    + int(id)
    + string(id)

rules
  Rewrite-1 : appl(nt, t, chs, [x | xs]) -> annotated(appl(nt, t, chs, []), [x | xs])

  Rewrite-1 : appl(nt, t, chs, []) -> appl(nt, t, chs)

  Rewrite-1 : lcontent(l, i) -> anno(term(l), i)

  Rewrite-1 : istar(is) -> list(is)

  Rewrite-1 : iplus(is) -> list(is)

  Rewrite-1 : iopt(None()) -> opt(None())

  Rewrite-1 : iopt(Some(i)) -> opt(Some(i))

  Rewrite-1 : string(s) -> string(s)

  Rewrite-1 : int(x) -> int(x)
```

Figure 6.10: Rewriting of IRHG to IRTG

6.5 Syntax Definition Conversions

The RHG and RTG language are designed to be an intermediate language for abstract syntax definitions, which are usually called schema languages in the context of XML . The concrete syntax for RTG and RHG has been designed as an afterthought, but it is possible to define an abstract syntax by hand in RHG or RTG . Of course the XML processing tool kit must be extended with conversion tools to transform existing schemas and syntax definitions to RTG or RHG . The reuse of RHG and RTG is interesting because validation and interpretation are difficult to implement. Implementing interpretation of XML to a structured ATerm for each schema language requires a lot of work. By chosing RHG or RTG as an intermediate syntax definition language, all the tooling that is developed for RHG and RTG is available for all syntax definitions that can be translated into RTG or RHG .

Some of the conversion tools have already been implemented, but a lot of work remains to be done, especially for RELAX NG and W3C XML Schema. The implementations of the conversion tools will not be discussed, but some of the more interesting challenges and highlights of the implementations will be mentioned.

6.5.1 DTD - Document Type Definitions

A DTD can be converted to an RHG using *dtd2rhg*. To ensure maximal standard compliance with minimal effort the DTDinst [Claa] tool (and language) of James Clark is used to parse a DTD into an XML instance format. The dtdinst2rhg transformation itself is implemented on a structured ATerm representation of the DTDinst language. *xml-interpret* is used to structure the XML output of DTDinst. DTDinst is not distributed as part of the xml-tools package. It is invoked using a JNLP[Pro] specification and Netx [Max], an open source JNLP client that can be used from the command line.

Example 23: DTDInst

```
<!ENTITY % author "author">

<!ELEMENT %author; (first-name, middle-name?, surname)>
<!ELEMENT first-name  (#PCDATA)>
<!ELEMENT middle-name (#PCDATA)>
<!ELEMENT last-name   (#PCDATA)>
```

The XML DTDinst representation of this DTD is shown in figure 6.11. The result of processing this representation to a structured ATerm is shown in figure 6.12.

6.5.2 SDF2 - Syntax Definition Formalism

SDF2 [Vis97b] is used by many Stratego/XT applications for the specification of concrete syntax. Stratego signatures, parse tables for SGLR [Vis97a, vdBSVV02], and pretty printers for Generic Pretty-Printer Package[dJ00] (GPP), are generated from SDF2 syntax definitions. Transforming SDF2 to an abstract syntax definition is not trivial because SDF2 is a very rich language. The translation has to handle modular definitions, renamings, aliases, lexical syntax, context-free syntax and many other constructs.

```
<?xml version="1.0" encoding="UTF-8"?>
<doctype>
  <nameSpec name="author">
    <name>author</name>
  </nameSpec>
  <element>
    <nameSpecRef name="author"/>
    <sequence>
      <elementRef>
        <name>first-name</name>
      </elementRef>
      <optional>
        <elementRef>
          <name>middle-name</name>
        </elementRef>
      </optional>
      <elementRef>
        <name>surname</name>
      </elementRef>
    </sequence>
  </element>
  <element>
    <name>first-name</name>
    <pcdata/>
  </element>
  <element>
    <name>middle-name</name>
    <pcdata/>
  </element>
  <element>
    <name>last-name</name>
    <pcdata/>
  </element>
</doctype>
```

Figure 6.11: DTD in the DTDinst representation

```
doctype(
  [ nameSpec(name("author")){(name, "author")}
  , element(
      nameSpecRef{(name, "author")}
    , sequence(
        Some(
          ( elementRef(name("first-name"))
          , [optional(elementRef(name("middle-name"))), elementRef(name("surname"))]
          )
        )
      )
    )
  , element(name("first-name"), pcdata)
  , element(name("middle-name"), pcdata)
  , element(name("last-name"), pcdata)
  ]
)
```

Figure 6.12: DTD in the DTDinst structured ATerm representation

The SDF2 to Stratego signature transformation that has been used until now did not handle all these constructs and as a result it produced invalid signatures. The Stratego compiler does not really care about this, and because no other tools were using the abstract syntax definition the transformation of SDF2 into Stratego signatures has never been improved. Comparible translations are implemented in JJForester [KV01] and ApiGen [dJO02]. All these translations have their minor problems that make the generated abstract syntax definitions not applicable for heavy use in a tool kit centered around more formal tree language theory.

Implementing the translation of SDF2 to RTG is not easier then any of the existing translations, but the advantage of a translation into RTG is that this language is not targeted at a specific language or application. As an intermediate abstract syntax definition language, it can be used for many different purposes. The tooling required for translating a concrete syntax definition in SDF2 to an abstract syntax definition in Stratego, Java, Haskell, or whatever language, is now separated into several components that do just one thing and try to do it well. Because of this separation it is worth the effort to invest some more time into a good translation of SDF2 syntax definitions into RTG .

Tool Support

The transformation from SDF2 to RTG is a composition of several transformation components. The major work is done in a new implementation of SDF2 normalization. This normalization component reduces a modular SDF2 syntax definition to an SDF2 grammar. This grammar is then transformed into RTG .

core-sdf-normalize
: Applies a selective number of standard SDF2 normalizations as defined in [Vis97b]). This normalization step produces, given a main module and an SDF2 definition a non-modular grammar. The applied normalizations are modular normalization, basic normalization and alias-normalization (see [Vis97b]).

core-sdf-grammar2rtg
: Transforms the partially normalized SDF2 grammar to RTG . The grammar still contains informartion that is irrelevant for the abstract syntax, therefore some special filters must be used. For example layout, lexical, bracket and reject productions must be ignored.

sdf2rtg
: Composes *parse-sdf*, *core-sdf-normalize*, and *core-sdf-grammar2rtg* into a single tool that transforms an SDF2 syntax definition into an RTG abstract syntax definition.

Example 24: SDF2 to RTG
Figure 6.13 shows an example SDF2 syntax definition for a simple expression language. The RHG abstract syntax definition that is generated from this SDF2 syntax definition using the *sdf2rtg* and *rtg2rhg* tools is

```
regular hedge grammar
  start Exp
  productions
    Id  -> <string>
    Exp -> Mul (Exp Exp)
    Exp -> Div (Exp Exp)
    Exp -> Plus (Exp Exp)
    Exp -> Min (Exp Exp)
    Exp -> Var (Id)
```

6.5.3 Stratego Signatures

The Stratego language has its own algebraic signature language for defining the abstract syntax of languages. Some of the languages introduced in this thesis have been defined in this signature language. The signatures are used by the Stratego compiler to check the arity of the function symbol applications that are used in Stratego programs.

The algebraic signatures of Stratego are comparable to the RTG language. There are however some minor differences:

- Stratego signatures do not allow the declaration of the structure of annotations.
- Stratego signatures can have parameterized sorts (called non-terminals in RTG), like parameterized data types in Haskell. These parameterized data types are mostly used for optional terms (`Option(a)`) and list of terms (`List(a)`). Parameterized sorts have been used in the definition of the abstract syntax of the regexp and iregexp languages.

Tool Support

The conversion from and to Stratego signatures to RTG and vice versa is implemented by the following tools:

sig2rtg
: Translates a Stratego signature into RTG . Parameterized non-terminals are removed by generating production rules for every actual parameter.

rtg2sig
: Translates an RTG into a Stratego signature. Because Stratego signatures do not allow the declaration of annotation they are currently just left out.

Stratego signatures can be handwritten, but in typical Stratego program transformation systems they are generated from an SDF2 concrete syntax definition. From the introduction of the RTG language and *sdf2rtg* the preferred way to generate a Stratego signature from a SDF2 concrete syntax definition is by first translating it into an RTG using *sdf2rtg* and then translate the RTG into a Stratego signature using *rtg2sig*.

```
definition
module Expression
  imports Identifier Layout
  exports
    sorts Exp

    context-free syntax
      Exp "*" Exp -> Exp {cons("Mul")}
      Exp "/" Exp -> Exp {cons("Div")}
      Exp "+" Exp -> Exp {cons("Plus")}

      Exp "-" Exp -> Exp {cons("Min")}
      Id          -> Exp {cons("Var")}

    context-free priorities
      {left:
        Exp "*" Exp -> Exp
        Exp "/" Exp -> Exp
      }
    > {left:
        Exp "+" Exp -> Exp
        Exp "-" Exp -> Exp
      }

module Identifier
  exports
    lexical syntax
      [a-zA-Z\_] [a-zA-Z0-9\_]* -> Id

    lexical restrictions
      Id -/- [a-zA-Z0-9\_]

module Layout
  exports
    lexical syntax
      [\ \t\n] -> LAYOUT

    context-free restrictions
      LAYOUT? -/- [\ \t\n]
```

Figure 6.13: SDF2 definition of simple expressions

6.6 All Together Now

All tools and languages that are used for the processing of an XML document to a structured ATerm have been presented now. On page 88 the bigger picture of the XML processing tool kit is shown. The large arrow indicates the flow of data when an XML document is being processed into a structured ATerm for consumption by a tool that expects data in a structured ATerm representation.

First an XML document is parsed to an xml-doc representation or immediately into xml-info. Next the xml-info is interpreted against a regular hedge grammar in RHG . This results in an IRHG term. The IRHG term is rewritten to an IRTG term. The IRTG is then imploded into the final structured ATerm that can be processed by the ATerm tool. The result of applying this pipeline of tools to our running examples has already been shown in the examples 15 and 16, so unfortunately there are no exciting terms left to show.

If the XML language is not defined in RHG then the definition must be converted to RHG . The options for this are also illustrated in the big picture. The *xml-interpret* tool composes all these tools into one tool that can be configured with command line arguments. It composes parse-xml-info, xml-info2irhg, irhg2irtg, and implode-irtg into one pipeline. All the schema languages that are supported by our tool kit can be specified on the command line so they do not have to be processed to RHG separately.

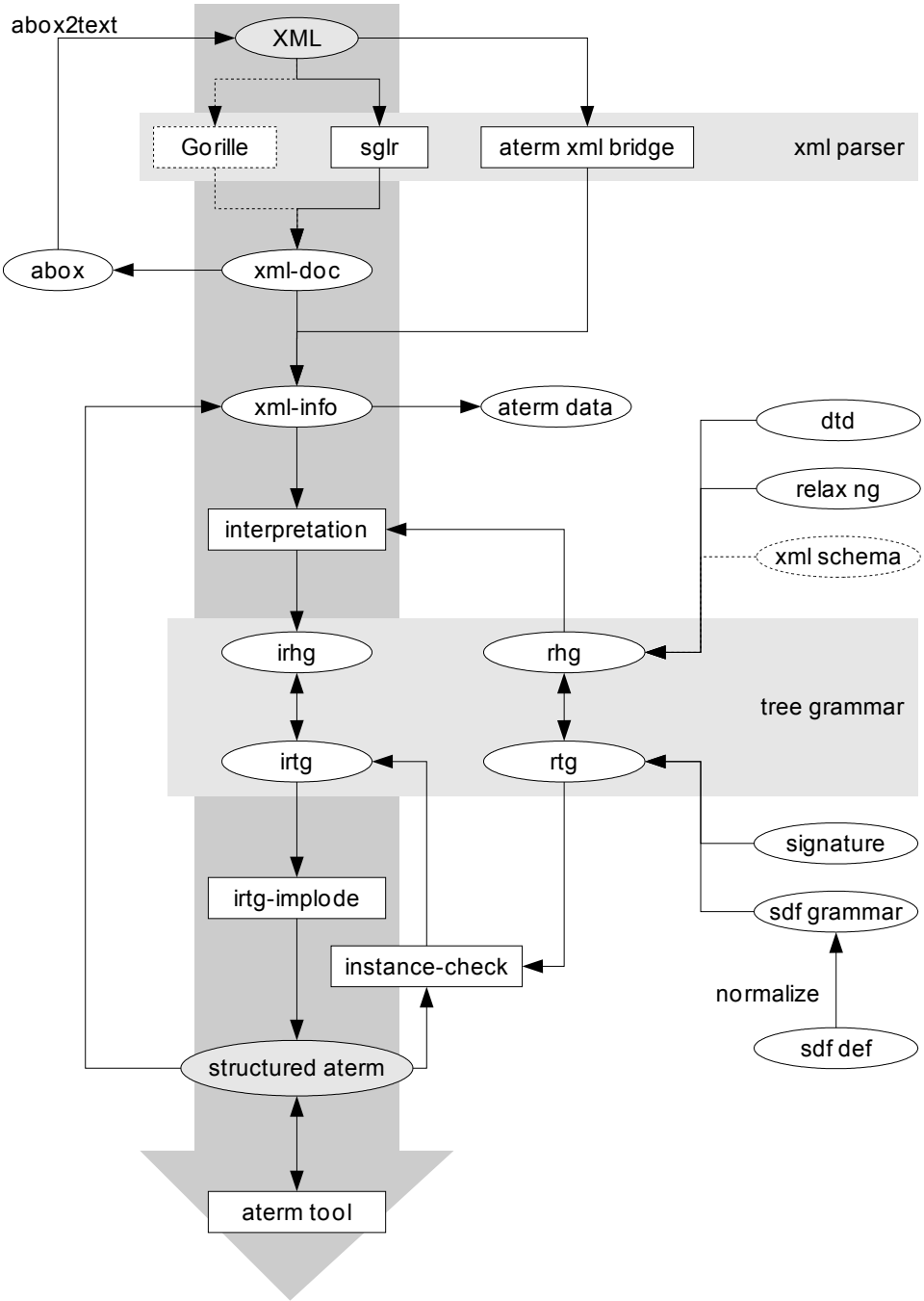
6.7 Related Work

6.7.1 XML Data Binding

Finding a natural representation of an XML document is also the goal of XML data binding [Bou]. An XML data binding solution seeks for a customary representation of the data in an XML document in the target programming language. The main difference between XML and ATerm interoperability and XML data binding solutions is that the first is not related to a representation in a programming language. Adding the required explicit structure to XML is a more clear, stand-alone and language independent problem. The ATerm XML interoperability tools we have presented in this chapter do just one thing: interpret an XML document against a schema. These tools can be reused in the implementation of XML data binding solutions.

6.7.2 ATerm-XML

ATerm-XML [Hen02] has been developed by a student at the CWI, where the ATerm library has been developed at well. As far as we know this package is not really used in practice. This package offers two tools: xml2aterm and aterm2xml. The rewriting of xml2aterm is almost equivalent to `parse-xml-info | xml-info2data`. The other way around is equivalent to `data2xml-info | pp-xml-info`. The ATerm representation of an XML document as produced by xml2aterm is thus not structured. The tools do not apply any schema knowledge. Also this package does not have representations of an XML document like xml-doc and xml-info. An advantage of ATerm-XML is that the conversion tools have been implemented by using the GNOME XML library (libxml2). The performance of the tools is therefore probably much better then our SGLR and Java based solutions. We did however not test the performance of the tools.


abox2text
XML
Gorille
sglr
aterm xml bridge
xml parser
abox
xml-doc
xml-info
aterm data
dtd
relax ng
xml schema
interpretation
irhg
rhg
tree grammar
irtg
rtg
signature
irtg-implode
sdf grammar
instance-check
normalize
sdf def
structured aterm
aterm tool

6.7.3 ADT/ApiGen

At the CWI the Annotated Data Type (ADT) and ApiGen [dJO02, www03] have been developed. The purpose of this work is to generate APIs that hide the generic ATerm library behind a typed interface. Hand-written code in programming languages like C and Java is in this way type-checked. When using the generic ATerm library from such a language there is no static guarantee that the terms that the program works with are valid terms of a language defined by some abstract syntax definition. The type system of the programming language is therefore to a certain extent useless. The same problem occurs when working with XML content using a generic XML library like the W3C DOM.

Relevant to our work is the ADT language. This language is used for some kind of abstract syntax definitions. These ADT definitions are generated from SDF2 syntax definitions. We have evaluated these tools, but they did not meet our requirements at that time. In some small tests we observed that the ADT definitions are not always correct because injections of strings into lexical sorts were missing. This is apparently not a problem in the tooling that generates the typed APIs, but in our tool kit incorrect or incomplete abstract syntax definitions are not going to work. Also, for our purposes, the SDF2 syntax definition was normalized too much. Structure that is important in our tool kit was lost because the distance to the original syntax definition has become too large after applying the normalizations.

6.8 Conclusion and Future Work

We have achieved interoperability of ATerm and XML software tools. From the ATerm point of view the ATerm tools can now accept data in the the XML language because they are able to transform it to the structured ATerm representation they are accustomed to work with. For this a definition of the XML language must be available in some form and there must be tools to convert this definition into a regular hedge grammar. The XML document is interpreted against the regular hedge grammar to make structure, that was implicit in the XML document, explicit in a structured ATerm representation.

From the XML point of view the XML tools can invoke ATerm tools as if the ATerm tools return data in the XML language. Making this possible was however not really a big challenge because in the conversion from ATerm to XML only explicit structure is lost. There are no schema aware tools involved and a plain invocation of `data2xml-info | pp-xml-info` will do the job.

As a side-effect of this project we have developed a more formal and concise way of working with abstract syntax definitions in Stratego/XT program transformation systems. The RTG and RHG languages and the related tools will be applied in program transformation systems that have nothing directly to do with XML . The formalization of abstract syntax definitions opens new opportunities for meta abstract syntax definition programs. Stratego signatures were usually incorrect because of the poor implementation of the generation of Stratego signatures from SDF2 concrete syntax definitions. Because of these incorrect Stratego signatures few meta program generators, which generate code for a certain language defined in an abstract syntax, have been implemented and the tools that have been implemented are rarely used because they require hand written signatures. For example,

format checkers used to be written by hand for a certain language. Now there is a generic well-formedness checker that verifies that an ATerm is an instance of a RTG .

Also we have given a comprehensive overview of hedge and tree languages and their application to XML and the ATerm format. We have implemented the tooling required for working with these formalisms as real languages in transformation systems. Most publications that apply tree or hedge language theory do not apply both formalisms and their relationship and difference has not been illustrated in such a practical way as we have done in this chapter and the preliminary introduction to these formalisms. As the core of our tools we have defined a clear mapping from regular hedge languages to regular tree languages.

But, still a lot of work remains to be done. We present some ideas for future work.

6.8.1 Ambiguous Grammars

The RTG and RHG languages allow ambiguities. There is no requirement that regular expressions in a regular hedge grammar must be 1-unambiguous or any less restrictive form of unambiguity. The implementation of XML interpretation, where xml-info2irhg is the key component, does however not handle ambiguous cases correctly. The matching of regular expressions is greedy. In practice this is not a big deal because all the languages we work with are 1-unambiguous, but having a more powerful implementation is high on our priority list because the design goal of the RTG and RHG languages is to handle the full class of regular tree and hedge grammars without any restrictions. Optimization of implementations operating on a given RTG and RHG grammar in a certain subclass is an implementation detail that must have no influence on the specification language.

The Stratego language has some interesting constructs for implementing support for ambiguities. Besides the local choice operators + (non-deterministic) <+ (prefer left), and +> (prefer right) Stratego supports *global* choice operators ++, <++, and ++> [BV02]. The local choice operators never withdraws a choice that has been made. If the left or right argument of the local choice succeeds, the other option will not be reconsidered if the continuation fails. This means that `(s1 <+ s2); s3` will fail if `s1` succeeds but `s1; s3` fails, even if `s2; s3` would have succeeded. Thus `(s1 <+ s2); s3` is not equal to `(s1; s3) <+ (s2; s3)`. The global choice operators are non-committing. If one of the options has already succeeded, but the continuation fails, the strategy backtracks to the other option. Thus for the global choice `(s1 ++ s2); s3` is equal to `(s1; s3) ++ (s2; s3)`.

The `bagof(s)` operator of Stratego produces the list of all possible results by forcing a backtrack to the last choice point. Thus, by changing all choices in our implementation into global choices, we can handle all ambiguous constructs. The implementation will then always return *an* interpretation if there is one according to the definition. By using the bagof operator we will get a set of *all* interpretations. Inspired by disambiguation filters for context-free word languages used in scannerless generalized LR parsing [vdBSVV02, KV94], such an implementation might optionally apply a set of user defined disambiguation filters to remove undesired interpretations from the set of possible ones.

6.8.2 Profitable Subclasses of Tree Grammars

When performance problems occur in the future we could implement optimized interpreters for subclasses of the regular tree grammars with profitable properties. In the current implementation we have completely ignored possible performance issues. We have not applied tree automata theory, the operational counterpart of regular tree grammars, in our implementation. We have chosen to ignore tree automata for now because accounts of tree automata usually focus on recognition of tree languages. In the interpretation of XML it is however important to know *in what way* a tree is an element of the tree language specified by a tree grammar. A lot of work has however been done in the area of (tree) automata and we definitely should be able to profit from this work.

6.8.3 Beyond Strings

The RHG and RTG languages do not allow the specification of the value of strings: a string is a built-in non-terminal. This design choice is more or less influenced by papers on the formalization of XML schema languages, which usually define a special string construct as well, often called pcdata. Also in typical Stratego program transformation systems the abstract syntax definition of a programming language does not involve the structure of the characters in a string.

The consequence of this design choice is that RTG and RHG cannot be used to define the language of applications of parameterized tree languages like AsFix, xml-doc and xml-info. It is for example not possible to define the XHTML language in an xml-info representation, or the Java language in an AsFix representation.

Too late in this project we found out that this is serious shortcoming. Although RTG and RHG were at first just intended for the interpretation of XML , these languages and tools can be applied in a much broader way. RTG and RHG will for example be used to specify the structure of input and output terms of transformation tools in a tool repository. We need to make RHG and RTG more general by considering strings to be lists of characters. The string non terminal will be a list of any character. In this extension of RHG and RTG a specific character will probably be a special terminal.

6.8.4 Built-in Non-Terminals to Built-in Production Rules

Related to the issue in string non-terminals is the way built-in non-terminals are used. Having built-in non-terminals without associated production rules causes all kinds of problems. For example, injections of such a built-in non-terminal into another non-terminal cannot be removed in the normalization of an RTG . Also the structure of a list cannot be specified because there is no way to specify the structure of parts of a tree.

This problem can be solved by letting built-in non-terminals actually refer to built-in production rules. In this case the built-in list non-terminal refers to two production rules for cons and nil constructs. A character non-terminal might refer to a large set of production rules for all the characters in a character set. In this approach the structure of lists, strings, and integers can be defined in RTG and RHG grammars.

6.8.5 Back-end Opportunities

Concrete syntax definitions in SDF2 have long served as the contracts between transformation components of Stratego/XT [dJV01]. The idea behind having concrete syntax definitions as contracts has proven to be successful because some of the infrastructure in typical Stratego/XT projects can be generated from concrete syntax definitions in SDF2 .

Most program transformation tools however exchange abstract syntax trees. Source code in a concrete syntax is immediately parsed to this abstract syntax and only at the end of the transformation the abstract syntax trees are pretty-printed to the concrete syntax of the language. Thus the only infrastructure that is concerned with the concrete syntax of a programming language is to be found at the beginning and the end of a typical composition of program transformation components. For this part of a pipeline there is also a need for an abstract syntax definition language that serves as the contract between the components that operate on some language. From such an abstract syntax definition tools and language specific code can be generated more easily because a generator is not bothered with the additional complexity of a concrete syntax definition.

Stratego signatures are the abstract syntax definition language of Stratego/XT. These signatures are however designed specifically for the Stratego language. Also signatures generated from an SDF2 concrete syntax definition are almost always incorrect as mentioned before. Now we have a separate abstract syntax definition language and a relatively good implementation of SDF2 to RTG , the grammars as contracts principle can become even more useful. Parallel to concrete syntax definition as contract principle we now have an abstract syntax definition as contract principle. This abstract syntax definition is a tree or hedge grammar, depending on the nature of the transformation system. For each tool and generator we can choose the appropriate contract layer.

6.8.6 Syntax Definition and XML Schema Languages

Currently DTD is the only XML schema language that can be translated into RHG . Of course we would like to have more implementations, especially for RELAX NG and W3C XML Schema. Implementing a rewriting of RELAX NG to RHG is not difficult. RELAX NG is a well defined and small language, especially if the rewriting operates on simplified RELAX NG as defined by the RELAX NG specification. Also it would be interesting to implement a simplification of RELAX NG in Stratego and support the Compact Syntax of RELAX NG [Cla02].

W3C XML Schema will definitely be more difficult to implement. We have been looking for simplification tools that we can reuse in our implementation. Unfortunately we have not found anything useful yet. Even Trang [Clac], which can convert almost anything in almost anything, does not allow W3C XML Schema as an input language. Implementing a high quality simplification of W3C XML Schema to some regular hedge grammar-like language seems to be a serious issue and is thus challenging.

Almost all XML schema languages allow the declaration of unknown content. RTG and RHG do not allow this: the language must be defined completely in the grammar. Currently we have no intention to provide constructs for this because it does not fit in the basic formalisms of hedge and tree grammars and we do not want to pollute our languages with new ‘features’. If these features are required for a certain application then RTG an

RHG must not be used. This will be a problem in the transformation of XML schemas to RHG . Probably implementations that convert schemas to RTG or RHG will disallow schemas that allow undefined content.

Other interesting front-ends are the data type definitions of programming languages. The transformation of Stratego signatures to RTG and vice versa is in this category. We would like to implement these transformations for Haskell data types and generate an additional (de)serialization of Haskell data in the ATerm format. This work is related to the Haskell ATerm Library [2].

[2] Available at
http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/HaskellATermLibrary

Chapter 7

XML Transformations using Stratego

7.1 Introduction to Stratego

Stratego [Vis01b, VBT98, www] is a transformation language that is being used for the implementation of complex transformation systems. Most of them are program transformations, for example compilers, optimizers, and documentation generators. Stratego is based on the paradigm of term rewriting with programmable rewriting strategies. The main ideas of Stratego are: (1) terms for data representation, (2) rewrite rules to define basic transformation steps, and (3) rewriting strategies to control the application of these rules.

Terms are used to represent the abstract syntax tree of a program or other structured data. In Stratego terms are encoded in the ATerm [dBdJKO00] format. Implementing a transformation system in Stratego requires an ATerm representation of the data that is to be transformed. Stratego started off as a program transformation language and thus transformation systems that have been implemented in Stratego usually transform or generate code in some programming language. The data transformed in such a system is then an abstract syntax tree representation of the program.

Rewrite rules define the basic transformation steps of terms into other terms. A rewrite rule consists of two parts: a term pattern to transform (left hand side) and a new term pattern to replace the old one (right hand side). If a term matches the left hand side pattern of a rewrite rule then it can be replaced by an instantiation of the term pattern at the right hand side of the rule. This is the basic transformation step in Stratego. Rewrite rules are however not atomic in Stratego. They are defined in terms of first-class pattern matching, pattern instantiation, and variable scope. These constructs are the primitives of transformation and are used in several other language constructs, like congruences, which apply strategies to specific terms in a term pattern.

Rewriting strategies control the application of rewrite rules. A rewriting strategy and the selection of rules to be used in a specific transformation are determined by the programmer using a strategy language based on a small set of fundamental combinators. Stratego has strong support for defining generic transformations using the generic traversal combinators `all`, `one`, and `some` to decend one step in a term. These combinators can be used to define highly generic strategies. The separation of rules and strategies allows their reuse in different transformations. The paradigm of term rewriting with programmable rewriting strategies is useful for implementing complex transformations as has been illustrated by the

kinds of transformations that have been developed concisely using Stratego.

7.2 Why Implement XML Transformations in Stratego

Stratego has several interesting features that are desirable in a language for implementing XML transformations: strategic programming (strategies and rules), generic traversals, and concrete object syntax. Several XML transformation languages have been proposed [HP00, BCF03, MS03a, MS03b] recently, but they provide limited support at these points as will be discussed in the related work section.

The strategic rewriting paradigm separates traversal control from rewrite rules. This allows the basic transformation steps to be defined concisely. The separation of strategies and rewrite rules makes rewrite rules reusable in different transformations. XSLT for example combines traversal control and basic rewriting in templates. Templates in XSLT rewrite XML content to other content *and* control how and where to proceed.

The generic traversal combinators allow the definition of generic traversal strategies. These strategies can be applied in transformations over any type of term. The Stratego library contains a large set of generic traversal strategies, which are defined in Stratego itself. Many generic traversal strategies can be customized by providing additional strategy arguments beyond a set of rules to apply.

Using the concrete object syntax facilities of Stratego, term patterns can be written in the concrete syntax of a language. Any object language can easily be embedded in the Stratego meta language. This makes term patterns much more clear if the meta programmer is familiar with a concrete syntax for the object language that is being manipulated in a Stratego program.

Stratego is an untyped language. This might sound odd as an advantage. Almost all proposals for an XML manipulation language that have been proposed focus on the type system of the language, which consists of a type system for XML and a type system for the constructs to manipulate XML . These type systems do not allow 'any' content, which XML schema languages such as RELAX NG, W3C XML Schema, and DTD do. Forcing the language of an XML document to be fully defined is a mismatch with the way many XML languages work. Because XML has no type system, unknown content can be handled. Only in the structured representation of an XML document a language definition is required.

The advanced transformation facilities of Stratego allow the specification of more complex XML transformations in an XML manipulation language, where more complex XML transformations, like for example RELAX NG simplification, are often not implemented in XML transformation languages but in general purpose languages like Java.

7.3 Overview

Implementing XML transformations in Stratego consists of two challenges:

1. The XML exchange format differs from the ATerm format. In particular, XML documents are less structured.

2. The data being exchanged is different. The ATerm format is mostly used for exchanging program representations. The XML language is used for documents and structured data.

In order to transform XML documents in Stratego there must be an ATerm representation of the XML document. In the previous chapters we have presented three different representations: xml-doc, xml-info, and structured terms. These representations can all be transformed in Stratego, because the representations are just ATerms.

Transforming the xml-doc and xml-info representations in the ATerm syntax is however not very convenient. The xml-doc and xml-info representations are very verbose. Even simple XML documents will result in enormous xml-doc and xml-info terms. Implementing generic XML transformations using these representations is still quite feasible because there are usually no large XML fragments involved in such a transformation. If larger XML fragments are involved, the xml-doc and xml-info representations become increasingly annoying. This situation is comparable to constructing larger DOM-like representations of an XML document in a language like Java. Generating large amounts of XML by creating a DOM is very verbose, while writing XML processing code where no large XML fragments are involved is feasible.

The issue of writing XML in xml-doc and xml-info is solved by using the concrete object syntax facilities of the Stratego language and additional processing of the concrete object fragments. This method allows the XML syntax inside Stratego code, where the underlying representation is however xml-doc or xml-info. By using one of the embeddings of the XML syntax in Stratego, XML representations can be manipulated by using the actual XML syntax. While Stratego code transforms xml-doc or xml-info.

The structured ATerm representation does not have this problem. In fact it is even more compact then the original XML document, as is illustrated by many of the examples in this thesis. If this representation is chosen, implementing XML transformation in Stratego is almost nothing unusual. Because of its additional structure this is in many cases the most appropriate representation for transforming XML in Stratego.

7.4 Concrete XML Syntax for xml-doc

Any language for which there is an SDF2 syntax definition can be embedded in Stratego using the concrete object syntax facility [Vis02]. Embedding the XML syntax for the xml-doc representation syntax in Stratego is just an application of the tools developed for the concrete object syntax facility and is therefore easy to implement. For applying the concrete object syntax facilities of Stratego only an SDF2 syntax definition has to be created that combines the object language, which is XML , and the meta language, that is Stratego . The purpose of this new syntax definition is to extend the Stratego language to allow XML syntax at all places where a term is allowed in the basic Stratego language. Thus for a Stratego term this new syntax definition also allows XML constructs.

This embedding of an object language into the meta language is completely user definable. Every embedding can choose its own quotation, anti-quotation and disambiguation notation. Most embeddings in Stratego use |[...]| for quotation, ~ for anti-quotation, and meta-variables for variables of the meta language in the concrete object fragments . An embedding of a certain object language is however free to choose symbols that are most appropriate

and visually attractive. It is even allowed to use no explicit symbols at all. In this case their might however occur ambiguities, which have to be solved. The embedding of XML in Stratego uses the template style proposed in [vW03]. In this template style the %> and <% symbols are used for quotation, that is switching from Stratego to XML , and <% and %> for anti-quotation, this is switching from XML back to Stratego. In this style a quotation with an anti-quotation looks in a striking way like two independent quotations. This is because the starting symbol of an anti-quotation is the same symbol as the last symbol of a quotation and the last symbol of a anti-quotation is the same symbol as the first symbol of a quotation. Although this might sound confusing this style feels very natural.

7.4.1 Underneath the Syntax

The parts of Stratego code that are written in the XML syntax are called concrete XML fragments. In general they are called concrete object fragments. In this case the object language is XML . The fragments are separated from the real Stratego code by using the meta transition markers. The meta transition markers are attached as constructors to the quotation and anti-quotation productions of an SDF2 syntax definition. For every meta language there is a set of meta transition markers, which correspond with the possible constructs of the language where concrete object syntax can be used.

The generic meta-explode tool for Stratego (discussed in [Vis02]) explodes concrete object fragments to the language constructs of the meta language. Meta-explode uses the meta transition markers to recognize what parts of the abstract syntax tree have to be exploded to the meta language.

Examples 25 and 26 illustrate how the XML syntax for xml-doc is processed by the Stratego compiler and what a concrete XML fragment actually stands for in Stratego abstract and concrete syntax.

Example 25: xml-doc in Stratego Concrete and Abstract Syntax
Before using the concrete syntax for xml-doc it is useful to see what a Stratego program that builds xml-doc terms looks like in concrete and abstract syntax. The example program that is used to illustrate the processing of Stratego with XML syntax creates the XML element

```
<h1>Tom Bombadil</h1>
```

This XML element is represented in an xml-doc term as

```
Element(
  QName(None, "h1")
, []
, [Text([Literal("Tom␣Bombadil")])]
, QName(None, "h1")
)
```

In Stratego terms are built using the `!` (build) operator. The argument of this operator is the term that must be built. The `bar` strategy in the following Stratego module builds the xml-doc representation of the XML element.

```
module foo
strategies

  bar =
    !Element(
      QName(None(),"h1")
    , []
    , [Text([Literal("Tom␣Bombadil")])]
    , QName(None(),"h1")
    )
```

This module is pure Stratego code: no syntax embeddings are used and the only thing that is required to compile this module is a signature for the terms that are used. The Stratego compiler parses this module to an abstract syntax tree in the ATerm format. The tree is shown in figure 7.1 on the following page. The details of this abstract are not important. The abstract syntax tree is just shown for comparison with the abstract syntax trees in the next example.

```
Specification(
  [ Strategies(
      [ SDef(
          "bar"
        , []
        , Build(
            Op(
              "Element"
            , [ Op("QName", [Op("None", []), Str("h1")])
              , Op("Nil", [])
              , Op(
                  "Cons"
                , [ Op(
                      "Text"
                    , [ Op(
                          "Cons"
                        , [Op("Literal", [Str("Tom Bombadil")]), Op("Nil", [])]
                        )
                      ]
                    )
                  , Op("Nil", [])
                  ]
                )
              , Op("QName", [Op("None", []), Str("h1")])
              ]
            )
          )
        )
      ]
    )
  ]
)
```

Figure 7.1: Stratego abstract syntax tree of the Stratego module in the examples 25 and 26. Note that the term in the Build term is an exploded representation of an ATerm. This is the representation of an ATerm is the Stratego language.

Example 26: XML syntax for xml-doc in Stratego
When using the embedding of XML in Stratego a term in Stratego can be replaced by XML content between the markers %> and <%. In the next Stratego module the xml-doc term of example 25 is replaced by the XML element it represents. Note that the ! operator is still used to build the term.

```
module foo
strategies

  bar =
    !%><h1>Tom Bombadil</h1><%
```

The meta transition marker associated with this quotation is `ToTerm`. This meta transition marker is used at a location in the abstract syntax tree where a Stratego term is expected, but concrete object syntax is used for this term. By convention all embeddings of an object language in Stratego use this marker is for this transition. After parsing this Stratego module, but before applying the meta-explode tool for Stratego, the abstract syntax tree of this module is:

```
Specification(
  [ Strategies(
      [ SDef(
          "bar"
        , []
        , Build(
            ToTerm(
              Element(
                QName(None, "h1")
              , []
              , [Text([Literal("Tom␣Bombadil")])]
              , QName(None, "h1")
              )
            )
          )
        )
      ]
    )
  ]
)
```

In this abstract syntax tree the term inside the `Build` in figure 7.1 is replaced by the `ToTerm` meta transition marker. `ToTerm` wraps the xml-doc representation of the concrete XML fragment.

The job of a meta-explode tool is to explode these fragments to the corresponding constructs of the meta language. An application of the meta-explode tool for Stratego to the abstract syntax tree above results in exactly the same abstract syntax tree as when no concrete XML syntax as used. This is thus the same abstract syntax tree as resulted from parsing the Stratego module in example 25, see figure 7.1.

In a Stratego program concrete syntax for an object language can be freely mixed with the abstract syntax representation of the object language in Aterms. This is possible because the concrete syntax of the object language is just a different syntax for the same ATerm . XML syntax for xml-doc can thus be mixed with the abstract syntax, in ATerms, for xml-

doc. This is useful because in some circumstances using the XML syntax is more confusing then helpful.

The xDoc documentation generator [Ver04] for Stratego shows for concrete object syntax fragments the plain Stratego representation of these fragments in a code browser. This is a great tool if a meta programmer does no longer understand what is going on behind the sometimes deceiving clear syntax.

7.4.2 Using XML Syntax for xml-doc in Stratego

Note that these productions are not really the productions of the SDF2 syntax definition of XML in Stratego .

Term Quotation

`"%>" Document "<%" -> Term`
: Document, whitespace is not relevant

`"%>" Content "<%" -> Term`
: Content, whitespace is relevant

`"%>" Content* "<%" -> Term`
: List of Content, whitespace is relevant

`"%>" Content* "<%" "::" "*" -> Term`
: List of Content. Explicit disambiguation. Whitespace is relevant.

`"@>" Attribute "<@" -> Term`
: Attribute

`"@>" Attribute* "<@" -> Term`
: List of attributes

Congruence Quotation

`"%>" Document "<%" -> Strategy`
: Document

`"%>" Content "<%" -> Strategy`
: Content, whitespace is relevant

`"%>" Content* "<%" -> Strategy`
: List of content, whitespace is relevant

Content Anti-Quotation

`"<%" Strategy "%>" -> Content`
: Apply the strategy to the current subject term and insert the result at this place. The result must be xml-doc Content.

`"<%" Strategy "::" "content" "%>" -> Content`
: Same as the previous construct, but uses an explicit disambiguation. In the case of Content anti-quotation this is seldom necessary.

`"<%=" Term "%>" -> Content`
: Insert the term between the anti-quotation symbols at this place. The term must be xml-doc Content.

`"<%=" Term "::" "content" "%>" -> Content`
: Same as the previous construct, but uses an explicit disambiguation.

```
"<%" Strategy "::" "*" "%>" -> Content*

"<%" Strategy "::" "content*" "%>" -> Content*
```

Character Data Anti-Quotation

`"<%" Strategy "::" "cdata" "%> -> CharDataText`
: Applies strategy to the current subject term. aka term wrap.

`"<%=" Term "::" "cdata" "%>" -> CharDataText`
: Term

Attribute Anti-Quotation

```
"<@" Strategy "@>" -> Attribute

"<@" Strategy "::" "*" "@>" -> Attribute*

"<@=" Term "@>" -> Attribute

"<@=" Term "::" "*" "@>" -> Attribute*
```

Attribute Value Anti-Quotation

```
"<@" Strategy "@>" -> AttValue

"<@=" Term "@>" -> AttValue
```

Attribute Character Data Anti-Quotation

```
"<%" Strategy "%>" -> DoubleQuotedText

"<%=" Term "%>" -> DoubleQuotedText
```

Name Anti-Quotation

```
"<%" Strategy "%>" -> NCName

"<%=" Term "%>" -> NCName
```

```
"<" Term -> NCName
```

7.4.3 Discussion

Transformation of XML documents in the xml-doc representation takes almost all syntactic details of an XML document into account. A transformation must only be implemented using the xml-doc representation if this actually matches the required level of detail in the transformation. If a transformation does not need to control the syntactical details of the XML document then the xml-info representation is probably more appropriate.

The problem of transforming XML in an inappropriate representation lies in the number of more or less equivalent xml-doc terms. If the required level of abstraction is higher then the level of xml-doc, then equivalent XML representation at the right level of abstraction might relate to different terms in the xml-doc representation. For an xml-info term there is a large set of xml-doc terms that will result in the same xml-info term when transformed to this representation. Pattern matches on xml-doc terms are therefore much more selective. Examples of these syntactical 'details' are the namespace notation and the use of entities. Pattern matching over xml-doc constructs involving namespace constructs will only succeed if the term it is applied uses the same namespace notation. Having the facility to transform XML at this level of syntactical detail is however still interesting to have. Most XML processing solutions like XSLT, SAX, DOM, XDuce, and Xen do not allow processing at this level of detail.

A disadvantage of working with the xml-doc representation is that xml-doc terms are not guaranteed to have a related well-formed XML document as already has been discussed in section 5.3 on page 41. Obviously there is absolutely no guarantee that the result is valid with respect to some schema.

7.4.4 Application

Besides the more syntactical XML transformations, where xml-doc is especially useful for, the approach is also sufficient for the generation of XML . In particular it is in the Stratego/XT project already used in various projects for the generation of XHTML.

- The xDoc documentation generator [Ver04] generates API documentation and code browsers in XHTML.
- Various dynamic XHTML websites have been implemented in Stratego. Using the stratego-net [Brad] package the Stratego programs run as CGI programs in a web-server.
  - XWeb, a generic interactive transformation demo application [1] written for the program transformation course at the Utrecht University. This instance of the demo application uses components of the Tiger in Stratego project [Vis], but any component based program transformation system can easily be plugged in.
  - The samples-net-xml package [2] illustrates the application of the xml-tools and stratego-net packages by a set of simple hello world like services.

[1] Available at http://catamaran.labs.cs.uu.nl
[2] Available at http://catamaran.labs.cs.uu.nl

– An educational SQL to relational algebra transformation site. The visualization is using MathML in XHTML, which can be viewed in modern web browsers like Mozilla.

7.4.5 Tool Support

parse-stratego-xml-doc
: Parses a Stratego module with concrete XML fragments using a parse table generated from the Stratego-xml syntax definition. In practice this tool is only invoked by other tools that will be introduced later. Invoking this tool is not necessary because Stratego programs using the XML in Stratego syntax (or any other concrete object syntax) can specify this in a meta file for this module. Because of this meta file all Stratego programs with concrete object syntax can be compiled in the same way.

stratego-xml-doc-strip-whitespace
: Removes whitespace (without schema knowledge) from a Stratego module with concrete XML fragments, which are represented in xml-doc in the abstract syntax tree before meta explosion. This tool does to a mixture of Stratego and xml-doc what xml-doc-strip-whitespace does to a plain xml-doc term. All character data literals that just contain whitespace characters are removed.

7.5 Concrete XML Syntax for xml-info

In the xml-info representation (see section 5.4) many syntactical details, which are sometimes irrelevant, are no longer present in the representation of the XML document. The actual namespace notation is no longer relevant, comments have been removed and entities have been rewritten to the XML content they refer to. If these details *are* relevant the program must not use the xml-info, but the xml-doc representation If a developer decides that the xml-info representation is most appropriate for implementing a certain XML transformation then he would still like to use the XML syntax in the program. Although xml-info is more compact then xml-doc, it is still tiresome to write xml-info terms by hand. If the concrete XML syntax in Stratego only works for xml-doc then making an objective choice out of the possible representations is not possible

In the previous section the embedding of XML in Stratego only required an XML syntax definition in SDF2 . This way of embedding XML in Stratego will no longer do the complete job. The concrete XML fragments are parsed to the xml-doc representation. An additional transformation is required to let the concrete XML syntax stand for an xml-info representation. This transformation rewrites these xml-doc terms between the meta transition markers to xml-info. After this transformation the xml-info fragments still have to be exploded to the meta language. Because these fragments are however still separated from the Stratego code by the meta transition markers, the generic *meta-explode* for Stratego can handle this.

There is thus no requirement that the terms in meta transition markers are the direct result of parsing concrete object syntax: any term is allowed, created by any means. In general for any object language it is possible to perform any transformation on the object fragments. Applying such a tool in the compilation process is at the time of writing this not yet supported by the Stratego front-end, which is implemented in the stratego-front package. Therefore we have created a new composition of components of the stratego-front

package and apply the xml-doc to xml-info transformation before doing the meta explosion to Stratego.

Example 27: xml-info representation
To get insight in how the concrete XML syntax for xml-info actually works it is again useful to see how the abstract syntax tree of a plain Stratego program that constructs xml-info terms using the ATerm syntax looks like. The example program creates the same XML as example 25:

```
<h1>Tom Bombadil</h1>
```

The xml-info representation of this XML element is:

```
Element(Name(None, "h1"), [], [Text("Tom␣Bombadil")])
```

This xml-info term is built by the strategy in this module:

```
module foo
strategies

  bar =
   !Element(Name(None, "h1"), [], [Text("Tom␣Bombadil")])
```

The Stratego abstract syntax tree after processing the source file with the Stratego front-end is:

```
Specification(
  [ Strategies(
      [ SDef(
          "bar"
        , []
        , Build(
            Op(
              "Element"
            , [ Op("Name", [Var("None"), Str("h1")])
              , Op("Nil", [])
              , Op(
                  "Cons"
                , [Op("Text", [Str("Tom␣Bombadil")]), Op("Nil", [])]
                )
              ]
            )
          )
        )
      ]
    )
  ]
)
```

Note that the child of the `Build` term (the abstract syntax representation of the ! operator) is again the exploded representation of the xml-info term in Stratego abstract syntax.

Example 28: xml-info representation
If in a program the concrete XML syntax stands for xml-info, the syntax is just the same as in the case of xml-doc. The same concrete XML syntax stands for for different representations. The variant with concrete XML syntax of the Stratego program in example 27 is thus exactly the same as the program with concrete XML syntax in example 26 on page 101. For the sake of clearness we repeat it.

```
module foo
strategies

  bar =
    !%><h1>Tom Bombadil</h1><%
```

Because this is exactly the same module as in example 26 and i the same syntax definition is used, the abstract syntax tree directly after parsing is equivalent to the abstract syntax tree of example 26. This abstract syntax tree thus contains fragments with xml-doc. By applying process-stratego-xml-info the xml-doc fragments are rewritten to xml-info. The result is an abstract syntax tree with xml-info constructs:

```
Specification(
  [ Strategies(
      [ SDef(
          "bar"
        , []
        , Build(
            ToTerm(
              Element(Name(None, "h1"), [], [Text("Tom␣Bombadil")])
            )
          )
        )
      ]
    )
  ]
)
```

The generic meta-explode tool is used to explode the xml-info constructs to the meta language. This results is the abstract syntax tree in example 27.

In the xml-info representation every name of an element or attribute contains its complete namespace URI. The transformation of xml-doc terms to xml-info has to determine what namespace to use for all the xml-info attribute and element names. In the xml-doc representation this information is available in the way it is in XML : default namespace attributes, prefix declarations, and the use of prefixes in element names. The transformation uses this information to determine the namespace of element and attribute names.

A Stratego module usually contains a number of concrete XML fragments. The default XML namespace attributes and XML namespace prefix declarations are only used in the fragment in which they are specified. This means that in every concrete XML fragment the namespace must be specified if there is one. To prevent these tiresome declarations for each fragment there is a minor extension of the XML in Stratego syntax for programs that use XML syntax for xml-info in Stratego. This Stratego-xml-info language allows the declaration of the default namespace and namespace prefixes for a complete Stratego module. These declarations then apply to all concrete XML fragments as if they declared

in a parent element of the fragment. The namespace declarations are the only difference with the Stratego-xml language. The production for this is:

```
context-free syntax
  "module" StrategoModName Attribute+ StrategoDecl*
      -> StrategoModule {cons("Module")}
```

Example 29: Stratego XML namespace extension
The following Stratego module uses the namespace extension to declare a default namespace and namespace prefix:

```
module foo
  xmlns="xhtml-uri"
  xmlns:xsl="xsl-uri"

strategies

  bar =
    !%><ul><xsl:apply-templates/></ul><%
```

After parsing this module against the Stratego-xml-info syntax definition and processing the xml-doc terms to xml-info the abstract syntax tree is:

```
Specification(
  [ Strategies(
      [ SDef(
          "bar"
        , []
        , Build(
            ToTerm(
              Element(
                Name(Some("xhtml-uri"), "ul")
              , []
              , [Element(Name(Some("xsl-uri"), "apply-templates"), [], [])]
              )
            )
        )
      )
    ]
  )
]
)
```

7.5.1 Tool Support

parse-stratego-xml-info
: Parses a Stratego module against the Stratego-xml-info grammar. It does not perform any transformation of the concrete XML fragments.

process-stratego-xml-info
: Rewrites the XML fragments that have been written in concrete xml syntax, and are now in the xml-doc representation, to xml-info. The result still contains the meta transition markers. The content of the fragments has only been rewritten to the xml-info representation.

Although we do not discuss the concrete implementation of the various tools we have developed for this thesis, the implementation of process-stratego-xml-info is interesting to mention. Many transformations on concrete object fragments will have the same structure. Currently such transformations are not implemented often in the Stratego/XT project, yet this will change when the Stratego compiler has the facilities to plug such transformations at the appropriate places into the pipeline of the Stratego compiler. Performing a transformation on concrete object fragments at the moment still involves creating a completely new pipeline of the required components.

The implementation of this module is shown in figure 7.2. The program performs an alltd traversal over the abstract syntax tree, where it applies *xml-fragment2xml-info* to all the outermost meta transition markers. These markers are: ToTerm, ToBuild, and ToStrategy. They all indicate a different quotation from Stratego to the object language. The fragments are then rewritten using an topdownS traversal, which stops the rewriting at an anti-quotation marker. Anti-quotation markers indicate the transition from the object language, which is xml-doc in this case, to the meta language, that is Stratego. The meta transition markers are: FromTerm, FromStrategy, and FromApp. The rewrite rules of the *xml-doc2xml-info* tool (discussed in section 5.3 on page 39) are used to rewrite the xml-doc terms in the concrete XML fragments to xml-info. This makes the implementation of *process-stratego-xml-info* beautiful and small.

7.5.2 Discussion

Although transforming xml-info is suitable for a lot of XML transformations there are some issues. The developer must be careful to choose the right representation. If the syntax of an XML document must be controlled, the xml-doc representation must be used. On the other side a more abstract representation is necessary in some cases because of a lack of structure in the XML document. Examples of this are optional content and repeated occurrences of content. The structured ATerm representation presented in the previous chapter solves this problem. Of course a Stratego program can use this structured ATerm representation directly in a Stratego program, but in some cases this is not convenient. Also there is still no guarantee that the result is valid with respect to some schema. Schemas are not even used by the implementation, which in fact also counts as an advantage. These issues will be discussed in the next section.

In the xml-info representation there are no XML entities. All text in the xml-info representation is represented in ATerm strings. Apart from the standard XML entities for special characters (`<` and `&`), entities often refer to more exotic characters outside of the ASCII range. This is a problem because the ATerm format is unfortunately limited to characters in the ASCII range. There are no escape sequence for Unicode characters. Of course we could chose our own representation of characters on top of the ATerm format, but this would make the mapping very uncomfortable to users that are accustomed to the ATerm representation of strings. This is not just a theoretical problem: it already showed up in the implementation of a relational algebra presentation in MathML. Using the xml-doc representation with XML entities is the only solution at the moment.

```
module process-stratego-xml-info
imports xml-doc2xml-info Stratego stratego-meta-markers

strategies

  stratego-xml2xml-info =
      xml-fragments2xml-info
    ; stratego-xml-apply-namespaces

  xml-fragments2xml-info =
    rec x(
      alltd(
        all-quotation(xml-fragment2xml-info)
      + Module(id, map(rewrite-Attribute), x)
      )
    )

  xml-fragment2xml-info =
      (  ?Document(_, _, _) < rewrite-Document + id )
    ; topdownS(
        try(
          rewrite-Element
        + rewrite-Attribute
        + rewrite-Text
        + rewrite-Conc
        )
      , is-anti-quotation-helper
      )

  is-anti-quotation-helper(s) =
    is-anti-quotation
    ; xml-fragments2xml-info

  stratego-xml-apply-namespaces =
    rules( DefaultNamespace:  ()   -> None()
              ResolvePrefix: "xml" -> "http://www.w3.org/XML/1998/namespace"
    )
    ; rec x (
      {| ResolvePrefix, DefaultNamespace :
          ?Element(_, _, _)
        < xml-apply-namespaces-element(x)
        + try(remove-stratego-xmlns-decl)
          ; all(x)
      |}
      )

  remove-stratego-xmlns-decl :
    Module(name, xmlatts, decls) -> Module(name, decls)
      where <collect-namespace-decls> xmlatts
```

Figure 7.2: Implementation of processing xml-doc in a Stratego program to xml-info. Note that the implementation of xml-doc2xml-info is reused. This implementation is not shown here.

7.6 Structured ATerm Transformation

The approaches in the previous sections still transform the actual XML document. For some applications this is fine, but for transformations that can abstract from the encoding in XML document a representation in a structured ATerm (see chapter 6) is more attractive because Stratego has been designed to transform such structured ATerm . An xml-info representation is also a structured ATerm , but it represents an XML document in a structured way, not the data of the document. Using plain xml-doc and xml-info representations will result in a lot of list processing, for which Stratego currently has no special purpose syntax.

If an XML document is converted to a representation in a structured ATerm then transforming XML is nothing special anymore. The term patterns can be written in the usual term syntax. A problem with this is that the programmer must know the structure of the 'structured representation' of the language.

Example 30: Structured ATerm Transformation
The strategy in the following Stratego module generates the structured representation of the XHTML content that was generated in the previous examples as well.

```
module foo
imports xhtml
strategies

  bar = !h1(["Tom␣Bombadil"])
```

This module uses no syntax extensions of the Stratego language. It does require a signature of the XHTML language, which can be generated using the schema conversion tools presented in the last section. In this example the Stratego compiler also checks whether the term applications have the correct number of children or not.

The following client application invokes the example statename XML-RPC service at UserLand. The XML-RPC call is created in the structured ATerm representation of XML-RPC.

```
module statename-client
imports xml-rpc-client options

strategies

  io-statename-client =
    option-wrap(general-options, xtc-output(<statename> 40))

  statename =
    !methodCall(
       methodName("examples.getStateName")
     , params(
        [ param(value(i4(<id>))) ]
       )
     )
   ; xml-rpc(!URL("http://betty.userland.com/RPC2"))
```

The following set of simple rewrite rules are taken from an implementation that maps XML-RPC data types to an ATerm. The rules illustrate how implementing rewrite rules on the structured ATerm representation of an ATerm is not different from a usual set of rewrite rules in Stratego.

```
  xml-rpc-implode: value(struct(members)) -> ""#(members)
  xml-rpc-implode: value(array(data(vs))) -> vs
  xml-rpc-implode: value(int(x)) -> x
  xml-rpc-implode: value(i4(x)) -> x
  xml-rpc-implode: value(string(s)) -> s
  xml-rpc-implode: value(s) -> s
      where <is-string> s
```

The xml-tools package also contains support for interpreting concrete XML fragments to a structured ATerm representation. This implementation interprets XML embedded in Stratego. The implementation of this interpretation is interesting because the interpreter has to handle all the kinds of anti-quotations. This implementation is however not used in practice because this kind of XML embedding has little advantages over the use of structured

ATerms.

7.7 Related Work

7.7.1 CDuce

CDuce[BCF03] is a typed functional programming language that has been designed to manipulate XML , but can be used as a more general purpose language than XDuce , which inspired the design of CDuce. It has been designed specifically for XML and this is reflected by some neat features for manipulating XML . Strings are sequences of Unicode characters and therefore it is possible to apply the pattern matching constructs that are used for sequences of XML content to strings. XML elements are a constructed type and therefore generic programming is possible to a certain extent. Also CDuce has some interesting pattern matching constructs for attributes. Attributes are represented by open or closed record types. A closed record type does not allow any other entries then the ones specified in the record type. An open record also allows other fields. Optional record fields can be used in both record types to declare that if the field occurs then it must have a certain type.

The major difference between Stratego and CDuce is that in CDuce pattern-matching is not first-class like in Stratego. CDuce pattern-matching in comparable to the constructs of Haskell and ML and thus it combines several things in one language construct:

- the evaluation of an expression to be matched,
- all the patterns that this expression should be matched with,
- the scope of the variables in these patterns, and
- the continuation for each of the patterns

In Stratego such a construct is a composition of the more atomic language constructs of pattern matching, pattern instantiation and variable scope. These constructs can be combined at will into more abstract language constructs or can be applied directly by a programmer in specific situations.

Also CDuce has just limited support for implementing generic transformations and traversal. CDuce is not based on strategic rewriting and does therefore not promote the separation of rewrite rules, the basic transformation steps, and a strategy that controls the applications of the rules. To a certain extent it is possible to apply this strategic programming in CDuce, but it does not allow the specification of generic traversal strategies like Stratego. The capability to define such generic traversal is an essential feature that manipulates XML like data because the data is not always as structured as we would like it to be.

7.7.2 XML Tools for Haskell and Generic Haskell

HaXml [WR99] is a tool kit for working with XML in Haskell. Related to our work are the Haskell2Xml and Dtd2Haskell tools. Haskell2Xml translates Haskell data into an XML document and can generate a DTD that defines the language of the set of documents related to a Haskell data type. DtdToHaskell translates a DTD into an Haskell data type. The mapping of DTD to Haskell data types that is provided by these tools is comparable to our mapping of XML to a structured ATerm . Our mapping is somewhat more natural because

the ATerm format is closer to XML then Haskell data types. The ATerm format has for example a built-in facility for annotations. The Dtd2Haskell tool handles attributes a little bit less naturally. Another difference is that these tools just operate on DTDs, where our tools allow non-ambiguous regular hedge grammars. The DTD schema language just allow a local regular hedge grammar [MLM01], which is a very limited subset of regular hedge grammars. The HaXml utilities do not provide the tools to define transformations of XML documents using the XML syntax for data. An advantage of implementing an XML transformation in this way in Haskell, which is a typed higher-order functional language, is that the document is guaranteed to be valid with respect to the DTD. This feature does however not come for free: it is not possible to implement generic transformations.

When using the Generic Haskell [ACJ03] language for implementing the XML transformation this problem is solved more or less. Generic Haskell is a typed, generic, functional language that supports the definition of type-indexed programs. Generic Haskell provides type-safe access to the internal structure of data types, which enables the definition of functions that work for a range or even all data types. Generic Haskell thus combines the advantages of implementing an XML transformation that is guaranteed to produce valid XML with the abstraction and reduction of development time provided by generic programming. We are very interested in figuring out to what extent Generic Haskell supports strategic programming, that is, a separation of basic transformation rules and the control over the application of these rules. The tools described in [ACJ03] also provide a mapping of a subset of W3C XML Schema to Haskell data types. A disadvantage of using Generic Haskell is that there is still no way to use the XML syntax for the implementation of XML transformation. This is especially a disadvantage in the implementation of XHTML generating tools, where the developer typically does not want to be bothered with a structured representation of XHTML.

7.7.3 Xen

Xen [MS03b, MS03a] is an implementation of a proposal to extend object-oriented programming language with native support for XML-like data. Xen implements this idea as an extension of C#. Xen supports 'XML object literals', which is an XML syntax for objects in Xen. The XML object literals support embedded Xen expressions, which is comparable to anti-quotation in meta-programming with concrete object syntax. Xen extends the C# type system with sequence types, union types, the all type, streams (in possibly empty and nonempty variants), and an option type. These extensions allow a more natural binding of XML documents to objects in Xen than the typical XML data binding tools of this moment do. The structure of the XML language needs to be declared in Xen types.

The type system of Xen is in fact a new type system for XML , comparable to the way W3C XML Schema defines a type system for XML . Xen is however not compared to XML data binding tools, but is said to have native support for XML . In this way Xen defines yet another essence of XML [SW03]. Xen cannot handle arbitrary XML documents because it does, for example, not allow ambiguities in the definition of the XML language in Xen types. Of course XML documents that Xen cannot handle can still be manipulated using some of the standard XML APIs for .NET.

The power of the type system of Xen is not comparable to the one of Generic Haskell and therefore it does not allow the definition of generic traversal strategies. The different

embeddings of XML in Stratego offer a layered set of XML representations. Developers can choose what the essence of XML is to their application.

7.8 Conclusion and Future Work

The concrete syntax facilities of Stratego can be used to transform XML documents in the xml-doc and xml-info representations in a concise way. The tools have already proved to be successful in the implementation of dynamic XHTML documents for the web. Some of the generic XML processing tools presented in the last chapter also use concrete XML syntax.

Implementations of transformations of the structured ATerm representations of an XML document have been somewhat lagging behind because the structured representation and the required schema conversion tools have been implemented later in the project. Yet, also in this case some of the tools them self apply this way of transforming XML . DTDs are transformed to signatures in a structured ATerm representation of the DTDinst format as has been presented in the schema conversion section (6.5). The RELAX NG to RHG conversion will also be implemented on a structured ATerm representation of RELAX NG. Some XML-RPC demo services have been implemented, which are also based on the structured ATerm representation of XML-RPC. Yet, in the area of structured ATerm representation we need to implement more tools to illustrate the applicability of Stratego to XML transformations in this way.

7.8.1 Improved Annotation Matching

Annotations in term patterns are somewhat inconvenient currently. If annotations occur in a term pattern at a certain term then the actual list of annotations in the input term must exactly match these annotations. By exactly we mean really exactly: the annotations must occur in the same order and no other annotations are allowed. This is inconvenient because in the ATerm format as well as in XML the order of annotations is not significant and a programmer typically wants to allow other annotations in a term. This must thus be improved to allow more liberal matching of annotations. The approach of closed and open record types in CDucemight serve as a source of inspiration.

7.8.2 Improve List Matching of Stratego

In xml-info and xml-doc transformations all children of an element are in a list. This situation is comparable to the representation of XML in language like CDuceand Xen. These language have powerful pattern matching for lists by matching elements in a list against a regular expression pattern. Typical Stratego transformations operate on terms where the structure typically expressed in such regular expression patterns is already explicitly available in the term itself. Term pattern can in this case be used to match these structured terms. When transforming XML documents in the xml-doc and xml-info more powerful facilities for list matching would be useful. Stratego currently has no serious support for list matching although some very simple patterns like `x,xs*` can already be used. Although the transformation should probably be implemented using a structured ATerm representation, having better support for list matching is on our to do list.

7.8.3 Disambiguation and Interaction with Type Checker

Meta programming with concrete object syntax [Vis02] is used a lot in recent Stratego/XT applications. These applications have led to some new insights in the application of meta programming with concrete object syntax. In many cases the embedding of an object language in Stratego requires disambiguation constructs because the concrete fragments can be parsed as more then one non-terminal. These disambiguation constructs have also been used in the embedding of XML in Stratego. The `cdata`, `content*`, and `::*` literals in the embedding are examples of this. When this explicit disambiguation is required is not always clear and excessive use of the disambiguations makes the code less attractive.

An interesting solution to this problem is to solve the ambiguities by means of interaction with a type checker. If the meta language is typed, which Stratego is not, then the type checker can in many cases solve the ambiguity. By preserving the ambiguities until the type checking phase, which SGLR supports, they can be handled at this point.

Bibliography

[ACJ03] Frank Atanassow, Dave Clarke, and Johan Jeuring. Scripting xml with generic haskell. Technical Report UU-CS-2003-023, Institute of Information and Computing Sciences, Utrecht University, 2003.

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: an xml-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 51–63. ACM Press, 2003.

[BHL99] Tim Bray, Dave Hollander, and Andrew Layman. *Namespaces in XML*. World Wide Web Consortium, January 1999. http://www.w3.org/TR/REC-xml-names/.

[BM] David Brownell and David Megginson. SAX, simple API for XML website. http://www.saxproject.org.

[BMW] Anne Bruggemann-Klein, Makoto Murata, and Derick Wood. Regular tree and regular hedge languages over unranked alphabets: Version 1. http://citeseer.nj.nec.com/451005.html.

[Bor00] Jonathan Borden. *The Resource Directory Description (RDDL) for the XSet namespace*. The Open Healthcare Group, 2000. http://www.openhealth.org/XSet/.

[Bou] Ronald Bourret. XML data binding resources. http://www.rpbourret.com/xml/XMLDataBinding.htm.

[Boy01] John Boyer. *Canonical XML version 1.0*. World Wide Web Consortium, March 2001. http://www.w3.org/TR/xml-c14n.

[BPSM00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. World Wide Web Consortium, October 2000. http://www.w3.org/TR/REC-xml.

[Braa] Martin Bravenboer. ATerm database interface website. http://www.stratego-language.org/Stratego/ATermDatabaseInterface.

[Brab] Martin Bravenboer. java-front for Stratego/XT website. http://www.stratego-language.org/Stratego/JavaFront.

[Brac] Martin Bravenboer. Samples for xml-tools and stratego-regular website. http://www.stratego-language.org/Stratego/SamplesNetXml.

[Brad] Martin Bravenboer. stratego-net for Stratego/XT website. http://www.stratego-language.org/Stratego/StrategoNetworking.

[Brae] Martin Bravenboer. stratego-regular for Stratego/XT website. http://www.stratego-language.org/Stratego/StrategoRegular.

[Braf] Martin Bravenboer. xml-tools for Stratego/XT website. http://www.stratego-language.org/Stratego/XmlTools.

[Bra02] Martin Bravenboer. Making xt xml capable. http://www.students.cs.uu.nl/ mbravenb/docs/xml-pem-2002.pdf, 2002. Talk at CWI Programming Environment Meetings.

[dBdJKO00] M. G. J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[dBHdJ+01] M. G. J. van den Brand, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J.S. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The Asf+Sdf Meta-Environment: a component-based language laboratory. In R. Wilhelm, editor, *Compiler Construction (CC'01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–368, Genova, Italy, April 2001. Springer-Verlag.

[BV01] Martin Bravenboer and Eelco Visser. Guiding visitors: Separating navigation from computation. Technical Report UU-CS-2001-42, Institute of Information and Computing Sciences, Utrecht University, 2001.

[BV02] Martin Bravenboer and Eelco Visser. Rewriting strategies for instruction selection. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 237–251, Copenhagen, Denmark, July 2002. Springer-Verlag.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at http://www.grappa.univ-lille3.fr/tata, 1997.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, IT-2:3:113–124, 1956.

[Claa] James Clark. DTDinst. Thai Open Source Software Center. http://www.thaiopensource.com/relaxng/dtdinst/.

[Clab] James Clark. RELAX NG home page. http://www.relaxng.org/.

[Clac] James Clark. Trang, multi-format schema converter based on RELAX NG. Thai Open Source Software Center. http://www.thaiopensource.com/relaxng/trang.html.

[Cla02] James Clark. *RELAX NG Compact Syntax Specification*. Organization for the Advancement of Structured Information Standards (OASIS), November 2002. http://www.relaxng.org/compact-20021121.html.

[Cla03] Kendall Grant Clark. The long, long arm of SGML. http://www.xml.com/pub/a/2003/11/05/deviant.html, November 2003.

[CM01] James Clark and Murata Makoto. *RELAX NG Specification*. Organization for the Advancement of Structured Information Standards (OASIS), December 2001. http://www.relaxng.org/spec-20011203.html.

[Cou89] Bruno Courcelle. On recognizable sets and tree automata. In Hassan Ait-Kaci and Maurice Nivat, editors, *Algebraic Techniques*, volume 1, chapter 3, pages 93–126. Academic Press, 1989.

[CT01] John Cowan and Richard Tobin. *XML Information Set*. World Wide Web Consortium, October 2001. http://www.w3.org/TR/xml-infoset.

[dJ00] Merijn de Jonge. A pretty-printer for every occasion. In Ian Ferguson, Jonathan Gray, and Louise Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.

[dJO02] Hayco de Jong and Pieter Olivier. Generation of abstract programming interfaces from syntax definitions. Technical Report SEN-R0212, Centrum voor Wiskunde en Informatica (CWI), 2002.

[dJV01] Merijn de Jonge and Joost Visser. Grammars as contracts. In Greg Butler and Stan Jarzabek, editors, *Generative and Component-Based Software Engineering, Second International Symposion, GCSE 2000*, volume 2177 of *Lecture Notes in Computer Science*, pages 85–99, Erfurt, Germany, October 2001. Springer.

[Dod00] Leigh Dodds. Investigating the infoset. http://www.xml.com/pub/a/2000/08/02/deviant/infoset.html, August 2000.

[Hen02] Kasper Hendriks. aterm-xml website. http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ATermXML, 2002.

[HK00] T. Eisenbarth H. Kienle, J. Czeranski. Exchange format bibliography. In *Workshop on Standard Exchange Format (WoSEF)*, pages 2–9, Limerick, Ireland, June 2000.

[HP00] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. In *Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, 2000.

[Jel] Rick Jelliffe. *Schematron, A language for making assertions about patterns found in XML documents.* Academia Sinica Computing Centre. http://www.schematron.com.

[JS01] Johan Jeuring and Doaitse Swierstra. *Grammars and Parsing*. Utrecht University, 2001. ps.

[KP76] Brian W. Kernighan and P.J. Plauger. *Software Tools*. Addison-Wesley, 1976.

[KV94] Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy, October 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

[KV01] Tobias Kuipers and Joost Visser. Object-oriented tree traversal with JJForester. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA).

[Lau] Simon St. Laurent. Gorille, XML unicode character checking and markup parsing. http://simonstl.com/projects/gorille/.

[Lau99] Simon St. Laurent. Toward a layered model for xml. http://simonstl.com/articles/layering/layered.htm, 1999.

[Lau03] Simon St. Laurent. What can you do with half a parser? In *Extreme Markup Languages*, Montreal, Canada, 2003. [html, pdf].

[LMM00] Dongwon Lee, Murali Mani, and Makoto Murata. Reasoning about XML schema languages using formal language theory. Technical Report RJ#10197, Log#95071, IBM Almaden Research, November 2000. [pdf].

[Mar01] Jonathan Marsh. *XML Base*. World Wide Web Consortium, June 2001. http://www.w3.org/TR/xmlbase/.

[Max] Jon A. Maxwell. Netx, an open-source jnlp client. http://jnlp.sourceforge.net.

[MET78] M.D. McIlroy, E.N.Pinson, and B.A. Tague. Unix time-sharing system forward. *The Bell System Technical Journal*, 57(6), 1978.

[MLM01] Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.

[MS03a] Erik Meijer and Wolfram Schulte. Programming with rectangles, triangles, and circles. In *Proceedings of XML Conference & Exposition 2003 (XML 2003)*, December 2003.

[MS03b] Erik Meijer and Wolfram Schulte. Unifying tables, objects, and documents. In *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL 2003)*, August 2003.

[Mur00] Makoto Murata. Hedge automata: a formal model for XML schemata. http://citeseer.nj.nec.com/murata99hedge.html, 2000.

[Oba03] Dare Obasanjo. Xml schema design patterns: Is complex type derivation unnecessary? http://www.xml.com/pub/a/2003/10/29/derivation.html, October 2003.

[Ogb03] Uche Ogbuji. XML data bindings in python. http://www.xml.com/pub/a/2003/06/11/py-xml.html, June 2003.

[Pro] Java Community Process. Java network launching protocol (JNLP) and API specification. http://www.jcp.org/en/jsr/detail?id=56.

[Ray03] Eric Steven Raymond. *The Art of Unix Programming*. Addison-Wesley, 2003. http://www.catb.org/ esr/writings/taoup/.

[Rit84] Dennis M. Ritchie. The evolution of the UNIX time-sharing system. *AT&T Bell Laboratories Technical Journal*, 63(6):1577–1593, October 1984.

[SW03] Jerome Simeon and Philip Wadler. The essence of XML. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13. ACM Press, 2003.

[Tho01] Henry S. Thompson. *XML Schema*. World Wide Web Consortium, May 2001. http://www.w3.org/TR/xmlschema-1/.

[TT01] Richard Tobin and Henry Thompson. *A Schema for Serialized Infosets*, May 2001. http://www.w3.org/2001/05/serialized-infoset-schema.html.

[VBT98] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

[vdB03] Mark van den Brand. Pgen, a parse table generator for sdf2. http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ParsetableGenerator, 2003.

[vdBSVV02] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.

[Ver04] Rob Vermaas. xdoc - an extendible documentation generator. Master's thesis, February 2004.

[Vis] Eelco Visser. Tiger in stratego, compilation by program transformation. http://www.stratego-language.org/Tiger.

[Vis97a] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[Vis97b] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[Vis01a] Eelco Visser. Scoped dynamic rewrite rules. In Mark van den Brand and Rakesh Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 2001.

[Vis01b] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[Vis02] Eelco Visser. Meta-programming with concrete object syntax. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[Vis03] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer et al., editors, *Domain-Specific Program Generation*, Lecture Notes in Computer Science. Spinger-Verlag, June 2003. (Accepted for publication).

[vW03] Jonne van Wijngaarden. Code generation from a domain specific language. designing and implementing complex program transformations. Master's thesis, Utrecht University, Utrecht, The Netherlands, July 2003. INF/SCR-03-29, [pdf].

[Wika] Wikipedia. Chomsky hierarchy. http://wikipedia.org/wiki/Chomsky_hierarchy.

[Wikb] Wikipedia. Formal grammar. http://wikipedia.org/wiki/Formal_grammar.

[Wikc] Wikipedia. Formal language. http://wikipedia.org/wiki/Formal_language.

[WR99] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34–9, pages 148–159, N.Y., 27–29 1999. ACM Press.

[www] Stratego website. http://www.stratego-language.org.

[www03] api-gen website. http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ApiGen, 2003.