Combinatorial Test Set Generation:

Concepts, Implementation, Case Study

Vadim Zaytsev,
Universiteit Twente, Enschede

June 22, 2004

Abstract

This project is about test data generation in a combinatorial way, with usage of specific mechanisms to control explosion. The work consists of adoption of existing concepts, description of the test data generator, application to the XML case study with XML Schema as grammar description formalism and practical usage of the tool. Actual results presented in the work show the differences and the common behaviour among three XML validators—the information that can be used to judge, choose, discard and upgrade them.

Hosting organisation: Vrije Universiteit, Amsterdam, The Netherlands
Supervisors: Dr.ing. Ralf Lämmel,
Vrije Universiteit, Amsterdam
Prof.dr. Hendrik Brinksma,
Universiteit Twente, Enschede

Note:
This Master's thesis contributes to a collaboration between Dr. Wolfram Schulte from Microsoft Research Redmond, Foundations of Software Engineering (FSE) group and Dr. Ralf Lämmel from Centrum voor Wiskunde en Informatica (CWI) and Vrije Universiteit (VU), Amsterdam.

The thesis contributes a possible extension of Geno and a case study on using the tool Geno being developed by the FSE group. In Chapter 2 and Chapter 3 the thesis describes and elaborates some of the Geno concepts, which are extracted from a Microsoft internal document [22].

Contents

List of Figures

Chapter 1

Introduction

This project applies testing theory to grammarware: namely, combinatorial test generation to XML Schema-based validators. It contains not only the theoretical background, but also the actual source code and purely practical results.

Test data generation is used in grammarware testing, where we test software operating upon a clearly distinguished grammar (a protocol, a data format, a programming language). Having faced lots of different XML validators, we do differential testing (let them find out one another's errors instead of checking them with the standard). Also, we do not want to pick up only some test cases randomly and use exhaustive approach (generate everything possible within the given limits). Chapter 2 tells a short story about those issues, including both the definitions from other sources and the description of our way to use them.

When our project had started, we had a tool called Geno, written in C# at Microsoft Research Redmond (see the disclaimer on the page i). It is a supporting tool for combinatorial test data generation theory with control mechanisms. Chapter 3 describes Geno's architecture in the words of the first chapter.

The XML and the XML Schema specifications are grammar description formalisms that do not map one to one to the BNF-like signatures that were used in the tool. Chapter 4 concentrates mainly on the case study, its distinct issues and the changes to the tool that have their origin in this work. Also theoretical considerations are put there, such as the most probable places for a bug.

The largest issue that we have encountered so far finds its place in Chapter 5: we learn to adapt the tree-like XML Schema structure during its parsing to the Geno's internal grammar representation in the form of a signature. This result is also original for this project and will be published after some more experiments.

The actual results together with the careful description of the way they have been acquired are put to Chapter 6, as well as the hope for the future. The last Chapter 7 says a few words about the related work that did not contribute to the basic theory we used, but is somehow linked to our project and may eventually become valuable for us, too.

Chapter 2

Background

This chapter includes some brief background description, literature survey results, as well as other issues concerning some research that had existed before us, but upon which this project is based.

2.1 Testing: terms and definitions

The case study of this project (as it will be put in the Section 4.1.1) is XML test data generation in .NET environment. We use XML [6] as a data language and XML Schema [12] as a data definition language, i.e., a grammar formalism.

2.1.1 Conformance testing

In general, there can be only two kinds of testing: *structural testing*, which is based on knowledge of the guts of the system under test, and *functional testing*, which can rely only on the externally observable behaviour [38]. Sometimes they are referred to as *white box testing* and *black box testing* accordingly, or even *program-based* and *specification-based* accordingly.

Conformance testing happens when we do have a specification of what a program should do and an implementation doing something its creators believe is right. It can belong to either of the classes described above. In the first case we possess some knowledge about how the implementation is constructed, in the latter we can rely only on what we can see from the outside.

Usually by testing an implementation it is meant that a set of tests, called a *test suite* is applied to the implementation with the aim of determining whether the implementation conforms to the relevant specification. Two completely different implementations of the same specification may be correct if they behave similar enough.

A *test case*, or a test, is a term used very often. A test case is, in general, a sequence of actions being performed upon a *system under test* together with the required inputs and the desired outputs in order to find a bug in the system.

More concrete definitions can rely on a final sequence of states [14], a pair of states [5], a set of events [28], etc. Unlike all those, we will consider *test data*, which consists of data inputs. This can always be done in grammarware testing, but software may operate with no explicit formalism behind it, and generation of sole inputs yields a problem of how to evaluate the correctness of the observed system behaviour [14].

As it will become clear in the sections to follow, we are going to use a grammar as a specification of the program.

Unlike conformance testing, the *differential testing* approach, as put in [26], is a form of stochastic testing that takes two or more different systems and feeds them with series of mechanically generated test cases. If (better say when) the outcomes are different (or one of the systems loops indefinitely or crashes), the tester has a candidate for a bug-exposing test. Differential testing can be used with either of the aforementioned approaches. We will use it together with combinatorial testing, application details and related discussion can be found in the next chapters.

2.1.2 Testing of grammarware

"Grammarware comprises grammars and all grammar-dependent software, i.e., software artifacts that directly involve grammar knowledge" [18]. In fact, by the word *grammar* any syntactic formalism may be meant, just to name a few: syntax definitions, interface descriptions, application programming interfaces, regular expressions, telematics protocols and XML Schema schemata. With the word *grammarware* we mean any software that operates upon such a grammar: a parser, a compiler, a pretty-printer, a lexical analyser, a run-time environment, a development environment, a browser, a type checker, a structural editor, a loader (load-and-go compiler), a debugger, a preprocessor, a profiler, as well as an interpreter and other numerous reverse engineering tools, code refactoring tools, slicing tools, (re)documentation tools, software analysis tools, static checkers, optimisers, etc. In grammarware testing the input is always a *test data* which consists of words in the language defined by the grammar.

All the grammars used to be forged using *grammar hacking* techniques where issues like testing and disciplined adaptation of grammars played a minor role [20]. Nowadays it is moving slowly towards *grammar engineering*, where the development, the maintenance, the recovery and the implementation of grammars are based on well-founded concepts [18, 20, 23][1]. We will concentrate on *grammar testing* from now on.

A grammar can be used as a specification, and that is exactly how we are going to look at it. But on the other hand, a grammar can be seen as a program (compiler compiler's input, for example). In the future work section we will describe another way: one can treat the "XML Schema schema for XML Schemas" [40] as just another XML language, for which we can generate test

[1] One being interested in such issues is referred to the project "Engineering of Grammarware" at the Free University, Amsterdam: http://www.cs.vu.nl/grammarware).

data. In this case, test data will be XML Schema schemata themselves, and afterwards we can use them to generate XML files).

Also, there can be no approved or standardised grammar for a language at all, that is the moment when *grammar recovery* techniques come into play [20, 23]. We do not consider this issue in this project, but it is worth mentioning—even the official version of XML Schema does not claim that it is correct, referring to hundreds and hundreds of pages with verbal description. Again, we know it, we point it out, but we do not deal with it within the current project.

Grammarware testing is not easy. Grammar-based functionality operates on richly structured, sometimes arbitrarily nested data, which leads to challenges which we still have to meet. There are also a lot of problems, as an example we can give generation of semantically correct terms (programs): given the fact that generated test data is a syntactically correct program, it still does not imply that its semantics is correct too. In other words, we can write it down, but it does not make any sense (just like in the real life languages).

Scenarios of grammar testing

We can think of quite a number of scenarios of grammar-based testing (the following list is partially taken from and totally inspired by [18] and [22]).

- ◇ Telematics protocol design.

  Interfaces, communication standards, interchange formats and the interaction protocols themselves are all very well described in grammar form. XML Schema schemata, ASN.1 formalism, IDL from CORBA, façade design pattern, UML diagrams, (W)SDL, API and programming libraries—they all can be seen as grammar formalism, and can be helpful in testing every piece of telematics protocol design [13, 18].

- ◇ Virtual processor implementation.

  Here by a virtual processor we mean anything that has an instruction set and is not a real processor: a virtual machine, a just-in-time compiler, a silicon processor, an intermediate code interpreter, a debugger and so on. During development of a virtual processor, its instruction set is continuously being enhanced and updated. If this is done in a systematic way, testing can easily find its place in a lifecycle. Having a specification of valid instructions and their semantic meaning, we can generate arbitrary long (in)valid instruction sequences and run them on the implementation under test [22, 32].

- ◇ Implementation testing.

  Almost any given idea can have a couple of implementations that differ drastically in performance. In order to guarantee that performance optimisation does not introduce any odd behaviour to our implementation, it seems appropriate to test the optimised and the original versions against each other in a sense of differential testing [22, 26].

◇ Compiler development for a known language.

Different compilers of the same programming language often behave differently: a program accepted by one of them may yield an error message with another. Thus, different compilers lead to different dialects (and eventually to extensions, sub- and super-sets, etc). But even if by making a compiler one changes the language, it is important to know the differences. Again, differential testing is very helpful. Alternatively, we may want to modify the language intentionally (as Modula-3 extends Modula-2, as C++ extends C, as Cω extends C#) and then put the effort on testing the new features exclusively.

◇ Browser development.

Browsers are not just markup language compilers. They are intended for showing the content no matter how clumsy and sloppy it is written: a good browser never says *"error on the page!"*, a good browser should always find some way to resolve any mistake a web page designer has made. Good error-fixing mechanism is perhaps more important for a browser than strong conformance to the markup language specification.

Therefore, we need to do stress testing with large amount of invalid (X)HTML, CSS and perhaps XSLT files, as well as programs written in supported scripting languages (JScript, JavaScript, ECMAScript, VBScript, etc). In this scenario the outcome of any test case comes usually in graphical form, which involves GUI testing techniques [28].

So, grammarware is not all about programming languages [18]!

2.2 Combinatorial vs. stochastic testing

We can classify all testing approaches in two vast groups. The first one is called *combinatorial testing* (or *exhaustive testing*). By taking this approach, we generate all possible combinations of test data, which is usually a huge test data set. Most of the time it is unacceptably big or even infinite, so that is why the other approach is used more often. That one is called *stochastic testing* (or *random testing*). Assuming that execution of all test cases is not possible (it is infinite or too costly), we instead pick some of them in sorry hope that they will point a bug. Such reduction of the size of generated test set is called *test selection* technique, which may include, for example, assigning costs and values to all test suites [38]. Despite the name of this approach (random testing), test selection should not be done at random but a strategy should be applied such that the resulting reduced test suite is valuable for conformance testing, in the sense that there is a large chance of detecting non-conforming implementations. A lot of different heuristics are known and used in such cases.

A project related to our work is *Massive Stochastic Testing for SQL* [33]. It is about automated generation of huge SQL test set, and related to us in a sense that it uses differential grammar-based testing [26]. It is being carried out at

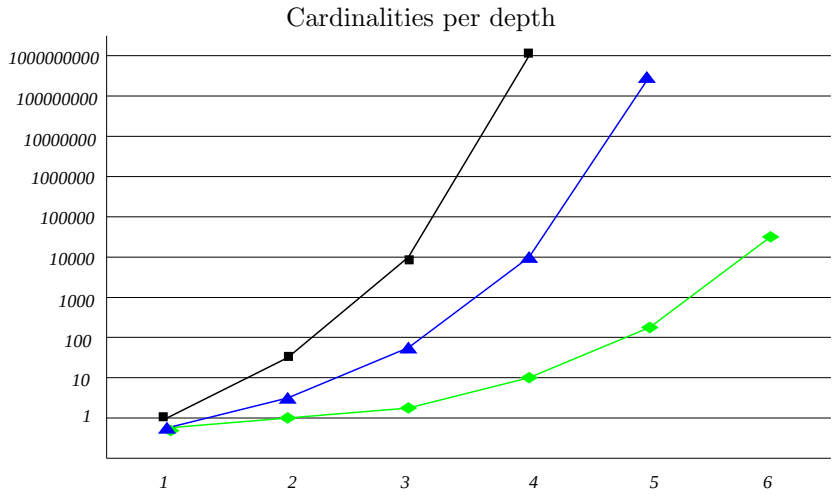

Figure 2.1: Combinatorial explosion in logarithmic scale: green diamonds represent a simple language with 1, + and −; blue triangles—its "upgrade" with 0, 1, 2 and all arithmetic operation symbols; black boxes describe the C#. Points not included are higher than 18446744073709551616.

the Microsoft Research Redmond. However, it uses stochastic testing techniques and we use the first approach, namely the combinatorial one [22]. Test data is going to be generated in a very systematic manner achieving coverage of the underlying grammar in a fundamental way [20].

Obviously, the straightforward combinatorial approach is way too thorough and inefficient to be used as is and its nature is too explosive to make it possible to use it in practice. We use the word *explosion* that means not only that the function value (number of terms) goes to infinity with the finite increment, but also that the number of generated test cases becomes unfeasible within a very small number of depth layers explored. In practice this means that there is always a depth for which the term generation process is not workable (number of terms is far beyond the amount with which we can deal).

As it can be seen from Figure 2.1, number of terms per depth level increases exponentially even on logarithmic scale (therefore, it is close to e^{e^x})! The lowest line (green diamonds) on the diagram corresponds to the most silliest language we could have imagined: the one that has 1 as a terminal, "−" as an unary operator and "+" as a binary one. Even given such a simple example, it is not possible to proceed beyond depth 6 (the number of terms on the depth 7 is outside the long integer range: $2^{64} = 18446744073709551616$). The next example (blue triangles on Figure 2.1) corresponds to the "upgrade" of the same language: it has $0, 1$ and 2 as terminals, two unary operators $(+, -)$ and all binary ones $(+, -, *, /)$. This minor improvement not only consumes one depth level entirely, but also dramatically increases the number of terms on the

last accessible ones (that might lead to severe consequences for these levels up to becoming *practically* inaccessible). The line above these two (black boxes on Figure 2.1) shows an object-oriented language like C# or C++ in all its complexity.

In order to keep up with this explosion we apply some control mechanisms to our situation.

2.3 Mechanisms to control explosion

The following control mechanisms are introduced in the Microsoft internal document [22]. We adopt them here and provide a semi-formal definition of them including some illustrative examples.

There are six major control mechanisms:

- ◇ *Depth control* — we limit the maximum number of constructors to be applied.
- ◇ *Recursion control* — we limit nested applications of recursive constructors.
- ◇ *Equivalence control* — we build equivalence classes.
- ◇ *Balance control* — we limit the preceding levels from which the terms are reused.
- ◇ *Combination control* — we limit the Cartesian product on argument terms. Pair-wise testing, ordering and duplicate controls also belong here.
- ◇ *Context control* — we enforce various context conditions and add context-dependent information.

The first three of them are shown on Figure 2.2 taken from [22]. The illustration conveys that precision of coverage can be restricted to certain parts of the combinatorial search space. More details to follow in the next sections.

We will show how all the control mechanisms work on a rather simple example of a grammar (it is supposed to represent a tiny subset of the HTML):

```
html = (head, body);
head = title;
body = (block*);
block = (data) | p(block);
data = em(data) | strong (data) | text;
```

2.3.1 Depth control

The depth is defined to be 1 for all literals (constants); it increases every time a new constructor is applied to something on the top-level. If a constructor is applied to several parameters, the resulting depth is counted as 1 + maximum depth of all its parameters. Therefore the depth of $c(t, c(t, t))$ is 3, as

well as the depth of $c(c(t,t),c(t,t))$. For our example, we have `title` (for sort `head`) and `text` (for sort `data`) as literals. Then, to step on the next depth level, we may apply one constructor on them: `html` is not yet accessible, so we can only generate the first term for the sort `block` and longer `data`s, having on the depth 2: `block(text)`, `em(text)` and `strong(text)`. On the next depth level we hit the `body` with terms like `body*(block(text))` and `body*(block(text),block(text))`. On this depth `block` is also allowed for three more sophisticated terms: `block(em(text))`, `block(strong(text))`, as well as for `p(block(text))`. They will contribute on the next depth, where the first `html` is going to be built.

The main idea that lies behind the combinatorial test data generation is to explore (to generate) all possible terms up to a certain depth (Figure 2.2 [22]). However, this is not exactly what we want: we would rather generate all possible terms of a given sort (`html` in the example above). Therefore it is possible to introduce depth control on constructor basis, too. Instead of defining and limiting the depth as the maximum reached depth in global (per sort), we can also manage depth per argument position to be considered separately. This is more specific and in ideal situation can lead to better coverage (since combinatorial test case generation is exhaustive, the coverage is always 100%, so we mean here that we can go deeper to more interesting directions and forget about less interesting ones), but it is much more complicated to specify.

The depth control mechanism seems to be the main limitation: it is definitely the most generous we could ever imagine, it is always applicable, very simple, and therefore—the least interesting. This is also the last control to be applied: it can stop the generation process entirely at any point that turns out to be last of our interest.

Some layers (sets of terms of the same depth) may not be interesting for us at all. For the example we use, depth levels before 4 look uninhabited with respect to the sort `html`, which is the one we need to explore. This happens quite often in practice: the grammar is written in such a way that the first several layers do not contribute at all to the root sort (the one we need), because at least on term of all its argument sorts should be constructed before that (we will say from now on: become *inhabited*).

2.3.2 Recursion control

By the word *"recursion"* here we mean a sort that has a constructor with one or more arguments of that sort. In the example from the previous page the sorts `block` and `data` are recursive.

The recursion control is another way to limit unnecessary generation of terms. As one can see from Figure 2.2 [22], it is a more sophisticated version of the depth control: limitation of the recursive exploration of sorts holds before we reach sorts of interest. By saying that, for example, no term can have subterms of the same sort, we can seriously decrease the number of terms to be generated. The impact of this is growing with the depth explored. In our example, we can present a table with numbers of terms for each sort per depth (no recursion

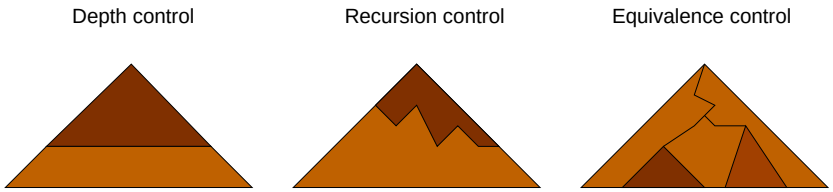

Figure 2.2: "The brighter—the better covered"—the precision of combinatorial coverage in signature exploration [22]. Depth control stops the generation process entirely at a given depth; recursion control limits nesting; equivalence control relies on the definition of equivalence classes.

control, sequence control (see below) set to 5):

	1	2	3	4	5	6	7	8	···
`html`	0	0	0	11	159	1225	6951	33717	···
`body`	0	0	11	159	1225	6951	33717	150219	···
`block`	0	1	3	7	15	31	63	127	···
`data`	1	2	4	8	16	32	64	128	···

For obvious reason, we do not include here the `head`, which has only one literal and does not acquire any more terms with depth. Then, we assign `MaxRecDepth=2` to the sorts `block` (due to its constructor `p(block)`) and `data` (due to its constructors `em(data)`) and `strong(data)`). The table changes as follows:

	1	2	3	4	5	6	7	···
`html`	0	0	0	11	159	652	0	···
`body`	0	0	11	159	652	0	0	···
`block`	0	1	3	4	0	0	0	···
`data`	1	2	4	0	0	0	0	···

To put it bluntly: we have applied a control mechanism to two sorts, and this has reduced the test space from infinitely large to finite and small. We would not lie to you: it rarely happens like this in real life, but at least it shows well how powerful those control mechanisms are.

NB: we did not even use depth control: it is not necessary now!

Another powerful part of a recursion control is applied to constructors rather than to sorts, it is called *sequence control*. In the case we have a sequence constructor (a star, in words of the BNF [30]), that can generate arbitrarily long sequences of subterms of the same sort, we can add an attribute to restrict the maximum length of such a sequence. The power and the meaning of this control strongly varies on how far is the constructor's sort from the root: for something deep inside the term we are interested in, it is natural to introduce as many restrictions as possible. On the other hand, if our root is a sequence of some other element, which has huge set of all kinds of constructors yielding different

subtrees, we will certainly think twice before taking away that root element's ability for production of long sequences.

Anyway, we can apply this to our grammar (restricted variant—with sort recursion control as described above), increasing `[MaxLength]` Geno attribute up to 50 to the `body`'s one and only constructor.

	1	2	3	4	5	6	7	···
`html`	0	0	0	1226	19554	91732	0	···
`body`	0	0	1226	19554	91732	0	0	···
`block`	0	1	3	4	0	0	0	···
`data`	1	2	4	0	0	0	0	···

As we see, it is impossible to ruin all the wonderful job of the sort recursion control, yet as a final result we have 137 times more terms for `html` than the number of terms with using some lousy limitations (5 is not a very strict limit, either). Obviously, no control at all (`MaxLength` $= \infty$, if it were possible) leads to immediate explosion as soon as the sort with a sequence constructor is accessible.

2.3.3 Equivalence control

Halting the term generation process entirely is not a perfect solution. Eventually we may still want to generate deeper terms of our interest (and not abandoning the depth control, too). In order to complete this task we introduce the equivalence relation between terms which states: *every two terms are considered equivalent if they are of the same shape up to a given depth.*

Having built such equivalence classes, we do not need to generate all terms for every depth any longer. Instead, we can pick up one representative from each of the equivalence classes. As it can be seen on Figure 2.2 [22], combinatorial exploration is terminated when we reach a certain depth.

For our example, we can consider all terms that look like `html(title, body*(p(em(strong(...)))))` to be equivalent (it actually makes sense, since "`...`" may have only `em`s and `strong`s, which are already there). So, only the term `html(title, body*(p(em(strong(text)))))` will be generated to represent this equivalence class.

2.3.4 Balance control

We can also limit the preceding levels from which the terms are reused such that only terms from the immediately preceding levels are taken. It can not only be specified generally, but also per sort (we might not need terms of some sort to have simple short subterms); per constructor (the same applies only to one constructor of a sort); or even per sort occurrence (per constructor argument).

If we use the balance control for our example in the most strict way (only one previous depth layer), this can rule out any occurrences of the shorter terms of the same sort if there are longer ones. Thus, on the depth 4 we will have `body*(block(em(text)),p(block(text)))`, but not

`body*(block(em(text)),block(text))`, because the subterm `block(text)` cannot be taken from the depth 2!

2.3.5 Combination control

Often we may have a constructor which has more than one argument. In that case, it is not always necessary to vary them all when constructing terms (i.e., to combine argument positions in a Cartesian product). The combinatorial completeness can often be relaxed using knowledge of test experts: for example, when testing binary expressions one would like to exhaust operators and operandi, but not both simultaneously [22]. The other extreme is called *one-way coverage*, when each argument position is exhausted separately (it is a rather aggressive way to avoid explosion).

However, there are other possibilities between those two: *multi-way coverage* notion exists and allows for specifying almost any combination requirements. The most simple scenarios include $\emptyset$ (no combination at all—leads to construction of a single term), $\{\{1\}, \ldots, \{n\}\}$ (one-way coverage exhausting all components), $\{\{1,2\}, \ldots, \{1,n\}, \{2,3\}, \ldots, \{2,n\}, \ldots, \{n-1,n\}\}$ (pair-wise coverage) and, of course, $\{\{1,2,\ldots,n\}\}$ (full Cartesian product).

In our example there is no constructor with two arguments (except for the only one in the `html` which is not interesting (because the `head` is not particularly rich with terms). We can introduce another example (which is a slightly modified version of the grammar we have used so far), where the `body` has a few attributes:

```
body = (blocks, onload, onunload, style, class);
blocks = (block*);
block = (data) | p(block);
data = em(data) | strong (data) | text;
onload = ""
       | "alert('hi');";
onunload = ""
         | "alert('bye');";
style = ""
      | "text-align: left;"
      | "text-color: blue";
class = "" | "a" | "b" | "c";
```

With all control mechanisms relaxed besides the sequence control set to 4 and two-way coverage, we have results as follows:

	1	2	3	4	5	6	7	8	···
`body`	0	0	0	34	390	2890	16158	77794	···
`blocks`	0	0	7	96	721	4038	19447	86292	···
`block`	0	1	3	7	15	31	63	127	···
`data`	1	2	4	8	16	32	64	128	···
`onload`	2	0	0	0	0	0	0	0	···
`onunload`	2	0	0	0	0	0	0	0	···
`style`	3	0	0	0	0	0	0	0	···
`class`	4	0	0	0	0	0	0	0	···

With one-way coverage, the number of terms of our interest (i.e., those from the `body` sort) decreases. It corresponds now exactly to the number of subterms of the most complex subterm's sort (`blocks` in our case):

	1	2	3	4	5	6	7	8	···
`body`	0	0	0	7	96	721	4038	19447	···
`blocks`	0	0	7	96	721	4038	19447	86292	···
`block`	0	1	3	7	15	31	63	127	···
`data`	1	2	4	8	16	32	64	128	···
`onload`	2	0	0	0	0	0	0	0	···
`onunload`	2	0	0	0	0	0	0	0	···
`style`	3	0	0	0	0	0	0	0	···
`class`	4	0	0	0	0	0	0	0	···

We see that it decreases not only the number of terms, but also the speed their number grows with depth.

2.3.6 Context control

We may also decide to enforce various context conditions and add context-dependent information. By doing so we can incorporate semantic meaning to the terms that are being generated. Before constructing a new term with a constructor and all the sub-terms which can serve as its arguments, we check the predicate associated with that particular constructor.

The reasonable predicates may be: implying a relationship between the arguments (should not be equal, should sum up to a certain number, should be homogeneous, etc); assuring a good place for a term (should be preceded by, should be followed by, should appear only once, etc); posing additional requirements on the arguments (one of them should be equal to, all should begin with, etc); introducing a relationship with a distant term (especially for references); recursive predicate (for all subterms of all the arguments it should be, etc).

For our example (we take the original one from the page 7, not the one from the preceding control mechanism description), we will use the so called *transform control*, which allows for term transformation or dropping. In order to do that, we write a method[2] that takes an array of newly generated terms

[2]According to the delegate: `public delegate Term[] FilterProc(Term[] ta);`

and filters it. For example, for some reason we want to discard all terms that use both `em` and `strong` constructors. Now we can refer to that method directly from the annotated grammar. Without using that filter option the results are (with `MaxLength=5` for `body` and `MaxRecDepth=5` for both `block` and `data`):

	1	2	3	4	5	6	7	8	···
`html`	0	0	0	11	159	1225	6951	33717	···
`body`	0	0	11	159	1225	6951	33717	85398	···

When we assign the filter described above to the `body`'s sole constructor, the statistics changes as follows:

	1	2	3	4	5	6	7	8	···
`html`	0	0	0	11	159	1204	6423	27885	···
`body`	0	0	11	159	1204	6423	27885	64174	···

Two results are acquired: first, we do not generate the terms we do not want and can concentrate on the desired remains; second, the number of generated terms is decreased—both are important.

Chapter 3

C# .NET-based Test Data Generation

3.1 .NET Framework overview

The Microsoft .NET Framework is an important new component of the Microsoft-hosted family of operating systems. It is the foundation of the next generation of applications running under Windows which are easier to build, deploy and integrate with other networked systems.

Microsoft claims that most of their consumers will never notice that the .NET Framework is running on their Pocket PC, smart phone or desktop computer, but may appreciate the reliability, ease of use and ability to connect to other systems that the .NET Framework helps bring to computers [29].

Anyway, the .NET Framework is the most recently developed software platform. Its core features, as well as the "official" programming languages (C++-like C# and assembly-like IL) have been standardised by the ECMA[1] and the ISO[2]. There exists an open source version (including compilers) that runs under FreeBSD and on Mac. It is also suitable for multi-language programming, being shipped with a bunch of compilers as well as providing unified access to the library or class contents [41].

The principal designers of the C# language were Anders Hejlsberg, Scott Wiltamuth and Peter Golde [4]. The first widely distributed implementation of C# was released by Microsoft in July 2000, as a part of the .NET Framework initiative. It is intended to be a simple, modern, general-purpose, object-oriented programming language. It is strongly statically typed, not strictly object-oriented language **with** support for polymorphism, operator overloading,

[1] **Standard ECMA-335** Common Language Infrastructure (CLI), 2nd edition (December 2002): http://www.ecma-international.org/publications/standards/Ecma-335.htm.
Standard ECMA-334 C# Language Specification, 2nd edition (December 2002): http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[2] **ISO/IEC 23271:2003**, Information technology — Common Language Infrastructure. **ISO/IEC 23270:2003**, Information technology — C# Language Specification.

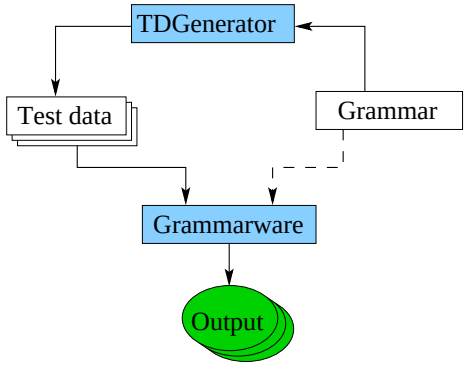

Figure 3.1: Grammarware testing tool: a test data generator takes a grammar and produces test data, which is fed to the application that operates on that grammar.

multiple threads, delegates, events, properties, exceptions, XML comments and automated garbage collection; **without** multiple inheritance, generics or type inference whatsoever, with its roots in C++, Delphi, Modula and Smalltalk, and some minor nifty features borrowed from other languages, too. It has syntactical dialect in Basic-style (not in C++ style) called Visual Basic .NET (not to be confused with the Microsoft Visual Basic 6).

A new version of the C# programming language in now being designed (it is claimed to have generics, a bit of type inference and functional programming features, co-routines, as well as some new operators), but is not yet available for public usage.

3.2 Test data generator architecture

A test data generator is supposed to be a program which computes a test data set achieving the desired coverage criterion [20]. We shall be dealing with the special test data generation project Geno under development at Microsoft Research Redmond (see the disclaimer on the page i).

We are now going to describe briefly Geno's architecture with more technical details and the changes we have brought to it are to follow in Chapter 4.

3.2.1 Grammarware testing in practice

Geno is a support application for "Controlled Explosion in Grammar-based Testing" [22]. It was a command-line tool written entirely in C# and consisted of some 3200 lines of code when our project started. It has now more then 5700 lines of code and has a simple GUI (which is another 80k of generated XML data).

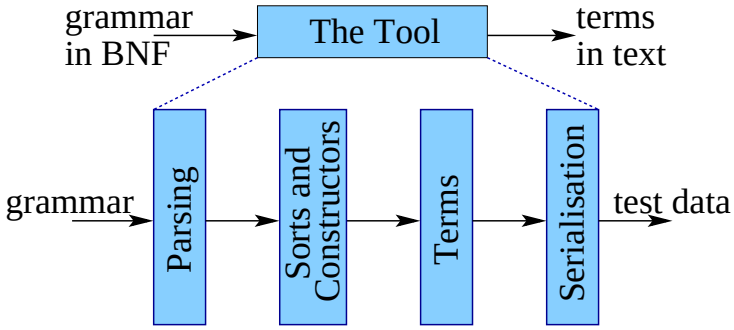

Figure 3.2: The original architecture of Geno broken into pieces: four main blocks.

The tool built for grammarware testing works as depicted on Figure 3.1. Its input is a grammar of test data it should generate (so, it is grammarware, too). After test data is generated, it can be fed to the system under test. The line going from the grammar to the system is not solid because this relationship can be implicit. The system under test produces then a sequence of outputs that should be somehow classified and eventually leads to a conclusion whether that piece of grammarware contains a bug.

Thus, we have a tool that takes a textual description of a grammar in BNF-like language as an input and produces sequence of terms, also in textual format. We can break it in four blocks for further understanding and development (see Figure 3.1).

The leftmost block is the first stage of dealing with the grammar: one must parse it (of course, it also involves scanning). Then, a set of sorts together with their constructors are passed to the next block, where they are well organised, filtered (not all of them might be of importance for us) and the whole engine is initialised. Then the terms generation mechanism is executed. It produces a lot of terms, which can be treated one by one in the next block (Serialisation), which transforms them to actual test data.

3.2.2 Grammar parsing and representation

As long as we are not concerned with the concrete syntax, we operate on simple signatures and not on the context-free grammars [22]. Therefore we have sorts and constructors. Sorts represent "types" of the expressions and constructors help to build up terms which correspond to those sorts.

The original version of Geno used a special BNF-like language for descriptions of the signature input (see Section 5.1.1 for more details). It allows for writing down sorts and their constructors in a very straightforward way, together with the specification of attributes (control mechanisms parameters).

In that version the grammar textual description was scanned, parsed and

only after that the actual core engine was initialised (the version of ours is very much the same, but it uses the standard XML parser (DOM).

3.2.3 Term generation algorithm

The algorithm for combinatorial exploration and all its enhancements and modifications can be found in [22].

3.2.4 Serialisation

The word *"serialisation"* actually means that we store a run-time object to its data description, using one way or another. People from electronic commerce and related area where information interchange plays a big role, are certainly familiar with such process, calling it *marshalling*[3]. It can be used to save memory, to exchange data, to backup data, to encrypt data, and to do thousand more things that you can do with data—but to do that with objects. The process of going the other way around (rebuilding run-time objects from text data) is called *"deserialisation"* (*demarshalling*).

In this project we have to serialise terms, because after the generation algorithm finishes, they are still objects. Also, as long as we talk about XML test data generation, we need to make an XML file out of every single term. The details on XML serialisation and the source code for it are found in Section 4.4.5.

3.3 Control mechanisms implemented

Section 2.3 has described the control mechanisms, now we will list the attributes one can use in the grammar description or directly inside Geno. Not every control mechanism has its bunch of attributes yet, but Geno is still being developed: we list only fully implemented and tested attributes, although.

3.3.1 Attributes assignable to sorts

- `MaxDepth`—this attribute represents the *depth control* (page 7), it limits maximum depth of terms, can be assigned to any sort (works not only at the top level) and has a default value of maximum possible integer (*System.Int32.MaxValue* in C#, that is 2147483647).

- `MaxRecDepth`—this attribute represents the *recursion control* (page 8), it limits nested applications of recursive constructors, can be assigned to any recursive sort and has a default value of maximum possible integer (*System.Int32.MaxValue* in C#, that is 2147483647).

[3] Talking about different words: our colleagues from Microsoft Research Redmond call it "serializ**z**ation", of course.

3.3.2 Attributes assignable to constructors

- ◇ `Oneway`—this attribute represents the *combination control* (page 11), it enforces the one-way coverage, can be assigned to any star constructor and is switched off by default.
- ◇ `Twoway`—its counterpart is always switched on (unless one-way is), i.e., no multi-way coverage is implemented yet.
- ◇ `Unordered`—the combination control parameter for two-way coverage, ignores the order of appearance, makes sense only for a star constructor.
- ◇ `NoDuplicates`—another combination control parameter for two-way coverage, ignores duplicate occurences, makes sense only for a star constructor.
- ◇ `MinLength`—this attribute represents the *recursion control* again (page 8), it tells the minimal length of a sequence, can be assigned to any product constructor, has a default value of 1.
- ◇ `MaxLength`—the other side of the same story, tells the maximal length of a sequence, can be assigned to any product constructor, has a default value of 2.
- ◇ `Filter`—this attribute represents the part of the *context control* (page 12), it gives the name of a C# function that should be run after the terms are generated but before they are passed outside the generator, it can be assigned to any constructor and is not used by default.

Chapter 4

The XML Case Study

4.1 Introduction into the topic

4.1.1 XML and XML Schema

The XML (eXtensible Markup Language) is a simple flexible text format for data representation and markup. It has been derived from the SGML (Standard Generalised Markup Language) by the W3C (World Wide Web Consortium) as early as in November 1996, but the current standard (W3C Recommendation confusingly called *Extensible Markup Language 1.0, Third Edition*) is dated February 2004 [6]. However, there is also an XML version 1.1 evolving independently, which tries to be compatible to Unicode of all versions. Anyway, it is in no way our goal to list or describe, let alone classify, all standards found on the W3C web site.

XML schemata express shared vocabularies providing a means for defining the structure, constraining the content and describing the semantics of XML documents. It had started in 1999 as a Document Definition Markup Language (DDML), followed by its introduction in practice as the Document Type Definition (DTD)[1]. Later it was enhanced, made XML-compliant and renamed to what we have today. Hence, we can safely say that XML Schema is an XML-compatible superset of the DDML.

The standard of the day contains over four honderd pages and consists of four parts. The first (and the most important) three are these W3C Recommendations of May 2001: *"XML Schema Part 0: Primer"* [11, 12], a non-normative document providing a sort of readable description of the XML Schema facilities in general; *"XML Schema Part 1: Structures"* [36, 37], which gives more formal definition of what one can do with XML Schema schemata and how this should be done; and *"XML Schema Part 2: Datatypes"* [2, 3] adds to those the data types used in the standard as well as the mechanisms introduced to define one's

[1] The terms DTD and XSD are purely practical and are nowhere to be found on the W3C official web-site. They usually denote a file containing a grammar in the DDML or the XML Schema, accordingly.

own data types (that might sound as a feature of secondary importance, but we shall not forget that any element that has sub-elements or even attributes, has a type, i.e., XML Schema without types and type definition is almost useless). These parts also exist as second edition versions [2, 12, 36], but those have not yet made it to a standard (now it is at a W3C Proposed Edited Recommendation level).

The fourth part is a requirements part: *"XML Schema Requirements"* [25], W3C Note of February 1999, or *"Requirements for XML Schema 1.1"* [8], W3C Working Draft of January 2003, about the purpose and goals of XML Schema and the possible scenarios of its use. One can check herself if the requirements were fulfilled, there exist different points of view about it [31].

Also we use the more formal description of the Part 1 [7], the XHTML 1.0 in XML Schema [16], as well as an anonymous resource that contains a DTD and XML Schema for XML Schema itself [40].

4.1.2 Testing concerns in the XML

We can put our effort on testing different corners of the XML. The XML Schema standard is huge, and there are lots of things that can go wrong in its implementation. We list here a few issues that we can test:

- ◇ *Well-formedness*

  As it will be described in details later (see page 23), there are two levels of XML file correctness, and we can try the simplest one first. Well-formedness assures that it is possible to make a schema such that the given file will be valid against it. Sometimes it makes sense to test this issue: if a validator gives a positive result for an ill-formed file, it is definitely possessed by a bug.

- ◇ *XML declaration*

  Every XML document should[2] have a header called XML declaration. It is just one line, but without that line we cannot tell the XML file from any other: it is a signature.

  ```
  <?xml version="1.0" encoding="UTF-8" ?>
  ```

  Many pieces of software still forget about this signature. One of them is the .NET Framework, surprisingly! For example, the class `XmlDocument` has a method `Save()` which takes a file name and is supposed to save the object contents right into the file. It does, but without a signature.

  The XML specification [6] gives also very strict description of how the document type declaration should conform to the document content.

  Despite all the recommendations that are given by the specifications, some grammarware developers may deliberately decide to omit those "tiny things" in order to improve their system's robustness. We do not discuss

[2] In the context of the RFC 2119.

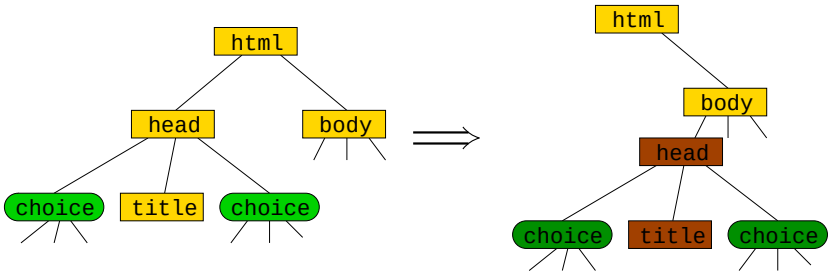

Figure 4.1: Subtree shifting example: the `head` tag which is expected as a sub-element of the `html` tag becomes a sub-element of the `body` tag.

the underlying reasons, but do not feel certain enough to call this an error regardless the circumstances.

◇ *Root element*

We can also check for the root element whether it is on its place. According to the XML standard [6] and the World Wide Web Consortium politics, every XML file should have one and only one root element and that element name must[3] be listed even before its occurrence, in the document header. An example is given below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html> ... </html>
```

The simplest test cases we can think of are: misplaced root element, no root element, misspelled root element name, several root elements, etc.

◇ *Grammar mutation*

It is very common in testing to stress boundaries by introducing additional test cases. We can do the same thing on a grammar level by slightly changing the grammar. This will lead to generation of lots of incorrect (not valid) files, which we should try on XML validators. With using differential testing [26] we do not even need to know which test data is correct and which is not. However, it would be good to know it just to have some certainly correct test data, too.

The list of things that can be mutated includes: occurrence conditions (`minOccurs` and `maxOccurs`), simple types, degradation of references, mixed content rules.

◇ *Subtree shift*

[3] In the context of the RFC 2119.

As a specific kind of mutation we can try to push some elements (element occurrences) from their places. For an element that is necessary on its place this already must yield an error (for example, this would happen if one tried to make an (X)HTML document without the `<body>` tag). For an optional element its occurrence on the wrong place should, too. An example of subtree shifting is shown on Figure 4.1.

◇ *Attributes*

Attributes launch a challenge to XML testers: they are plentiful, freely used and may belong to hundreds of types (called *simple types* in the XML Schema). Anyway, one may concentrate purely on checking attributes, their use, positioning, types, values, ordering, duplicates, omitting required attributes, etc.

◇ *References*

Since the beginning of the XML we have references. A reference in unlike any other XML feature, because one entity is actually broken into two parts: one with some element and the other with the reference to it. Testing references relies both on syntax and semantics, hence, requires the usage of the context control mechanism.

Two things can be tested upon the XML references: first, they should be *unique* (no two elements with equal `id`s should exist within the same namespace); second, they should be *consistent* (if there is a reference to some x, there should always be one element with its `id` attribute set to x within the same namespace). Obvious invalid tests include double declarations and referring to non-existing entity.

◇ *Stress testing*

We can also stress things that are not specific for the XML and that all test data share in common. In words of the XML it would be: the sequences that are too long, the patterns that are too complex, the simple types that are not that simple, the attributes that are too numerous, etc.

◇ *XML representation of schemata*

In the W3C Specification [36] there are three levels of XML schema-aware processors defined. The first one is *minimally conforming*, implementing XML Schema component constraints, validation rules and XML Schema information set contributors without additional features. The medium level is called *conforming to the XML Representation of Schemas*, they are minimally conforming and accept schemata represented in the form of XML documents as described in [2] (`<include>` item). The third level is described within the next issue:

◇ *Full conformance*

Fully conforming processors are network-enabled processors which conform to the XML Representation of schemata and are additionally capable of accessing schema documents from the World Wide Web according to [36]. Usage of additional external schemata should therefore be tested.

◇ *Namespaces*

Elements in an XML document usually belong to one namespace. For example, a common XML Schema schema will use only tags with the `xs` prefix, which is the short form for a namespace named `http://www.w3.org/2001/XMLSchema` (so, it would be `<xs:element>`, `<xs:attribute>`, `etc`). For XHTML [16] documents the prefix is even omitted, but it is possible anyway to use a schema element from another namespace, just by using a different prefix. This requires additional support effort from APIs, data binding frameworks and XML validators.

◇ *Schema-related markup in documents under validation*

XML Schema: Structures [36] also defines several attributes for direct use in any XML document. These attributes are in a different namespace, which has the namespace name `http://www.w3.org/2001/XMLSchema-instance` (short prefix: `xsi`). All schema processors should have appropriate attribute declarations for these attributes built in.

4.2 Applying Geno to the XML area

This section sketches the case study of this project as the next one explains the way it has been solved. One particular issue, namely the mapping process from XML Schema, is however shifted to the next chapter.

4.2.1 System under test

We assume an XML Validator (Figure 4.2) to be our system under test. It takes two arguments: the XML-marked text file and the grammar it is suspected to be valid against.

The XML files have two levels of conformance. The first one is called *well-formedness*: any file that has been written with all tags closed, proper tags nesting, and obeying some other rules of hygiene, is considered well-formed. Tags not closed at all or closed in improper place (like the all-famous construction `<b><i>text</b></i>` produced by the early versions of Microsoft Front Page) prevent a file from being well-formed. Sloppy way to write down attributes (`width=10` instead of `width="10"`) might do the same thing. This level is nowadays considered to be just courtesy and nothing more. Everyone is now expected to write well-formed XML code. Luckily, most of the time this is done for us automatically by computer programs.

The second level of conformance is called *validity*. This is the moment when the XML Schema comes into play. In short, validity is nothing more

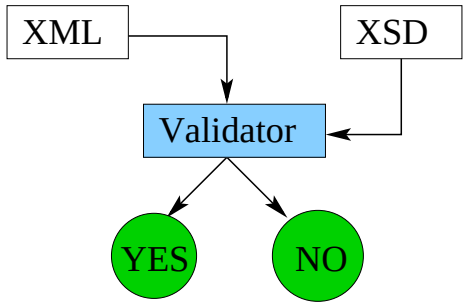

Figure 4.2: System under test: an XML validator which operates on an XML Schema schema, takes an XML document as an input and produces either a positive or a negative result.

sophisticated than conformance to a certain XML Schema schema (they say, *is valid against ...*). That means that every single tag in the XML file is an element defined in the schema, every attribute is defined there as well and the overal structure (how the elements are placed according to one another, what attributes do they have, etc) conforms to the schema too.

Talking in terms of grammarware, an XML Validator is a twisted parser. It parses the file according to the grammar, yes, but it does not construct the parse tree, just saying either *"yes, I might have done that"* or *"no, I wouldn't bother doing that"*. It just tries its best to parse the file, and gives the result whether is can be parsed (with the given grammar).

Alternatively, an XML Validator may be a part of a data binding architecture and actually construct a kind of parse tree (therefore being a real parser): in this case we can treat successful construction as a positive result and the presence of error(s) as a negative one.

4.2.2 Using Geno for the XML

The first possible solution proposal can be seen on Figure 4.3. It is very straightforward, we assume to have some kind of test data generator that can take the grammar as an input and produce lots of XML files according to it (perhaps by generating them with the combinatorial algorithm we have discussed earlier). Then that test data set is fed into an XML validator, which outcome (set of *yes/no* answers) leads to the deduction whether it is a good or a bad validator. In this scheme we need some kind of an Oracle that can tell for each XML validator's answer (being it *yes* or *no*) if it is correct, and eventually to let us decide whether the system is correct and bug-free. We do not use such scheme.

Figure 4.4 proposes something what we actually do use: test data set generated by our tool is fed into several systems under test. We run several (say, three) XML validators against one another and report the cases when their results are not the same. By the way, there is always a chance that either all

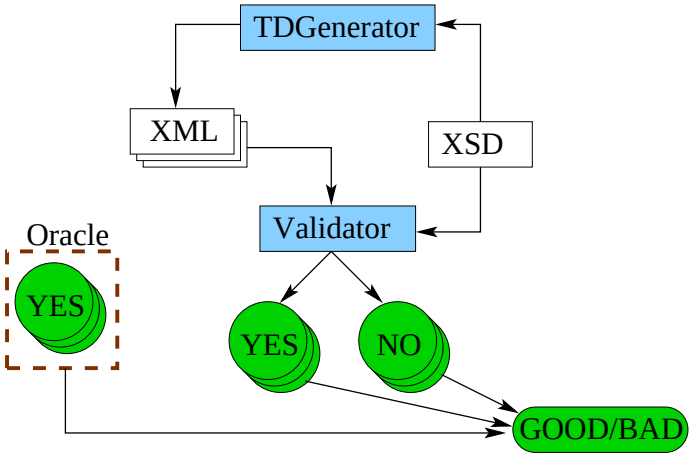

Figure 4.3: Generating XML documents from an XML Schema schema (conformance testing). We use an oracle to conclude that the system under test is bug-free.

XML validators agree on what is considered an error, or they have different opinions about something that is vague enough in the standard to be understood in several right ways. In the first case (all agree on an error) this might lead to changes in the standard—that is what the W3C standards are for: to give everyone equal chances to understand them equally good. In the second case (differ on a vague point) it is also important to report such difference and necessary to determine the right way to resolve the problem and to change the description in the standard. Now, may be that is the reason why the W3C Recommendations keep on growing so fast...

This problem solution lets us test all kinds of XML files with one XML Schema schema (one grammar).

4.3 Usage

Before we start doing the essential (the testing part), we should know with some degree of certainty about how it can be used, for what purpose, and what do we do with the results. The next sections are totally dedicated to those issues.

4.3.1 Possible scenarios

The reasoning behind one's willingness to apply combinatorial test data generation technique to several XML validators can be different:

- ◇ **Testing new XML validator**

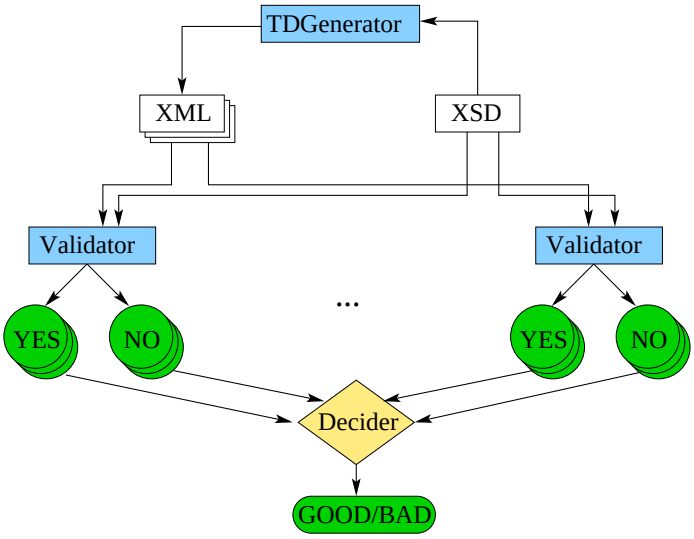

Figure 4.4: Generating XML documents from an XML Schema schema (differential testing). A bug-exposing test is the one on which the XML validators under test did not agree.

In this scenario we assume some kind of new XML validator which we cannot totally *trust*. We cannot rely on its output, so some testing technique must be applied. If we already have a couple of validators on hand, we can use differential testing technique to run this new XML validator against the trustworthy ones.

Why should we bother with the new one, already having a bunch of working pieces? There might be a lot of reasons: for example, they show worse performance or have inconvenient interfaces.

◇ **Choosing the XML validator**

This scenario is more about the user side: we have to choose one XML validator among the several existing. Almost all XML validators either are free or have some freely downloadable (albeit limited) version by which it should be possible to judge them.

Solving this problem, we may end up with quite a number of different XML validators with different functionality, interfaces, performance and other features, but all doing the same job. They are then executed against one another in a way differential testing does it. The outcome would be a set of files on which some of them cannot agree. Which one to choose, will be decided according to the requirements we have in mind: some reasonable balance should be found between the fast and buggy and the slow and correct; besides, the given licensing mechanism may strongly influence the

choice.

◇ **Reviewing the specification**

Even the best standardised specification should be reviewed with time. Especially with the World Wide Web Consortium standards it seems appropriate to review the Recommendations not only on a base of new features that the end users want to be introduced and old bugs that they want to be fixed, but also on a base of misinterpreted issues of the previously used specification.

There can be two kinds of such issues: those on which all available grammarware artifacts do something unacceptable, as well as those on which only some of them make mistakes. (The word "mistake" is used here as a synonym for the word "misinterpretation", because it would be too evil to point each of those as a real error.) In the former case we are powerless (already discussed earlier): differential testing cannot detect that kind of bug, period. However, in the latter case, when, say, the half of all XML validators produce a positive answer and the other half produce a negative one, this can help.

4.3.2 Postprocessing

As long as the description of scenarios above mostly considered procedures being executed before the actual testing phase, now we are going to list through the things we can do after it. This is not the major purpose of our project, we concentrate on searching for the bug-exposing test data case, and not on its postprocessing. Anyway, we list our thoughts considering what might be done afterwards.

NB: unless stated the opposite, we can have only two kinds of outcome: it is either valid or not valid, with nothing in between!

◇ **Absolute majority**

If we found a situation where all (more than two different) XML validators have one opinion, and just one disagrees. We most probably can just discard its opinion and assume it is a bug in that particular XML validator.

◇ **Majority report**

With less degree of confidence we can conclude that, given any odd number of systems under test, the correct result is the one for which there are more answers and the minority have made a mistake. Assuming this can be wrong, though.

◇ **More detailed analysis**

Sometimes XML validators do not only fetch *errors* (one or more of which lead to the negative result), but also *warnings* (usually by having a small number of them a positive result is still accessible). For example, as we

will see later, .NET Framework validation API throws a warning for some reason when it encounters an element that is not found in the corresponding XML Schema schema or a document type definition in DDML. Those warnings or any other intermediate results may be examined further.

◇ **Third solid outcome**

While we assume at the beginning that there can be only two outcomes possible, namely valid and not valid, in practice sometimes we can get the third one, namely: program crash. If the XML validator crashes, we cannot conclude whether the file under concern was valid. Although it seems quite appropriate to assume it to be invalid, this crash may have no apparent link with the file, but with the environment.

As we will see in the Section 6.2, C# APIs also give explicit third result: a warning (opposed to an error). We assume a warning belongs to the valid outcome, but it is not that obvious.

◇ **Localisation**

It must be very useful to try to localise the error, if we think we have found one. For doing so we need to collect all the files on which the XML validators disagree and by means of some analysis try to figure out to which schema details those correspond. Besides, as it has been shown above, the error may be outside the actual system under test, but here by localisation we mean, for example *"the validator X does not pay attention to Y feature of the supplied XML Schema schema"*.

◇ **Report to the implementor**

This is the simplest way, and the most probable one: testers do not have to be involved in the process of *fixing* the bug they have found. However, this is the last way and the previous item certainly helps to empower it.

4.4 Changes to the Geno's architecture

We can now recall the architecture that we had in Chapter 3 (changes to Figure 3.2 are shown on Figure 4.5). If we look into the center of the new figure, we see that there is everything perfect, so we just need to correct the input language parser, as well as the output serialisation (additional boxes on the bottom of the figure).

The first part (which brings the understanding of the XML Schema schema to the test data generation engine) is called the *XSD Mapping*, it parses the grammar and creates all necessary sorts and constructors with which the engine can be initialised. With this issue we deal mainly in Chapter 5 due to its utter complexity.

The second new block (which should serialise the generated terms and pack each of them to separate XML file) is called the *XML Mapping* and is just about serialisation. This issue is much easier and explained in Section 4.4.5.

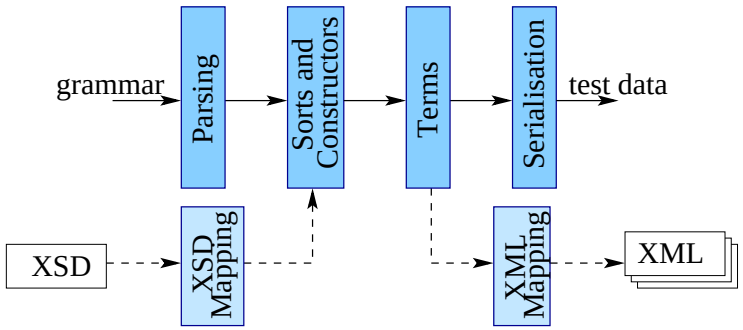

Figure 4.5: The changes brought to the original architecture of Geno: input and output blocks are substituted. Dashed lines represent new blocks relations with the old architecture.

4.4.1 Grammar input language

The data generator that should be the input of Geno is no more a special BNF-like plain textual language, but an XML Schema schema according to [3, 7, 37, 40]. It was our intent to change it that way and this task was successfully completed.

4.4.2 Internal structural changes

All changes that we have brought to the internal structures of Geno (the classes representing sorts and constructors) belong to debugging issues only and do not alter the core.

The environment (a special singleton class used for data exchange) has changed a lot, mainly due to the visualisation we want to develop (see Figure 4.6).

In the original design all different kinds of sorts were represented by one class, and the constructors had their onec class too, but the name of any constructor could have told a story about some special features (like being a star constructor, for example). We have inherited this tradition, and enhanced naming notation as follows:

Affix	Meaning	Applicability
A-	an XML attribute or an attribute group	sort, constructor
Aof-	all XML attributes of one element	sort
-S	an iteration, a star	sort
-R	a repeat, special case of a star	sort
-O	a choice (or), used when flattening	sort
-W	a sequence (with), used when flattening	sort

Note that some affixes which are applied to sorts only may also be encountered in constructors names, because those sometimes are composed by adding

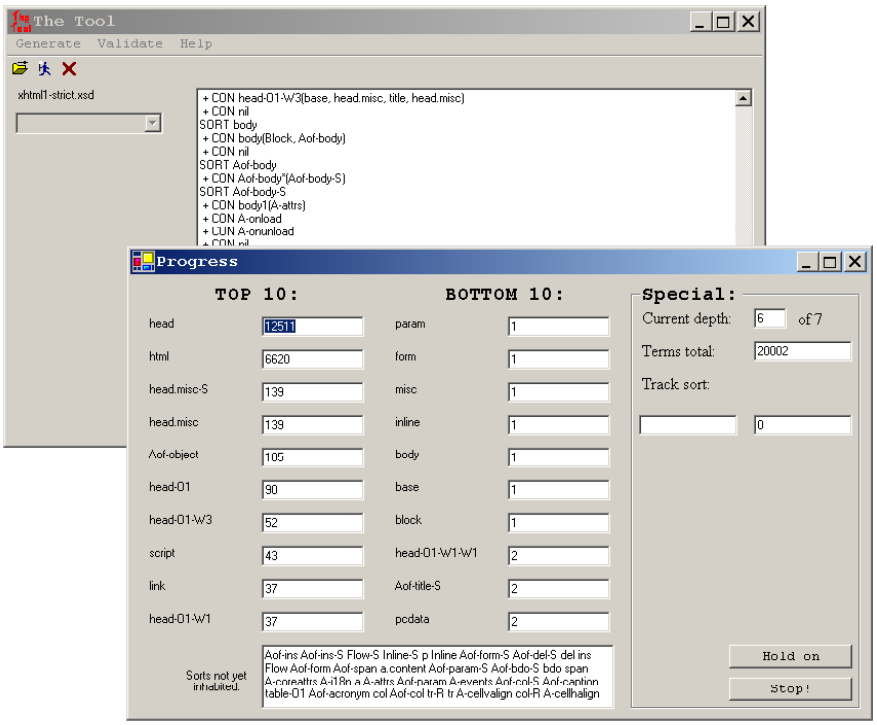

Figure 4.6: A screenshot: Geno generating terms for XHTML1 Strict: the main window on the left, the progress window on the right.

a number to its base sort name.

NB: the XML attributes shall not be confused with grammar attributes!

4.4.3 Terms generation algorithm

It was our intent not to change the terms generation algorithm, and we have kept it this way till now. Introducing some new features does change the way it behaves, but this is impelled by some mutations made on earlier stages.

4.4.4 Explosion visualisation

When we started experimenting with the new tool, we realised that the final results are not always comprehensible enough and they definitely do not give the whole picture (in the case of success we want to know some parameters of the resulting test data set, in the case of failure we want to know the reason for it). It turned out to be crucial for us to get the information before the actual explosion happens, to look on the parameters' influence without restarting the

application, to stop the application without the use of operating system features, to be able to proceed after the stop, etc.

The visualisation interface now looks like on Figure 4.6. Real term generation is executed in another thread, so we are able to stop it, to proceed, to manage it and, of course, to get information out of it easily. As it is shown on the figure, we have three main groups of number shown: the `TOP 10` with the sorts which have most terms. Usually there is one leader (or a group of leaders) and all the other sorts have much less terms; on some intermediate stages there is a set of sorts that keep collecting terms up to some number before that leader can be supplied with more terms again. The second group is a `BOTTOM 10` with the sorts that have least terms, and the third one shows completely uninhabited sorts (all sorts belong here when the generation begins and they all should be gone when it finishes).

If this is not enough, we can specify exactly the name of the sort we want to track and right on the next step we will have the number of terms that belong to it. The total number of terms generated so far and the depth (achieved as well as specified) are also shown to the tester.

4.4.5 XML Serialisation

A serialisation source code example for one of the very short grammars is shipped with Geno, it is a typical pretty-printer with some `switch`/`case` C# code, where each possible constructor of every sort is treated separately and specifically. Of course, when one changes the grammar or adds something to it, or even switches to another grammar, this code should constantly be rewritten.

Luckily for us, the XML provides us with three good aspects:

- ◇ All XML elements may be treated homogeneously.
- ◇ XML attributes are different from XML elements, but they are treated homogeneously too.
- ◇ There are special APIs for working with the XML in .NET Framework.

The former two let us write very compact code, which has only two types of treatment: one for all elements and one for all attributes (actually in practice we need the third one: for special constructors like `pcdata`—this can be seen in the code itself, explained below). The last one gives us good tools to implement that.

The actual code as is one can find in Appendix A.1, and in this section we will try to highlight the most important issues.

We have one class defined here with the three methods that are used from the outside. The first one instantiates the XML document (using Document Object Model, DOM, which provides us with the same features as Xerces), it is used at the beginning of serialisation process:

```
static public void Start()
 {
  doc = new System.Xml.XmlDocument();
 }
```

Its counterpart is used at the end of serialisation process, its result is a concrete XML file saved on a hard disk:

```
static public void Flush(string dir, int n)
{
 StreamWriter outXml =
  new StreamWriter(dir+"\\"+fillZero(n)+".xml");
 outXml.WriteLine("<?xml version=\"1.0\" encoding=\"UTF-8\" ?>");
 outXml.WriteLine("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 "+
 "Strict//EN\" \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd\">");
 outXml.WriteLine(doc.OuterXml);
 outXml.Close();
 }
```

`fillZero(n)` here is a silly method that concatenates zeros to the file name to give all the same length (for example, 12 will yield `00000012.xml`). DOM contains a special method for saving a file, but it does not provide either a correct header, nor a document type, so we have had to write it another way.

The last method used from the outside is the printing process executor:

```
static public void Print (Term t)
 {
  XmlElement el = doc.CreateElement(t.Op);
  if(t.Op=="html")
  {
  el.SetAttribute("xmlns","http://www.w3.org/1999/xhtml");
  el.SetAttribute("lang","en");
  }
  Print(t.Args, el);
  doc.AppendChild(el);
 }
```

Namespace should be constant for all XHTML documents [16], so we postpone its reference until the serialisation phase. Actually, this part may be removed if Geno were used for any other XML Schema schema, but it is not necessary: in that case we would never happen to have a constructor called `html`. However, in that case we would most probably want to write the namespace attribute of *that* schema.

Printing the bunch of subterms (the arguments) is easy:

```
static private void Print (Term[] ta, XmlElement parent)
 {
  for(int i = 0;i<ta.Length;i++)
   Print(ta[i],parent);
 }
```

And the last internal method prints a subterm into a ready parent XML element object. It also deals with the special kinds of constructors:

- ◇ `nil`—a dumb constructor representing element absence, nothing is added
- ◇ `pcdata`—a dumb constructor representing a text node (mixed content), "..." is added
- ◇ `A-`—constructor family representing XML attributes
- ◇ other dumb constructors of all kinds: mapping issue is described in more details in Chapter 5

The source code of the method looks therefore as follows:

```
static public void Print (Term t, XmlElement parent)
 {
  if (t.Op=="pcdata")
  {
   parent.AppendChild(doc.CreateTextNode("..."));
   return;
  }
  if (t.Op=="nil")return;
  if (t.Op.IndexOf("A-")>-1)
   {
    if(Env.Me.Real.Contains(t.Op))
     parent.SetAttribute(t.Op.Remove(0,2),"");
    else
     Print(t.Args,parent);
   }
  else
   {
    if(Env.Me.Real.Contains(t.Op))
     {
      XmlElement el1 = doc.CreateElement(t.Op);
      Print(t.Args,el1);
      parent.AppendChild(el1);
     }
    else
     Print(t.Args,parent);
   }
 }
```

The usage of serialisation and the test data generation is very simple too:

```
foreach(Term t in this.generator)
 {
  Serialisation.Shared.Start();
  Serialisation.Shared.Print(t);
  Serialisation.Shared.Flush("path\\", n++);
 }
```

4.5 XML Validators

First of all, we use the internal APIs of .NET Framework: it would be strange not to use them in a .NET-based project. Then, we list all Java data binding frameworks [27, 34] because of their omnipresence and importance, and give our short opinion about possibility to use each of them. Some of XML validation facilities in other languages are listed also.

4.5.1 C# API

The .NET Framework APIs certainly contain means to create, parse, check, modify and validate XML documents as well as XML Schema schemata. However, it lacks the ready to use class that we can instantiate with a schema and run a handy method every time we need to validate an XML document. As a part of our work we have written such a convenient wrapper in C#. It is a complicated class having special callback mechanism for reporting validation errors, but it still works much faster than any JVM-based tool, of course.

The core has been taken from the "Extreme XML :: Working with Namespaces in XML Schema" tutorial as a command-line tool. It has been adapted to our needs and transformed into a class that can be instantiated once for an XML Schema schema and has a method that can be executed to get the validation result for one XML file against that schema.

The actual code almost fits one page (it can be found in Appendix A.2). In consists of a definition of one class, `XSDValidator` with a handful of methods in it:

```
public class XSDValidator
 {
  private XmlSchemaCollection sc = new XmlSchemaCollection();
  private bool valid;
  public bool warn;
  ...
 }
```

The properties `valid` and `warn` are used to denote the presence of errors and warnings accordingly. The next important part is a standard callback method:

```
public void VCallback(object sender, ValidationEventArgs args)
 {
  if(!this.valid)return;
  if(args.Severity == XmlSeverityType.Error)
   this.valid = false;
  else if(args.Severity == XmlSeverityType.Warning)
   this.warn = true;
 }
```

The constructor of the class registers it as a delegate [41]:

```
public XSDValidator(string xsd)
 {
  if((xsd==null)||(xsd==""))return;
  sc.ValidationEventHandler +=
   new ValidationEventHandler(VCallback);
  sc.Add(null, xsd);
 }
```

`null` here corresponds to the namespace. If it is given, replace `null` with a string (in the real source code we have another constructor that is not presented here—we will not use namespaces anyway). And the actual method used for validation looks as follows:

```
public bool IsXmlValid(string xmlFile)
 {
  XmlValidatingReader vr =
   new XmlValidatingReader(new XmlTextReader(xmlFile));
  vr.Schemas.Add(sc);
  vr.ValidationType = ValidationType.Schema;
  this.valid = true;
  this.warn = false;
  vr.ValidationEventHandler +=
   new ValidationEventHandler(VCallback);
  while(vr.Read()&&this.valid);
  return this.valid;
 }
```

It creates a validating reader object—a standard XML parser that tries to build a parse tree according to the validation type specified (the XML Schema schema in our case) and calls a delegate in a way very similar with the exception handling mechanism. We try to read the file until it ends or until a non-conformance is found. The validation result is returned as is, and the presence of warnings can be understood by looking directly to the object's `warn` variable.

4.5.2 Java XML data binding frameworks

1. Castor, http://www.castor.org

   Castor is an open source data binding framework for Java. It is claimed to be the shortest path between Java objects and XML documents. Castor provides Java to XML binding, Java to SQL persistence and some more. Although it can be used in our project, this would involve quite some amount of programming in Java, which is in no way inside the intended scope of work.

2. Enhydra Zeus, http://zeus.objectweb.org

   Zeus is, in a nutshell, an open source Java-to-XML data binding tool. It provides means of taking an arbitrary XML document and converting that

document into a Java object representing the XML. That Java object can then be used and manipulated like any other Java object in the JVM. Then, once the object has been modified and operated upon, Zeus can be used to convert the Java object back into an XML representation.

3. Java Architecture for XML Binding (JAXB), http://java.sun.com/xml/jaxb

   A framework that provides a convenient way to bind an XML Schema schema to its representation in Java code. As it is stated by the authors, this makes it easy to incorporate XML data and processing functions in applications based on Java technology without having to know much about the XML itself.

   Since it is not the purpose of our project to write a new validator, we can take the official Sun Multi-Schema XML Validator 1.2 (MSV from now on), which is free for download and for use (AS-IS software with limitations on redistribution). The validator is very nice, stable, has plain text output that can be easily redirected to `grep` tool or something alike. However, we have to admit that it is as slow as a Java-based tool could possibly be. It cannot validate several XML documents against one XML Schema schema at once (as our own C#-based tool can). Nevertheless, this validator is approved for use. Can be downloaded freely from http://developers.sun.com/dev/coolstuff/schema.

4. JBind, http://jbind.sourceforge.net/

   A framework that generates Java code from XML documents (for our case, can be used to generate a validating parser). Impossible to download automatically, textual request to the author is sent.

   The author has replied that there is a special project where the JBind-based XML validator is developed, but no ready software is yet available.

5. Quick, http://qare.sourceforge.net/web/2001-12/products

   Data binding system for transforming XML into Java objects and Java objects into XML. Quick builds on QJML, a binding schema which connects XML elements to Java classes. Quick can be used to generate the Java code for data classes, but it keeps the serialisation code separate from data classes. Quick includes utilities for transforming DTDs into QJML, QJML into serialising logic, QJML into HTML documentation, and finally QJML into data classes. Too complicated, and no validation engine ready to use.

6. JiBX, http://www.jibx.org

   This is yet another Java data binding architecture, it is pretty new, developed by an expert after several reviews about all the others [34, 35, and others]. It claims to have the best performance, but due to its young age has no ready to use validator.

4.5.3 Validation facilities in other languages

- XSV, http://www.ltg.ed.ac.uk/~ht/xsv-status.html

  This is lightweight Python-based validation engine, simple and available under the GPL. It is intended to use with the CGI and requires additional packages to be installed (such as LtXML). Request for installation sent to the helpdesk.

- HaXml, http://www.cs.york.ac.uk/fp/HaXml

  An example of utilities collection for using XML in Haskell programming language. Unfortunately, contains only a DTD-based validator.

- ElCel Technology XML Validator, http://www.elcel.com/products/xmlvalid.html

  The ElCel Technology XML Validator is a free command-line utility built using OpenTop, which is a new C++ class library that provides the means to create powerful and robust network-oriented applications and greatly simplifies their development. It has been designed to highlight some of the strengths of the underlying XML Toolkit. It is based on DTD.

Chapter 5

XML Schema Mapping

This chapter mainly concentrates on the *XSD Mapping* block from Figure 4.5, on how it operates and how it is internally organised.

5.1 XML Schema vs. (E)BNF

5.1.1 Input language of Geno

The input language of Geno is very simple, but somewhat close to the BNF [30]. Grammar description is signature-oriented: for every sort there is given either a set of constructors (and a term of the sort may be constructed by any of them):

```
sort = constructor1(argument-sort1, argument-sort2)
     | constructor2(argument-sort3)
     | constructor3;
```

or a single sequence constructor:

```
sort = (another-sort*);
```

or a single product constructor:

```
sort = (another-sort, yet-another-sort);
```

Any sort name occurrence might be preceded by a bunch of attributes regarding control mechanisms which deal with depth (`MaxDepth`, `MaxRecDepth`, `MaxRecUnbalance`), and any constructor name occurrence might be preceded by a bunch of attributes regarding control mechanisms which deal with combinations (`Oneway`, `Twoway`, `Unordered`, `NoDuplicates`, `MinLength`, `MaxLength`). These all can be found in the documentation, but they are also described in Section 3.3 which deals with control mechanisms implemented in Geno.

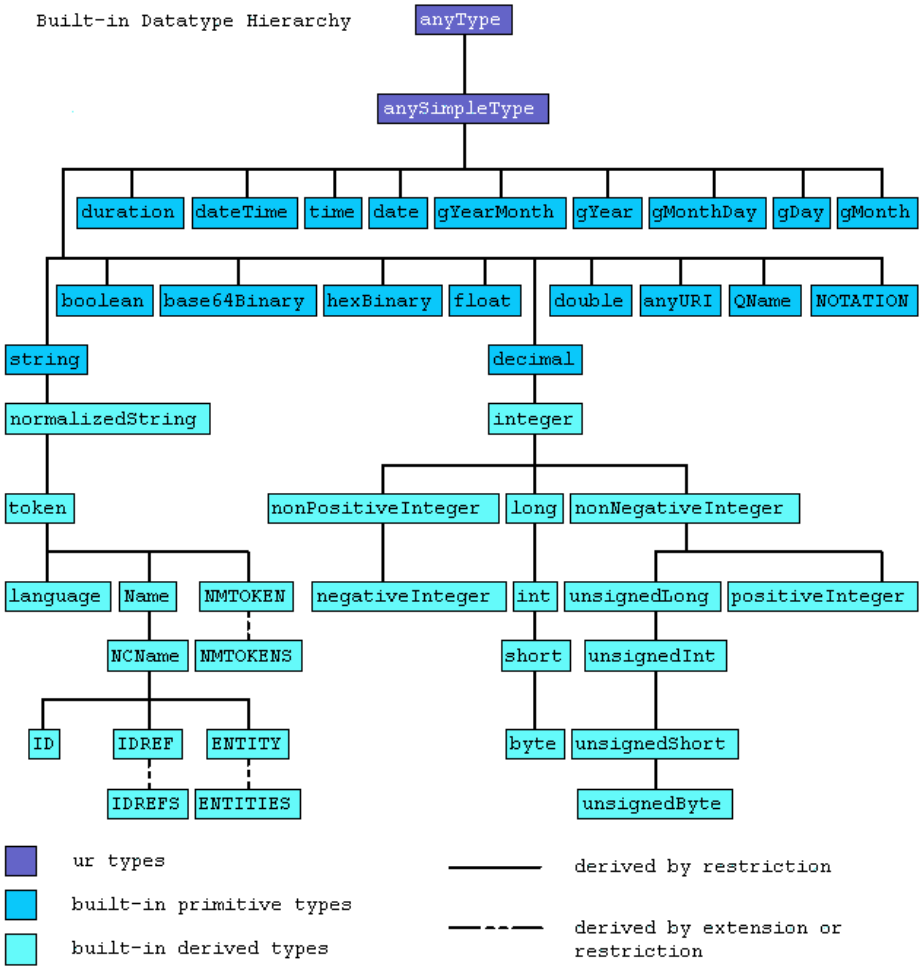

Figure 5.1: Datatype system of the XML Schema W3C Recommendation: only *simple* types presented. [2].

5.1.2 XML Schema

Unfortunately, the XML Schema language [2, 3, 7, 8, 11, 12, 25, 36, 37] cannot be described on one page. However, we can state it is an element-oriented grammar definition language.

There are *simple types* which describe how a single value can look like (for example, strings and integers are simple types). They are used mostly for the attributes. There are also *complex types* which tell about from which elements can their value be constructed and in what order (for example, in XHTML [16] `html` has a unique complex type that refers to the elements `head` and `body`, which also belong to their complex types).

Usually an element is defined far from the place where it is used, allowing several references to the same elements to appear in different places in the schema. The content of a complex type (called *complex content* when it has subelements and *simple content* if not; attributes are allowed for both) may be *mixed* (that means: may have plain text (*character data*) among its elements). Every element occurrence not only contains a reference to its definition (it might contain the actual definition instead), but also *occurrence constraints* saying how many of those elements may appear in that place, what their values should be (`fixed` attribute) and what are the `default` values. Complex types can be declared separately from their place of usage, too.

It may look silly, but the complex types are that simple. There are also additional features like *groups* of elements and attributes, which make life even easier. Unlike them, the simple types are very complex. There are 45 built-in simple types in the XML Schema standard—see Figure 5.1 [2]. They all can be used as they are (which rarely happens) or be extended/restricted directly in the place where they are used. If this is not enough, one can invent its own *atomic, list* or *union* datatypes.

5.1.3 XML Attributes

The XML attributes are somewhat different from the XML elements:

- ◇ They may appear only once per described occurrence, no sequences are allowed.
- ◇ They cannot belong to complex types.
- ◇ Their order of appearance in not important.
- ◇ They may not appear at all (unless explicitly stated otherwise).

5.2 Grammar adaptation

In order to map the complicated structure given by an XML Schema schema to a rather flat signature description, we need to introduce dummy sorts and dummy constructors. These are certainly to be eliminated later, on the serialisation

phase. However, on the first stage we need them for sure. Just to give a simple example:

$$A \rightarrow B^{\star}|C$$

cannot be parsed into the internal representation as is, just because the iteration (closure, $\star$) must be the only alternative. Therefore we bring a dummy sort in:

$$\begin{aligned} A &\rightarrow B_S|C \\ B_S &\rightarrow B^{\star} \end{aligned}$$

Also, Geno cannot understand the *brackets* in the extended BNF:

$$A \rightarrow B, (C|D), E, ((F|G)|H^{\star})$$

should be automatically transformed to:

$$\begin{aligned} A &\rightarrow B, CD, E, FGH \\ CD &\rightarrow C|D \\ FGH &\rightarrow FG|H_S \\ FG &\rightarrow F|G \\ H_S &\rightarrow H^{\star} \end{aligned}$$

The more complex the given structure is, the more dummy sorts we need to parse it well. In fact, for each complicated piece of grammar we need some way to deal with.

More complicated and solid grammar adaptation techniques are given in [19] and [24], they are described in the related work section.

5.2.1 Documentation

XML Schema standard allows for so-called annotations containing documentation about the element or complex type where they are encountered. As long as they do not provide any computer-intelligible information about the grammar, they are completely discarded.

5.2.2 Element

Obviously, every element becomes a sort. Not true. It gives a name to a sort, but the actual content of the sort is determined by the complex type of the element.

5.2.3 Complex type

If the complex type is encountered as a part of an element, it becomes the sort with the name of that element, if it is encountered by itself, it gives its name to the derived sort. The complex type cannot be used in the XML Schema schema by itself, but without a name (because then it would not be possible to refer to it).

In practice a complex type consists of either a particle (a sequence or a choice) or a complex content (that happens usually when the content must be mixed). The latter case is easy: it yields one constructor for the complex type's sort for being mixed and some others for XML attributes (if any).

The former case if, however, not that simple: any sequence produces one constructor for the complex type's sort. It has arguments that correspond to every sequence piece. Any choice produces set of constructors, one for each piece inside it. The structures that are too complicated must be flattened: additional dummy sorts are created in order to refactor them.

As an example we take a piece of XHTML grammar [16]:

```
<xs:element name="head">
 <xs:annotation>
  <xs:documentation>
   content model is "head.misc" combined with a single
   title and an optional base element in any order
  </xs:documentation>
 </xs:annotation>
 <xs:complexType>
  <xs:sequence>
   <xs:group ref="head.misc"/>
   <xs:choice>
    <xs:sequence>
     <xs:element ref="title"/>
     <xs:group ref="head.misc"/>
     <xs:sequence minOccurs="0">
      <xs:element ref="base"/>
      <xs:group ref="head.misc"/>
     </xs:sequence>
    </xs:sequence>
    <xs:sequence>
     <xs:element ref="base"/>
     <xs:group ref="head.misc"/>
     <xs:element ref="title"/>
     <xs:group ref="head.misc"/>
    </xs:sequence>
   </xs:choice>
  </xs:sequence>
  <xs:attributeGroup ref="i18n"/>
  <xs:attribute name="id" type="xs:ID"/>
  <xs:attribute name="profile" type="URI"/>
 </xs:complexType>
</xs:element>
```

(Quoted exactly as in [16, lines 571–602])

Or, in somewhat more familiar EBNF notation (*hm* as a short form for

`head.misc`):

$$\begin{aligned} head \quad \rightarrow \quad & hm, \\ & ((title, hm, (base, hm)?)|(base, hm, title, hm)), \\ & (i18n|id|profile)^\star \end{aligned}$$

It would be parsed as follows:

```
head = (head.misc, head-O1, Aof-head);
head-O1 = head-O1-1(head-O1-W1)
        | head-O1-2(head-O1-W2);
head-O1-W1 = (title, head.misc, head-O1-W1-W1);
head-O1-W1-W1 = (base, head.misc)
              | nil;
head-O1-W2 = (base, head.misc, title, head.misc);
Aof-head = [Unordered, NoDuplicates] (Aof-head-S*);
Aof-head-S = head1(A-i18n)
           | A-id
           | A-profile
           | nil;
```

This can be understood much easily if one is familiar with our naming notation: O means sequence (from the word "or"), W means choice (from the word "with"), A means an attribute, Aof—a set of attributes of one element. nil is a special constructor that can make nothing out of nothing, but the result can be a term of any sort. Groups are described below, right after the attributes. We deal with the occurrence constraints in the last sections.

5.2.4 Attributes

For every element we create a special sort representing all its attributes. It is an iteration (a star), so this always gives one more sort. Every actual attribute becomes a constructor of that inner sort, if it is declared inside the parent element declaration, or it becomes a sort in the place where it is defined and the constructor with the name of the element and that sort as its only argument, if it is just referred to. Every attribute sort (the inner one, Aof-*element*-S) has an additional nil constructor for the case of no attributes. The outer sort (Aof-*element*) is granted the control mechanism parameters limiting its pair-wise coverage.

If an element has no attributes, no additional sorts are created.

Attributes can be combined into named *groups*. Referring to an attribute group is equivalent to referring to all of its attributes. The straightforward substitution, however, cannot be done because we cannot predict that the point where the attribute group is defined is earlier in an XML file than its usage. Therefore, we pull the same trick as with reference to attributes: the constructor named after the element has an argument of a sort, which corresponds to the attribute group itself.

5.2.5 Group

XML Schema enables groups of elements to be defined and named so that the elements can be used to build up the content models of complex types. Unnamed groups of elements can also be defined, and along with elements in named groups, they can be constrained to appear in the same order (sequence) as they are declared. Alternatively, they can be constrained so that only one of the elements may appear in an instance [16].

In practice almost always a group is a long choice of different elements (or other groups) which is used more than once throughout the XML document.

A group can only occur standing alone. It can therefore be treated the same way the reference to the element or complex type is: the definition of a group yields a sort with all kinds of constructors; the reference to a group yield an argument of that sort (if used inside the sequence) or a constructor taking one argument of that sort (if used inside another choice).

5.2.6 `maxOccurs="unbounded"`

Almost any element in the XML may have `minOccurs` and `maxOccurs` attributes (it seems to be mostly used inside the sequences, however). By default they both are equal to 1, but can be given any number as well as special "∞" value called `unbounded`. Actually, `minOccurs=0` and `maxOccurs=unbounded` means $\star$ in EBNF, while `minOccurs=1` (or default) and `maxOccurs=unbounded` means $+$. For good or bad be it, but unlike the EBNF, XML Schema allows its users to specify exact minimum and maximum number of element occurrences. Usually this is a big pain for XML API developers, because then such information must be stored with any schema element, but it does not give us a fright: the internal representation of a grammar in Geno stores a lot of grammar attributes anyway. Among others there are `MaxLength`, `MinLength`, which are pretty much equivalent to what we have in the XML Schema.

In Geno grammar description language [22] we do not have different means for $\star$ and $+$, but we have one constructor *type* which name ends with a star. More details can be described with grammar attributes.

Suppose we have this element in XML Schema:

```
<xs:element name="ol">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="li" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

(Adapted from [16])

In EBNF this would have looked like:

$$ol \rightarrow li^{+}$$

In the input language of Geno it becomes an attributed grammar:

```
ol = [MinLength=1]ol*(li);
```

However, it is nearly impossible to encounter such a peaceful situation in real life. Even this simplest *ordered list* element in XHTML1 Strict looked as follows:

```
<xs:element name="ol">
  <xs:annotation>
    <xs:documentation>
      Unordered list
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="li" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attributeGroup ref="attrs" />
  </xs:complexType>
</xs:element>
```

(Exactly as in [16])

Let alone the comments, we have a reference to a bunch of attributes here! That means that the constructor of *ol* actually takes two parameters, one of them being attributes and the other being the closure. Thus, we have in EBNF:

$$ol \rightarrow li^{+}, attrs$$

Which yields a data generator description:

```
ol = ol(attrs, li-S);
li-S = [MinLength=1]li-S*(li);
```

5.2.7 `minOccurs="0"`

Actually, we could have done the same thing in this case, if it would not be so: `maxOccurs` is 1 by default, that means for us that if only `minOccurs` is present, this is not a closure, but a possible non-terminal:

```
<xs:element name="table">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="caption" minOccurs="0" />
      ...
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

(Adapted from [16])

which gives us in an EBNF:

$$table \rightarrow caption?, \cdots$$

or, closer to Geno:

$$\begin{aligned} table &\rightarrow caption_M, \cdots \\ caption_M &\rightarrow \varepsilon | caption \end{aligned}$$

and in the input language of Geno:

```
table = table(caption-M, ...);
caption-M = nil | caption-M(caption);
```

where `nil` is a special dummy constructor that evaluates to nothing (typed nothing, actually) during the serialisation phase. Lucky enough, this is all we need for `minOccurs`: unlike its cousin, it cannot be unbounded.

5.2.8 Arbitrary numbers in `minOccurs` and `maxOccurs`

For any other case we can have one general rule: map the numbers to attributed grammar used in Geno.

```
<xs:element name="foo">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="bar" minOccurs="i" maxOccurs="j"/>
      ...
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

where i and j are some natural numbers. This cannot be expressed in the EBNF, but in the input language of Geno it becomes:

```
foo = foo(bar-N, ...);
bar-N = [MinLength=i,MaxLength=j]bar-N(bar);
```

We must admit this is done for completeness's sake only: we have not seen any real XML Schema schema that makes use of that opportunity. We must admit, too, that the possibility to give precise values to any element occurrence may be useful for the testers when they realise that a schema does not work at all with our test data generator (the complex structure drives it out of memory before any meaningful terms are actually generated) and they need to restrict the combinatorial generation somehow. For more details please consult Section 2.3 about control mechanisms.

Chapter 6

Results

6.1 Experiments: directions and details

6.1.1 Chosen scenarios

We have decided to concentrate on three scenarios, namely:

- **Huge valid test data set**

  We want to generate a lot of presumably valid XML files from one XML Schema schema and to put them into three XML validators: the C#-based one, the Java-based one and the Python-based one.

- **Grammar mutation**

  We want to mutate the grammar in order to generate lots of invalid XML files mixed with valid ones, put them into the same three validators in order to see of they can distinguish between them.

- **Point-wise stress testing**

  We want to use the control mechanisms in such a way that everything besides the needed point is not explored (minimum amount of terms is generated) and everything linked with the point of interest is exhausted (up to the maximal possible depth).

The total summary of how big the grammars and the corresponding test data sets were is shown on the table below. Depth is defined as on the page 7; the numbers of sorts and constructors are given to get an impression about how big the grammars are; the number of terms: generated and belonged to the root sorts—the latter can be seen as the actual size of a test data set in files:

	Depth reached	Sorts in the signature	Constructors in the signature	Terms total	Terms of the root sort
Valid	8	234	478	9914261	37240
Mutation	5	234	684	347339	64247
Stress	1000	5	6	1500	499

It turned out to be infeasible to generate valid test data set for a complex grammar like the XHTML [16]: in order to make all sorts inhabited, we had to go till depth 29, which leaves us with almost infinite number of terms to be generated. If a grammar is too big to be handled as a whole, we may try to *slice* it into feasible parts and test them separately (it is somewhat related to multi-way coverage). In XHTML we do not have always to vary `head`s and `body`s at the same time. Therefore, we have decided to concentrate on generating `body`s and use the shortest completion algorithm to get one `head` for all of them. Even then, we could not cover all the sorts and went down to depth 8.

With grammar mutation technique (ε rule added to each sort) we were able to generate more (mostly invalid) terms of the root sorts earlier with the depth. We can easily see the result in the table: the terms of the root sort form larger part of a whole.

For stress testing we picked up another schema, so that we could concentrate better on the nesting of one constructor and forget about the rest:

```
<?xml version="1.0" encoding="ISO-8859-5"?> <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">
 <xs:element name="root">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="rec" maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="rec">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="rec" minOccurs="0"/>
    <xs:element ref="foo"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="foo">
  <xs:complexType mixed="true"/>
 </xs:element>
</xs:schema>
```

Then we tweaked all the control mechanisms in such a way that the test data generator explored the grammar as deep as possible considering the recursion, but ignored all items of secondary interest. The great thing is that by doing so we brought the test data generator down to linear growth (check the table)!

6.1.2 Implementation details

Hardware

We will use the term *Windows machine* to denote an AMD Athlon 2200^{+}-based Fujutsu-Siemens personal computer with 1.8GHz CPU, 1Gb memory and Windows XP Professional installed and the term *UNIX machine* to denote a

Sun UltraSPARC-III+ workstation with two 900MHz CPUs, 4Gb memory and SunOS 5.8 installed.

.NET-based XML validator in C#

This one was home-made, therefore it did not require any additional strange movements. It was executed on a Windows machine and produced a log file with file names and their results. So far it proved to be the fastest one of the three.

JVM-based Sun XML validator in Java

This validator ships as a JAR file, so we had to re-execute it for each XML file. It consumes time, which does not make the JVM any faster. Some tests were executed on the same Windows machine using the batch file generated by another batch file. This complicated system was necessary to use because of the operating system's `FOR/IN` command. For example, the straightforward way:

```
FOR %%A IN (c:\generated\*.xml)
 DO  java -jar msv\msv.jar x.xsd %%A
  |  grep -E "validating|document"
  >> result.txt
```

gives in somewhere between 10000 and 20000 files (depends on the size of every file). Besides, it silently skips some files (unacceptable for us—this can ruin the whole experiment). Therefore, actual execution in the batch file is replaced with the `echo` shell command that makes another batch file of the form:

```
@echo off
java -jar msv\msv.jar x.xsd c:\generated\0000000.xml
     | grep -E "validating|document"
java -jar msv\msv.jar x.xsd c:\generated\0000001.xml
     | grep -E "validating|document"
java -jar msv\msv.jar x.xsd c:\generated\0000002.xml
     | grep -E "validating|document"
...
```

Surprisingly, this one works perfectly. Alternatively, we may run experiments with MSV on a UNIX server. The batch file uses the `ksh`'s `for/in` loop:

```
#!/usr/bin/ksh

for xf in generated/*.xml
 do java -jar msv/msv.jar x.xsd $xf
  |  /usr/xpg4/bin/grep -E "validating|document"
  >> result.txt
done
```

The UNIX server works a bit slower that the Windows machine, but has tremendously useful `at` mechanism, as well as all advantages of a remote system.

XML validator in Python

Windows version of the XSV is shipped as an EXE file intended for CGI use (which output can still be redirected with the `2>` command). UNIX version comes as a Python module that can be used easily:

```
#!/usr/local/bin/python
from XSV.driver import runit

class zzz:
 def write(self,z):
  if z.find("bug")>-1:
   self.z=''
  if self.b==1:
   self.b=0
   self.s=z
  if z.find("instanceErrors")>-1:
   self.b=1
 def __init__(self):
  self.b=0

for i in xrange(0,37241):
 xmlfile = 'RV/'+fillZero(i)+'.xml'
 print "validating", xmlfile
 res = runit(xmlfile, ["x.xsd"])
 o = zzz()
 res[0].printme(o,[],[])
 if o.s.split('"')[1]=='0':
  print "the document is valid."
 else:
  print "the document is NOT valid."
```

All we need is to format the output in the same way the other two validators use.

6.2 Results: environment and validators

6.2.1 Differences in validators

Lax validation

In some cases the XSV (for example, if it gets an empty XML Schema file), switches to another mode called *lax* validation (as opposed to *strict* validation),

which allows it to accept almost anything it can find in XML files. An example of such a schema follows:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" />
```

Unknown element

If the .NET APIs see an element that is nowhere to be found in the given XML Schema file, it throws a *warning* (as opposed to an *error*), which some can interpret as a positive result. An example of a XML file that can be "valid with warnings" against the XHTML schema follows:

```
<?xml version="1.0"
      encoding="UTF-8" ?>
<!DOCTYPE html
          PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<nil />
```

The UNIX version of the XSV always tries to contact the actual URL of the document type definition and fails.

Duplicate attribute

If we generate an invalid test data which has at least one element with two or more attributes with the same name (in other words: if we negate `NoDuplicate` parameter of combination control), the XSV does not report an error, but breaks:

```
<xsv xmlns="http://www.w3.org/2000/05/xsv"
     instanceAssessed="false"
     schemaDocs="y.xsd"
     target="/home/vadim/valid/RV/0000071.xml"
     version="XSV 2.7-1 of 2004/04/01 13:40:50">
 <bug>validator crash during target reading</bug>
 <XMLMessages>Error: Repeated attribute style
 in unnamed entity at line 3 char 102 of
 file:///home/vadim/valid/RV/0000071.xml
 </XMLMessages></xsv>
```

Note that there is no `instanceErrors` line in the output, but an `instanceAssessed="false"` instead: therefore, XSV tells us it did not succeed reading the file. From the point where we have found this odd behaviour on, we consider this result as a negative one.

The C# APIs also react differently by raising an exception `XmlException` — again, after noticing that we start treating this exception as a negative validator result.

Stress nesting

The XSV turned out to be vulnerable for point-wise stress testing (upon our own schema, not the XHTML one). On the 493^{rd} nesting level of the `<rec>` element

the XSV broke, but the other two XML validators did not. The problem is not in a validator, but in the underlying architecture: no error was reported by the XSV itself, but by the Python instead. Besides, it did not break under UNIX (at least, up to 499^{th} nesting level).

6.2.2 Environmental errors

`FOR` command in Windows

Two errors have been found in this point. The first one is: using the `FOR` command for the whole test data set (rarely smaller than 20000) compels halting the validation process somewhere around the 15000^{th} XML file with the error message (for Windows XP Professional):

```
Not enough storage is available to process this command.
Out of memory.
```

The second one is: it skips about 0.03% of all files if executed with a bearable amount of files (say, 10000). The skipped part does not seem big, but with a big number of files this lead to severe consequences. For instance, since that bug was encountered, we had to log not only the output of the validation, but also the XML file name.

The log file in such a case could look as follows (real example listed):

```
...
the document is NOT valid.
validating c:\generated\0011009.xml
the document is NOT valid.
validating c:\generated\0011010.xml
the document is NOT valid.
validating c:\generated\0011012.xml
the document is NOT valid.
validating c:\generated\0011013.xml
the document is NOT valid.
validating c:\generated\0011014.xml
...
```

You can see that no file called `0011011.xml` has been executed: and yes, it existed.

Chapter 7

Related Work

7.1 XML Conformance Testing

The World Wide Web Consortium itself has an initiative about XML conformance testing [39]. The test data suite contains over 2000 test files and an associated test report that contains background information on conformance testing for XML as well as test descriptions for each of the test files included in this release.

Its analysis and documentation have shown that it was made exclusively by hands during a discussion in the mailing list. Besides, there is no single XML Schema schema in it, but DTDs only.

7.2 Testing hypotheses

There is a notion of hypothesis introduced in series of works about algebraic abstract data types [1, 9, 10, 38]. Hypotheses represent and formalise common test practices, they usually have the form of *"if a property holds for every item in a subset, it holds for the whole set"*. In more practical words it will be: "if a system under test behaves normally for every test case in a test set, it always behaves normally". The strongest hypothesis is that the program is already correct (it leads to the smallest test set: $\emptyset$), the weakest one is associated with the exhaustive test data set. Usually people use something in between:

- **Σ-adequacy hypothesis**

  This non restrictive hypothesis means that all exported operations of the program under test are specified [1].

- **Regularity hypothesis**

  This hypothesis translates our depth control (page 7) on the formal ground of algebraic data types [1].

- **Ω-Regularity hypothesis**

  Stronger version of the same: defined operations are not taken into account [1].

- **Uniformity hypothesis**

  It states that if a formula is true for some value, it holds always. This usually leads to replacement of variables of imported sorts by ground terms (assuming there are already several regularity hypotheses that eliminate non-imported sorts) [1, 9]. Can be applied to one variable or to a domain.

These hypotheses give the good consistent picture, but they are not everything that is possible! As an example of informal test hypothesis [38] gives the following: we can assume that our program is a finite state machine (if the specification if a FSM, it would be then easier to compare them).

We can see (and use) the hypotheses as a foundation for our control methods.

7.3 Coverage criteria

Coverage criteria are sets of rules that help to determine whether a test suite has adequately tested a program and guide the testing process [28]. In general, they provide an assessing mechanism for a tester to calculate how good the test suite cover the wanted aspects. We list here some of them that we think relate to our project.

- **Rule coverage**

  Rule coverage simply means that a test set explores all rules of a grammar [14, 15, 20].

- **Branch coverage**

  This coverage criterion is a generalisation of a rule coverage (that is applicable only to context-free grammars) [21].

- **Position coverage**

  A minor generalisation of branch coverage by taking parameter positions of the function symbols into account [21]. Equivalent to the branch coverage.

- **Context-dependent branch coverage**

  Another generalisation taking all possible classes into account: a context-dependent version of the branch coverage [20, 21].

- **Reachability coverage**

  While context-dependent branch coverage relates only adjacent symbols in terms, this one, being the obvious generalisation, looks also for remote pairs of function symbols. [21].

- **Unfolding coverage**

  This criterion introduces special treatment for recursion [21].

- **Two-dimensional approximation coverage**

  The criterion is used for attribute grammars and to other first-order declarative programs. The two dimensions are syntax and semantics [15].

- **Abstract domain coverage**

  This one proposes that we apply the search algorithm to find a new test case and add it to the test set only if it improves the coverage, and keep on repeating that until the full coverage is achieved [15].

- **Rule update coverage**

  Rule updates represent the system reaction to particular events or conditions. If a test set tests every function update of each rule, this criterion is fulfilled [14].

- **Parallel rule coverage**

  This criterion assures the testing of interaction between rules and therefore helps in discovering inconsistent domains [14]. The natural extension exists called **strong parallel rule coverage**.

- **Modified condition/decision coverage**

  This criterion wants that every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once and each condition in a decision has been shown to independently affect the decision's outcome [14, 17].

7.4 Miscellaneous

There is a work thread considering formal grammar adaptation techniques. [19] is a fundamental formal paper in that area, while [24] describes a particular framework for grammar transformations in SDF. Also, some ongoing work is about combinatorial test data generation with the same control mechanisms in the ASF+SDF Meta-Environment (not yet published).

Chapter 8

Conclusion

As the title page says, this Master's thesis is about the concepts, the implementation of them and the XML case study. This short chapter will summarise the contributions to those three aspects of our work.

The major contributions of this project are:

- The infrastructure of the XML-based test data generator is designed and implemented. Geno has now full support for XML and XML Schema.
- The XML-based case study (XHTML Strict 1.1). Geno has been tried on a real example of a grammar, and its output was used to test three XML validators. Bugs were successfully found.

The contributions not of major importance are:

- Generation process visualisation is a completely new extension for Geno.
- Illustration and rationalisation of control mechanisms for combinatorial test data generation.

Further points of research include, among the others:

- **XML Schema schema for XML Schemas**

  One can treat the "XML Schema schema for XML Schemas" [40] as just another XML language, for which we can generate test data. In this case, test data will be XML Schema schemata themselves, and afterwards we can use them to generate XML files.

- **Control mechanisms**

  Specifying control mechanism parameters with the XML Schema features seems very promising. On the one hand, we should somehow distinguish the original grammar entities from the control mechanism arguments that are introduced by a tester. On the other hand, it may be better not to mess with the schema file itself and invent another way to specify them. This issue can also be linked with the next one.

- **Full interactiveness**

  It may be interesting to introduce more interactive explosion visualisation mechanisms to the existing application. They proved to be very helpful during this project and we hope to get more of them later.

Appendix A

Source Code (C#)

A.1 Serialisation.cs

```
using System;
using System.IO;
using System.Xml;
using TDGenerator;

namespace Serialisation {
    public class Shared
    {
        public static XmlDocument doc;

        static public void Start()
        {
            doc = new System.Xml.XmlDocument();
        }

        static public void Flush(string dir, int n)
        {
            StreamWriter outXml = new StreamWriter(dir+"\\"+fillZero(n)+".xml");
            outXml.WriteLine("<?xml version=\"1.0\" encoding=\"UTF-8\" ?>");
            outXml.WriteLine("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Strict//EN\""
                            +" \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd\">");
            outXml.WriteLine(doc.OuterXml);
            outXml.Close();
        }

        static private String fillZero(int n)
        {
            if(n<0) return n.ToString();
            if(n<10)return "000000"+n;
            if(n<100)return "00000"+n;
            if(n<1000)return "0000"+n;
            if(n<10000)return "000"+n;
            if(n<100000)return "00"+n;
            if(n<1000000)return "0"+n;

            return n.ToString();
```

```
        }

        static public void Print (Term t)
        {
            XmlElement el = doc.CreateElement(t.Op);
            if(t.Op=="html")
            {
                el.SetAttribute("xmlns","http://www.w3.org/1999/xhtml");
                el.SetAttribute("xml:lang","en");
                el.SetAttribute("lang","en");
            }
            Print(t.Args, el);
            doc.AppendChild(el);
        }

        static public void Print (Term t, XmlElement parent)
        {
            if (t.Op=="pcdata")
            {
                parent.AppendChild(doc.CreateTextNode("..."));
                return;
            }
            if (t.Op=="nil")return;
            if (t.Op.IndexOf("A-")>-1)
            {
                if(Env.Me.Real.Contains(t.Op))
                    parent.SetAttribute(t.Op.Remove(0,2),"");
                else
                    Print(t.Args,parent);
            }
            else
            {
                if(Env.Me.Real.Contains(t.Op))
                {
                    XmlElement el1 = doc.CreateElement(t.Op);
                    Print(t.Args,el1);
                    parent.AppendChild(el1);
                }
                else
                    Print(t.Args,parent);
            }
        }

        static public void Print (Term[] ta, XmlElement parent)
        {
            for(int i = 0;i<ta.Length;i++)
                Print(ta[i],parent);
        }
    }
}
```

A.2 XSDValidator.cs

```
using System;
using System.IO;
using System.Xml;
```

```
using System.Xml.Schema;

/// <summary>
/// "Extreme XML :: Working with Namespaces in XML Schema"
/// Extracted by VVZ (as a command-line app)
/// Changed by VVZ (to a useful object)
/// </summary>
namespace TheTool {
    public class XSDValidator
    {
        private XmlSchemaCollection sc = new XmlSchemaCollection();
        private bool valid;
        public bool warn;

        public void ValidationCallback(object sender, ValidationEventArgs args)
        {

            if(!this.valid)return;
            if(args.Severity == XmlSeverityType.Error)
                this.valid = false;
            else if(args.Severity == XmlSeverityType.Warning)
                this.warn = true;
        }

        public XSDValidator(string xsd)
        {
            if((xsd==null)||(xsd==""))return;
            sc.ValidationEventHandler
                += new ValidationEventHandler(ValidationCallback);
            sc.Add(null, xsd);
        }

        public XSDValidator(string xsd, string ns)
        {
            if((xsd==null)||(xsd==""))return;
            if(ns==null)ns="";
            sc.ValidationEventHandler
                += new ValidationEventHandler(ValidationCallback);
            sc.Add( ns, xsd);
        }

        public bool IsXmlValid(string xmlFile)
        {
            XmlValidatingReader vr
                = new XmlValidatingReader(new XmlTextReader(xmlFile));
            vr.Schemas.Add(sc);
            vr.ValidationType = ValidationType.Schema;
            this.valid = true;
            this.warn = false;
            vr.ValidationEventHandler
                += new ValidationEventHandler(ValidationCallback);
            while(vr.Read()&&this.valid);
            return this.valid;
        }

    }//XSDValidator
}//ns
```

Bibliography

[1] G. Bernot, M.-C. Gaudel, and B. Marre. Software Testing Based on Formal Specifications: a Theory and a Tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[2] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes Second Edition. *W3C Proposed Edited Recommendation*, 18 March 2004. http://www.w3.org/TR/2004/PER-xmlschema-2-20040318.

[3] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes. *W3C Recommendation*, 2 May 2001. http://www.w3.org/TR/xmlschema-2.

[4] J. Bishop and N. Horspool. *C# Concisely*. Pearson Addison-Wesley, 2004.

[5] M. R. Blackburn and R. D. Busser. T-VEC: A Tool for Developing Critical Systems. In *Eleventh Annual Conference on Computer Assurance*. National Institute of Standards and Technology, 1996.

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). *W3C Recommendation*, 04 February 2004. http://www.w3.org/TR/2004/REC-xml-20040204.

[7] A. Brown, M. Fuchs, J. Robie, and P. Wadler. XML Schema: Formal Description. *W3C Working Draft*, 25 September 2001. http://www.w3.org/TR/2001/WD-xmlschema-formal-20010925.

[8] C. Campbell, A. Malhotra, and P. Walmsley. Requirements for XML Schema 1.1. *W3C Working Draft*, 21 January 2003. http://www.w3.org/TR/2003/WD-xmlschema-11-req-20030121.

[9] P. Dauchy, M.-C. Gaudel, and B. Marre. Using Algebraic Specifications in Software Testing: a Case Study on the Software of an Automatic Subway. *Journal of Systems and Software*, 21(3):229–244, 1993.

[10] R.-K. Doong and P. G. Frankl. The ASTOOT Approach to Testing Object-Oriented Programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, Apr. 1994.

[11] D. C. Fallside. XML Schema Part 0: Primer. *W3C Recommendation*, 2 May 2001. http://www.w3.org/TR/xmlschema-0.

[12] D. C. Fallside and P. Walmsley. XML Schema Part 0: Primer Second Edition. *W3C Proposed Edited Recommendation*, 18 March 2004. http://www.w3.org/TR/2004/PER-xmlschema-0-20040318.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] A. Gargantini and E. Riccobene. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence. *Journal of Universal Computer Science*, 7(11):1050–1067, Nov. 2001.

[15] J. Harm and R. Lämmel. Two-dimensional Approximation Coverage. *Informatica*, 24(3), 2000.

[16] M. Ishikawa. XHTML™ 1.0 in XML Schema. *W3C Note*, 2 September 2002. http://www.w3.org/TR/xhtml1-schema.

[17] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *Proceedings of the 1994 international symposium on Software testing and analysis*, pages 95–107. ACM Press, 1994.

[18] P. Klint, R. Lämmel, and C. Verhoef. Towards an Engineering Discipline for Grammarware. Draft, Submitted for journal publication; 32 pages, Aug.17 2003.

[19] R. Lämmel. Grammar Adaptation. In *Proceedings of Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.

[20] R. Lämmel. Grammar Testing. In H. Hussmann, editor, *Proceedings of Fundamental Approaches to Software Engineering (FASE) 2001*, volume 2029 of *LNCS*, pages 201–216. Springer-Verlag, 2001.

[21] R. Lämmel and J. Harm. Test Case Characterisation by Regular Path Expressions. In E. Brinksma and J. Tretmans, editors, *Proceedings of Formal Approaches to Testing of Software (FATES'01)*, Notes Series NS-01-4, pages 109–124. BRICS, Aug. 2001.

[22] R. Lämmel and W. Schulte. Controlled Explosion in Grammar-based Testing. Microsoft Research Redmond, internal document, 20 pages, 24 Oct. 2003.

[23] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[24] R. Lämmel and G. Wachsmuth. Transformation of SDF Syntax Definitions in the ASF+SDF Meta-Environment. In M. van den Brand and D. Parigot, editors, *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA'01), Genova, Italy, April 7, 2001, Satellite event of ETAPS'2001*, volume 44 of *ENTCS*. Elsevier Science, Apr. 2001.

[25] A. Malhotra and M. Maloney. XML Schema Requirements. *W3C Note*, 15 February 1999. http://www.w3.org/TR/NOTE-xml-schema-req.

[26] W. McKeeman. Differential Testing for Software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.

[27] B. McLaughlin. *Java and XML Data Binding*. O'Reilly & Associates, May 2002.

[28] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage Criteria for GUI Testing. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 256–267. ACM Press, 2001.

[29] Microsoft. Microsoft .NET Information. http://www.microsoft.com/net, 2004.

[30] P. Naur. Revised Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960.

[31] J. Siméon and P. Wadler. The essence of XML. In *Proceedings of the 30th symposium on Principles of Programming Languages (POPL'03)*, Annual Symposium on Principles of Programming Languages, pages 1–13. ACM Press, 2003.

[32] E. G. Sirer and B. N. Bershad. Using Production Grammars in Software Testing. In *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL '99), October 3–5, 1999, Austin, Texas, USA*, pages 1–13, Berkeley, CA, USA, 1999. USENIX.

[33] D. Slutz. Massive Stochastic Testing for SQL. Technical Report MSR-TR-98-21, Microsoft Research, Redmond, 1998. A shorter form of the paper appeared in the Proceedings of the 24th VLDB Conference, New York, USA, 1998.

[34] D. Sosnoski. XML and Java technologies: Data binding, Part 1: Code generation approaches—JAXB and more. In *developerWorks — XML or Java Technology*. IBM, 2003.

[35] D. Sosnoski. XML and Java technologies: Data binding Part 3: JiBX architecture. In *developerWorks — XML or Java Technology*. IBM, 2003.

[36] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures Second Edition. *W3C Proposed Edited Recommendation*, 18 March 2004. http://www.w3.org/TR/2004/PER-xmlschema-1-20040318.

[37] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures. *W3C Recommendation*, 2 May 2001. http://www.w3.org/TR/xmlschema-1.

[38] J. Tretmans. A Formal Approach to Conformance Testing. In O. Rafiq, editor, *Protocol Test Systems, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems,*, volume C-19 of *IFIP Transactions*, pages 257–276. North-Holland, 1994.

[39] W3C. Extensible Markup Language (XML) Conformance Test Suites. *W3C*, 10 December 2003. http://www.w3.org/XML/Test.

[40] W3C. XML Schema schema for XML Schemas: Part 1: Structures. *W3C*, 13 February 2001. http://www.w3.org/2001/XMLSchema.

[41] D. Watkins, M. Hammond, and B. Abrams. *Programming in the .NET Environment*. Addison-Wesley, 2002.