Static Guarantees for Algorithms on Constrained XML Documents

Patrick Michel

December 11, 2006

Contents

Listings

List of Figures

1. Introduction

XML is an interesting and widespread technology. Not only the language itself but a host of domain-specific languages tailored towards XML make up the success of it and are referred to when talking about XML in general.

At first XML was used as common interchange format, as a generic way for data encoding and transportation. As in the compiler technologies, where an intermediate language greatly simplifies the translation of many source languages to many different target platforms, XML simplified the transportation of data from many different kinds of sources to many different targets.

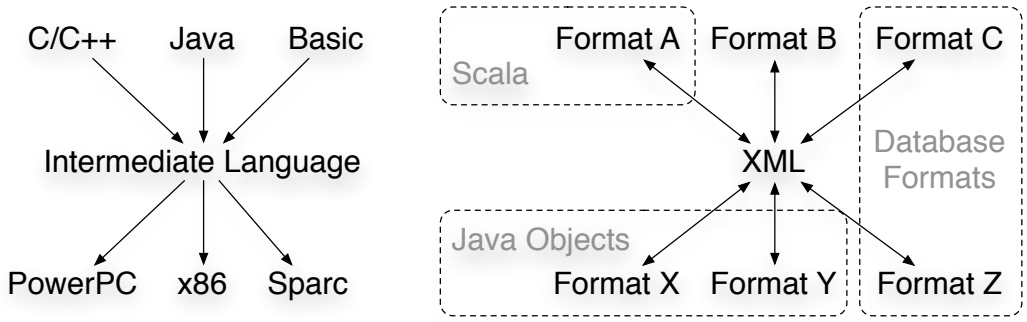

Figure 1.1.: XML as Intermediate Language

But in contrast to a program written in a programming language, which is translated once to a binary for a target platform, XML data keeps being processed, passed around and modified. Each program involved has to be able to read from XML sources and output to XML, but the data itself is most often explicitly translated to an internal representation.

XML does no longer only specify a generic way to interface the data of different languages, but it describes a way to formulate and look on data. Data in the form of XML is a first class object in the system of very different programs working with it. Modern languages will need a way to process XML data directly, new languages doing this are already being researched, while better support and integration is added to the old ones.

With many different languages involved, however, the task to keep the data meaningful and well formed is distributed to all of them. Depending on the application domain, the data used has to comply to various constraints, regarding the structure as well as consistency and integrity.

We have to be able to express such constraints in a generic way and supply developers of the various languages with them, such that each program dealing with the data

is aware of the constraints and does not violate them. If not each program should check all constraints whenever it gets data, we have to ensure that every manipulating program does not invalidate any constraints. If we can statically guarantee, either by proof or general arguments, that the algorithms used in programs preserve the constraints, we can simply combine programs in a safe and efficient way.

Again we will need to find a generic way to formulate algorithms manipulating XML data, which is simple enough to allow us to actually prove things, yet powerful enough to express common operations. At the moment there are few XML technologies which are actually concerned about local manipulations of data. The main focus is still on information retrieval and the transformation of documents.

Once both constraints and algorithms are specified, we can bring both together and prove the algorithms correct. It will be possible to provide generic arguments, which allow to disregard some constraints for certain algorithms altogether. Nevertheless we will have to prove some constraints correct for each algorithm, as there will always be those constraints, which are directly connected to the task the algorithm solves.

This work therefore splits into three major parts, which are all guided by a case study, featuring a student management system for the Bachelor's course of studies of the University of Kaiserslautern. The first part will evaluate present technologies to formulate constraints on XML documents and fix one or more to use for the case study. In the second part, technologies are evaluated to formulate algorithms for local manipulations on XML data. We will then bring both together in the third part and develop arguments to prove the algorithms of the case study correct with regard to the constraints defined.

The example system consists of relevant student data, which is needed for the students to succeed with their studies, as well as the module handbook, which contains all lectures, seminars, etc. which are unified under the term "module" and thereby provide the means the students need to progress.

There are many different parties involved in such a system, which all deal with either the students or the handbook, both retrieving information and changing it. Modules have to be created and maintained, student data is entered into the system by the examination office and other parties. Each of them is at first glance independent, but again XML is not only used to pass data between them, but it is a major aspect of the system. The data has to stay valid and has to comply to the specified constraints, so the different parties can work seamlessly together.

In addition to the data and the constraints proposed for it, the example system will feature the role of the examination office, which contains enough algorithms to guide students from registration to success or failure.

The following subsections give a short introduction to XML and the relevant concepts of it used in this work.

1.1. The Technology XML

XML started off as a language focused on the representation of data and its interchange [34], concerned with many crucial aspects like character encoding, white spaces and modularity, which directly lead to the success of the language in distributed, amorphous and evolving systems. It is now common to represent and store data in XML and to get data in XML over networks or shared file systems.

James Clark[1] describes the complexity of XML as follows:

> Back in February 1998, when the XML 1.0 Recommendation was first introduced, XML was radical in its simplicity compared to SGML. The innovation in XML was not so much in what it added to SGML but rather in what it took away. However, XML is now part of a much larger family of standards from the W3C. Collectively, these are much more complex than SGML ever was. It is hard for a newcomer to understand what is the right way to use XML and what are the core ideas.

The basic intention of the technology XML is a success, and attention can turn to other concerns and higher abstraction layers. The data and the way it is structured is now of interest. As it becomes common to work with data in the XML format it has to be conveniently accessed, manipulated and transformed from within common languages, which is quite hard if the structure of a document is statically unknown. The early technologies lack convenience as well as static guarantees for this exact reason and languages constraining XML data had to be reviewed and evolved.

Each technology working on domain-specific concrete data in XML will need to know how the data is structured and which rules apply to it. Only then information can safely be retrieved, stored or manipulated, in the sense that read-operations are secured to have a meaningful and valid result and that write-operations leave the structure and integrity of the data intact.

Common XML applications implicitly follow structural contracts and generic rules, but either do not express them explicitly or only check them explicitly at runtime in an isolated technology but do not connect this knowledge with other technologies. The benefits of connecting the knowledge from different technologies are static guarantees regarding algorithms, rendering runtime checks unnecessary and additionally testifying that certain erroneous situations do not arise at all.

1.2. The Right Abstraction

I assume the reader is familiar with some ideas of XML and has already seen one or two XML documents. The XML specification is not really helpful when trying to get an idea of what XML is all about. All concerns taken care of in there are no longer of common interest, which is a very good sign for the success of a specification. The W3C itself therefore offers another document about XML, in which it describes the XML information set [35].

[1] 2nd paragraph of the “Foreword by James Clark” in [30].

When talking about XML, we don't want to talk about how well it handles character encoding, but how well it is suited to represent and store data. The XML information set gives an abstract introduction to what the content of an XML document is.

In this work I want to follow an even more abstract view, best described by James Clark[2]:

> I would argue that the right abstraction is a very simple one. The abstraction is a labelled tree of elements. Each element has an ordered list of children in which each child is a Unicode string or an element. An element is labelled with a two-part name consisting of a URI and local part. Each element also has an unordered collection of attributes in which each attribute has a two-part name, distinct from the name of the other attributes in the collection, and a value, which is a Unicode string. That is the complete abstraction. The core ideas of XML are this abstraction, the syntax of XML, and how the abstraction and syntax correspond. If you understand this, then you understand XML.

The only thing I want to add to this view is, that it is sometimes useful to see each element as an entity, like in the database domain, and the parent-child relationship, as well as the sibling relationship, as especially important connections between the involved entities. To become aware of this fact allows to design more suitable data schemas, whenever this is the intended use of XML technology.

Throughout this work I will therefore stick to the terms *element*, *attribute* and *document* to refer to parts of XML, like defined in [35], neglecting most of the other information items.

1.3. Elements vs. Attributes

Attributes do not really add anything to XML, if at all they are just easier to write than elements. The information contained in an attribute can always be represented in an equally named child element. This, of course, also works the other way round: Each single child element, not having children, can be turned into an attribute of its parent.

To distinguish those two is often a cause of trouble only and when designing XML documents, it's non-trivial to decide which one to use for a certain information. Many XML technologies honor this insight, by making it possible to treat attributes and elements in the same way.

Still there are good reasons to use attributes at all. While attributes might not add much to the technical aspects of XML, they are conceptually quite different to elements. Attributes are much closer related to their parents as elements are and they are unlikely to be regarded as independent entity. This becomes even more true, when regarding a document as a collection of entities, tied together by many relations, in which case attributes are no entities by their own, but are a part of one.

An attribute will never have children and in many cases will be an important property, which can't be omitted, of an element. In contrast to that, a child element might

[2] 5th paragraph of the "Foreword by James Clark" in [30].

be refined in the future, to contain elements of it's own and can often be either omitted or added to the parent more than once.

Throughout this work, *elements* and *attributes* will rightfully describe two different concepts, but the reader should be aware that talking about "attributes" of an element is therefore meant conceptually and might as well resolve to a child element of another element in a concrete representation of the discussed data, especially in situations where this representation is still unknown.

2. Constraining XML

This chapter strives to formulate the constraints of the example system in an adequate language. To achieve this goal I will first describe the example system in more detail and derive both different kinds of constraints for the system and requirements for the technologies which could be used. After giving an overview of the candidates I will then fix one or more technologies and show how they are used for the system. The result of the chapter will be constraint specifications containing the logic of the application domain.

2.1. Application Domain

The application domain of the example system is the Bachelor's degree in computer sciences of the University of Kaiserslautern. It is regulated in three major documents, which are maintained by different parties, influencing the likelihood of changes in the future.

The document containing the *examination regulations* is the most static one and describes the procedure of obtaining the Bachelor's degree, by choosing and passing exams with regard to all restrictions. It regulates the structure of the study, the time constraints to fulfill, how and when examinations are done, how often they can be done, etc.

The *study plan* is maintained by the department of computer sciences and defines the more concrete parameters of the Bachelor's study, like the mandatory modules of blocks or the currently available areas a student can take as focus. The plan can be changed by the department at any time.

Last but not least, the *module handbook* gives an account of all available modules. It contains, for example, the mandatory modules, the specific modules of an area, which are repeated irregularly only, and the modules of other departments of the University, which are needed by students in the area they focus on. The handbook is a logical part of the study plan, but kept separate from the latter, as it is likely to change more often.

2.1.1. Data

We will now have a look at the data of the example system, which is modelled in XML. Figure 2.1 gives an abstract overview of the relevant data and already suggests some "belongs to" relationships by including parts in others.

The concrete structure of the data strongly depends on the involved technologies working with it and is a important design decision. The figure at hand already incorporates several decision made in the final system but should nevertheless give an introduction to the example system.

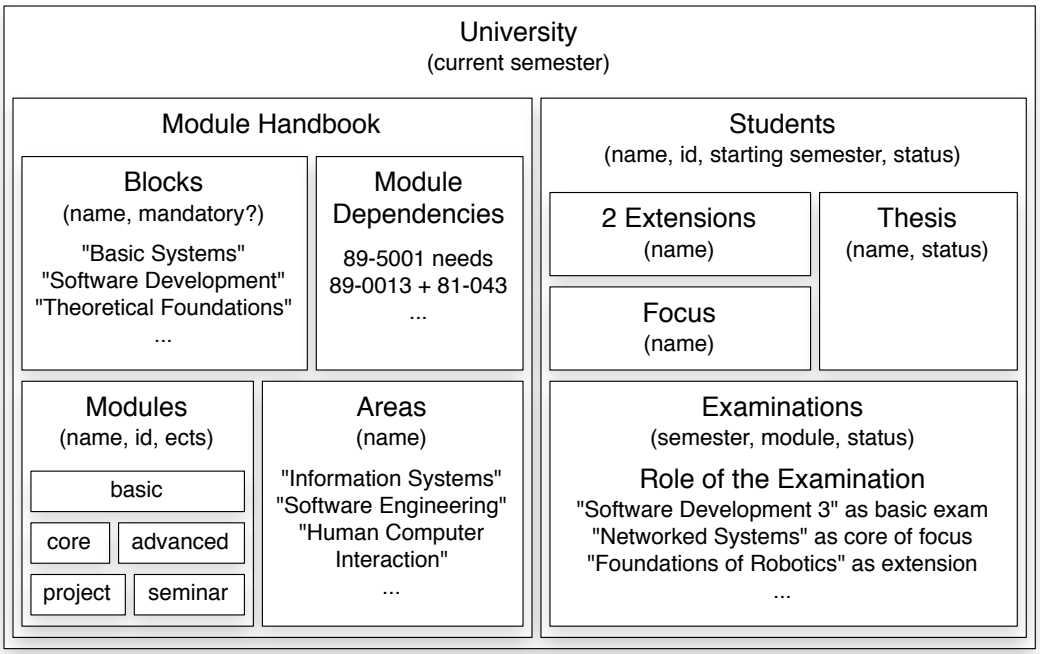

Figure 2.1.: An Overview of the Data in the Example System

The modules are described in the module handbook and are essential for the Bachelor's regulations and represent a major part of the pure data defined in the documents taken into account. While most parts of the documents are concerned with rules and constraints, the study plan also contains two recommended abstract schedules.

Each student should have such a schedule in mind when planning his studies and it has to comply to the module handbook and the Bachelor's regulations. Together with the module data they pose an interesting data themselves, which is subject to a number of constraints. Students, however, don't need to fix a schedule in advance, but just start doing examinations. When they are done with their studies, the entirety of the examinations resembles a study plan, as each examination represents a module for a given semester.

Students enroll for examinations but can also withdraw from them and examinations have to be done while certain semesters are running and need to be in the correct order. To express such constraints the XML document has to contain the current time and I judge those categories of constraints as too fine grain and complex for this example.

I therefore decided to operate on the much larger timescale of semesters and disregard enrollment and withdrawal of examinations nearly completely. The example system will therefore contain parts of the module handbook as more static data, together with students of the Bachelor's course of studies. Students will start with an empty schedule and build up a valid one by adding and passing examinations over time.

The following sections analyze different aspects of the application domain to delimit specifics of a valid system and derive requirements of XML technology used to describe

it. There need not be a single technology satisfying all requirements or a technology for each requirement at all, but it will be possible to support an interesting part of the example system.

2.1.2. Structure

As mentioned in the previous section and also seen in Figure 2.1, the relations between different artifacts are not fixed and it is a design decision how the layout of the data is chosen. We will try to find as many constraints as possible which help delimiting the structure, looking at the three regulating documents.

S1 **Section 3 of the study plan** tells us that there are different kinds of modules (basic, core, advanced, seminar, project and minor) and that modules have a fixed number of ECTS, which are each equivalent to about 30 hours of work.

S2 **Section 5** mentions different blocks, to which modules can belong. The blocks "Software Development", "Basic Systems", "Theoretical Foundations" and "General Foundations" consist of basic modules, which have to be done by all students. There is also a focus block as well as an extension block, in which modules can be chosen. The modules a student chooses for those two blocks are not tied to a specific block, but to an area of computer sciences.

S3 **Section 7** mentions all currently known areas, to which non-basic modules can belong.

S4 The **module handbook** reveals that there are special dependencies between modules, some modules can be selected by a student only when others are already selected.

S5 In rare cases these dependencies can be more complicated, such that some modules can be selected only when enough ECTS of another group of modules are already selected.

S6 The **module handbook** explicitly states, that a module either belongs to a basic block or an area of computer sciences. There is a third category mentioned but it is only relevant for modules of the master program.

S7 A module is not always offered every semester, but can be offered less frequent or even without fixed schedule.

A module will therefore need a name, an ECTS value, a type, an associated block or area and optional dependencies to other modules. The concrete structure of an XML document describing modules is still not fixed by these observations, as for example the relation between modules, blocks and areas is unclear (see Figure 2.1). Constraint S6, however, suggests to put modules within blocks and areas, while those remain siblings on the same level.

In the end such a structure is best chosen dependent on the used technologies to express constraints or manipulations, to augment to the conciseness of such specifications.

To fix a given design we need a language capable of constraining the content of an XML document, which can define the following properties:

- The names of elements which are anticipated in a document.
- The position in which elements can appear and how they interact with other elements.
- The root element.
- The attributes of an element.
- The allowed values for an attribute.
- A way to express the relation between certain elements and a set of data. We need to define all possible blocks and areas in the example and express the relation of modules to them.
- A way to represent strongly related elements, which are yet not completely equal. We need to express types of modules in the example.

In contrast to modules, students and their schedules are not directly described by a document, but they have to comply to various constraints defined in those. As far as the structure of a schedule or a set of examinations is concerned, we have the following information:

S8 **Appendix 2 of the study plan** shows schedules as tables, denoting the current semester on one axis and the structural blocks of the Bachelor's program on the other. The table then contains modules planned to be done in a semester for a specific block. In the example those entries in the table will reflect passed examinations of the student.

S9 So a schedule will contain examinations with references to modules, which contain the additional information in which semester they were done and in some way for what *reason*.

S10 The document needs to contain a global time value, to which examinations refer.

S11 **§8(1) of the examination regulations** defines a regulation of progress for each student, so it will be necessary to relate a student to the global time value, to get an idea about how long the student is already studying.

S12 Students will of course have a name and an unique identifier.

Students are therefore much more variable with regard to their structure, though it again has to be fixed, to be able to manipulate and constrain schedules automatically. The choice of the structure has an impact on both tasks and depends on the technologies chosen for them.

There will be more structural constraints, which arise in the context of domain-specific constraints or simply represent design decisions, which also have to be enforced once they are fixed[1].

[1] See Section 2.3.1 for structural design decisions made after technologies are fixed.

2.1.3. Integrity

There are a lot of constraints to the content of an XML document, which are not purely structural, but closely related to it. They constrain the structure of the document on basis of two or more concrete values of the document, so they are different from, for example, general rules on the combination of elements.

Such constraints ensure the integrity of a document besides the basic structure, but directly depend on the design of it. They are therefore not directly found in the Bachelor's documents, if at all they are implicitly given by the regulations or the study plan.

I1 Module names have to be unique in a specific scope. From the module handbook it is obvious, that module names need not be unique in global scope, as there are several examples of different modules with the same name, but different types. A good approach seems to be that for each type all module names have to be unique. Additionally, to simplify referencing modules, each module will get a unique identifier.

I2 Some module types will never appear together with certain block references or areas in which they are offered. Basic modules, for example, are never associated with an area, whereas core modules never belong to a basic block. Depending on the structure of the module data this might already be ensured. It depends on the realization to which category of constraints this property belongs, so it might also be counted to purely structural constraints.

I3 Modules can have optional dependencies to other modules. As such a dependency is at least associated with two modules, it is inevitable that the structure of an XML document representing modules contains references from one element to another. The target of such a reference has to exist and has to be unique.

In the case of schedules, the observation made in I3, together with its consequences, are even more obvious and represent an essential part of the validity of the presented data: A planned module of course has to exist in the module handbook and it also has to be unique. A schedule is defined only on top of a module handbook and its data is a hopefully valid combination of references into the handbook.

I4 Examinations contain a "time" value, when they were done, which references the global time, so this is only valid when the global time is always greater or equal than all examination times. The global time should always be a positive value, to avoid confusion.
This is not a domain-specific constraint, as it arises from a design decision in the structural model, i.e. it is an already known example for I6.

I5 Students and modules have unique identifiers, but the structure might not be able to express that, but only the fact that they must have an identifier. The uniqueness has then to be explicitly enforced.

I6 Depending on the structural model, some integrity constraints may arise. Whenever a decision is made how to represent data, that representation might have

implicit constraints on its usage.
The reader should note, that this is a fairly big source of constraints, as most design decisions will yield such a constraint. As we will see in the example system, the constraint splits into 12 different ones.

A valid XML technology to use for the example system should therefore also be able to express that:

- The set of attribute values of a specified set of elements does not contain duplicates (uniqueness).
- The attribute value of an element has to match one of those of a set of existing elements (foreign keys).
- Some elements can occur only in a "valid" combination (relational correctness). This is a far more general category than the former two and is likely to need a much more complex technology than used for those.

2.1.4. Domain-specific Validity

The last group of constraints is implied by the application domain itself, representing its logic. Depending on the chosen structure of the XML data, some constraints may be implicitly taken care of. This class of constraints needs the most powerful class of features for their definition, as they can be arbitrarily complex, depending on the application domain.

There are some constraints concerning modules:

D1 **§3(1) of the examination regulations** mentions that blocks always have at least one module and that a module has between two and 14 ECTS. These two constraints are a good example of constraints, which can but need not be taken care of by structural constraints.

D2 **§4(2)** defines that the sum of ECTS of all modules of a basic block has to be within specific bounds, which depend on the name of the basic block. For the other two blocks it defines what kind of modules have to be done and how many ECTS each type needs to have in total.

D3 **§4(4)** defines all possible names of areas, modules can belong to. This list, however, can be changed by the department of computer science.

D4 **§4(5)** restricts that all modules chosen to be in the focus block have to belong to the same area. This is, of course, a restriction for schedules, but a valid module handbook should therefore contain enough modules in each area to allow this area as a focus at all.

D5 **§4(6)** demands that a student chooses modules of three different areas, so there should be at least three areas to chose from in any valid module handbook.

To formulate such constraints a XML technology needs to be able to allow the following constructs:

- The existence and number of child elements can be constrained.
- Depending on the structure of the data, it might be necessary to constrain the existence and number of child elements with certain properties.
- Such properties have to be expressed and might be more complex than inherently given facts of the XML structure (i.e. more complex than, for example, sibling or child relations)
- Constraints must be able to include attribute values.
- The sum of a set of attribute values is fixed or within a certain range. In general it has to be possible to build expressions on attribute values of arbitrary types and to build boolean expressions with them.
- Arbitrary sets of nodes, besides those implied by the structure, are needed to express all constraints (i.e. more complex than, for example, the set of children).

Most constraints for students and their schedules are given within the examination regulations, as they define which module combinations are valid and what dependencies on modules exist. The study plan and module handbook, however, add some constraints too:

D6 **Section 4 of the study plan** mentions six semesters to be usual and that a student needs 180 ECTS to get the Bachelor. This includes the Bachelor's thesis.

D7 **Section 5** defines that all modules of mandatory blocks have to be present in a schedule.

D8 **§4(2) of the examination regulations** fixes the total number of ECTS in each block.

D9 **§4(5)** defines that all chosen modules of the focus block must belong to the same area.

D10 **§4(6)** defines that each eight ECTS of core modules need to belong to a different area. The student should choose three different areas in which he chooses core modules and then select one area as focus.

D11 **§6(4)** forbids to choose any advanced modules in the focus block, as long as the core modules of the block are not done. Additionally every examination a student enrolled for has to be passed.

D12 **§8(1)** restricts the possible number of different semesters, as it defines how many ECTS a student needs to have after a certain number of semesters. Some of these constraints are formulated even more strict, as the student needs at least a number of ECTS from basic modules only.

D13 **§9(6)** defines that the student needs all mandatory modules as well as at least 120 ECTS, to get a topic for his thesis.

D14 The **module handbook** defines even more dependencies between modules, in rare cases with complex summation criteria. They should be fulfillable and have to be fulfilled by students.

From these constraints we get additional requirements for a XML constraining language:

- All references contained in a set of elements have to reference elements sharing a common property. Which property depends on the actual structure of the data, so this might be, for example, either the element type or some common value of one of the attributes of the element.
- Properties of a set of referenced elements have to differ (uniqueness of referenced elements).
- Some elements are not allowed certain attribute values or they are not allowed in certain positions in the structure, depending on the attribute values or position of other elements. This is necessary to express the dependencies between modules, but the concrete kind of constraint is again dependent on the data model.

An application domain like the Bachelor's course of studies is guaranteed to have inherent constraints not explicitly stated in the regulations at hand, which represent common sense or constraints so common to the domain they are formulated in a much more general document. Those constraints might include, but are not restricted to:

D15 A student cannot have two passed examinations to the same module, i.e. examinations have to be unique.

D16 There should be at least one schedule, which is valid and leads to the Bachelor's degree, i.e. all constraints are fulfillable with the given data of the module handbook. This is a general statement about all other constraints and it is unlikely that any constraint language used for them can formulate it. There are, however, many simpler ones, that are part of it and ensure some facets of it.
Constraints like D4 or D5 already contain such facets, which were easily derivable from parts of the examined documents.

D17 There should be no "useless" modules, e.g. core modules with an ECTS value which can't be combined with others to both reach at least 8 ECTS and not have done the module in vain, as the sum of the rest also totals 8 ECTS or more.

D18 Areas should offer combinations of core modules, which are exactly eight ECTS, so they can be chosen as focus or extension, without doing "too much". Focus areas should additionally offer at least one combination of advanced modules with a value of eight.

D19 Modules of the type "project" or the type "seminar" are not enrolled for by the students, but they rather get a certificate for successful participation. Those modules thereby stay outside of D11, i.e. a student can attempt to do them and then choose to do another, without the system noticing.

Such constraints are even more demanding to formulate:

- Without further limitations, we will need quantifiers on sets of objects, to express the fact that any subset of a set of elements has to fulfill a given property.
- Though it is true, that a total sum of eight ECTS of modules, which each have at least two ECTS, can have only four elements, it would be extremely cumbersome to formulate without such quantifiers.

2.2. Choosing the XML Technologies

To actually supply data as XML and manipulate it, only the XML specification itself and a simple text editor are needed. This was also one of the design goals of the language, so it is a text-based language which is human readable, as well as tolerant against formatting and editing by humans.

The XML standard does not provide any means to access documents programmatically, as this is not necessary for its purpose and could even be considered harmful. This is but one way to explain the host of technologies which arose around XML out of necessity and the constant evolution of technologies to adjust to arising needs.

This section will introduce more categories of XML technologies and introduce some of their members. They are then reviewed with regard to the necessary qualities regarding the example system and especially the specification of constraints, as needed in this chapter.

Such a specification should be

- **readable**, such that it is adequately easy to understand that the specified constraint is equivalent to the informally given one.
- **declarative**, i.e. the focus on the formulation of a constraint is the constraint itself, not the way it is checked.
- **executable** by an interpreter or indirectly by a compiler, i.e. it is possible to check a constraint for a given document, without the need to implement it again in another language.
- **adequately complex**, i.e. the constraints should be formulated in a language sufficiently complex to support the constraints but not containing arbitrary more other constructs.
- **easy to analyze** by tools, which is heavily dependent on the complexity mentioned before, but also includes that the syntax isn't too complex and can be analyzed with standard technologies.

2.2.1. Basic APIs

Common technologies associated with XML are the APIs to get access to documents from within common programming languages. They acknowledge the tree structure

of XML and grant access to the nodes and arcs of it. They are unaware of domain-specific restrictions to documents, which results in them being powerful but generic frameworks.

Like most generic frameworks, they are a good starting point to work with XML, but are not particularly suited for any task. They are merely the connection between the XML standard definition and the programming world, granting generic access to all its features.

The description of the "Document Object Model", as defined by the W3C[2], starts as follows:

> This specification defines the Document Object Model Level 1, a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents. The Document Object Model provides a standard set of objects for representing HTML and XML documents, a standard model of how these objects can be combined, and a standard interface for accessing and manipulating them. Vendors can support the DOM as an interface to their proprietary data structures and APIs, and content authors can write to the standard DOM interfaces rather than product-specific APIs, thus increasing interoperability on the Web.

The "Simple API for XML" (SAX) [23] is a direct competitor to DOM, while following another design principal. Opposed to DOM, SAX does not grant random, active access to all parts of the document, but traverses it and uses an event mechanism to grant passive, linear access to a documents structure and content.

The design of SAX also disregards manipulations of a document, but is therefore a much more lightweight approach than DOM. As a result both technologies are valid choices for certain projects, but DOM turned out to be more general as interface specification. In fact SAX is often used as API only, while DOM is often referred to as an API, but is also the de facto standard interface specification to combine XML technologies.

For this work I judge both technologies ill suited to concisely express the creation, manipulation or transformation of XML, as well as to express constraints on documents. Especially concerning the latter, which is the focus of this section, neither DOM or SAX are in any way declarative or adequately complex and their readability is debatable at best.

2.2.2. Schema Languages

Schema languages are the first example of more specific languages in the context of XML. They concentrate on the definition of classes of valid XML documents, as needed for a certain application. At first glance such a language has to be perfectly suited for the needs of the example system.

These classes are defined differently in different schema languages, some define constraints each valid document has to fulfill, others define all allowed elements and

[2]First paragraph of the abstract in [31].

attributes, as well as their combination. Those languages differ in the kind of criteria allowed to define restrictions and thereby allow slightly different sets of classes to be defined at all.

Schema languages are for the most part declarative languages, such that they allow to specify properties of XML documents, but are not directly related to any implementation. It is, however, true that some languages are influenced by the need of efficient implementations and therefore forbid the definition of too complex classes of allowed documents.

An XML schema in general is the basis for all other technologies. Only if the structure of a document is fixed in at least some dimensions, declarations of properties or algorithms working on it can have a meaningful interpretation in an application context.

Eric van der Vlist[3], the author of a book about XML Schema [29], summarizes the need for schema languages in the following way:

> XML, the Extensible Markup Language, lets developers create their own formats for storing and sharing information. Using that freedom, developers have created documents representing an incredible range of information, and XML can ease many different information-sharing problems. A key part of this process is formal declaration and documentation of those formats, providing a foundation on which software developers can build software.

Being declarative is already a plus for schema languages and most of them come together with runtime checkers, which also makes them executable. However, they severely differ in their readability, how easy it is to analyze them and in their capability to express needed constraints at all, i.e. their complexity.

DTD The "Document Type Definitions" (DTDs) are part of the original XML language specification [34]. Though being a generic language, or even because of being one, XML itself is concerned with defining classes of valid documents for an application context.

DTDs are therefore the most basic schemas for XML documents. They are situated in the prologue of an XML document, between XML declaration and actual data elements. The syntax is specifically defined for this purpose and does not reuse XML itself.

The class of allowed documents is defined in a grammar based approach. All allowed elements are listed, with one special being the starting point, and regular expression like rules are given to describe what child elements and attributes the element can have.

The syntax lacks the simplicity of XML itself and has to be learned by the user as it is not even possible to intuitively read the declarations. Many directives are concerned with whitespace handling or the allowed kind of data or text in an element, though DTDs don't support simple data types like integer.

[3] 1st paragraph of Chapter 1 in [29].

All in all DTDs are very awkward to use and rely on a special syntax. Definable classes of documents aren't very precise and lack many important features. There is no concept of element classes, subtyping or simple shared use of common expressions.

Though DTDs are sufficient for most, but not all, requirements formulated in Section 2.1.2, they are not suited for integrity constraints or even domain-specific constraints. DTDs have a notion of unique identifiers and foreign keys, but it is too basic to be practically applicable. It only supports globally unique names and references to them.

As there are many technologies striving to replace and enhance DTDs, I decided they are not suited for the example system.

XML Schema XML Schema is the most widespread substitute for DTDs. It enhances their capabilities and is defined in XML itself, so it is not needed to learn a special syntax, but only to know the semantics of each element. XML Schema was influenced by various other proposed schema languages and has adopted parts of most of them.

It is a recommendation of the W3C, split into the structural definition [36] and the data type definition [37]. The introduction of data types is one of the important contributions of XML Schema, as only with data types it is possible to define meaningful algorithms and expressions on XML documents.

Because of the various contributions, XML Schema is a complex and powerful language. It is possible to reuse definitions and XML Schema supports several forms of inheritance of elements, but offers no subtyping alongside it. XML Schema meets most of the requirements formulated in Section 2.1.2, but surprisingly enough one of its shortcomings is the inability to define a designated root element. Every document matching one of its productions is valid in the sense of XML Schema.

XML Schema supports powerful uniqueness and foreign key constraints, so it is, for example, possible to define uniqueness based on multiple attributes, without restriction. But XML is still a grammar based approach and besides many complex reuse and inheritance concepts relies on the definition of complex productions to define a class of allowed documents.

To be able to formulate foreign key constraints in such a setting, XML Schema uses embedded XPath expressions, to specify the target of a foreign key and the targets of a uniqueness constraint. XML Schema does not allow to formulate logical expressions and constraints cannot be based on attribute values at all.

The combination of the various production definition elements is quite complicated and some restrictions apply, which are motivated by the runtime effort to validate. Whenever there is a choice of different child elements, the validity checker has to be able to decide in linear time which option was chosen in a document.

The overall complexity of XML Schema originates in such shortcomings and the somewhat artificial and arbitrary definition of the language. Even simple element definitions have to be expressed by nesting various elements, leading to fast growing, complex specifications, which are no more easy to read or understand cause of many different techniques like inheritance or substitution groups.

XML Schema therefore is a vast improvement in potential and syntax, but too com-

plex for the tasks that can be accomplished by it. The usage of XPath for uniqueness and foreign keys only is necessary but awkward and does not utilize the potential of XPath in a desired way.

XML Schema would be powerful enough to meet most of the structural and integrity requirements formulated in Sections 2.1.2 and 2.1.3, but also lacks some parts of both. As there are better ways to express the constraints in both categories and its overall complexity, I decided against the use of XML Schema.

RELAX NG A short introduction to the history of RELAX NG is best given by one of the authors[4] itself:

> Although I was a member of the XML Schema Working Group, I also felt that we need a simpler alternative to W3C XML Schema. I thus designed RELAX Core, which was a simple schema language based on hedge automata, an aspect of tree automation theory. James Clark then designed TREX, which embodied many improvements [...]. To provide a powerful alternative to W3C XML Schema, RELAX Core and TREX were unified into RELAX NG at OASIS in 2001. Recently ISO/IEC JTC1 has published RELAX NG as a Final Draft International Standard without making any technical changes.

RELAX NG is also inspired by the Xduce type system, which is based on tree automata. This underlying theory virtually guarantees a sound and consistent feature set, but is more ignorant to runtime efficiency compared to XML Schema.

The benefit for the user is an easy to understand, yet powerful schema language, which is convenient to use and perfectly solves the tasks it decided to solve, while leaving to complicated features out right from the start. The syntax is again XML itself, with all the advantages and disadvantages this offers, but there is also an abbreviated special syntax one can use.

With regard to structural constraints, RELAX NG is more powerful than XML Schema, i.e. many interesting features which XML Schema forbids are offered and only some minor features are left out, which are unsupported by the underlying theory, but turn out to be conceptually unsound in most cases anyways.

RELAX NG does not offer any capabilities regarding integrity constraints or domain-specific validity, but meets all requirements formulated in 2.1.2. It also allows to share common parts in definitions and merge different schemas.

Grammar-based Approaches By now it becomes apparent that grammar based schema languages are inherently good at structural constraints, with only minor differences, but are unusable for integrity constraints or domain-specific validity, without leaving the core idea. Only by adding additional concepts some languages where capable to handle at least integrity constraints to a certain extend.

Though grammar based approaches are well understood and controllable, we have to look at more powerful concepts to be able to express integrity constraints conveniently and to express domain-specific validity at all.

[4] Parts of the 2nd paragraph of the "Foreword by Murata Makoto" in [30].

The realization that there are many different tasks to be solved by schema languages is well documented in a paper by Rick Jelliffe [18]. The three categories of constraints used in this work match the layers used in the paper in the following way: Structural constraints are expressed using the "Regular Structures" capability of a language, integrity constraints are called "Local Reference Integrity" and the "Co-Occurrence Constraints" are essentially what I call domain-specific validity.

The paper backs up all revelations stated so far about DTDs, XML Schema and RELAX NG and also shows that there are few languages satisfying all needed aspects, even only regarding those three needed for this work.

Jelliffe himself introduced Schematron as new schema language, which is depicted as language capable of handling all three categories of constraints and being especially strong in co-occurrence constraints, a category where most others are lacking.

Schematron and XPath Schematron is a completely different approach to constraining XML documents, which is best described as being a rule-based approach. A schema contains arbitrary many rules containing assertions which have to hold for all valid documents in the specified context of the rule.

To specify contexts of rules and conditions for assertions, Schematron excessively uses the XPath language [32], which is a read-only navigation and expression language for semi-structured data like XML. So the context of each rule is simply one XPath expression and the condition of an assertion is given as XPath expression which has to evaluate to a boolean value, which is in general possible for all valid XPath expressions.

A Schematron file itself is then given as XML document, where the XPath expressions are embedded in text attributes. The language itself has only few elements concerning the definition of rules and the structuring of rules in patterns and so called phases, giving more possibilities and freedom when to check certain rules at all or in which order.

XPath as domain-specific language, solving two specific problems only, is not really usable for complete projects or tasks, but relies on being embedded in a host language. XSLT [33], for example, is a transformation language for documents, focusing on this functionality, while leaving the details to the embedded XPath.

The reference implementation of Schematron is implemented as XSLT meta-stylesheet, i.e. this sheet is used to transform a schema to another XSLT sheet, which is then used to validate documents, by transforming them. The backend of the reference implementation, i.e. the output of the generated stylesheet, is customizable, so the user can generate error messages for failed assertions or create report documents, or simply transform a valid document to the empty document, thereby showing the absence of errors.

A XPath expression can locate every element and attribute in a document and it is thereby possible to access attribute values and express conditions on them. This trivially allows all the requirements formulated in 2.1.3 and furthermore allows to formulate many of the domain-specific constraints by meeting many of the requirements formulated in 2.1.4.

Schematron is thereby the first schema language to express most of the constraints, but this capability essentially comes from the embedded XPath language, to which

Schematron only adds few essential aspects. Any schema language embedding XPath will most likely be capable of expressing the same categories of constraints Schematron can express.

XML Schema already uses XPath expressions to formulate integrity constraints, as this is difficult with a pure grammar based approach. It does, however, not allow to use XPath more freely, so though XML Schema uses XPath, it is not capable of formulating more complex integrity constraints and domain-specific constraints.

The exclusive use of XPath in Schematron to formulate constraints leads to awkward and cumbersome definitions of structural constraints. Each element needs to be the context of a rule, in which child elements and attributes are tested for existence or other properties. It becomes apparent very quickly, that XPath is too powerful for such a task and that good grammar-based languages do a far better job constraining the structure of documents. The constructive nature of grammar-based approaches alone is a huge advantage when defining structure, whereas declarative rule-based approaches are more suited for higher level constraints.

DSD, Examplotron, etc. Document Structure Descriptions (DSDs) [26] and Examplotron [28] schemas are only two of much more available schema languages with comparable capabilities to what we looked at so far. They differ slightly in their syntax or the definable class of valid documents, but not enough to be worth considering in this work.

Examplotron is also one of the few schema languages, which do not follow the grammar based approach, but allow specification by prototypes or examples. Though being an interesting idea, it is no more capable of expressing needed constraints than the already analyzed languages. The main advantage of Examplotron is the ease of specification, especially for inexperienced users.

Interestingly enough, Examplotron also allows the specification of assertions within the example documents structure, using XPath expressions. In this case, the context of an assertion need not be specified by an XPath expression itself, but the condition needs the power of XPath. Examplotron is thereby the first approach combining two concepts to one capable of handling most of the desired constraints.

It is, however, debatable if such a combination is necessary or even useful, as the proposed combination of Examplotron is quite basic. Interleaving structural constraints with more complex ones does not necessarily increase the readability of either and could even be considered harmful. Technically a separation of both does not take away any capabilities and results in a freedom of choice for both parts.

In fact the proof of concept implementation of Examplotron is implemented as XSLT sheet, that translates schemas to RELAX NG specifications. The embedded assertions are not really standardized and marked as implementation dependent, but a translation to Schematron is suggested.

DSDL The Document Schema Definition Languages (DSDL) ISO/IEC project acknowledges the fact that single schema languages are unable to satisfy all arising needs and gives four major categories of concern: Structure, Data types, Integrity Constraints and Business rules. The project now tries to give a multi-part standard

to solve the general problem and improve the interaction of technologies involved.

Part 2 of DSDL is a ISO/IEC standard for regular-grammar-based validation, for which RELAX NG was chosen. For part 3 of DSDL, a standard for rule-based validation, the project chose to standardize Schematron as ISO/IEC norm [15]. The DSDL overview document is still only available as working draft [6].

The RELAX NG standard wasn't technically changed much compared to the initial version of the language, whereas Schematron was improved in various ways. Schematron was also opened to be a host language for arbitrary domain-specific languages, not only XPath 1.0.

2.2.3. Languages integrating XML support

A third category of technologies for XML, which is possibly suited to express constraints on XML, are full grown programming languages offering support for XML data, either via API or the language itself, eventually by extension of a base language.

By the nature of this category, such specifications are much more concrete, i.e. less declarative and more depending on specific implementations. They are easy to validate at runtime, as the sole purpose of the used languages is the creation of executable code, but harder to analyze statically, as the constraints are embedded into source code of the language, which has to be parsed. The semantics of the formulated constraints is virtually embedded into the semantics of the programming language, which is far too complex for the task at hand.

Programming languages are ill suited to specify constraints, but focus on creation and manipulation of XML. An introduction to some languages will therefore be given in Section 3 and though it is possible to define constraints within them, they are unlikely to be of more use than the schema languages analyzed above.

2.2.4. Result

The most reasonable choice for the example system is to use RELAX NG for the structural constraints and Schematron for the integrity constraints and the domain-specific validity. Both being recommended by the DSDL project furthermore approves this choice.

The choice of Schematron, however, is guided by the choice of XPath, which can't be used alone to formulate the constraints, so a simple embodiment is needed. Schematron is a good first approach for this but quickly approaches its limits.

Without the possibility to share common expressions in assertions, without being able to define either variables, methods or another form of abstraction, a schema is quickly getting out of hands, when constraints grow more complex. ISO/IEC Schematron at least got a let construct, which allows the definition of variables, but unfortunately there is no implementation of it available yet.

Additionally Schematron allows each node of a document to match only one rule context, which has to be the most specific one. The implementation of Schematron in XSLT certainly benefits from this feature or restriction, as XSLT patterns follow the same semantics. For the example system, however, this is a severe limitation,

affecting the structure of the schema document and the location of assertions for different constraints.

Last but not least Schematron as it is available hosts the XPath 1.0 language only, which proved to be a big step compared to other schema languages, but yet not powerful enough to express even relatively simple constraints conveniently or at all. XPath 2.0 [38] is a revised and enhanced version, which is already used in XSLT 2.0 [43] and XQuery [39] and is sufficiently powerful to express all constraints which are likely to be checkable at all without the need for higher order logic.

As I don't even used most of the features of Schematron I decided to define an own host language for XPath 2.0, based on a small subset of Schematron, which basically only consists of rules and assertions, enhanced by variables and multiple matching contexts. The implementation is done in Java, using the XPath 2.0 and DOM APIs, within a method of under 100 lines of code, which even includes comments.

Most of the constraints formulated in Sections 2.1.2, 2.1.3 and 2.1.4 are formulated in two documents, being a RELAX NG schema file and a schema file for the Schematron enhanced subset language. Constraints not expressed there include only some very powerful statements of the domain-specific constraints.

2.3. Resulting Constraints

2.3.1. Structural Constraints

The structural constraints S1 to S12 are formulated in RELAX NG for the most part. Such constraints could be checked one by one, in a rule-based language, but they heavily rely on an integrated design of the complete system. The RELAX NG schema represents one possible structural concept, which incorporates the structural rules revealed in Section 2.1.2.

The constraints S3 and S7 are both not represented in the schema, as I decided to discard both constraints for the example system. S3 defines the list of all known areas, which is technically not of interest and not needed for any other constraint. The relevant aspects of the existence of such a list are captured in constraints like D5, which states that the list has to contain at least three valid items. To realize constraint S7 would be an additional effort, not adding any new facets to the problem of constraining data.

The schema file is annotated informally, stating for each block which constraint is realized by it. There are also many design decision made in the structure, which thereby are new structural constraints not mentioned in Section 2.1.2:

DD1 To support domain-specific validity conditions, we represent the two possible end stati "succeeded" and "failed" of a student in an optional attribute value. As long as the student is still studying, this status field is absent.

DD2 The main body of a student element will be categories in which the student can accumulate ECTS credits. Each category will consist only of similar sources of ECTS and state the current sum of ECTS in a dedicated attribute value of the category. The focus block, for example, is not one of those categories, but a

kind of meta-category, holding categories for each type of module a focus block has.

DD3 Managing minor subject modules in the example system would render it unnecessary complex, as it would not add any new category of constraints, but complicate the existing ones. The system therefore does not include minor subject modules, but needs an alternative way to express examinations in them.

DD4 A student can have up to three extension blocks and will have to change one into a focus block at some point. To keep it simple, however, the structure allows the student to have up to three extension blocks and a focus area at the same time. The integrity constraint I6-2 restricts this possibility.

DD5 The thesis is handled differently than modules. It can be added at any time, but thereby activates constraints that have to hold, like D13. The status of the thesis is involved in various constraints regarding the progress and status of the student.

DD6 The status of an examination is simply one of "enrolled", "passed" or "failed". The idea is that any enrolled examination has to lead to one of the two other results and an examination cannot be cancelled as whole, only single attempts can be cancelled. The details of different attempts and time constraints involved are thereby disregarded.

DD7 The example system will be hosted in a single document, containing the handbook as well as the students. Splitting the parts into several documents does not contribute to the core of the task at hand.

Most of the structure is defined in an inlined notation, such that subelement definitions are most of the time defined within the definition of their parent element. In the following few examples, how constraints are formulated in RELAX NG, the definition of child elements and irrelevant parts is collapsed to three dots ("...").

The following part of the schema realizes constraint DD7, as it says that only documents representing a university (`uni`), containing a handbook and a possibly empty list of students are valid:

```
start =
  element uni {
    attribute semester { xs:integer },
    element students {
      element student {
        ...
      }*
    },
    element handbook {
      ...
    }
  }
```

The `start` definition is essential, as it defines that the one top level element of each valid XML file has to be `uni`. By transitivity this also guarantees that certain other elements are also included in each valid document.

The constraint DD1, which states that it must be possible to set the end status of a `student`, is expresses in this way:

```
element student {
  ...
  attribute status { "failed" | "succeeded" }?,
  ...
}
```

The `@status` attribute is optional and can only have those two values.

The last example, which shows constraint DD2, demonstrates how common definitions can be shared in RELAX NG:

```
ects-and-exams = (
  attribute ects { xs:integer },
  examinations
)
```

The common block "ects and exams" specifies that an integer attribute for the sum of ects has to be present and that examinations are allowed in this context, which is an already defined structural definition. Each category mentioned in DD2 references this definition so it is ensured they can all be treated equally with regard to this aspect of DD2.

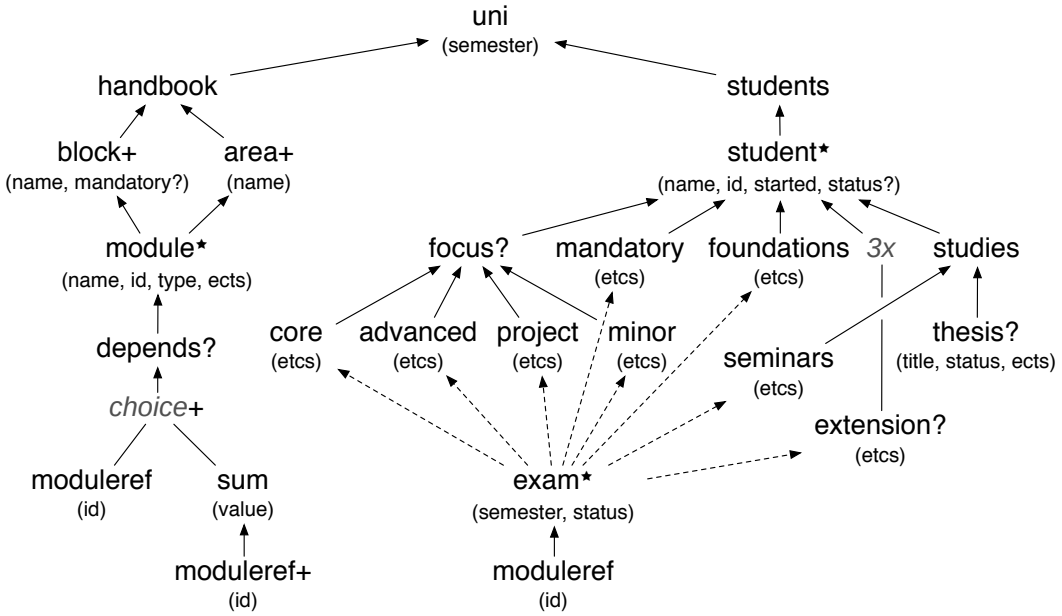

Figure 2.2.: Visualization of the Structural Schema

The complete annotated schema file can be found on the electronic medium. Figure 2.2 shows a visualized version of it using an informal notation which neglects some details.

Attributes are shown without types in parenthesis under their respective elements. Some parts of the of the syntax used are inspired by the abbreviated RELAX NG notation. Listing A.2 on Page 83 additionally shows a valid example document.

2.3.2. Integrity and Domain-Specific Constraints

The integrity constraints I1 to I6 and the domain-specific constraints D1 to D19 are formulated in an enhanced subset of Schematron, whose syntax can quickly been given as RELAX NG schema:

```
start =
  element schema {
    element rule {
      attribute context,
      attribute name,
        element variable {
          attribute name {
            xs:string { pattern = "[a-zA-Z]+" }
          },
          attribute value
        }*,
        element assert {
          attribute test,
          text
        }+
    }*
  }
```

A schema is simply a list of arbitrary many rules, which have a context and a name. Each rule can additionally have arbitrary many variables and at least one assertion. Variables are name value pairs, which can be used in the XPath 2.0 test expressions of assertions.

A rule has to hold in every matching context, which can also be none at all. For technical reasons, a variable can only hold exactly one element. It is neither allowed to assign the empty sequence, nor any sequence with two or more elements. Assertions contain a text message, which is reported whenever an assertion fails, to give details about the failure.

Contrary to RELAX NG specifications and the structural constraints, it is now possible to simply "translate" each constraint into one independent set of rules. I will demonstrate how constraints are formulated within this language:

The structural constraint DD4 needs an accompanying integrity constraint, as the structure is kept simple at that point. This integrity constraint is formulated as a simple rule:

```
<rule name="I6-2" context="student">
  <assert test="not(count(extension) > 2 and focus)">
    Student %@id% should have up to three extensions or up to
    two and a focus field.
  </assert>
</rule>
```

The error message can contain XPath expressions, which can again utilize variables defined in the rule, so they can easily output relevant information about the context in which they occurred. The syntax used is that each occurrence of the percent ("%") character changes from text mode to XPath evaluation mode.

For the remainder of this work, we will use an abbreviated syntax for the constraint language, which should be easier to read and focuses on the essential aspects. We will

completely neglect the error messages and simply give the name of the rule, followed by its context in parenthesis, as well as variable definitions starting with the ":" character and asserts by the "!" character:

```
I6-2 (student)
! not(count(extension) > 2 and focus)
```

The integrity constraint I5 specifies that student identifiers and module identifiers each have to be unique, which is done in the following way:

```
I5-1 (student)
! count(//student[@id = $current/@id]) = 1

I5-2 (handbook//module)
! count(//module[@id = $current/@id]) = 1
```

The special variable `$current` mustn't be redefined as normal variable and always represents the matched context of the rule. The uniqueness is expressed as "there is exactly one of this id".

A last example, containing a variable, is the constraint D12, which asserts that a student who has not yet succeeded or failed with his studies is progressing:

```
D12 (student[not(@status)])
: $semester = /uni/@semester - @started
! $semester < 2 or mandatory/@ects >= 30
! $semester < 4 or mandatory/@ects >= 50 and
    sum(.//@ects) >= 60
! $semester < 6 or mandatory/@ects =
      sum(//handbook/block[@mandatory]/module/@ects) and
    sum(.//@ects) >= 110)
! $semester < 9
```

In this case the variable is used in each of the four assert statements to guard the following condition. The context of the rule is guarded by an expression, such that only students match the context which do not have a `@status`.

All other constraints are formulated in the same way, using increasingly difficult test expressions and more features which are exclusive to XPath 2.0. The complete specification file can again be found on the electronic medium. A pretty printed version is also given in Listing A.1 on Page 79.

3. Manipulating XML

In this chapter the specification of algorithms is the main focus. Again we will start with the application domain to both see what algorithms are needed for the example system and determine requirements for technologies we could use. After getting an overview of available languages, I will fix one and show how algorithms are formulated in it. The result of the chapter is an algorithm specification containing the necessary algorithms of the example system.

3.1. Application Domain

The example system will be able to manage arbitrary many students in different phases of their studies. To achieve this, the system needs a number of algorithms, which are capable of modifying the document in an appropriate way, without invalidating any constraints on it. The needed operations are:

- **addStudent:** It has to be possible to add a new student to the system, which starts with an empty schedule in the current semester.
- **enrollExam, passExam, failExam:** Students will then enroll for examinations to various modules and they will either pass those exams or fail in them.
- **addExam:** Because of D19, there are also examinations, which are simply added as "passed" to the student.
- **addThesis, passThesis, failThesis:** To succeed with his studies, the student has to get a thesis topic one day and will either pass the thesis or fail.
- **markSucceeded:** Once a student has done everything needed, he can be marked as "succeeded".
- **endOfSemester:** At the end of each semester, the "global time" has to be incremented, which also means that some students might invalidate the regulation of progress, so they fail with their studies.

Most of the algorithms will need at least one parameter to specify the correct intention of the generic operation, like the student identifier. There will be virtually no algorithm which will always be able to do a modification on the document, regardless of actual parameters or the state of the document.

An operation adding a student, for example, while being very simple, will still have to check if there already is a student of the given id and abort if so. Even if the id is not a parameter, but supplied by the algorithm itself, it still could be impossible to

insert a new student, if the document already consists of a student for each possible id.

A technology able to formulate algorithms on XML documents will need to support the following concepts:

- It has to be able to create, manipulate and delete elements and attributes of a document. It may also be necessary to move elements and attributes to other elements.
- It has to be able to check conditions of the same complexity as the constraints which are defined on documents.
- It has to be able to formulate conditional operations to adjust to the state of the document and to the given parameters.

3.2. Choosing the XML technologies

Some of the technologies reviewed in Section 2.2 are also suited to formulate algorithms on XML, in fact the focus of some of them lies only on manipulations, but since they are remotely suited to also express constraints, they are already analyzed there.

Unfortunately there are far less technologies available which are capable of formulating algorithms than there are, for example, schema languages. Additionally we have to keep in mind, that the ultimate goal is not only to formulate algorithms at all, but also to analyse them with regard to the constraints formulated using the technologies of Section 2.2.

This section will introduce more XML technologies and analyze them regarding their capabilities to support the goals of the example system, especially the specification of the needed algorithms of Section 3.1.

As the ultimate goal is to show static properties of the algorithms, concerning the formulated constraints, a specification of an algorithm should again be

- **readable**, such that the presented algorithm only shows the relevant logic for the application domain and it is adequately easy to trace the idea of the algorithm and why it is plausible it correctly does what it is supposed to do.
- **declarative**, i.e. the focus of the formulation of the algorithm is the idea itself, not the way it is implemented in detail.
- **executable** by an interpreter or indirectly by a compiler, i.e. the example system can be executed without a need to implement the algorithms again in another language.
- **adequately complex**, i.e. the algorithms should be formulated in a language sufficiently complex to support all needed checks and operations but should not contain arbitrary more other constructs or powerful concepts.
- **easy to analyze** by tools, as standard technologies together with an adequate syntax for the level of complexity are appreciated for analysis.

- **closely related** to the constraint specifications, such that no translation of concepts is needed and it is easier to show that the algorithm does not invalidate a specific constraint.

3.2.1. DOM and SAX

An introduction to both technologies is already given in Section 2.2.1, both are too basic and low level APIs concerned about the finer details of XML and the support of all features of it. Opposed to that are a few simple algorithms manipulating some parts of an XML document whenever all context conditions are fulfilled.

To formulate these algorithms in DOM or SAX, together with a host language like Java would both be quite error-prone and awkward. Such a combination of a full-grown programming language with an API is in no way declarative, adequately complex or closely related to constraints formulated in XPath. Even the readability is lacking, as the logic of the program is hidden in Java control structures, references and API calls.

3.2.2. XSLT

XSLT (XSL Transformations) and even XSLT 2.0 [43], as transformation languages, are much more suited to express manipulating algorithms on XML data, but still suffer from being to generic. A local manipulation is a more specific task than a complete transformation, resulting in the formulation of such algorithms being a bit awkward too.

The usage of XPath in XSLT is an advantage, as it automatically brings operations closer to the constraints they might violate and thereby supports automated analysis. Still, the basic idea of XSLT does not match with those of manipulating algorithms, which results in XSLT being not really suited for them.

Without optimizing implementations, for example, an XSLT sheet realizing an algorithm will always copy the majority of a document, which is not affected by it. Another example is the formulation of conditional operations, which is also not a focus of the language and cumbersome to use.

Using XSLT as technology to formulate algorithms on XML should only be considered an option if there are really no better suited technologies available.

3.2.3. XQuery and XUpdate

XUpdate [14] is a first XML technology, which explicitly targets the manipulation of XML documents. It is a host language for XPath and formulated in XML itself. XPath expressions are used to locate and select nodes of a document, which are to be manipulated.

The language was created in the context of the XML:DB initiative for XML databases, a context which needs a language to define database operations, which are obviously also manipulating. Unfortunately the initiative seems to be discontinued since 2003 and the main website is already hijacked by domain grabbers. The project is also hosted on Sourceforge, where some information can still be found [13].

Though being the first good approach to formulating algorithms to manipulate XML, which are as well readable and easy to analyze, XUpdate has several downsides. The biggest technical downside is the lack of conditional processing capabilities, which are also stated in the working draft (of September 2000) as "open issues".

Without such capabilities XUpdate is not suited to formulate complete algorithms, but merely operations or blocks of operations. Such operations are of no use in the example system, as the goal is to prove each invocation of an algorithm as correct, no matter what the parameters are or which valid state a document is in. This can obviously only be done in cases where the algorithm is capable of checking some essential conditions before actually doing operations.

Other practical problems with XUpdate are the lack of implementations around and I had no luck in getting the reference implementation called "Lexus" of XML:DB to work. In any case it is not advisable to select a technology which is discontinued for over two yours by now, though the general idea is still one of the best fitting around.

As the name suggests, XQuery [39] is a query language only, meaning it was not designed to formulate operations upfront. There is, however, a so called "XQuery Update Facility" [42], which is still a working draft only up to the date of this work. It adds the same kind of operations available in XUpdate to the XQuery language, making it a good candidate for the example system.

Unfortunately this technology also suffers from several downsides. XQuery's syntax is not based on XML but a separate new one, embedding XPath in it. This results in algorithms formulated in XQuery being harder to analyze than, for example, XUpdate and having to deal with much more syntax, which is simply not needed. XQuery is far too complex and the interesting parts for this work would only be the update facility extension together with some language constructs, not so much the query language itself.

Being a fairly new technology, which is still not standardized, there are no implementations for XQuery's update facility, so it would only be usable to define algorithms, but not to deliver a running system as proof of concept. This pretty much discards XQuery at this point.

3.2.4. Languages integrating XML support

Such languages could be quickly discarded as constraint specification languages in Section 2.2, but still the specification of algorithms is a task they all are designed for. The focus of languages in this category is the convenient and safe utilization of XML data or XML structured data within common programming languages.

Most often this simply means that the type system of proposed languages is enhanced to include XML data or semi-structured data in general. The languages can thereby guarantee statically, that structural constraints are never violated. This is done by either importing XML data types formulated as schemas or by allowing the definition of complex XML like data types within the language.

XJ XML Enhancements for Java (XJ) [10] is an enhanced version of Java 1.4 and allows to import XML Schemas as data types. All defined elements from the schema

are usable as classes in XJ, as subtypes of `XMLObject`, and it is thereby possible to navigate on and manipulate XML data with static guarantees.

Those guarantees include, for example, most structural constraints, but lack some of the finer details of XML Schema. XJ cannot check everything statically but also relies on runtime checks for certain properties. Updates are possible in various ways and XJ also supports XQuery to formulate the update values.

Cω Cω [27] is a successor of C#, adding several concepts to the type system, including streams, anonymous structs, choice types, context classes and generalized member access. With these concepts it is possible to materialize semi-structured data in the Cω type system and therefore support XML in the language itself.

The language is strongly typed, guaranteeing static properties about semi-structured data. Cω also incorporates XPath and XQuery into the language, using the new concepts to interface the returned values with the language, but changed the syntax of both. Manipulations in Cω can be done as usual on objects.

Linq Language Integrated Query for XML Data (Linq) [8] is a project targeting consistent query mechanisms for object, relational as well as XML data from within common programming languages. As such, the project and the languages arising in it are not inherently suited to express manipulating algorithms on XML data.

Based on the idea of Linq, Xlinq [24] especially focuses on XML and compares itself directly to the DOM API. The introduction given in [24] is an interesting read on the downsides of the generic DOM API, which are also mentioned in Section 2.2.1.

XDuce XDuce [11] is a functional language, using a tree automata based type system to handle semi-structured data. A pattern matching facility is used to retrieve data and the language is based on a powerful type inference system to allow programmers to write practical applications on complex data types.

RELAX NG uses the same underlying mathematics to describe valid documents as XDuce uses to define types. It should therefore be possible, though not done at the moment, to import a RELAX NG schema into a XDuce program as data type.

Scala The official website[1] describes Scala as follows:

> Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages.

The language is compatible to Java and .NET and uses a separate compiler to produce compatible compilation artifacts. The features of interest for XML development are the XML Processing facilities. It is possible to create XML documents with the native syntax of XML and represent XML documents as values in the language in general.

[1] 1st paragraph of the front page at [20].

The language offers support for many other paradigms and approaches, which are useful to work with XML values. Unfortunately the language is again neither adequately complex nor closely related to the constraint specifications.

Though being interesting technologies in the field of XML, all these languages are not suited to express the algorithms of the example system, at least not in the context of this work. They are all executable, which is a big plus, but do not meet the requirements for being adequately complex, easy to analyze or closely related to the constraint specifications.

The latter is mostly due to the fact that the majority of the constraints at hand isn't structural but much more complex. The languages do a good job at giving static guarantees that a valid program does not harm the structural constraints of a document. These constraints, however, while being very important, are not the most difficult to verify statically. In fact the languages at hand show that many solutions for this category of constraints are already ready to use and incorporated into languages.

3.2.5. Result

There is not really a good technology available, which allows the declaration of algorithms on XML documents, the analysis of the algorithms in an easy way and also closely relates the algorithms to the constraints on the target documents. The technology closest to those constraints is XUpdate, but it is neither finished yet, nor usable.

XUpdate's idea is essentially the same as the one of Schematron, to embed XPath in a XML based language, to provide a powerful new language with only few added concepts. So I decided to implement an own update language with XML syntax, using XPath 2.0 and DOM for the implementation.

The language features parametrized methods, containing only variable definitions and assertions as well as a handful of manipulating commands. The structure of method bodies follows strict rules to keep them simple and to support static analysis. The essential idea of a method is that it checks all parameters, and the conditions that have to hold on the document to successfully execute the method, and then calls the needed operations or aborts.

The interpreter for this language is implemented in Java, within several methods of together about 500 lines of code, including extensive code comments. The implementation is straight-forward, delegating all tasks needed to either XPath or DOM whenever possible, only the scoping rules within methods, regarding variable definitions, cause some trouble and are responsible for this language being a bit more complicated then the XML constraining language.

3.3. Resulting Algorithms

3.3.1. Overview of Algorithm Features and Syntax

The language used to specify algorithms is inspired by XUpdate and is a bit more complex than the constraint specification language. A file containing algorithms has

the following abstract structure:

```
start =
  element algorithms {
    element method {
      attribute name,
      element parameter {
        attribute name { ident },
        attribute type { "string" | "integer" }
      }*,
      commandblock
    }*
  }
```

A specification of algorithms is a list of methods only, which all have a name and arbitrary many parameters of a fixed type, which is `string` or `integer` only at the moment. The rest of the method consists of a command block only:

```
commandblock =
 ( element assert {
     attribute test,
     text
   }* &
   element variable {
     attribute name { ident },
     attribute value
   }* ),
 element node {
   attribute name { ident },
   node
 }*,
 ( ( element append { ... } |
     element remove { ... } |
     element move { ... }    |
     element update { ... } |
     element call { ... }
   )* |
   element for { ... } |
   element if { ... }
 )
```

A command block contains arbitrary many assertions, containing a XPath 2.0 test as well as an error message, and variables as name value pairs, in any order. Next to that nodes can be constructed, which can be used by referencing their name later for `append` operations.

The last part of a command block is either a list of operations, which ends the nesting of command blocks, or a control structure containing one or more command blocks. The idea is that control structures can divide the control flow using conditions, but it is never recombined. So after control reaches a block of operations, the method is done and all the conditions, variables, assertions and nodes checked or defined on the way can be used to analyze the effect of this execution of the method.

For loops are necessary to express all needed algorithms but do not really fit into this conception, so they have to be used with caution only. The same goes for method calls, which are also very convenient but have to appear in “safe” contexts only. Both concepts can make the analysis of the algorithm very hard when used too freely.

The `for` loop is used to bind arbitrary many values to a variable, one after the other, in the order they appear in the document:

```
element for {
  attribute variable { ident },
  attribute in,
  commandblock
}
```

The `@in` expression specifies the sequence of values to use, while `@variable` is the name of the special variable holding the current value.

The `if` construct can be used to fork the control flow, such that only one of the command blocks is executed in any given execution of a method or `for` loop body:

```
element if {
  attribute test,
  element then
    commandblock
  }
  ( element elseif
      attribute test,
      commandblock
    }*,
    element else {
      commandblock
    }? )?
}
```

The `if` consists of a test expression and a `then` block, which is executed whenever the test evaluates to true. The `if` body can then contain arbitrary many other blocks (`elseif`) to be executed whenever no block before it matched so far and the own test expression evaluates to true. At last the `if` body can contain a block (`else`) which is executed whenever no guarded block matched.

All blocks which are possibly executed thereby are at the same depth in the XML tree, only the test expression for the `then` block is not located in the block, but in the `if` itself.

The available operations are:

- **append**

  ```
  element append {
    attribute context,
    ( node |
      element attribute {
        attribute name { ident },
        attribute value
      } )
  }
  ```

  Appends a single element or attribute value as the last child to a node specified by `@context`.

- **remove**

  ```
  element remove {
    attribute context
  }
  ```

Removes all nodes specified by `@context`, so this may also be a sequence.

- **move**

```
element mode {
  attribute context,
  attribute to
}
```

Moves all nodes specified by `@context` to the node specified by `@to`. All nodes are appended to the targets context and document order is maintained.

- **update**

```
element update {
  attribute context,
  attribute value
}
```

Updates the content of an attribute or element specified by `@context` with the result of the `@value` expression.

- **call**

```
element call {
  attribute method { ident },
  element parameter {
    attribute value
  }*
}
```

Calls a method from the same algorithm specification, with given name (`@method`) and parameters set to the results of the `@value` expressions.

Elements are created using a syntax inspired by RELAX NG:

```
node = (
  element element {
    attribute name { ident },
    element attribute {
      attribute name { ident },
      attribute value
    }*
    ( node* |
      element content {
        attribute value
      } )
  } |
  element copy {
    attribute node { ident }
  }
)
```

Elements and attributes are created in their current context, using the name given in the attribute. Attributes are set to the result of the `@value` expression, while elements can either contain other elements and attributes or consist only of the result of a `@value` expression, in which case the element is used like an attribute.

Whenever an element is expected it is also possible to `copy` an existing node referenced by name (`@node`). The node needs to be visible on the execution path to the operation needing the node.

3.3.2. Example Algorithms

One of the most simple methods, which does not have to check much, is addStudent, which adds a student to the system:

```
<method name="addStudent">
  <parameter name="name" type="string"/>
  <parameter name="id" type="string"/>

  <assert test="not(//student[@id = $id])">
    Their should not already be a student with id %$id%.
  </assert>
  <assert test="matches($id, '[0-9]{6}')">
    The given id (%$id%) has to conform to student id rules.
  </assert>

  <append context="/uni/students">
    <element name="student">
      <attribute name="name" value="$name"/>
      <attribute name="id" value="$id"/>
      <attribute name="started" value="/uni/@semester"/>
      <element name="mandatory">
        <attribute name="ects" value="0"/>
      </element>
      <element name="foundations">
        <attribute name="ects" value="0"/>
      </element>
      <element name="studies">
        <element name="seminars">
          <attribute name="ects" value="0"/>
        </element>
      </element>
    </element>
  </append>
</method>
```

The method has two string parameters, one being the name of the student, the other being his identifier. The only situations, in which such a method should fail are those where the identifier is invalid or already present in the system.

Both conditions are checked with assertions, so these cases can be eliminated and would abort the method execution giving a reason what went wrong. The method then appends a new student to the list of students, initializing all needed sections to default values.

This setting shows the principal used to formulate all algorithms, no matter how complex:

- First of all, the parameters are checked for validity, this can be either the format of the parameter, if not expressed by the type, or the existence of an element specified by the parameter.

- After that all conditions on the document, which have to hold to make this method call valid are checked.
- Finally the necessary operations to execute the operation are done.

Most of the constructs not used in this method are only added for convenience reasons. They make the specification of algorithms easier and more readable, while being justifiable complex to handle in the analysis of the algorithm.

For the remainder of this work, we will use an abbreviated syntax, which should be easier to understand for the reader. Many aspects of the syntax are inspired by Java, such as the notation of methods, parameters and `if` constructs. Variable definitions, assertions and node constructions are preluded by keywords. Operations are stated by name and contain their context in parenthesis, followed by the second parameter, if any. All statements are terminated with the character ";", like it is done in Java. Constructed XML elements are provided in XML notation, while attribute values can be XPath expressions.

Listing 3.1: Method: addStudent

```
addStudent(string name, string id) {
  assert not(//student[@id = $id]);
  assert matches($id, '[0-9]{6}');

  append (/uni/students)
    <student name=$name id=$id started=/uni/@semester>
      <mandatory ects="0"/>
      <foundations ects="0"/>
      <studies>
        <seminars ects="0"/>
      </studies>
    </student>;
}
```

The passThesis method is only slightly more complex:

Listing 3.2: Method: passThesis

```
passThesis(string sid) {
  assert //student[@id = $sid];
  var student = //student[@id = $sid];

  assert $student/studies/thesis[@status = 'topic'];
  var thesis = $student/studies/thesis;

  update ($thesis/@status) 'done';
  update ($thesis/@ects) 12;
}
```

The method only takes a student identifier as parameter, which has to exist in the list of students and the student obviously needs to have a `thesis` in the correct state. If these conditions hold, the thesis `@status` is updated, as well as the `@ects` value of the category.

Variables and assertions can appear in any order, which is significant for execution. Variables can be used only after they are declared, but they can only be declared if it

is secured they evaluate to exactly one value. So the variable `$student` can be defined only after the first assertion. Because of I5-1 there cannot be more than one `student` with that `@id`, but there could be none at all.

The `$thesis` variable can only be defined after it is checked the `student` has a `thesis` at all, however in this case the assertion checks more than is needed to define the variable.

The `if` construct is an advanced assertion mechanism. If the assertion holds, the command block of the `then` element is evaluated, if it does not hold, the method is not aborted but enters an alternative execution path. The addModule method contains many nested `if` statements to examine where exactly the module has to be added to the student. One of those `if` statements looks like the following:

```
if ($mtype = 'core') {
    ...
  } else if ($mtype = 'advanced') {
    ...
  } else if ($mtype = 'project') {
    ...
  } else if ($mtype = 'seminar') {
    ...
  } else {
    assert false;
  }
}
```

The `$mtype` variable holds the type of the module taken from the handbook. The `if` is located in another `if` statement, which already guarantees the module to belong to an area, so I2-2 guarantees the module type is not basic and together with S1 the `else` element in the `if` should never be chosen.

The `if` construct is thereby used to combine several different methods into one, which are very similar to each other. The addModule method could be split to 11 very similar methods, which is the reason why the method is much longer and more complex than most others.

All available methods of the example system are supplied on the electronic medium. Pretty printed versions of several methods can be found in the listings included in this work.

4. Static Guarantees

After having constrained the data stored in XML documents of interest to our system and having provided algorithms to manipulate those documents, it is time to bring both pieces together and prove that the algorithms at hand are "safe" with regard to the constraints, without checking the constraints dynamically at all, like it is usually done in present systems.

4.1. General Considerations

Invoking any method on a XML document for which all constraints hold has to leave the document in a state in which again all constraints hold. So the constraints hold at any given time between two method invocations and most important whenever information of the document is retrieved (read-only access). The constraints form a global invariant of the system.

To achieve this goal we actually need to show two things:

1. The invariant holds for the start document.
2. Each method leaves the document in a state in which the invariant holds, whenever the invariant was fulfilled before the invocation of the method.

This first approach of course assumes a sequential call of methods and consciously leaves out the much more complex situation of concurrent method calls.

The first point we have to show can be easily proven by checking the start document. This is nothing to worry about, as it is only one dynamic check for the lifetime of the document, provided we prove the second point and only use such methods to manipulate the document.

The second point, however, is much more complex. The following sections will analyse the different types of constraints and how they can be statically shown to hold for each method. The algorithm specification language was explicitly kept simple to analyze, such that algorithms are more likely to follow a simple scheme and concentrate all operations into one block. From all constructs available in the algorithm specification, only few are actually suited to invalidate constraints, as the rest is completely read-only.

Proving methods correct has three major facets:

- The technical aspects, containing:
  - The correct implementation of the algorithms interpreter and the constraints checker.
  - The syntactical correctness of the constraint and algorithm specifications, including the XML part as well as the embedded XPath part.

  - The correctness of the underlying runtime environment, operating system, hardware, etc.

- The basic logical aspects, containing the following:

  1. XPath expressions should not abort with dynamic errors or type errors.
  2. XPath expressions used as context of an operation have to evaluate to exactly one node or a set of nodes, as requested by the type of operation. XPath expressions used in other contexts need to fulfill other constraints, which are not all listed here and are dealt with whenever the need arises.
  3. Variable definitions need to evaluate to exactly one node or value, especially not the empty sequence.
  4. Variable definitions must not be invalidated by operations, i.e. the variable still holds exactly one value, which should be the same as before the operation.

- The major logical aspect, being:

  - The semantics of the algorithm and especially its operation blocks don't invalidate any constraints.

Throughout this work, we neglect all technical aspects and assume them to be correct. These aspects are not the focus of this work; Some are even ongoing research topics, while others are not hard to prove in general. The correctness of the Java implementations is probably the weakest point of those listed and is supported only by the running example system.

The major logical aspect is the most important one and the one of interest to this work. It will be the focus of the following sections, but without the basic logical aspects done, we will not be able to show much at all. So whenever we want to prove any method correct, we first have to check the list of these aspects.

In order to show basic aspects we do the following:

1. The XPath expressions used in the example system are most often navigating in the source document, containing only simple guarding conditions. Such expressions do not need much attention concerning dynamic errors and type errors. Especially test expressions, however, include function calls and other non-basic constructs, which need a bit of attention regarding errors and will be looked at when the need arises.

2. All contexts of operations have to be validated, which is either done by looking at the global invariant, which includes the structural schema, or the assertions and conditions already passed in the algorithm.

3. The global invariant together with passed assertions and conditions will also be used to guarantee variables to be correctly defined.

4. The algorithms of the example system will avoid manipulating the context of variables altogether, so a comparison of the involved XPath expressions will show that variables cannot be invalidated.

All these steps have to be done for each method and basically rely on the same technique. The global invariant, which holds at least to the first operation, the additional assertions of the algorithm and the conditions checked in control structures are sufficient to show that XPath expressions and variable definitions are correct and don't lead to any errors.

4.2. SX Constraints (Structural)

Of all constraints, which make up the invariant, the structural constraints are easiest to check. In many cases, it is sufficient to analyze the block of manipulating operations only and completely disregard the rest of the method. Whenever it is necessary to know something about the current structure to see if a manipulation is valid, the condition should be checked in an assertion or in a control structure.

The relevant operations which have to be examined in the algorithms of the example system are `append`, `update`, `move` and `remove`. They all have an XPath expression as context of the operation, which relates the operation with the RELAX NG schema. In the case of `append`, for example, we have to check if the appended attribute or element, together with all contained elements, is allowed in this context by the schema.

The same goes for `remove` operations, though they are often used in combination with other operations like `append` or `move`, such that elements which cannot be removed in general are supplied again and the structure stays valid. From a structural point of view, the `move` operation is a combination of `remove` and `append`, but it cannot simply be replaced by a sequence of the two operations.

`Update` operations only manipulate the value of attributes or special elements, so the type of value has to be checked for consistency with the schema. The same applies to values generated in the context of an `append` statement or values moved into a new context with the `move` statement.

The example system uses either free text, patterns, enumerations or integer as value types, where free text does not need to be checked at all. Pattern constrained text values are explicitly checked by an assertion in the algorithm and enumerations are easy to check as well. Integer values should be given as XPath expressions which result to values and are often sums of other integers, so as well not a big problem.

Isolated method calls, which are used in situations where the structure is currently valid, are not a problem, as the called method is analyzed as well and therefore can't invalidate the structure. The example system must not show any situation, in which operations are structurally only valid in conjunction with a `call` statement.

We will now look at several methods and prove them correct with regard to the structural constraints.

4.2.1. Appending Elements: addStudent

The method as shown in Listing 3.1, takes an `id` as well as a `name`, which are both free text parameters, and adds a new `student` to the list of students, whenever possible with the given parameters. Only one operation is used for this, which is an `append` in the context of `/uni/students`.

The RELAX NG schema tells us that each valid document starts with the `uni` tag and that this element always has exactly one child element of the name `students`, so this context is always exactly defined. The operation adds a `student` element, which is the only allowed child element of `students` and can appear arbitrary often. So this operation can so far be done without looking at the document at all and we know it to be correct. In fact this is true for the complete method, at least from the structural point of view.

To prove the rest of the `append` operation, and to prove `append` operations in general, we have to match the constructed tree of XML elements and attributes against the tree automata defined in the RELAX NG schema.

The next parts, which are defined as mandatory by the schema, are the three attributes of `student`. The algorithm adds three attributes of correct names, so we only need to show that their type is correct. The `@name` attribute is free text, so there is nothing to show, but the `@id` attribute only accepts strings of a specific pattern. The value assigned for this attribute is taken from a string parameter of the method, which is explicitly checked against that pattern, so this assignment is valid as well. The last attribute is `@started`, which has to be integer. The value assigned is taken from the `@semester` attribute of `uni`, which is also known to be integer. The schema additionally tells us that this attribute is not optional and always present in any valid document.

The algorithm then adds all other child elements, which are not marked as optional by the schema, and initializes the necessary integer attributes to 0.

So the element constructed is valid with regard to the part of the schema where it is added and it is always valid to add the element at all, since we append it to an arbitrary long list.

4.2.2. For Loop and Conditions: checkProgress

The method checks the progress of all students and can set the `@status` of each `student` when necessary. It's the only method using the powerful `for` control structure, which has to be used with caution to allow verification of algorithms. This, however, is even more of a problem for integrity and domain-specific constraints and will be covered there.

Listing 4.1: Method: checkProgress

```
checkProgress() {
  var semester = /uni/@semester;

  for (student in //student[not(@status)]) {
    var diff = $semester - $student/@started + 1;

    if ($diff = 2 and $student/mandatory/@ects < 30) {
      append ($student) status='failed';
    } else if ($diff = 4 and ($student/mandatory/@ects < 50 or
                sum($student//@ects) < 60) {
      append ($student) status='failed';
    } else if ($diff = 6 and ($student/mandatory/@ects !=
                sum(//handbook/block[@mandatory]/module/@ects) or
                sum($student//@ects) < 110) {
```

```
      append ($student) status='failed';
    } else if ($diff = 9 and (sum($student//@ects) != 180 or
               $student//exam[@status != 'passed']) {
      append ($student) status='failed';
    } else if ($diff = 9) {
      append ($student) status='succeeded';
    }
  }
} // D12 will hold with incremented @semester
```

From a structural point of view, the method uses only `append` operations, which add an attribute called `@status` in the context of an element of the sequence `//student[not(@status)]`. The schema tells us, that `student` elements only appear as child of `/uni/students`, so we know what `student` elements have to look like. It wouldn't harm the algorithm to have used an absolute path right away, but it is not necessary as longs as the `student` element has only one role in the schema.

As the sequence from which the contexts are taken can contain only students, which do not have the optional attribute `@status`, the algorithm can safely add this attribute, without invalidating the structure. The type of the attribute is an enumeration, which contains both possible values, which are assigned to it.

4.2.3. Using Variables, Asserts and Updates: addMinor

There are three operations to look at in this method. The last one is a method call, which we can ignore for structural considerations. If the other two methods are correct and the called method is always correct too, the combination cannot invalidate anything.

Listing 4.2: Method: addMinor

```
addMinor(string sid, string mname, integer mects) {
  assert //student[@id = $sid];
  var student = //student[@id = $sid];

  assert $student/focus;
  assert $mects >= 2 and $mects <= 15;

  append ($student/focus/minor)
    <certificate modulename=$mname moduleects=$mects />
  update ($student/focus/minor/@ects)
    min((20, $current + $mects))
  balanceFS($sid);
}
```

The first operation is an `append` having a rather complex context. It starts with the student variable, which we have to show to contain exactly one element anyways. The value assigned is the sequence of `student` elements anywhere in the document, having a fixed `@id` given as parameter. The constraint I5-1, however, takes exactly these elements as context and states that for each given `@id` of such a `student`, there is no other `student` with the same `@id`. The assertion directly in front of the variable definition additionally states, that there is at least one `student` with the given `@id`, so the variable definition correctly selects exactly one student.

The context of the `append`, however, also navigates downward from the `student`, so we need to check if the selected child elements exist and are unique. `student` elements need not have a `focus` element, as it is optional, but there is another assert statement saying the `student` element in question has such a child element. Once the student contains a `focus` element, the schema guarantees it also has the `minor` element.

We don't go into detail of the rest of the `append`, as there is nothing new to show, but the following `update` statement is of interest. The context of it is nearly the same but shows an additional navigation step to an attribute of the `minor` element. This attribute is non-optional, so it has to exist and it has the type integer. The new value for the attribute is a minimum of an integer literal and the sum of two variables. One is the old value of the attribute, the other comes from an integer parameter, so the result is as well integer.

4.2.4. Dependent Operations, Remove and Move: addModule

This is a complex method consisting of many different cases, telling what to do with the module given as parameter and where to add it in the `student`. The method could easily be split into 11 different ones, so we will stick to analyzing one case only.

```
if (name($entry/..) = 'block') { ... }
else {
  if ($mtype = 'core') { ... }
  else if ($mtype = 'advanced') {
    if ($student/focus[@name = $fbname]) { ... }
    else if ($student/extension[@name = $fbname] and
             not($student/focus)) {
      // case 7:
      append ($student) #focus;
      append ($student/focus/advanced) #exam;
      move ($student/extension[@name = $fbname]/*)
        $student/focus/core;
      remove ($student/focus/core/@ects);
      move ($student/extension[@name = $fbname]/@*)
        $student/focus/core;
      remove ($student/extension[@name = $fbname]);
    } else assert false;
  }
  else if ($mtype = 'project') { ... }
  else if ($mtype = 'seminar') { ... }
  else assert false;
}
```

Case 7 is one of the bigger operation blocks present in the list of algorithms. It consists of two `append` operations, two `move`s and two `remove`s. The `append` operations offer nothing really new, but the elements appended are not defined at the `append`s themselves but before the control structures, so they can be used in any case:

```
node exam =
  <exam semester=$semester
        status=(if ($passed) then 'passed' else 'enrolled')>
    <moduleref id=$mid/>
  </exam>;
```

```
node focus =
  <focus name=$fbname>
    <core ects="0"/>
    <advanced ects="0"/>
    <project ects="0"/>
    <minor ects="0"/>
  </focus>;
```

The context of the first `append` is guaranteed to exist, the context of the second `append` is made available by the first `append`, which adds exactly the elements needed for it.

Both `append` operations don't invalidate the structure, but to prove the second correct we need the first. Besides appending elements or attributes to the structure, such statements form assert statements for following operations. Of course they also invalidate asserts and conditions which included the manipulated context. In this case the last part of the control structures condition is no longer valid, as the state of the `focus` part of the student changed.

The next operation is a `move`, so we have to look at the `remove` facet of the operation, as well as the `append` facet of it. Context of the `move` is an `extension` element, which is guaranteed to exist by the control structures condition and is uniquely defined cause of constraint D10. The `extension` element does not need any child elements, so removing them all does not invalidate its structure.

The `@to` context of the operation needs to specify exactly one element, which is guaranteed by the `append` operation and the fact that there was no `focus` at the `student` before it. The schema tells us, that an `extension` element and a `core` element can have exactly the same children, so the `append` part of the `move` is structurally valid too.

The next operation is a `remove` of the `@ects` attribute of the `core` element, which is invalid. The next `move` operation, however, moves all attributes from the same context as the last `move` operation, to the same target, which is the `core` element. As `extension` elements and `core` elements also have exactly the same attributes, which is only the non-optional `@ects` attribute, the `core` element is valid again.

On the other side the `extension` element is now invalidated, as it is missing a necessary attribute. The last operation of the operation block removes the `extension` element, which is valid for its parent as `extension` elements are always optional, so the missing attribute is no longer of interest.

4.2.5. Summary

To verify manipulating operations with regard to structural constraints, the context of the operations is the first important property, as it locates the structural constraints of interest to the operation. Only those constraints at the context, the parent of the context and all child constraints are relevant and can be invalidated.

For `append` the relevant question is if the appended element or attribute can appear in this context and if the structure of the element matches the corresponding tree automaton.

`Remove` operations will either remove optional elements or appear together with `move` or `append` operations targeting their context. `Move` operations can structurally

be divided into one `remove` and one `append` operation, though the `append` operations value depends on the schema, which is more general than a newly constructed element and therefore harder to prove for correctness in general.

Besides having a well defined context, `update` operations only need to preserve the correct type of the attribute or element.

4.3. IX and DX Constraints Discarded by Structure

The context of operations made it easy to find out which structural constraints need to be checked at all, as these constraints where specified in a grammar-based approach. Integrity and domain-specific constraints, however, are specified in a rule-based way.

This is not a shortcoming of the notation method used for these constraints, but is an inherent property of them. Such constraints bring parts of the document together, which are not always locally connected. They can be arbitrary complex and include many different elements and attributes. Structural constraints are far more limited with regard to these dimensions, for the exact reason to keep them simple and well suited for the particular task.

The consequence of this is that we need a way to determine which constraints are of interest regarding an operation and which can be safely ignored. The basic prove obligation, when showing a method correct, splits into many separated goals, namely one for each constraint of the global invariant combined with the method.

So the first approach is to discard as many constraints as possible for each method, simply because the method "has nothing to do" with those constraints. We will use structural arguments for this.

All other constraints, which cannot be excluded by those arguments, have to be proven correct one by one, using the global invariant as well as the checked assertions and conditions. If we cannot prove the constraint to hold with this information, the algorithm is most probably incorrect, which occurred several times while verifying them.

4.3.1. First Structural Arguments

There are many constraints, which are very situational and unlikely to be violated. I5-1 and I5-2, for example, only specify that `student` and `module` `@id`s are unique in their respective environment. No method will invalidate any of those, if it is not adding a `module` or `student` or changing the `@id` of one or the other.

Many constraints even secure the validity of the module handbook only, which is never manipulated by any of the methods of the example system. Such constraints can be disregarded altogether, though they might still be needed to prove others.

So one observation is that there are several different parts of the document, which have differing relations to each other. There is the module handbook on the one hand and the students on the other. But the students themselves are also fairly independent from each other. In fact constraint I5-1 is the only constraint combining elements from two or more separate students.

The context of a constraint is the anchor point of the constraint, it says how often the constraint is instantiated and can be used to toggle constraints on or off for parts

of the document. There are, for example, a lot of constraints which are only checked for a specific student if he already has a topic for his thesis or if he chose one or more extensions.

The context thereby also says where a rule belongs to. Together with the separation of the complete document into parts, we get a graph of references. Each node of the graph is a major part of the document, each arc represents one or more rules, which have their context in the starting part of the arc, while their assertions are using references into other parts of the document, especially the part where the arc ends.

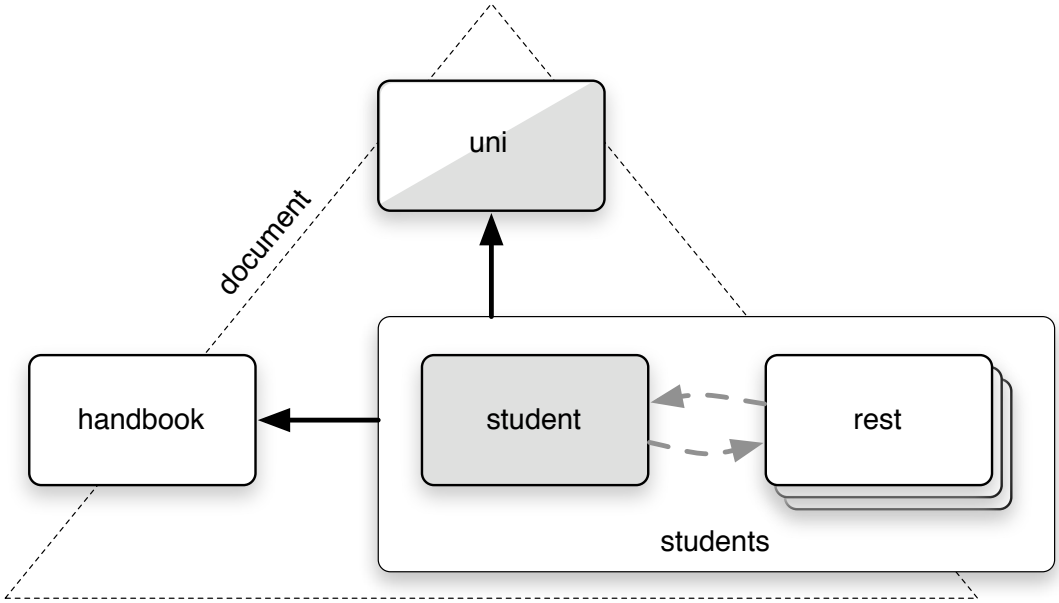

Figure 4.1.: The Major Parts of the Document and their Connection

Considering that the only references between students are formulated in I5-1, we can safely ignore these references and get a much more simple structure with two remaining arcs only. Closed operation blocks are focusing on one student only, with the exception of one method, so all constraints having a context in the handbook or in another student (besides 15-1) can be completely ignored for all these methods.

Our example system does not feature any methods on the handbook, which are in general necessary to maintain it. For those methods, of course, such a simplification would not be possible, as there are many constraints located in the student parts, which reference the handbook. But this is the essence of our example domain. Methods working on the handbook will have a much harder time being verified. Manipulating modules which are already being used by students is also a complicated thing in the real life equivalent.

To verify such methods, the graph would most probably have more nodes and arcs within the handbook part, as it is also splitting into fairly independent parts. So we can always separate all methods into classes which utilize the view of major parts on the system to their best advantage. For the example system, and the methods used there, we can thereby state that the following rules cannot be invalidated:

- **S2**, which is the only structural rule being realized outside RELAX NG, mostly for convenience reasons. It states which blocks a handbook needs to have and if they are mandatory or not.
- **I1, I2-X, I5-2**, concerning the integrity of the handbook and its modules.
- **D1-1, D2-X, D3, D4, D5, D14-1, D16**, being domain-specific constraints for the module handbook.
- **I4-1**, for all methods **except endOfSemester**, as the constraint operates outside of students.

To show these results in a more technical way, rather than using arguments from the application domain and common sense, we have to look at the XPath expressions involved and how they relate to each other.

The first part of the argument used is the centralization of the operations used in a method on one single major part of the document. This is shown by the context expressions of operations and the target expression of the move operations:

- All operations, except endOfSemester, either have a `student` or one of its child elements or attributes as context or are append operations appending a `student` to the list of students. The manipulation of the document in the latter case is therefore still located at a `student` element.
- As a result we can assume all parts of the document, which are not contained in a `student` element, to be static. All references a constraint has to static data cannot be the cause for this constraint being invalidated by the method.

The second part of the argument used is that several constraints are operating completely on static data, i.e. their context is within static data and all assertions only reference static data. This is found by comparing all involved XPath expressions and their subexpressions with the path to a `student` and realizing that the latter is not a prefix of the former.

Constraint D4, for example, includes the following XPaths, which all not have `/uni/students/student` as prefix.

```
/uni/handbook/module/@type
/uni/handbook/module/@ects
/uni/handbook/module
```

A specialization of this argument, which is in most cases sufficient, is that all XPath expressions involved in constraints share a common prefix, which is statically known to be incomparable with the path to a `student`.

- The constraints excluded above share the common prefix `/uni/handbook`, while all students share the common prefix `/uni/students`.

A shared prefix for all involved XPath expressions essentially states a localization property of the constraint to the element described by the prefix and its subtree. The `uni` element and its constraint I4-1 do therefore not apply to this kind of argumentation.

- Constraint I4-1 operates on static data for all methods except endOfSemester, as `/uni/student` is not a prefix of `/uni`.

For all structural arguments we are using elements and paths to elements only. Attributes are seen as part of their parent element only and do not themselves add to the structure. Constraint I4-1 is therefore considered to operate on the element `/uni` only.

Following this argumentation and the considerations made in Section 1.3 we can also neglect some child elements, which are only complex attributes of another element. Module references, for example, are modelled as child elements, but could as well be real attributes of XML. In any case they are conceptually seen as attribute of examinations only, not stand-alone entities.

4.3.2. Generic Structural Argument

The underlying concept of the previous sections arguments, which rule out a number of constraints completely, can easily be generalized to one strong structural argument.

A closed block of operations, which should be verified together to leave all constraints intact, manipulates only small parts of the document. All parts of the document not affected by any operation are static with regard to this block of operations. All constraints containing only references to this static data can be disregarded while verifying the block of operations, as they will still hold after the operations are done.

Technically this means checking all involved XPath expressions of a constraint against affected parts of the document. Involved XPath expressions are:

- A path expression given as test to an assert statement.
- The path expression given as context of a rule, whenever it is implicitly or explicitly used by XPath expressions in the assert statements.
- The path expressions given within other XPath expressions, examples being
  - guards in axis steps being path expressions
  - sequences for quantifiers and for statements

Affected parts of the document are:

- For `append` operations the path to the added element or attribute as well as all child elements and attributes of it.
- For `remove` operations the path to each removed element, as well as all child elements and attributes.
- For `move` operations the combination of the above.
- For `update` operations the path to the manipulated element or attribute.

All structural arguments used to rule out constraints with regard to one method or closed operation block are based on these general considerations. As we have already seen, it is, however, often possible to give much more simple arguments, which are

both easier to understand and to check. We will therefore look at some more classes of structural arguments, which can be of use for several methods with regard to constraints not touched by them.

4.3.3. Situational Arguments

The arguments used in Section 4.3.1 split into two categories:

- Special properties of the constraints, which could be ruled out, concerning the structure of the document. These constraints where localized to parts of the document, namely the subtree `/uni/handbook` or the element `/uni`.

- Special properties of the algorithms or operation blocks concerning the structure of the document. Algorithms also focused on certain parts of the document only, namely the subtree of `/uni/students/student` or the element `/uni`.

The used parts of the document where quite large, being subtrees of certain elements, containing major parts of the document. When looking at single constraints or single algorithms, we find that their localization properties are much more precise.

To rule out more constraints for some methods, we need to refine the used arguments and we will do so with arguments based on properties of constraints. The general algorithm argument, that all parts of the document not located in the subtree of `/uni/students/student` are static, is both sufficient and necessary to support the next considerations.

A very common property of constraints is that they focus on a small subtree of a student only. The difference to constraints already ruled out is that these constraints are located in the student, so there are methods which will be necessary to be checked against them.

The constraints are characterized by having their context prefixed with `/uni/students/student`, while all parts of the document referenced within their assertions are either located in the subtree of their context or in static data. So whenever the context of a rule is not a prefix of the context of an operation, the operation does not violate the rule.

When talking about the prefix of an XPath expression, used as context of a rule for example, we talk about the normalized prefixed, which results from the combination of the actual expression together with the structural RELAX NG specification. The path `//student` will always expand to `/uni/students/student` in our example system, which is either one `student` element or a sequence of `student` elements.

Constraints which have the property described above are local constraints and operate on their context tree only:

- **I3** states that module references have to reference an existing `module` in the `handbook`. The rule is special as module references can appear anywhere in the document, but for those of interest, situated in the `student`, the rule is localized to its context tree. All incarnations of the rule matching outside of the `student` operate on static data and can be discarded anyways.

- **I6-3** operates on the `student/mandatory` subtree only, saying that all contained examinations have indeed to be mandatory. To check this, it is of course necessary to reference information contained in the module handbook, which is not situated in the context tree of the rule, but is considered as static.
- **I6-4 to I6-9** are similar constraints compared to I6-3, each stating that their respective contexts only contain examinations which are allowed in the category associated with it.
- **I6-10** guarantees that the `@ects` attribute of `student/mandatory` is indeed the sum of `@ects` of the contained examinations, so again all references used in the constraint are located in the context tree or the module handbook.
- **D1-2** is a local property of the `@ects` value of certificates for minor subject modules.
- **D8-1 to D8-8** are similar constraints compared to I6-10, each guaranteeing that the `@ects` attribute of the respective category is the correct sum of `@ects` from all examinations.
- **D9** states that all examinations in each `extension` or `focus` block belong to the same area.
- **D19** relates modules of an examination to the `@status` of the examination.

Many rules in this category of constraints can be eliminated on a method-by-method basis, such that only few constraints have to be verified for each method. But there are still constraints left, which we need to take care of.

There are constraints like I4-2, which could be characterized as being localized to their context tree, but in this case the complete `student` is the context, so we gain nothing by doing so. The constraint, however, is still very localized, as it only needs the `@started` attribute of the student and does not even descend to any of its child elements.

This leaves us with a last group of constraints, which utilize the full potential of the rule-based approach, to connect elements which are not already closely related by the structure. The example system was designed in such a way that these constraints only need very specific single elements, such that the general argument of Section 4.3.2 can easily be applied.

Together with the observation that all data outside of the subtree `/uni/students/student` is static for most methods, many remaining constraints are even left with only one single element referenced in the `student`. When verifying these last constraints for each method, we can therefore use the following arguments:

- The context of the rule does not yet exist in the `student`, i.e. all incarnations of the rule operate on static data and can be discarded.
- To rule is based on references to several single elements, which are all not manipulated. Manipulations of children of these elements or parents are both not invalidating the constraints.

- If neither of the two above is true for the method at hand, the rule has to be checked.

We will now go through the constraints left and look which argument might be applicable for a method:

- **I4-2** has only to be checked when either the `student` element itself is changed or the `uni` element.
- **I4-3** only when an `exam`, `student` or `uni` element is changed.
- **I5-1** only when the `student` element is changed.
- **I6-1 and I6-2** only when the `student/extension` or `student/focus` elements are changed.
- **I6-11** only applies to students which failed their thesis, but even if that's the case it only needs to be checked when the `student` or the `thesis` element of the `student` are manipulated.
- **I6-12** combines the `foundations` and `minor` elements of a `student`, as these are the categories which do not have a fixed `@ects` number given and the student has a certain freedom of choice.
- **D6** only applies to students which succeeded with their studies and then only needs to be checked if the `student` element, or an element containing an `@ects` attribute is changed.
- **D7** only applies to students having a thesis and needs to be checked whenever the `thesis` element is changed or a change is made in the *subtree* of the `mandatory` element.
- **D10** is essentially the same as I6-1 and I6-2, with regard to when it needs to be checked.
- **D11-1** uses references to `exam` elements within the `focus` element only and therefore also only applies to students with a `focus` element.
- **D11-2 and D11-3** both need to be checked only when the `student` element or an `exam` element of the student change. The former also only applies to succeeded students and the latter only to failed examinations.
- **D12** is the regulation of progress and therefore only applies to students which did not succeed or fail yet. For all other students it only needs to be checked whenever the `student` element or any element with an `@ects` value are modified.
- **D13** applies to students having a `thesis` element only and needs to be checked whenever the `thesis` element or any element having an `@ects` attribute are modified.
- **D14-2 and D15** need to be checked whenever an `exam` element is changed.

The argument of constraint D7 is actually more difficult than stated above, as it does not only name single elements referenced, but also a complete subtree. So to check if the constraint can be discarded for a method, we need to check for specific single elements, but also do a prefix check, i.e. a combination of techniques used so far.

Using specialized arguments instead of the single general arguments in all cases does not allow us to prove more combinations of constraints and methods, but only simplifies the proofs we can already do with the single argument alone.

4.3.4. An Example: passThesis

The method allows to mark a students `thesis` as "passed" and grants the correct amount of ECTS credits. Being a very specific task, the method should not be able to invalidate many constraints at all, besides those which are reasonable to look at in this context.

All constraints listed in Section 4.3.1 can be discarded upfront, using the generic argument explained there. So we have to look at the remaining constraints listed in Section 4.3.3, using situational arguments. To do this we need to know what parts of the document can be manipulated at all.

The algorithm specification of Listing 3.2 tells us that only the students `thesis` element can be manipulated, so using this information we can look at the context tree constraints:

- **I3, I6-3 to I6-10, D1-2, D8-1 to D8-7, D9 and D19** do not have the path to a `thesis` as prefix, so they do not match.
- **D8-8** has the path to the `thesis` element as prefix, so it matches and has to be checked. This is the result we like to have, as this constraint states that the `@ects` value of the `thesis` element is correct.

For the remaining constraints we get:

- **I4-2, I4-3, I5-1, I6-1 and I6-2, I6-12, D10, D11-X, D14-2 and D15** do not match.
- **I6-11, D6, D7, D12 and D13** match because either the `thesis` element or `@ects` attributes in general are referenced. So they are all valid concerns whenever we change the `thesis` element and especially the `@ects` attribute.

In the case of the passThesis method we therefore have only six constraints left to check at all, using only flat arguments. We could as well check the general structural argument for those remaining, but we neglect to do so here and look at another method instead.

4.3.5. A Worse Case: failExam

If a student fails an examination, he also fails with his studies. The method both manipulates an `exam` anywhere in the `student` and the `@status` of the students top element, i.e. the `student` element itself.

Listing 4.3: Method: failExam

```
failExam(string sid, string mid) {
  assert //student[@id = $sid];
  var student = //student[@id = $sid];

  assert $student//exam[@status = 'enrolled' and
                        moduleref[@id = $mid]];
  var exam = $student//exam[moduleref/@id = $mid];

  if (not($student/@status)) {
    update ($exam/@status) 'failed';
    append ($student) status='failed';
  } else if ($student/@status != 'failed') {
    update ($exam/@status) 'failed';
    update ($student/@status) 'failed';
  }
}
```

For this method we will have a lot of matches using the flat arguments and need to use the general structural argument more often. This is also caused by the fact, that we neglected attribute elements completely for flat arguments and counted them to their parent element only.

The failExam method only manipulates the `@status` attribute of one `exam` element and the `@status` attribute of the `student`. So considering the flat arguments, the method manipulates the `student` element and the `exam` element, though both operations do not invalidate many constraints using them:

- **I3** is located at `moduleref` subtrees, which can be child of `exam` elements, but the algorithm does not manipulate children of the `exam` element, so the rule does not match and can be discarded.
- **I6-3 to I6-9** have to be considered for this method, as the structural specification tells us that an `exam` element can be contained in any of the contexts. If we use the generic structural argument, however, we can discard them altogether, as the method only manipulates the `@status` attribute of the `exam`, which is not referenced by the constraints.
- **I6-10, D8-1 to D8-5 and D8-7** cannot be discarded and have to be verified.
- **D8-6, D8-8 and D1-2** cannot have `exam` elements in their context trees, so they can be disregarded.
- **D9** is similar to I6-3 in that it cannot be discarded by flat arguments, but by the general argument, as the constraint does not reference the `@status` attribute of the `exam`.
- **D19** has to be verified.

For the remaining constraints we get:

- **I4-2, I4-3, I5-1, D12, D14** match, but can nevertheless be disregarded by the general argument, as neither the `@status` of the `student`, nor the `@status` of the `exam` are referenced.

- **I6-1, I6-2, I6-12, D10 and D13** do not match.
- **I6-11, D6, D7, D11-X and D14-2** have to be verified.

We are left with much more constraints, which could not be discarded, but considering the magnitude of the consequences of the operation this was expected. We also had to use the general argument much more often, which was expected by the fact, that the manipulations were not localized to a specific part of the `student` only, but could be in any part containing `exam` elements.

4.4. IX and DX Constraints Verified by Proof

Whenever a pair of a method and a rule cannot be discarded statically by any structural argument, we need to show that the rule holds for the method, using a logical argument. One of the major problems of the structural arguments is their ignorance of the present state of the document.

Especially the category of constraints, which only have to be checked whenever the student has a certain element, or an element with a certain attribute value, cannot be discarded by structural arguments only. Algorithms often `assert` that the student has a special state, e.g. that he did not yet succeed or fail with his studies. Such an assertion can immediately be used to discard constraints like D6 or D11-2, whenever the algorithm does not manipulate the `@status` of the `student`.

Algorithms are formulated in such a way that:

1. They first gather data on properties of the parameters and the state of the document. This is done by giving assertions which have to hold for the method to proceed, whereas the method aborts with a message when they don't hold.
2. The method then tries to figure out what exactly it has to do, by testing conditions and branching control flow on them.
3. Only at the the end of execution the operations are done and the document is actually manipulated, if at all.

We thereby accumulate usable constraints for proofs of the operations, which are dependent on the state of the document and are not general invariants of it. These three sources of constraints

- global invariant
- assertions and
- conditions

have to be sufficient to prove the operation block correct. Operations can invalidate some of the available constraints, which can be checked by the general structural argument, but also provide new constraints.

Knowledge gained in any form can be used in many different kinds of techniques:

invalid context The knowledge about the document shows that the context of a constraint is not present, i.e. the constraint can be discarded. This is a very simple argument and closely related to the structural arguments. The big difference to structural arguments is that by gaining the knowledge to discard the constraint, we reduced the number of documents the method will actually work on. This is no shortcoming of the method, but is a part of the application domains logic. A method adding an examination for a non-existent student, for example, can't just create a student to circumvent that fact.

uniqueness Many checks are done to assert that a certain expression returns exactly one value, such that it can be used in a variable. If there is a uniqueness constraint in the integrity constraints, the check can furthermore be loosened to an existence check.

explicit check Some assertions are explicit checks of constraints, so we know them to hold at the end of the method. Those checks, however, have to be done before the document is even manipulated, so they can't copy the check used in the constraint itself. All references to elements and attributes, which are not yet appended, or which will be changed by an operation, have to be replaced with the value that will be present after the operation finishes.

partial checks The are many constraints which are formulated using quantifiers, sums, or other constructs which can be incrementally proven correct. For these constraints it is not necessary to check the complete constraint, but it is sufficient to check the part added by operations. If adding an additional value to a sum, for example, we have to show that the new sum will still have the property stated in the constraint, using the fact that the former sum did not invalidate it. For quantifiers we can likewise state an additional property to prove the constraint together correct, using the fact that it was already fulfilled with the unmodified document. A uniqueness constraint, for example, can incrementally be proven correct by checking that each added value did not exist so far.

4.4.1. Proving Constraints Correct: passThesis

In Section 4.3.4 we excluded the majority of constraints, leaving only six to prove. We will now go through these constraints one by one and see why they hold after the execution of the method.

D8-8 The constraints context is the `thesis` and it states that the `@ects` attribute of the `thesis` has to be 12 whenever the `@status` is "done" and 0 otherwise. As the algorithm contains only two `update` operations, which set the `@status` of `thesis` to "done" and the `@ects` value to 12, the constraint obviously holds.

I6-11 The constraint matches only whenever the `@status` of the `thesis` is "failed". The algorithm, however, sets the status of the thesis to "done", so the constraint does not match the `thesis`.

D6 The constraint applies only to students who succeeded with their studies and asserts that they have exactly 180 ECTS credits. The algorithm does *not* state that it only operates on students who not yet succeeded or failed, though this would be the easiest solution to discard the whole constraint.

What the algorithm checks is that a `thesis` exists and that the `@status` of it is "topic", so the constraint D8-8, proven correct above, tells us, that the `@ects` value of the `thesis` was 0 so far. Common sense tells us, that a student cannot have succeeded without having done the thesis, but we have to prove it using the constraints we have. At this point it might be the simplest solution to just add an assertion to the algorithm to trivialize the prove, but we can prove it nevertheless.

The structural specification tells us, that there are 11 sources of `@ects` for each `student`, which are summed up in the constraint. Each source has an upper bound defined in constraints which we know to hold:

- The most obvious source is the `thesis` and we know the `@ects` value to be 0.
- The `mandatory` block can host only examinations to modules of mandatory blocks of the module handbook (I6-3). Also each module can only have one examination in the context of one `student` (D15). The `@ects` value of the mandatory block is the sum of all passed examinations (I6-10) and as the modules of mandatory blocks in the module handbook cannot total more than 100 ECTS credits (D16), we have an upper bound for the block.
- Each `extension` block cannot have more than 8 ECTS (D8-1) and we can have up to three of those.
- The `focus/core` block cannot have more than 8 ECTS (D8-2).
- The `focus/advanced` block cannot have more than 8 ECTS (D8-3).
- The `focus/project` block cannot have more than 8 ECTS (D8-4).
- The `foundations` block cannot have more than 12 ECTS (D8-5).
- The `focus/minor` block cannot have more than 20 ECTS (D8-6).
- The `seminar` block cannot have more than 4 ECTS (D8-7).
- The `focus` block can only be present to replace an `extension` (I6-2).

So there are 10 possible sources of `@ects` totaling 176, i.e. constraint D6 cannot be fulfilled whenever the thesis is not done and the student therefore cannot be done with his studies. As the method does not change the `student` elements `@status` attribute, D6 still does not match.

As a side note, adding the 12 ECTS of the thesis would possibly total more than 180 ECTS, which is prevented by I6-12.

D7 This constraint does not contain any references to the `thesis` in its assertions, but only activates when a `thesis` is present. The algorithm asserts that all mandatory modules are done for the student, so the constraint still holds for the student and is not manipulated by the algorithm.

D12 The constraint consists of four assertions, which formulate the four steps of progress a student has to show. Each constraint is therefore guarded by a condition stating at which `@semester` count the constraint activates for the student. As the algorithm does neither manipulate the `@started` attribute of `student`, nor the `@semester` attribute of `uni`, the same criteria are relevant for the student after the method finishes.

- After two semesters only the `@ects` of the `mandatory` element are relevant, which are not manipulated by the algorithm.
- After four semesters we also have a summation criterion, which is relevant. The sum of all `@ects` of the `student` has to be bigger than a certain value. As we have already shown, the only manipulated `@ects` attribute went up to 12 from 0, so the sum was increased by 12, which cannot invalidate the assertion.
- After six semesters there is another summation criterion, but the student also needs to have done all mandatory modules. The algorithm does not change the sum of `mandatory` `@ects`, so this will still hold.
- After nine semesters the student needs to have succeeded. Again it would be easiest to assert that the student did not yet succeed or fail for this algorithm, but we have already shown for constraint D6, that the students status cannot be "succeeded", i.e. the student cannot have studied nine semesters yet and the assertion holds.

D13 The constraint states that any `student` with a `thesis`, where it is not important in which state, has to have enough ECTS credits done. As the algorithm asserts that the student already has a `thesis`, the constraint did hold at the start of the method. Again we only added to the total sum of `@ects`, so the new sum is still greater than needed for the assertion.

4.4.2. Discarding Constraints by Simple Assertions: failExam

The failExam method had much more constraints left to verify, though the method does not have more assertions or conditions. So it is most likely that many constraints hold in any case or are easy to verify:

I6-10, D7, D8-1 to D8-5, D8-7, D11-1 and D14-2 These constraints work with "passed" examinations, as they are the only ones yielding ECTS credits for the student. The failExam method, however, asserts that `exam` elements have to be "enrolled" and changes the status to "failed". All constraints referencing the `@status` attribute only to check for "passed" can be disregarded by logical argument, as the `@status` change does not change the result of those constraints.

D19 The constraint can only be invalidated when the `@status` of the `exam` is "enrolled". As the algorithm does not set the `@status` to "enrolled", the constraint can't be invalidated by it.

I6-11 The constraint only matches students with a failed thesis, but we don't know if the student failed it. The assertion, however, only states that the `@status` of the `student` has to be "failed", whenever he failed his thesis. As the algorithm sets the `@status` of the `student` to "failed", the constraint will hold, no matter if it matches the student or not.

D6 and D11-2 These constraints match only students who succeeded with their studies. As the method sets the `student @status` to "failed", the constraints will not match after the method finished.

D11-3 The constraint states that a student who failed an examination also failed with his studies. As the algorithm sets the `@status` of the `student` to "failed", all constraints on failed examinations will hold.

4.4.3. Method Dependencies: balanceFS

Method calls are a powerful language feature in general, but they are not designed to be that powerful in the example system. We do not allow recursive calls and only use calls for convenience reasons or to preserve the fact that each method finishes after the first operation block.

Listing 4.4: Method: balanceFS

```
balanceFS(string sid) { // I6-12 need not hold
  assert //student[@id = $sid];
  var student = //student[@id = $sid];

  if ($student/focus) {
    if ($student/focus/minor/@ects +
        $student/foundations/@ects > 180 - 56 -
        sum(//handbook/block[@mandatory]/module/@ects) {
      update ($student/focus/minor/@ects) 180 - 56 -
        sum(//handbook/block[@mandatory]/module/@ects) -
        $student/foundations/@ects
    }
  }
}
```

In the case of balanceFS, we could as well inline the method to all places it is called from. The method is designed to work on documents, for which the constraint I6-12 does not hold, but guarantees the global invariant after it finishes, like all methods do.

The method can be called by the user of the system, though the method will not do anything in that case: The only operation of the method is guarded by a condition, which essentially states that I6-12 does not hold. As the global invariant always holds for the user, whenever he only uses verified methods, balanceFS is useless to him.

The method is called from other methods, whenever the `@ects` attribute of the `foundations` block or the `minor` block is manipulated. We will not be able to verify I6-12 for these methods alone, but need to include the method call to balanceFS at the end of their respective operation blocks.

4.4.4. Method Dependencies: checkProgress

Instead of working on documents which lack one or more constraints, the checkProgress (Listing 4.1) method guarantees additional constraints upon termination. The method is therefore not used at the end of an operation block, but at the beginning, so we can use the additional constraint to verify.

The method is used in endOfSemester, which increments the `@semester` attribute of the `uni` element only. The major concern for this operation is the regulation of progress for each student. Invoking checkProgress guarantees that the regulation of progress (D12) will hold for the incremented `@semester` attribute.

Listing 4.5: Method: endOfSemester

```
endOfSemester() {
  checkProgress();
  update (/uni/@semester) $current + 1;
}
```

With the current syntax of the algorithm language, it would not be possible to inline the checkProgress method into endOfSemester. This is intended to preserve the property that no more than one operation block in a method can be executed in an invocation.

Dependencies of operation blocks thereby have to be stated as properties of methods, which can then be used to verify operation blocks.

4.4.5. For Loops: checkProgress

In Section 4.2.2 we already covered the structural implications of having a for loop as control structure. Essentially the argument used there included the fact that operations were independent and each iteration of the loop left the structure intact.

The example system exclusively uses for loops to formulate operations, which could be done in any order or even in parallel. The operations used are not only structurally, but also logically independent. For the method at hand it will be sufficient to show, that each iteration of the loop does not invalidate any constraint, thereby guaranteeing that the complete execution has the same property.

The method additionally guarantees a constraint, which is used by endOfSemester. The constraint is a version of D12, with all references to the `@semester` attribute of `uni` replaced by the value of `@semester` incremented by one. The context of rule D12 includes all students which don't have a `@status` yet.

The for loop is therefore also necessary to guarantee the modified version of D12 for each `student` separately, as it visits all student without a `@status`. For students with a status the modified version of D12 holds in any case.

Considering the normal global invariant, we can leave out many constraints by structural considerations, as the only operations are those manipulating the `@status` attribute of `student`. We can easily show those which are left:

D6 The constraint matches only succeeded students and as the algorithm does not manipulate students with a status, this can only match after the last case in the for loop. This case could only be selected whenever either `$diff != 9` or `sum($student//`

`@ects) = 180` and `not($student//exam[@status != 'passed'])` and as the guard of the last case itself is `$diff = 9`, we know the condition of D6 to hold.

I6-11 Whenever the student failed his thesis, he also failed with his studies, so his `@status` has to be "failed". The algorithm either sets the status to "failed" anyways, or he sets it to "succeeded", but for constraint D6 we have already shown that he will have 180 ECTS credits in that case. In Section 4.4.1 we have shown that a student can't reach 180 ECTS credits without the 12 credits of the thesis. D8-8 tells us that he cannot get the credits of his thesis, without its status being "done".

D11-2 The constraint matches only students which have succeeded with their studies, so again this can only happen in the last case of the for loops body. As pointed out in the proof of D6, the the inverse right hand side of the preceding guard has to be true, so the student does not have any examinations with a `@status` different to "passed" and the constraint holds.

D11-3 Students manipulated by the algorithm do not have a `@status` yet, so by D11-3 they have no failed examinations. As the algorithm does not change the `@status` of any `exam`, the constraint does not match for manipulated students.

D12 The constraint holds for all students at the start of the method. The method can only add a `@status` to students, so the constraint does not match these students anymore.

4.5. Complete Example: Case 6 of addExam

As mentioned earlier, the addExam method is a combination of several methods, which share the same idea and need about the same assertions and variables. Because of the layout of algorithms, however, it is possible to show all cases of executed operation blocks correct separately. The advantage of this procedure is that more constraints can be ruled out by structural arguments. Case 6 looks as follows:

```
if (name($entry/..) = 'block') { ... }
else {
  if ($mtype = 'core') { ... }
  else if ($mtype = 'advanced') {
    if ($student/focus[@name = $fbname]) {
      // case 6:
      append ($student/focus[@name = $fbname]/advanced) #exam;
    }
    else if ...
  }
  else if ...
  else assert false;
}
```

To prove the case correct we need to follow the steps described in Section 4.1, starting with the general correctness of XPath expressions, such that no dynamic or

type errors can occur, etc. A good approach seems to follow the method execution and go through the relevant points whenever they occur. When encountering conditions we fix the outcome to evaluate to case 6, as this is the target of the verification.

```
assert //student[@id = $sid];
```

The method starts with an assertion, which tests if a student of given `id` exists. The expression cannot yield any errors, as attributes can always be cast to string and compared with a string, which is the type of the parameter used. The `@id` attribute always has to be present for `student` elements. If the assertion holds we know that there is exactly one `student` having the `@id` given as parameter, using I5-1.

```
var student = //student[@id = $sid];
var semester = (if ($sem >= 0) then $sem else /uni/@semester)
               cast as xs:integer;
```

The first variable definition uses the same expression, so we know it to evaluate to exactly one element. The `$semester` variable evaluates either to the integer parameter or to the always present integer attribute of the uni element. In both cases the variable holds a value greater or equal to zero, as this is guaranteed by the condition of the `if` statement and I4-1. Both variables are not manipulated by the only operation of case 6, which appends to an `advanced` element.

```
assert not($student/@status);
```

The following assertion states that the student must not have a `@status` attribute, which is marked as optional in the structural specification. The expression itself again can't yield any errors, as no operations are done and no special types are needed.

```
assert $semester <= /uni/@semester and
       $semester >= $student/@started;
```

The next assertion is very similar to constraint I4-3, which states that the `@semester` attribute of an examination has to be within certain bounds. The expression compares two integer attributes which are always present with a variable shown to be integer.

```
assert //handbook//module[@id = $mid];
```

The method proceeds with an assertion which tests the `handbook` for a `module` having the `@id` given as parameter. The attribute exists for all modules in the handbook and has the type string, which is also the type of the parameter used.

```
var entry = //handbook//module[@id = $mid];
var fbname = $entry/../@name;
```

`$entry` is defined with the exact same expression, which evaluates to exactly one module element, using I5-2. `$fbname` uses `$entry` and navigates one step upwards, which is guaranteed to be either a `block` or an `area` element. Both always have the `@name` attribute, so the variable is correctly defined. Both variables are again not modified by case 6.

```
assert not($student//exam/moduleref[@id = $mid]);
```

The next assertion is a part of constraint D15, stating that the student can only have one examination for each module. It tests that the student does not have an examination for the given module id. The `@id` attribute of a `module` is always present and shares the same base type with the parameter used in the comparison.

```
assert every $dependmod in //handbook//module[@id = $mid]
                            /depends/moduleref satisfies
         some $exammod in .//exam[@status = 'passed' and
                                  @semester < $semester]
                           /moduleref satisfies
            $dependmod/@id = $exammod/@id;

assert every $sum in //handbook//module[@id = $mid]
                      /depends/sum satisfies
         sum(//handbook
              //module[some $module in $sum/moduleref satisfies
                         @id = $module/@id]
                     [some $module in
                        //student[@id = $sid]
                        //exam[@status = 'passed' and
                               @semester < $semester]
                         /moduleref satisfies @id = $module/@id]
             /@ects) >= $sum/@value;
```

The following two assertions are similar to those of rule D14-2, with the exception that all references to the context examination of the rule are substituted with defined variables of the method, sharing the same type. As far as the involved XPath expressions are concerned, they use many comparisons of values which are guaranteed to exist and share the same base type. The used quantifiers correctly use sequences of elements and the used `sum` function indeed gets a sequence of integer attributes.

```
var mtype = $entry/@type;
```

`$mtype` selects the `@type` attribute of the `module` element stored in `$entry`, which is always present. The variable definition is again not invalidated by the operation of case 6.

```
assert $mtype != 'advanced' or
       sum(//handbook/area[@name = $fbname]/module[@type = 'core']
         [some $module
          in //student[@id = $sid]
             //exam[@status = 'passed']/moduleref
          satisfies $module/@id = @id]/@ects) >= 8;
```

The assertion resembles D11-1, with all references to the context replaced by variables. The expression does not show any new kind of construct or potential error source and is correct.

```
var passed = ($mtype = 'seminar' or $mtype = 'project');
```

`$passed` evaluates to exactly one boolean value, which is defined by comparing a variable shown to be string with to string literals. As the used variable wasn't manipulated by the operation of case 6 and the variable definition does not use other parts of the document, the variable is also not manipulated.

```
node exam =
  <exam semester=$semester
        status=(if ($passed) then 'passed' else 'enrolled')>
    <moduleref id=$mid/>
  </exam>;
```

```
node focus =
  <focus name=$fbname>
    <core ects="0"/>
    <advanced ects="0"/>
    <project ects="0"/>
    <minor ects="0"/>
  </focus>;
```

The following `node` definitions use only correct expressions for attribute values and cannot be analysed any further, as they are not yet related to any structural specification.

```
if (name($entry/..) = 'block') { ... }
else {
```

The method closes with an `if` statement, holding several different cases and conditions. The first condition uses the `name` function, which takes a node as parameter. The parent of `$entry`, holding a `module` element, was already shown to always be either a `block` or an `area` element. We assume the condition to be false, i.e. the name of the parent element is not "block", so it has to be "area" and we continue with the `else` case.

```
  if ($mtype = 'core') { ... }
  else if ($mtype = 'advanced') {
```

The next cascade of conditions compares the `$mtype` variable with string literals, so we assume `$mtype` to contain the string "advanced", i.e. the `@type` attribute of the `module` in the `handbook` contains this value.

```
    if ($student/focus[@name = $fbname]) {
      // case 6:
```

Finally we assume the condition to hold, which states that the `student` indeed has a `focus` element as child, which additionally has a `@name` attribute containing the content of `$fbname`. As `$fbname` contains the `@name` of the `area` the `module` is in, the `focus` element has the same `@name` as attribute. All conditions are simple XPath expressions, which can't produce any errors on execution.

```
append ($student/focus[@name = $fbname]/advanced) #exam;
```

We finally arrived at the operation block of case 6, which hold only one `append` operation. The context of the operation is based on the `$student` variable, which is guaranteed to hold exactly one `student`, which can only be situated at the `students` element of `uni`. The last condition has shown that the `student` has the optional `focus` element with the correct `@name` attribute, which is also unique for each `student`. The `focus` element always contains exactly one `advanced` element, so the context is defined and consists of exactly one element.

To verify that the structure is not violated, we need to look at the element appended and compare it to the structural specification of the context. The algorithm copies the `exam` node, which is an `exam` element. There can be arbitrary many `exam` elements in the context, so this is not a problem.

The `@semester` attribute is taken from the variable `$semester`, which was shown to be integer. The `@status` attribute is either the string literal "passed" or "enrolled",

which are both valid. The `exam` element also contains the necessary `moduleref` element, which contains the attribute `@id`. The attribute is set to the content of the `$mid` parameter, which is only guaranteed to be string. As one assertion stated that there is at least one `module` in the `handbook`, whose `@id` attribute is equal to the parameter, the parameter has to be valid with regard to the pattern of the `@id` attribute of `module`. As the pattern is the same as the one for `moduleref`, the assignment is correct.

At this point we are done with structural considerations and the basic logical aspects. We now have to find out which constraints are not ruled out by structural arguments and have to verify them:

I3 The `@id` attribute of the `moduleref` element has to exist as `@id` attribute of a `module` in the `handbook`. This is guaranteed, as the value of the `@id` attribute was taken from the `$mid` parameter, which was used in an assertion stating that there exists such a `module`.

I4-3 The constraint was explicitly checked in an assertion. The assertion in the algorithm therefore substituted all references to the context with variables, which were then used in the creation of the `exam` element.

I6-7 The constraint states that each module reference of an examination in `focus/advanced` needs to have a `module` in the `handbook`, whose `@id` is that of the reference and whose `@type` is "advanced". A condition which led us to case 6 states that the variable `$mtype` contains the string "advanced" and as `$mytpe` is set to the `@type` attribute of a `module` of the handbook, which has the `@id` given as parameter, the constraint also holds for the added `exam` element.

D8-3 The constraint states that the `@ects` attribute of `focus/advanced` has to be the sum of `@ects` of modules which have passed examinations. Adding an `exam` element, however, did not increase the sum, as only `exam` elements with the `@status` "passed" are counted in the sum. The algorithm added "enrolled" as `@status` of the `exam` element, as the variable `$mtype` has the value "advanced" and neither "seminar" nor "project".

D9 Each `module` referenced by an examination in an `extension` of `focus` block needs to belong to the area of the block. The area of the module added was stored in `$fbname` and proven by a condition to be the same as the `@name` of the `focus` block it was added to.

D11-1 The constraint was explicitly checked in an assertion, with all references to the context replaced by variables, which where used in the construction of the `exam` element.

D11-2 The constraint does not match, as an assertion explicitly states that the `student` had no `@status`, so he also hasn't succeeded yet.

D11-3 The constraint does not match the appended `exam` element, as its `@status` cannot be "failed".

D14-2 The constraint was again explicitly checked by two assertions, using the same technique as before.

D15 The `student` mustn't have two examinations to the same module. Considering that the constraint was valid before, only the added `exam` element can cause conflicts. An assertion stated, however, that there was no examination for the `module` in the `student`, so adding one cannot invalidate the constraint.

D19 An examination of seminar or project modules cannot be "enrolled". As the module `@type` was shown by a condition to be "advanced", the constraint holds for the added `exam` element.

5. Conclusion

Semi-structured data, especially in the form of XML, is now a first class object in many systems of very different clients and programs. The correctness, integrity and consistency of the data has to be ensured throughout the lifetime of data and has to be guaranteed by each involved program separately.

There are some languages suited to express constraints on data but none is suited to cover all different forms of constraints. RELAX NG, as representative for grammar-based approaches, is very well suited to formulate structural constraints, while XPath 2.0 turned out to be the language of choice for integrity and domain-specific constraints. Schematron was the first language to embed XPath for that purpose, but any host language defined to formulate rules on XML documents is likely to be sufficient.

There are only few languages which allow the formulation of conditional local manipulations on XML documents and they are either unfinished or discontinued. To keep proofs and analysis simple for a start, I proposed a small language which again utilizes the power of XPath, which is complex enough to implement the example system, yet preserves several important properties.

Proving algorithms correct with regard to specified constraints is a well researched task for structural constraints but still a difficult task for more complex integrity and domain-specifc constraints. It is still possible to simplify proofs by excluding many constraints for each algorithm, which can't be violated. I judge it likely to be possible to automatically analyse algorithms and constraints to generate such results, whenever there is a structural schema present and the XPath expressions involved adhere to some general guidelines.

In this context it would be interesting for future work to develop a tool suite for XPath expressions on RELAX NG constrained XML documents. The presence of a schema allows to check XPath expressions for meaningfulness and makes it possible to compare different XPath expressions to reveal properties like inclusion or incomparability. Many steps contained in proofs will be able to be retrieved automatically, while it is unlikely or even impossible to automate complete proofs.

Constraints which could not be excluded by general structural argument have to be proven correct individually. This was supported by the layout of algorithms, which concentrates all operations into blocks and allows to utilize the global invariant as well as assertions and conditions already checked. Proofs could be categorized into several most common cases, which could also be supported by static analysis of the involved XPath expressions.

We have also seen a few more complex algorithms, which utilized for loops and a call mechanism, which both needed stronger arguments. Methods which work on documents, which are not completely valid or ensure even more constraints on their termination, play an essential role in formulating algorithms of growing complexity.

Future work might also take an interest in the creation of a full-grown XML manipulation language, which could also include concepts to combine operation blocks to facilitate proofs. Especially operation blocks with strongly related manipulations, which had to be verified together, could be especially formulated in a way which emphasizes their relation.

A. Additional Listings

Listing A.1: Constraints

```
S2 (handbook)
! block[@name = 'Software Development' and @mandatory]
! block[@name = 'Basic Systems' and @mandatory]
! block[@name = 'Theoretical Foundations' and @mandatory]
! block[@name = 'General Foundations' and not(@mandatory)]
! count(block) = 4

I1 (handbook//module)
! count(//handbook//module[@name = $current/@name and
                           @type = $current/@type]) = 1

I2-1 (handbook/block)
! not(module[@type = 'advanced'])
! not(module[@type = 'core'])

I2-2 (handbook/area)
! not(module[@type = 'basic'])

I3 (moduleref)
! //handbook//module[@id = $current/@id]

I4-1 (uni)
! @semester cast as xs:integer >= 0

I4-2 (student)
: $started = @started cast as xs:integer
! $started <= /uni/@semester and $started >= 0

I4-3 (student//exam)
: $semester = @semester cast as xs:integer
! $semester <= /uni/@semester and $semester >= ancestor::student/@started

I5-1 (student)
! count(//student[@id = $current/@id]) = 1

I5-2 (handbook//module)
! count(//module[@id = $current/@id]) = 1

I6-1 (student/extension | student/focus)
! //handbook/area[@name = $current/@name]

I6-2 (student)
! not(count(extension) > 2 and focus)

I6-3 (student/mandatory)
! every $module in exam/moduleref satisfies
    some $entry in //handbook/block[@mandatory]/module satisfies
      $module/@id = $entry/@id

I6-4 (student/foundations)
! every $module in exam/moduleref satisfies
    some $entry in //handbook/block[@name = 'General Foundations']/module
      satisfies $module/@id = $entry/@id
```

```
I6-5 (student/extension)
! every $module in exam/moduleref
    satisfies //handbook//module[@id = $module/@id]/@type = 'core'

I6-6 (student/focus/core)
! every $module in exam/moduleref satisfies
    //handbook//module[@id = $module/@id]/@type = 'core'

I6-7 (student/focus/advanced)
! every $module in exam/moduleref satisfies
    //handbook//module[@id = $module/@id]/@type = 'advanced'

I6-8 (student/focus/project)
! every $module in exam/moduleref satisfies
    //handbook//module[@id = $module/@id]/@type = 'project'

I6-9 (student/studies/seminars)
! every $module in exam/moduleref satisfies
    //handbook//module[@id = $module/@id]/@type = 'seminar'

I6-10 (student/mandatory)
! sum(//handbook
      //module[some $id in $current/exam[@status = 'passed']
                                    /moduleref/@id satisfies @id = $id]
       /@ects) = @ects

I6-11 (student//thesis[@status = 'failed'])
! ancestor::student/@status = 'failed'

I6-12 (student)
: $minor
= if (.//minor) then .//minor/@ects cast as xs:integer else 0
: $foundation
= if (.//foundations) then .//foundations/@ects cast as xs:integer else 0
: $bound
= 180 - 56 - sum(//handbook/block[@mandatory]/module/@ects)
! $minor + $foundation <= $bound
! $minor <= $bound - 8
! $foundation <= $bound - 16

D1-1 (handbook//module)
! @ects >= 2 and @ects <= 14

D1-2 (student//minor/certificate)
! @moduleects >= 2 and @moduleects <= 14

D2-1 (handbook/block[@name = 'Software Development' or @name = 'Basic Systems']
: $sum = sum(module/@ects)
! $sum <= 32 and $sum >= 24

D2-2 (handbook/block[@name = 'Theoretical Foundations'])
: $sum = sum(module/@ects)
! $sum <= 40 and $sum >= 35

D2-3 (handbook/block[@name = 'General Foundations'])
: $sum = sum(module/@ects)
! $sum >= 8

D3 (area)
! count(//handbook/area[@name = $current/@name]) = 1

D4 (handbook/area)
! sum(module[@type = 'core']/@ects) >= 8
! sum(module[@type = 'advanced']/@ects) >= 8
```

```
! sum(module[@type = 'project']/@ects) >= 8
! module[@type = 'seminar' and @ects = 4]

D5 (handbook)
! count(area) >= 3

D6 (student[@status = 'succeeded'])
! sum(.//@ects) = 180

D7 (student//thesis)
! every $module in //handbook/block[@mandatory]/module satisfies
    ancestor::student/mandatory/exam[@status = 'passed']
                                /moduleref[@id = $module/@id]

D8-1 (student/extension)
: $sum = sum(//handbook
             //module[some $exam in $current/exam[@status = 'passed']
                        satisfies $exam/moduleref/@id = @id]
              /@ects)
! @ects = min(($sum, 8))

D8-2 (student/focus/core)
: $sum = sum(//handbook
             //module[@type = 'core']
                     [some $exam in $current/exam[@status = 'passed']
                        satisfies $exam/moduleref/@id = @id]
              /@ects)
! @ects = min(($sum, 8))

D8-3 (student/focus/advanced)
: $sum = sum(//handbook
             //module[@type = 'advanced']
                     [some $exam in $current/exam[@status = 'passed']
                        satisfies $exam/moduleref/@id = @id]
              /@ects
! @ects = min(($sum, 8))

D8-4 (student/focus/project)
: $sum = sum(//handbook
             //module[@type = 'project']
                     [some $exam in $current/exam[@status = 'passed']
                        satisfies $exam/moduleref/@id = @id]
              /@ects
! @ects = min(($sum, 8))

D8-5 (student/foundations)
: $sum
= sum(//handbook
      //module[some $module in $current/exam[@status = 'passed']/moduleref
                  satisfies $module/@id = @id]
       /@ects
! @ects <= min(($sum, 12))

D8-6 (student/focus/minor)
: $sum = sum(certificate/@moduleects)
! @ects <= min(($sum, 20))

D8-7 (student/studies/seminars)
: $sum
= sum(//handbook
      //module[some $module in $current/exam[@status = 'passed']/moduleref
                  satisfies $module/@id = @id]
       /@ects
! @ects = min(($sum, 4))
```

```
D8-8 (student/studies/thesis)
! if (@status = 'done') then @ects = 12 else @ects = 0

D9 (student/focus | student/extension)
! every $id in .//exam/moduleref/@id satisfies
    //handbook/area[@name = $current/@name]/module[@id = $id]

D10 (student)
! every $name in (focus | extension)/@name satisfies
    count((focus | extension)[@name = $name]) = 1

D11-1 (student/focus//exam)
: $module = //handbook//module[@id = $current/moduleref/@id]
! $module/@type != 'advanced' or
  sum($module/ancestor::area/module[@type = 'core']
    [some $exam in $current/ancestor::focus
                          //exam[@semester <= $current/@semester and
                                 @status = 'passed']
     satisfies $exam/moduleref/@id = @id]/@ects) >= 8

D11-2 (student[@status = 'succeeded'])
! every $exam in $current//exam satisfies $exam/@status = 'passed'

D11-3 (student//exam[@status = 'failed'])
! ancestor::student/@status = 'failed'

D12 (student[not(@status)])
: $semester = /uni/@semester - @started
! $semester < 2 or mandatory/@ects >= 30
! $semester < 4 or mandatory/@ects >= 50 and
    sum(.//@ects) >= 60
! $semester < 6 or mandatory/@ects =
      sum(//handbook/block[@mandatory]/module/@ects) and
    sum(.//@ects) >= 110
! $semester < 9

D13 (student//thesis)
! sum(ancestor::student//@ects) >= 120

D14-1 (handbook//module/depends/sum)
: $sum = sum(//handbook//module[some $module in $current/moduleref
                                satisfies $module/@id = @id]/@ects
! $sum >= @value

D14-2 (student//exam )
: $entry = //handbook//module[@id = $current/moduleref/@id]
! every $depmod in $entry/depends/moduleref satisfies
    ancestor::student//exam[moduleref/@id = $depmod/@id and
                            @status = 'passed' and
                            @semester < $current/@semester]
! every $sum in $entry/depends/sum satisfies
    sum(//handbook
        //module[some $module
                 in $current/ancestor::student
                           //exam[@status = 'passed' and
                                  @semester < $current/@semester]
                            /moduleref
                 satisfies @id = $module/@id]
                [some $module in $sum/moduleref satisfies @id = $module/@id]
         /@ects) >= $sum/@value

D15 (student//exam)
! count(ancestor::student//exam[moduleref/@id = $current/moduleref/@id]) = 1

D16 (//handbook)
```

```
: $mand = sum(//handbook/block[@mandatory]/module/@ects)
: $found = min((12, sum(//handbook/block[@name = 'General Foundations']
                        /module/@ects))
! $mand <= 100
! $mand >= 104 - $found

D19 (student//exam)
! //handbook//module[@id = $current/moduleref/@id]/@type != 'seminar' or
  //handbook//module[@id = $current/moduleref/@id]/@type != 'project' or
  @status != 'enrolled'
```

Listing A.2: Excerpt of a Valid Document

```
<?xml version="1.0" encoding="UTF-8"?>
<uni semester="9">
  <students>

    <!--=============================-->
    <!--=== Successful student ===-->
    <!--=============================-->

    <student id="123456" name="Mustermann" started="0" status="succeeded">
      <mandatory ects="98">
        <exam semester="0" status="passed">
          <moduleref id="89-0001"/>
        </exam>

        ...

        <exam semester="0" status="passed">
          <moduleref id="81-043"/>
        </exam>
        <exam semester="1" status="passed">
          <moduleref id="89-0002"/>
        </exam>

        ...

        <exam semester="4" status="passed">
          <moduleref id="89-0020"/>
        </exam>
        <exam semester="5" status="passed">
          <moduleref id="89-0010"/>
        </exam>
      </mandatory>

      <foundations ects="10">
        <exam semester="1" status="passed">
          <moduleref id="89-0017"/>
        </exam>
        <exam semester="2" status="passed">
          <moduleref id="89-0016"/>
        </exam>
      </foundations>

      <studies>
        <seminars ects="4">
          <exam semester="7" status="passed">
            <moduleref id="89-1211"/>
          </exam>
        </seminars>
        <thesis ects="12" status="done" title="My Thesis"/>
      </studies>

      <extension ects="8" name="Software Engineering">
```

```
            <exam semester="3" status="passed">
               <moduleref id="89-3001"/>
            </exam>
         </extension>

         <focus name="Algorithmics and Deduction">
            <core ects="8">
               <exam semester="5" status="passed">
                  <moduleref id="89-5001"/>
               </exam>
            </core>
            <advanced ects="8">
               <exam semester="5" status="passed">
                  <moduleref id="89-5531"/>
               </exam>
            </advanced>
            <project ects="8">
               <exam semester="4" status="passed">
                  <moduleref id="89-5145"/>
               </exam>
            </project>
            <minor ects="16">
               <certificate moduleects="5" modulename="minor subject I"/>
               <certificate moduleects="7" modulename="minor subject II"/>
               <certificate moduleects="6" modulename="minor subject III"/>
            </minor>
         </focus>

         <extension ects="8" name="Embedded Systems and Robotics">
            <exam semester="6" status="passed">
               <moduleref id="89-6001"/>
            </exam>
            <exam semester="8" status="passed">
               <moduleref id="89-6002"/>
            </exam>
         </extension>
      </student>

      <student id="000000" name="Neumann" started="0" status="failed">
         <mandatory ects="0"/>
         <foundations ects="0"/>
         <studies>
            <seminars ects="0"/>
         </studies>
      </student>
   </students>

   <handbook>

      <!--====================-->
      <!--=== Basic Blocks ===-->
      <!--====================-->

      <block mandatory="" name="Software Development">
         <module ects="10" id="89-0001" name="Software Development 1"
                 type="basic"/>
         <module ects="8" id="89-0002" name="Software Development 2"
                 type="basic"/>
         <module ects="4" id="89-0003" name="Software Development 3"
                 type="basic"/>
         <module ects="8" id="89-0020" name="SW-Development Project (Project)"
                 type="project">
            <depends>
               <sum value="12">
                  <moduleref id="89-0001"/>
```

```
            <moduleref id="89-0002"/>
            <moduleref id="89-0003"/>
         </sum>
      </depends>
   </module>
</block>

...

<block name="General Foundations">
   <module ects="6" id="89-0016" name="Project Management"
           type="basic"/>
   <module ects="4" id="89-0017" name="Working Techniques"
           type="basic"/>
</block>

<!--=======================-->
<!--=== Teaching Areas ===-->
<!--=======================-->

<area name="Software Engineering">
   <module ects="8" id="89-3001"
           name="Foundations of Software Engineering" type="core">
      <depends>
         <moduleref id="89-0001"/>
      </depends>
   </module>
   <module ects="8" id="89-3145"
           name="Foundations of Software Engineering (Project)"
           type="project"/>
   <module ects="4" id="89-3411"
           name="Components, Frameworks and Technologies
                 for Distributed Systems(Seminar)"
           type="seminar"/>
   <module ects="4" id="89-3231"
           name="Advanced Aspects of Object Oriented Programming"
           type="advanced"/>
   <module ects="4" id="89-3131"
           name="Project Management and Quality Assurance"
           type="advanced">
      <depends>
         <moduleref id="89-3001"/>
      </depends>
   </module>
   <module ects="4" id="89-3331"
           name="Safety and Reliability of Embedded Systems"
           type="advanced"/>
   <module ects="4" id="89-3431"
           name="Software Architecture of Distributed Systems"
           type="advanced"/>
</area>

...

<area name="Algorithmics and Deduction">
   <module ects="8" id="89-5001"
           name="Design and Analysis of Algorithms" type="core">
      <depends>
         <moduleref id="89-0004"/>
         <moduleref id="81-043"/>
      </depends>
   </module>
   <module ects="4" id="89-5111" name="Computer Algebra (Seminar)"
           type="seminar"/>
   <module ects="8" id="89-5145" name="Computer Algebra (Project)"
```

```
                type="project"/>
        <module ects="8" id="89-5131"
                name="Computer Algebra" type="advanced"/>
        <module ects="8" id="89-5531" name="Logic and Deduction"
                type="advanced"/>
        <module ects="8" id="89-5532"
                name="Reactive Systems" type="advanced"/>
        <module ects="8" id="89-5331" name="Analytical Complexity Theory"
                type="advanced"/>
    </area>
  </handbook>
</uni>
```

B. Electronic Medium

Attached to this work is a CD-Rom, containing all essential files of the example system as well as the specifications included in this work. The structure of the included files is as follows:

- **./build.xml** An Ant build file to compile and run the example system.
- **./src** The source folder containing the relevant Java sources of the example system, including the *Toolkit*, which implements the checker and interpreter. The folder also contains the XML specifications of constraints and algorithms, as well as an initial document:

  [**schema.rng**] The schema file of the structural constraints.

  [**conditions.rng/.xml**] The schema file of the constraint specification language, as well as the constraint specification file of the example system, containing integrity and domain-specific constraints.

  [**algorithms.rng/.xml**] The schema file of the algorithms specification language, as well as the algorithm specification of the example system.

  [**uni(.updated).xml**] The sample document before and after the execution of the `main` method of the example system.
- **./classes** The compiled example system, ready for execution.
- **./lib** The necessary libraries, containing:

  [**saxonb8-8j**] The open source part of the saxon project [19], which includes XPath 2.0 support.

  [**jing-20030619**] A RELAX NG validator called Jing, written by James Clark [17].
- **./doc** Additional documents, containing:

  [**JavaDoc**] The Javadoc of the `XML` package, with all private members included.

  [**Constraints**] The pretty printed constraint specification.

  [**Method: <Name>**] The pretty printed methods used in this work, also containing the complete version of addExam.

Bibliography

[1] Maria Adriana Abrão, Béatrice Bouchou, Mirian Halfeld Ferrari Alves, Dominique Laurent, and Martin A. Musicante. Incremental constraint checking for XML documents. In Zohra Bellahsene, Tova Milo, Michael Rys, Dan Suciu, and Rainer Unland, editors, *XSym*, volume 3186 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2004.

[2] Denilson Barbosa, Gregory Leighton, and Andrew Smith. Efficient incremental validation of XML documents after composite updates. In Sihem Amer-Yahia, Zohra Bellahsene, Ela Hunt, Rainer Unland, and Jeffrey Xu Yu, editors, *XSym*, volume 4156 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006.

[3] Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. Adding updates to XQuery: Semantics, optimization, and static analysis. In Daniela Florescu and Hamid Pirahesh, editors, *XIME-P*, 2005.

[4] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in C omega. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311. Springer, 2005.

[5] Tim Bray. The annotated XML specification, 1998. `http://www.xml.com/axml/testaxml.htm`.

[6] Alex Brown. Iso/iec 19757 - DSDL - document schema definition languages. `http://dsdl.org/`.

[7] Luca Cardelli. Transitions in programming models, November 2003.

[8] Microsoft Corporation. The LINQ project. `http://msdn.microsoft.com/data/ref/linq/`.

[9] The Apache Software Foundation. Apache Xindice. `http://xml.apache.org/xindice/`.

[10] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G. Burke, Rajesh Bordawekar, Igor Pechtchanski, and Vivek Sarkar. XJ: Facilitating XML processing in Java. In *Proceedings of the 14th International World Wide Web Conference*, pages 278–287, Chiba, Japan, May 2005. ACM Press. `http://www2005.org/cdrom/docs/p278.pdf`.

[11] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn*, 3(2):117–148, 2003. `http://doi.acm.org/10.1145/767193.767195`.

[12] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn*, 3(2):117–148, 2003.

[13] The XML:DB Initiative. XML:DB initiative for XML databases. `http://xmldb-org.sourceforge.net`.

[14] The XML:DB Initiative. XUpdate - XML update language. `http://xmldb-org.sourceforge.net/xupdate/`.

[15] International Organization for Standardization. Information technology — document schema definition languages (DSDL) — part 3: Rule-based validation — schematron. ISO/IEC 19757-3, April 2006.

[16] Marta Henriques Jacinto, Giovani Rubert Librelotto, José Carlos Leite Ramalho, and Pedro Rangel Henriques. Constraint specification languages: Comparing XCSL, schematron and XML-schemas. In *Proceedings of XML Europe 2002*, Barcelona, Spain, May 2002.

[17] Thai Open Source Software Center Ltd James Clark. Jing - a RELAX NG validator in Java. `http://www.thaiopensource.com/relaxng/jing.html`.

[18] Rick Jelliffe. The current state of the art of schema languages for XML, 2001. `http://chinese-school.netfirms.com/articles/RickJelliffe.pdf`.

[19] Michael Kay. SAXON, the XSLT and XQuery processor. `http://saxon.sourceforge.net/`.

[20] LAMP/EPFL. The Scala programming language. `http://scala.epfl.ch/`.

[21] D. Lee and W. W. Chu. Comparative analysis of six XML schema languages. *ACM SIGMOD Record*, 29(3):76–87, September 2000.

[22] Bill Lindsey and James Clark. XT, a fast, free implementation of XSLT in java. `http://www.blnz.com/xt/index.html` and `http://www.jclark.com/xml/xt-old.html`.

[23] David Megginson. Simple API for XML (SAX). `http://www.saxproject.org/`.

[24] Erik Meijer and Brian Beckman. XLinq: XML programming refactored. XML 2005 Conference and Exhibition, November 2005. `http://www.idealliance.org/proceedings/xml05/ship/63/Monoids.PDF`.

[25] Sun Microsystems. Java architecture for XML binding (JAXB). `http://java.sun.com/webservices/jaxb/`.

[26] AT&T Labs Research and University of Aarhus at BRICS. Document structure description (DSD). `http://www.brics.dk/DSD/`.

[27] Microsoft Research. Comega. `http://research.microsoft.com/Comega/`.

[28] Eric van der Vlist. Examplotron. `http://examplotron.org/`.

[29] Eric van der Vlist. *XML Schema*. O'Reilly, June 2002.

[30] Eric van der Vlist. *RELAX NG*. O'Reilly, December 2003.

[31] W3C. Document object model (DOM) level 1 specification, October 1998. `http://www.w3.org/TR/REC-DOM-Level-1/`.

[32] W3C. XML path language (XPath) version 1.0, November 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[33] W3C. XSL transformations (XSLT), nov 1999. `http://www.w3.org/TR/1999/REC-xslt-19991116`.

[34] W3C. Extensible markup language (XML) 1.1, February 2004. `http://www.w3.org/TR/2004/REC-xml11-20040204/`.

[35] W3C. XML information set, February 2004. `http://www.w3.org/TR/2004/REC-xml-infoset-20040204/`.

[36] W3C. XML schema part 1: Structures second edition, October 2004. `http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/`.

[37] W3C. XML schema part 2: Datatypes second edition, October 2004. `http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/`.

[38] W3C. XML path language (XPath) 2.0, June 2006. `http://www.w3.org/TR/2006/CR-xpath20-20060608/`.

[39] W3C. XQuery 1.0: An XML query language, june 2006. `http://www.w3.org/TR/2006/CR-xquery-20060608/`.

[40] W3C. XQuery 1.0 and XPath 2.0 formal semantics, june 2006. `http://www.w3.org/TR/2006/CR-xquery-semantics-20060608/`.

[41] W3C. XQuery 1.0 and XPath 2.0 functions and operators, june 2006. `http://www.w3.org/TR/2006/CR-xpath-functions-20060608/`.

[42] W3C. XQuery update facility, july 2006. `http://www.w3.org/TR/2006/WD-xqupdate-20060711/`.

[43] W3C. XSL transformations (XSLT) version 2.0, june 2006. `http://www.w3.org/TR/2006/CR-xslt20-20060608/`.