POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica e Informazione
Scuola di Ingegneria Industriale e dell'Informazione

Specl:
An Assertion Language for Service Data Objects

Relatore: Prof. Sam Guinea

Tesi di Laurea di:
Riccardo Brunetti, matricola 787293

Anno Accademico 2012-2013

A mio fratello, ai miei genitori, ai miei nonni, a parenti ed amici. A chi mi ha supportato e sopportato in questi anni.

Contents

List of Figures IV
List of Tables VI
Listings VIII

Abstract **XI**
0.1 English Version XI
0.2 Italian Version XI

1 Introduction **1**

2 State of the Art **3**
2.1 XPath 4
2.2 XQuery 6
2.3 JML 8
2.4 OCL 9
2.5 Spec# 11
2.6 Requirements 12

3 Specl Language **15**
3.1 Data Types 15
3.1.1 SDO 16
3.1.1.1 Mapping Examples 16
3.1.1.2 Functions 17
3.1.2 Array 18
3.1.2.1 Functions 18
3.1.3 Strings 19
3.1.3.1 Functions 19
3.1.4 Double 20
3.1.4.1 Functions 20
3.2 Other Lexical Elements 21
3.2.1 Comments 21

3.2.2 Keyword 22
3.2.3 Operators 22
3.3 Features 22
3.3.1 Navigation and Query 23
3.3.2 Declarations 23
3.3.3 Assertion 24
3.3.3.1 Quantified Assertions and Aggreagated Functions 25
3.3.3.2 Numeric Expressions 27

4 Specl Interpreter 29
4.1 Features 29
4.2 Specl Grammar 30
4.3 API 31
4.4 Errors 32
4.4.1 Static Errors 32
4.4.2 Runtime 33

5 Examples 35
5.1 Didactic example: Bookstore 35
5.1.1 Book.xml 35
5.1.2 First Assertion 37
5.1.3 Second Assertion - Aggregated Functions 39
5.1.4 Third Assertion - Negated Assertion 40
5.1.5 Fourth Assertion - Numeric Aggregate Functions . . . 40
5.1.6 Fifth Assertion - Numeric Expressions 41
5.2 SOAP Handler Example 42
5.2.1 SOAP 42
5.2.2 Server Side 43
5.2.2.1 Web Service Methods 43
5.2.2.2 Server SOAP Handler 43
5.2.3 Client Side 45
5.2.3.1 Client SOAP Handler 45
5.2.4 Results 46
5.3 Twitter API's Example 46
5.3.1 REST 47
5.3.2 Twitter 47
5.3.3 Code 47
5.3.3.1 Authentication Process 47
5.3.3.2 Fetch Data 49

5.3.3.3 Result . . . 54
5.3.4 Web Site . . . 54

6 Conclusion **55**

A XText **57**
A.1 EMF . . . 58
A.2 ANTLR . . . 60
A.3 Google Guice . . . 60
A.4 XText Generator . . . 61
A.5 Interact with XText models programmatically . . . 62
A.5.1 Loading and using a model . . . 62
A.5.2 Create a ResourceSet . . . 63
A.5.3 Loading a resource . . . 63
A.5.4 Referencing the root . . . 64
A.5.5 Working with the Grammar . . . 64

B Specl Grammar **65**

Bibliografia **69**

List of Figures

2.1 OCL Example - Bank Account 10
2.2 Navigation Example 10

5.1 SOAP Architecture 42

A.1 Ecore Classes 59
A.2 XGL-Ecore Mapping 60

List of Tables

3.1 Available Operands for each Type 24
3.2 Quanfied Assertions and Aggregated Functions 26

4.1 Runtime Errors and Examples 33

A.1 XGL Grammar - EMF Mappings 59

Listings

2.1 XML Example File . . . 5
2.2 XQuery - Orders XML Example . . . 7
2.3 XQuery - Query Example . . . 7
2.4 XQuery - Query Result . . . 7
2.5 XQuery - Quantified Expressions Examples . . . 8
2.6 JML Specification Example . . . 8
2.7 JML Quantifiers Examples . . . 9
2.8 Spec# Specification Example . . . 12
3.1 SDO Mapping - Original XML Document . . . 16
3.2 SDO Mapping - Original JSON Document . . . 16
3.3 SDO Mapping - Derived SDO . . . 17
3.4 Navigational Expression Example . . . 23
3.5 Variable Declaration Examples . . . 24
3.6 Assertion Examples . . . 25
3.7 Quantified Assertions and Aggregated Functions Examples . . 26
4.1 Java Snippet for Retrieving Specl's Model . . . 30
4.2 Simplified Specl Grammar . . . 30
4.3 Specl Syntax Error Example . . . 32
5.1 Main Example File Book.xml . . . 35
5.2 SDO Tranformation - Book.xml . . . 36
5.3 First Assertion . . . 38
5.4 First Assertion - Alternative 1 . . . 38
5.5 First Assertion - Alternative 2 . . . 38
5.6 First Assertion - Wrong Predicates . . . 39
5.7 First Assertion - Alternative 3 . . . 39
5.8 Second Assertion . . . 39
5.9 Third Assertion . . . 40
5.10 Fourth Assertion . . . 40
5.11 Fifth Assertion . . . 41
5.12 BookstoreWS Handler Snippet . . . 43

5.13 BoostoreWS Client Handler - Conditions 45
5.14 Twitter's Bearer Code Request Snippet 48
5.15 Getting Timelines Tweets Snippet 49
5.16 Tweets Search Snippet 50
5.17 Trends in Milan Snippet 51
5.18 Twitter's API Limits Snippet 52
B.1 Specl Grammar 65

Abstract

0.1 English Version

Service Oriented Architecture (SOA) is a modern approach to build distributed application and greatly reduce the time, effort and cost to develop the software projects. SOA provides the dynamic paradigm for composing distributed application even in heterogeneous environments. This is also introducing challenges: due to its distributed nature, service-oriented applications does not give a direct control over each third-party service integrated in the architecture (for example, they might change during time and be incorrect with respect to their initial functionality, the one for which they were chosen).

Starting from a language invented at Politecnico di Milano for facing the problem of monitoring and recovery in specific situation of service coordination, we built a more general language for observing and asserting conditions on system operation in a wider range of possible contexts, disconnecting itself from a sole framework.

In this text, we will explain the grammar and the features of the language, and even its working environment. We also provide a series of examples and tutorials for support the understanding, and for showing its power and flexibility.

0.2 Italian Version

Oggigiorno, l'approccio moderno e piú utilizzato per lo sviluppo di applicazioni é denominato Service Oriented Architecture (SOA): i principali benefici ottenuti sono la realizzazione di applicazioni distribuite in un minor tempo e con minor sforzo, piú flessibili e facilmente integrabili. SOA fornisce un paradigma dinamico per la composizione di applicazioni distribuite anche in ambienti eterogenei.

Questo porta con se anche delle difficoltá e introduce nuove problem-

atiche: data la loro natura distribuita, le applicazioni service-oriented che sfruttano e integrano servizi di terzi-parti non hanno ne garanzie, ne controllo diretto su queste componenti esterne. La giurisdizione su ciascun servizio é del fornitore, che, nell'arco del tempo, puó modificare o dismettere il servizio; di conseguenza, questo puó causare malfunzionamenti e complicazioni in tutte quelle architetture composte che sfruttano tale servizio.

All'interno del Politecnico di Milano é stato sviluppato un linguaggio proprio per far fronte al problema del monitoraggio e del recupero da errori, concentrandosi sull'ambito della coordinazione di web services.

In questa tesi, partendo dal linguaggio di cui sopra, andiamo a costruire un linguaggio di piú generico uso, che fornisce funzioni di monitoraggio e analisi di precise condizioni sul funzionamento del sistema. L'insieme dei possibili contesti raggiungibili viene ampliato, slegandosi quindi dal solo ambiente della composizione di web services.

Mostreremo e spiegheremo la grammatica del linguaggio, le sue caratteristiche e il funzionamento d'insieme. Tutto questo sará fatto seguendo un approccio pragmatico: ogni passaggio sará chiarito con opportuni esempi e casi d'uso che ne mostreranno le capacitá e la flessibilitá. Saranno anche brevemente descritti dei tutorial, realizzati appositamente e resi disponibili online, per mostrare come il linguaggio sia adattabile a diverse situazioni e in diverse circostanze.

Tutto il materiale sará inoltre liberamente accessibile su un sito web dedicato, nato proprio con l'idea di facilitare l'uso e l'apprendimento di questo strumento.

Chapter 1

Introduction

Modern applications are increasingly embracing the "everything-as-a-service" paradigm. This allows them to offer complicated and flexible functionality by composing different kind of services. One of the main peculiarities of using service-based technology is that the services are provided by their original developers over the Internet; when we decide to integrate them into our applications we do so through remote invocation. This means they are not under our jurisdiction, and that they are free to evolve according to the service provider's own business priorities.

There are many advantages to services, such as reduced amount of design work, separation of concerns, loose coupling and interoperability, the downside is that this is leading to potentially critical dependencies between services. A change in one service could cause side effects, faults, and errors in the other services in the system. Moreover, this phenomenon can be unpredictable; it could arise without any notice, and remain true for an unknown amount of time.

One way to cope with these complications is to enable service adaption, i.e. the services should be able to adapt to changes in their environment or in the system's requirements. One possible approach to self-adaptation is to hard-code adaptation strategies that can be triggered when a known fault occurs. This solution is limited since it requires that we known a priori the problems that could occur. Nowadays, it is common to use probes to get runtime information about a service's execution and operation, so that we can analyze its behavior, and launch a defined recovery or compensation strategy that exploits actuators that have been deployed to the system. In the analysis phase the runtime data are examined and filtered according to conditions and rules imposed by developers.

The work we are doing in this thesis goes in this direction. We give

designers a means to define runtime data handling, querying, and analysis. The work was inspired by previous work achieved at Politecnico di Milano called WSCoL (Web Service Constraint Language) [9, 10, 11]. WSCoL was a monitoring language for BPEL processes that was instrumental in the development of self-supervising BPEL processes, i.e. ≪special-purpose compositions that assess their behavior and react through user-defined rules≫.

Starting from WSCoL, we decided to extend its potential, and go beyond the context of BPEL processes. We wanted to cover all the possible implementations of SOA, and therefore we developed Specl - a general purpose specification language. To achieve our goals we used Service Data Objects (SDOs) [16] as our main data representations. They provide the needed level of abstraction on data, as well as a single interface for accessing it.

The Specl notation is simple and intuitive. Assertions cover the propositional logic constructs, and use a syntax inspired to XPath. Developers can predicate over data using a navigational path notation, query the SDOs to retrieve data, and then analyze them.

The interpreter of the language uses components that were built using XText [5, 6], a framework for language development. The included tools are a lexer, a parser, a validator and an AST builder. A proper API is offered, so that designers can include Specl as easily as possible in their projects.

To foster the adoption of Specl we are supporting interested developers with a vast set of examples that demonstrate all the language's main features. We also developed two complex real-world examples: a pre- and post-condition validation tool for SOAP calls between web services and clients, and a data extraction tool for JSON data such as that provided by Twitter's REST APIs. Both the language and its tutorials are provided through a GitHub web site.

Chapter 2

State of the Art

Inside Specl could be identified two natures: the first is a great facility in data handling, with simple navigation and query statements; and secondly it is a monitoring language useful for analysis on retrieved data. In these two areas, there is a lot of powerful tools and methods available, but none of them as the abstraction level of our language (given by the use of SDOs) and integrates both these skills.

The first capability of Specl is data individuation, manipulation and extraction. In this dimension, it is competing with query and navigation languages for XML documents and JSON. For the former set, exists many powerful idioms as XPath (from which the navigation and queries form is inspired), for search and obtain data from XML nodes, and XQuery for query, transform and selectively extract data out of XML documents. For the latter, we are in an early phase, but are raising many languages with the objective to port an XPath-like syntax and abilities to JSON. Instead, Specl gives a unique query form, homogeneous and independent from the underling file to interrogate. The user could be completely unaware of the type: he/she is using and retrieving data from an SDO without making any distinction.

For what concern the monitoring and statements checking, an important and notable approach (adopted by many general-purpose programming languages) is the design by contract paradigm introduced by C.A.R. Hoare, which introduce the use of pre-conditions (statements that must be true when a method is called) and post-conditions (conditions that must be true at the end of a task) [14]. "By contract" means that if the operation is correct and the pre-conditions met, then the post-conditions are guaranteed to be true; otherwise, if pre-conditions are false, there is no assurance on the result. Checking pre- and post-conditions of procedures is an important

technique for improving the reliability of software. It is available in many languages, including Java (with JML) and C# (with Spec#) [17]. Several languages, including in particular Eiffel [13], have supported runtime-checked pre- and post-condition contracts since their inception. UML has an associated language called OCL (Object Constraint Language) in which constraints can be represented formally. OCL brought a language to describe additional constraints about the objects in the model, which are not possible to describe in a graphic way.

In the next sections there will be an explanation of XPath and XQuery; after those, will be concisely presented JML for Java, OCL and Spec#, covering capabilities similar to the assertions validation of Specl.

2.1 XPath

XPath, also known as XML Path Language, is a query language for finding information in an XML document. It is defined and maintained by the World Wide Web Consortium (W3C) [20].

XPath use a syntax for defining parts of an XML document and uses path expressions to navigate through it and select nodes or node-sets. The language contains also a vast library of standard functions, which gives it the ability to compute in certain way the values retrieved from documents.

XPath operates on a logical representation of the XML file that is modeled to a tree structure and the language syntax for accessing to it. These expressions took the name *location path*, but from version 2.0, are called *path expressions*. Its expanded structure is `axis::node-test[predicate]`: `axis` says the relationship between the target node and the current node; `node-test` specify the type or the name of the target node; and `predicate` contains zero or more filter (inside square brackets) for imposing selecting conditions on the look up of the target node.

Main relationships in axis are:

- *Ancestor*: refers to each ancestor of the current node, i.e. each parent node before it in the tree;
- *Attribute*: every attribute of actual node;
- *Child*: refers to each child node of the current node;
- *Descendant*: refers to every child node descending from the current node and after it in the tree;

- *Parent*: refers to the node immediately before the current node, i.e. its parent;

- *Self*: refers to the current node itself.

Exists also an abbreviated syntax for path expression, using a compact notation with the list of the name of the nodes to access separated by a slash character (/). This list describe the path to the target node.

XPath 2.0 (the actual recommended version by W3C) is larger than the first version, and some concept is changed. Now, in particular, there is the built-in support for querying XML documents, because the 2.0 version is a subset of XQuery.

Given a sample XML document (taken from `http://en.wikipedia.org/wiki/XPath`)

Listing 2.1: XML Example File

```
<?xml version="1.0" encoding="UTF-8"?>
<wikimedia>
  <projects>
    <project name="Wikipedia" launch="2001-01-05">
      <editions>
        <edition language="English">
          en.wikipedia.org
        </edition>
        <edition language="German">
          de.wikipedia.org
        </edition>
        <edition language="French">
          fr.wikipedia.org
        </edition>
        <edition language="Polish">
          pl.wikipedia.org
        </edition>
        <edition language="Spanish">
          es.wikipedia.org
        </edition>
      </editions>
    </project>
    <project name="Wiktionary" launch="2002-12-12">
      <editions>
        <edition language="English">
          en.wiktionary.org
        </edition>
```

```
        <edition language="French">
          fr.wiktionary.org
        </edition>
        <edition language="Vietnamese">
          vi.wiktionary.org
        </edition>
        <edition language="Turkish">
          tr.wiktionary.org
        </edition>
        <edition language="Spanish">
          es.wiktionary.org
        </edition>
      </editions>
    </project>
  </projects>
</wikimedia>
```

The XPath expression

```
/wikimedia/projects/project/@name
```

selects name attributes for all projects, and

```
/wikimedia//editions
```

selects all editions of all projects, and

```
/wikimedia/projects/project/editions/↩
    edition[@language="English"]/text()
```

selects addresses of all English Wikimedia projects (text of all edition elements where language attribute is equal to *English*). And the following

```
/wikimedia/projects/project[@name="Wikipedia"]/↩
    editions/edition/text()
```

selects addresses of all Wikipedias (text of all edition elements that exist under project element with a name attribute of *Wikipedia*)

2.2 XQuery

XQuery (XML Query Language) is to XML what SQL is to database tables, is a language for finding and extracting elements and attributes from XML documents - not just XML files, but anything that can appear as XML, including databases [21]. XQuery is the language for XML querying and is built on XPath expressions. It was developed and recommended by W3C. Notice that, as said in the previous section, XQuery 1.0 and XPath 2.0 share the same data model and support the same functions and operators, so it is easy to pass from one to the other.

XQuery's expressions use XPath for the nodes identification then additional functions, peculiar of XQuery, for retrieve node's information.

The main expressions used for making complex queries are called FLWR expressions (For-Let-Where-Return), a generalization of the `SELECT-FROM-WHERE-HAVING` constructs.

Image to have an XML document called *orders.xml* and containing information about orders with this base structure:

Listing 2.2: XQuery - Orders XML Example

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
   <date>...</date>
   <product>...</product>
   <size>...</size>
   <color>...</color>
   <quantity>...</quantity>
</order>
```

The following query search for `<order>` nodes where the quantity is greater than 50 and sort the results according to the date of the order: `document("orders.xml")//order[quantity > 50] SORTBY (date)`

Let us consider another example:

Listing 2.3: XQuery - Query Example

```
FOR $c IN document("orders.xml")//color
LET $o := document("orders.xml")//order[color=$c]
WHERE count($o) > 50
RETURN
        <result>
                {$c}
        </result>
```

The first instruction creates a list (variable `$c`) holding all the colors in the archive. The second line associates at each color, the orders containing it, generating an ordered list of pairs color-order (`$c`, `$o`). Then, from this list, are selected (`WHERE`) only the tuples respecting the imposed filter i.e. appears in more than 50 orders. Finally, the `RETURN` instruction, the resulting colors are printed inside the `<result>` element. For example:

Listing 2.4: XQuery - Query Result

```
<result>
        <color>red</color>
        <color>blue</color>
```

```
</result>
```

The `count()` function is one of the main functions offered by XQuery for operating on elements and list of elements. The other ones are `avg()` for the average, and operations on sets like `union()`, `intersect()` and `difference()`.

XQuery supports other constructs like `IF-THEN-ELSE` for conditional expressions, existential expressions with `SOME-IN-SATISFIES` and universal with `EVERY-IN-SATISFIES`.

Listing 2.5: XQuery - Quantified Expressions Examples

```
some $product in doc("orders.xml")//product
satisfies ($product = "T-Shirt")

every $product in doc("orders.xml")//product
satisfies ($product = "T-Shirt")
```

The first one of previous queries check if exists at least one order with a *product* "T-Shirt", and the second if all of them are "T-Shirt".

2.3 JML

The Java Modeling Language (JML) is ≪a behavioral interface specification language that can be used to specify the behavior of Java modules ≫[2, 3]. It is used as a design by contract (DBC) tool for Java. A contract in software specify both duties and rights of clients and implementors.

JML specifications are written in special annotation comments that starts with an at-sign (`@`), and uses the `require` clause for the client's obligations and `ensure` for implementor's obligations. Considering the contract, `require` specifies the method's pre-conditions while `ensure` its post-conditions. In JML, specifications are typically written just before the header of the method they specify. The following is a known example for the sqrt method:

Listing 2.6: JML Specification Example

```
/*@
   requires
      x >= 0.0;
   ensures
      JMLDouble.approximatelyEqualTo(x, \result * \result, eps);
 @*/
public static double sqrt(double x) {
```

```
        /*...*/
}
```

In this example pre-condition is that the argument `x` is a positive number, and the post-condition says that the client has the right to get a square root approximation as the result (`/result`). The static method `approximatelyEqualTo` of the class `JMLDouble` tests whether the relative difference of the first two double arguments is within the given epsilon, the third argument.

JML combines the practicality of DBC language like Eiffel with the expressiveness and formality of model-oriented specification languages. As in Eiffel, JML uses Java's expression syntax to write the predicates used in assertions, such as pre- and post-conditions and invariants. The advantage of using Java's notation in assertions is that it is easier for programmers to learn and less intimidating than languages that use special-purpose mathematical notations. However, Java expressions lack some expressiveness that makes more specialized assertion languages convenient for writing behavioral specifications. JML solves this problem by extending Java's expressions with various specification constructs, such as quantifiers.

JML supports several forms of quantifiers: universal and existential (`\forall` and `\exists`), general quantifiers (`\sum`, `\product`, `\min`, `\max`) and numeric quantifier (`\num_of`).

Listing 2.7: JML Quantifiers Examples

```
// says that a[] is sorted at indexes between 0 and 9
// (the second parameter, separated by ';', is range)
(\forall int i,j; 0 <= i && i < j && j < 10; a[i] < a[j])

(\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
(\product int i; 0 < i && i < 5; i) == 1 * 2 * 3 * 4
(\max int i; 0 <= i && i < 5; i) == 4
(\min int i; 0 <= i && i < 5; i-1) == -1
```

2.4 OCL

The Object Constraint Language (OCL) started as a complement of the UML notation with the goal to overcome the limitations of UML (and in general, any graphical notation) in terms of precisely specifying detailed aspects of a system design [12]. Since then, OCL has become a key component of any model-driven engineering (MDE) technique as the default language

for expressing all kinds of (meta) model query, manipulation and specification requirements. Among many other applications, OCL is frequently used to express model transformations (as part of the source and target patterns of transformation rules), well-formedness rules (as part of the definition of new domain-specific languages), or code-generation templates (as a way to express the generation patterns and rules)[7].

The following example (taken from "OCL for Java Tutorial" on ParlezUML - `http://www.parlezuml.com/tutorials/umlforjava/java_ocl.pdf`) is referring to the common case of a bank account pre and post-conditions verification:

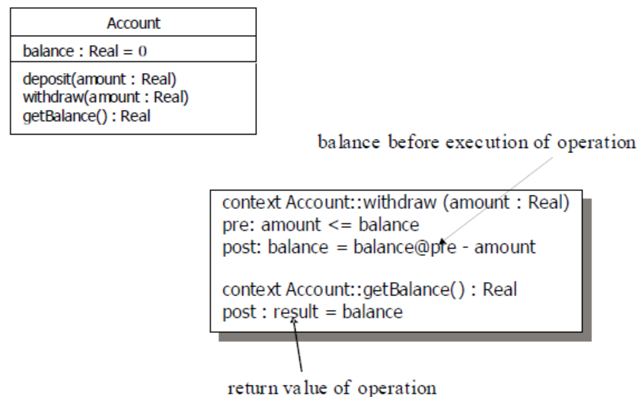

Figure 2.1: OCL Example - Bank Account

Where `context` is used for the specification of the interested method and its header; `pre` and `post` for the conditions to check before and after the execution of the procedure.

OCL offers predefined mechanism for retrieving the values of an object, navigating through a set of related objects and iterating over collections (e.g., `forAll`, `exists`, `select` functions). Values from objects are obtained using dot notation: for example, `account.balance` gives the balance value of the account object.

Even navigation is done through the dot notation, according to the cardinality of the relationships between objects: given the situation in figure, `account.holder` evaluates to a `Customer` object who is in the role *holder* in that situation. In the same way, `costumer.accounts` evaluates to a collection of `Account` ob-

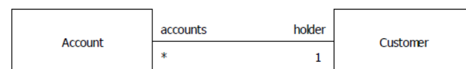

Figure 2.2: Navigation Example

jects in the role *accounts* of that relation.

Functions over Collection are many (see the specifications of the language), but a little example could be: `customer->exists(a:accounts | a.balance=0)`.

OCL includes also a standard library with set of object type and operations on them. Types could be primitive (as `String`, `Integer`, `Real` and `Boolean`) or collection (as `Set`, `Bag`, `OrderedSet` and `Sequence`).

Main advantages given by OCL are:

- A typed language: each expression evaluates to a type and conforms to the rules and operations of that type
- A language for specification: does not include any implementation details nor implementation guidelines
- A language for expressing necessary extra information about a model
- A precise and unambiguous language that can be read and understood by developers and customers
- A language that is purely declarative, i.e. it has no side-effects (in other words it describes what rather than how, OCL expressions can query and constrain the state of the system but not modify it)

2.5 Spec#

Spec# (pronounced Spec Sharp) extends C# with contracts allowing programmers to document their design decisions in their code (with support for non-null types, checked exceptions and throws clauses, method contracts and object invariants) [15].

The Spec# system consists of:

- The Spec# programming language. Spec# is an extension of the object-oriented language C#.
- The Spec# compiler. Integrated into the Microsoft Visual Studio development environment for the .NET platform, the compiler statically enforces non-null types, emits run-time checks for method contracts and invariants, and records the contracts as metadata for consumption by downstream tools.
- The Spec# static program verifier. This component generates logical verification conditions from a Spec# program. Internally, it uses an

automatic theorem prover that analyzes the verification conditions to prove the correctness of the program or find errors in it.

A unique feature of the Spec# programming system is its guarantee of maintaining invariants in object-oriented programs in the presence of callbacks, threads, and inter-object relationships.

Non-null types consists in declaring that an element is not null: `type T!` contains only references to objects of `type T` (not null). This puts the responsibility to make sure an argument is not null to the user. For example `int[]!xs` means that the xs array must not be null, otherwise an error message is sent.

Method contracts let the designer specify pre- and post-condition of a method. The used keywords are `requires` and `ensures`. In the example below a simple swap method is showed:

Listing 2.8: Spec# Specification Example

```
static void Swap(int[]! a, int i, int j)
       requires 0 <= i && i < a.Length;
       modifies a[i], a[j];
       ensures a[j] == old(a[i]);
```

Where `old` refers to the initial passed value of the variable. The contract could also have a statement signaling what variables are going to be modified (with `modifies` keyword).

Inline assertion is another feature: while method contracts indicate conditions that are expected to hold on method boundaries, the assert statement can be used in code to indicate a condition that is expected to hold at that program point. The statement `assert r <= x;` will cause the program verifier to check that `r` is bounded by `x`.

Even expressions using quantifiers are possible in Spec#, are available `forall` and `exists` quantifiers.
For example:

```
forall{int k in (0:a.Length); a[k] > 0};
exists{int k in (0:a.Length); a[k] > 0};
```

Exists also aggregated functions, using the same form as the previous: `sum`, `product`, `min`, `max` and `count`.

2.6 Requirements

Summing up, *Specl* is built has a mixture of features presented above, providing a language able to

- Identify and isolate data with precision;
- Select a single value or a subset of whole data;
- Compute obtained information using different expressions;
- Evaluate conditions on certain data;
- Simple to use and understand;
- Easy to integrate in various use scenarios.

Chapter 3

Specl Language

The main strength of this language is that it relies on the concept of Service Data Object (SDO). This gives the language a single consistent view of data obtained from a variety of sources, and takes *Specl* to a more high level of abstraction. *Specl* acquires power and flexibility thanks to SDOs, allowing it to be used in various different contexts.

In this chapter, we will explain the language characteristics and features, decorating explanations with little examples (extracts of the didactic example at Chapter 5.1) for a better understanding. It would be useful at least a basic knowledge of navigational path and XPath, if not see Section 3.3.1.

The presentation will be done in a bottom-up fashion, from the data types to the whole grammar structure.

3.1 Data Types

The data types used inside the language are

- SDO
- Array
- String
- Double
- Boolean

Each of them as associated its own functions, that expands the power of the language and that could be easily upgraded and extended. SDO, as a containment object, could have values of each of the previously showed type, even another SDO.

3.1.1 SDO

SDO was born for simplify and unify Service Oriented Architecture (SOA) data access and code, i.e. its purpose is to facilitate communication and, for this reason, are designed to deliver a uniform data access layer for heterogeneous data sources in a service-oriented architecture. Provides flexible data structures that allow data to be organized as graphs of objects (called *data object*) that are composed of *properties*. Properties can be single or many valued and can have other data objects as their values.

3.1.1.1 Mapping Examples

The Specl interpreter, provided as final result of this thesis, gives the possibility to translate XML documents and JSON objects to SDO. Furthermore, thanks to the generality and abstraction of SDOs, developers could build their own translator for any type of data.

For example, starting from the next XML and JSON code could be obtained a simple SDO.

Listing 3.1: SDO Mapping - Original XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<container>
  <elem1>String</elem1>
  <elem2>1</elem2>
  <elem3>A</elem3>
  <elem3>B</elem3>
  <elem3>C</elem3>
  <elem4>true</elem4>
  <elem5>
    <sub-elem1>String</sub-elem1>
    <sub-elem2>
      <sub-sub-elem1>10</sub-sub-elem1>
    </sub-elem2>
    <sub-elem3>1</sub-elem3>
    <sub-elem3>2</sub-elem3>
  </elem5>
</container>
```

Listing 3.2: SDO Mapping - Original JSON Document

```
{
  "elem1": "String",
  "elem2": "1",
  "elem3": [
```

```
      "A",
      "B",
      "C"
   ],
   "elem4": "true",
   "elem5": {
      "sub-elem1": "String",
      "sub-elem2": [
         "10"
      ],
      "sub-elem3": [
         "1",
         "2"
      ]
   }
}
```

Here is the structure of the SDO derived from the previous languages:

Listing 3.3: SDO Mapping - Derived SDO

```
{
   container = {
      elem1 = "String",
      elem2 = 1,
      elem3 = ["A", "B", "C"],
      elem4 = true,
      elem5 = {
         sub-elem1 = "String",
         sub-elem2 = {
            sub-sub-elem1 = 10
         },
         sub-elem3 = [1, 2]
      }
   }
}
```

As you can see, the representation is pretty similar to JSON: sets of key-value entry and ordered lists of values.

3.1.1.2 Functions

SDO's functions let the designer to operate both on the whole object and on the elements included into it (could be checked the presence of a certain property or get its value).

The following is the list of available functions (plus a short example for each one):

`boolean contains(Object elem)`

returns true if an object with `elem` value is contained (the research is deep)

```
// search "Pocket" string in the first book of the inventory
/inventory/book[1].contains("Pocket")
```

`Object get(int index)`

returns the value of the `index`-th node (the first element of the array has index equals to 1)

```
// returns the first book of the inventory
/inventory.get(1)
```

`Object get(String node)`

returns the value of the node with name `node`

```
// get the value of the key "name"
/inventory.get("name")
```

`double cardinality()`

returns the number of elements contained

```
// get the number of values contained
/inventory.cardinality();
```

3.1.2 Array

The used type coincide with the classical kind of array, containing a finite number of elements. It is defined writing the elements of it inside square brackets and separated by commas (`[elem1, elem2, ...]`).

3.1.2.1 Functions

Available functions for arrays covers the basic general-purpose language functions for this kind of data type, working on elements or getting the size of the array:

`boolean contains(Object elem)`

returns *true* if the passed `elem` object is contained

```
// check if the second book is written by a certain author
/inventory/book[2]/authors/author.contains("Larry Niven")
```

`Object get(int index)`
: returns the `index`-th element of the array

```
// get the second author from an array of authors,
// from the second book
/inventory/book[2]/authors/author.get(2)
```

`double cardinality()`
: returns the length of the array

```
// get the length of the authors array of the second book
/inventory/book[2]/authors/author.cardinality()
```

3.1.3 Strings

Represents a string constant obtained from a sequence of characters. Everything inside quotes is considered as string (double or single quote: `"..."` or `'...'`).

3.1.3.1 Functions

Even in this case, the functions are simple and common in many other contexts:

`String uppercase()`
: turns every character to uppercase

```
// inventory's name to uppercase
/inventory/name.uppercase()
```

`String lowercase()`
: turns every character to lowercase

```
// inventory's name to lowercase
/inventory/name.lowercase()
```

`double length()`
: returns the length of the string

```
// inventory's name length
/inventory/name.length()
```

`boolean startsWith(String prefix)`
: returns *true* if the string starts with `prefix`

```
// check inventory's name prefix
/inventory/name.startsWith("Inv")
```

`boolean endsWith(String suffix)`
: returns *true* if the string ends with `suffix`

```
// check inventory's name suffix
/inventory/name.endsWith("YZ")
```

`boolean contains(String s)`
: returns *true* if the string contains `s`

```
// check if inventory's name contains "ABC"
/inventory/name.contains("ABC")
```

`String concat(String s1, String s2, ...)`
: returns the concatenation of the string with `s1`, `s2`, etc...

```
// concatenate two strings to the title of the first book
/inventory/book[1]/title.concat("is", "the 1st Book")
```

`String substring(int beginIndex, int endIndex)`
: extracts the characters, between two specified indexes, and returns the new substring

```
// substring of the first book's title
/inventory/book[1]/title.substring(0, 2)
```

`String replace(String target, String replacement)`
: replace every occurrence of `target` with `replacement`

```
// substitutes blank spaces with underscores,
// in first book's title
/inventory/book[1]/title.replace(" ", "_")
```

3.1.4 Double

The general representation of every number, which are parsed and used as a signed double.

3.1.4.1 Functions

For numbers, the available functions can be used for manipulations and rounding. The list is a subset of the functions offered by XPath.

`double abs()`

gives the absolute value of the number on which it is applied

```
// suppose variable $a is defined equals to -1
$a.abs() // result is 1.0
```

`double ceiling()`

returns the smallest integer that is greater than the number argument

```
// ceiling of the first book's price (suppose 9.9)
/inventory/book[1]/price.ceiling() // result is 10.0
```

`double floor()`

returns the largest integer that is not greater than the number argument

```
// floor of the first book's price (suppose 9.9)
/inventory/book[1]/price.floor() // result is 9.0
```

`double round()`

rounds the number argument to the nearest integer

```
// rounded price of the first book (suppose 9.9)
/inventory/book[1]/price.round() // result is 10.0
```

`double roundHalfToEven()`

rounds a number with .5 fraction to the nearest even integer

```
// rounded price of the first book (suppose 9.5)
/inventory/book[1]/price.roundHalfToEven() // result is 10.0
```

`String toString()`

returns the string representation of the number

```
// string representation of the first book's price
/inventory/book[1]/price.toString() // result "15.0"
```

3.2 Other Lexical Elements

3.2.1 Comments

There are two type of comments:

- **Line Comments** start with the sequence `//` and stop at the end of line

- **General Comments** start with the sequence `/*` and continue until character sequence `*/` , it could contain more than one line

Comments do not nest.

3.2.2 Keyword

The following keywords are reserved and may not be used as identifiers

```
let       max       startsWith
in        min       endsWith
forall    sum       contains
exists    avg
numOf     product
```

3.2.3 Operators

The following character sequences represent operators, delimiters, and other special tokens:

```
==    +    &&    ,    ( )
!=    -    ||    ;    [ ]
<     *    !     .
>     /
<=    %
>=
=
```

3.3 Features

Specl structure present two main part: declaration part and assertions part.

Declarations part has the purpose to let the user define variables that could be used later for making assertions. The use of variables is not mandatory for the grammar, but is useful for simpler and clearer usage of the language.

Assertions part of the grammar contains the set of constraints and values to check and keep controlled. It is the kernel of the language. Assertions are composed by using logical operators: NOT (`!`), AND (`&&`), OR (`||`). These have the usual priority: NOT>AND>OR. With brackets for change the priority, depending on the user's needs.

Each assertion consist of a comparison, using various operators, between two expressions.

3.3.1 Navigation and Query

At first, we will discuss about one of the basic functionality of the language: navigation path expressions and query. Those let the user querying and retrieving data values from an SDO using a notation inspired by XPath.

The queries are built by steps, separated by a slash (/): at each step, the specified element (conforming to a key property in the SDO) is selected and the corresponding value is returned.

As in XPath, there are *predicates* used to restrict the node-set by selecting only those nodes for which a specified condition is true; acceptable predicates enable the selection of a node by its value or by its position:

Listing 3.4: Navigational Expression Example

```
// select the 'step1' whose 'step2' is greater or equal than 1
let $a = /step1[step2 >= 1];

// select the 'step1' whose 'step3' is equal to "String"
let $b = /step1[step3 == "String"];

// select the first of the nodes of 'step1'
let $c = /step1[1];
```

The form in the first two cases is

`[ node op value ]`

Where:

- **node**: the name of the node to test
- **op**: the operation to apply (note that only `==` and `!=` has sense when value is a String)
- **value**: the value to compare, could be a String, a number or a Variable

In the last example, the parentheses contain just a number, that number refers to the index of the item to select.

3.3.2 Declarations

Allows you to assign a value to a variable. This variable can then be used almost anywhere in the code that follows it, and there can no longer be assigned a different value.

Declaration syntax:

`let $var = value ;`

Where:

- `let`: is the keyword for starting a declaration
- `$var`: is the name of the assigned variable
- `value`: is the value to store
- `';'` : is the termination character

The value could be directly declared when it is a constant (a number, a string, a Boolean or an array - see data types at Section 3.1), or it could be the result of a statement (the value corresponding to a certain navigation path or the result of an expression or a quantified assertion).

Here is some examples:

Listing 3.5: Variable Declaration Examples

```
// a number literal
let $price = 10.5;

// a string literal
let $string = "Snow Crash";

// an array
let $author = ["Larry Niven", "Jerry Pournelle"];

// a navigational path to resolve
let $a = /inventory/book/title;

// multiply 2 and (10.5 - 1)
// (note: $price is the variable previously defined, equal to 10.5)
let $expr = 2 * ($price - 1);
```

3.3.3 Assertion

With the term assertion, we intend *true/false* statement achieved with a comparison between expressions. The comparison can be made with the use of operations like equal (`==`), unequal (`!=`), greater (`>`), lower (`<`), greater or equal (`>=`) and lower or equal (`<=`).

Each operation has sense with respect to the expressions type. The numeric assertions are the only ones that has access to the whole set of operands. In the next table, for each data type, are shown available operations and explained particular cases (see boolean expressions):

Table 3.1: Available Operands for each Type

Operation	SDO	Array	Number	String	Boolean
==	✓	✓	✓	✓	see ⋆
!=	✓	✓	✓	✓	see ⋆
>			✓		
<			✓		
>=			✓		
<=			✓		

⋆ in the case of a comparison between `Boolean` expressions, the operation and the result expression becomes implicit (the default value is true) and must be removed (for example the right one is `$a.contains("string")` and not `$a.contains("string") == true`).
If the assertion checks false the expression is negated (`!($a.contains("string"))`).

The members of an assertion can be:

- query (see Section 3.3.1)
- constant (numeric or string)
- array
- quantified assertion (see Section 3.3.3.1)
- numeric expression (see Section 3.3.3.2)
- variable

Examples of few possible statements composition

Listing 3.6: Assertion Examples

```
// query == string constant
/inventory/name == "InventoryABCD";

// variable >= numeric constant
$price >= 2;
```

3.3.3.1 Quantified Assertions and Aggreagated Functions

These are assertions that iterate over a finite set of values and check every contained element with respect to the conditions. The objective is to give the possibility to express constraints using universal and existential quantifier or aggregated functions.
The statement is in the following form:

`name (alias in var, conditions)`

Where:

- **name**: the name of the used quantifier/function (see the table below for details)
- **alias**: a temp variable that will be used as parameter in the upcoming `conditions`
- **var**: the variable to which the iteration is applied, it defines the range of values that the `alias` can assume (`var` must be an array)
- **conditions**: the set of assertions to test at every iteration

List of functions and their characteristics:

Table 3.2: Quanfied Assertions and Aggregated Functions

Name	Returns	Description
`forall`	`Boolean`	*True* if, for every element, the conditions are respected
`exists`	`Boolean`	*True* if, for at least one element, the conditions are respected
`numOf`	`Double`	The number of elements that respects the conditions
`max`	`Double`	Returns the maximum element that respects the conditions
`min`	`Double`	Returns the minimum element that respects the conditions
`avg`	`Double`	Returns the average of the elements that respects the conditions
`sum`	`Double`	Returns the sum of elements that respects the conditions
`product`	`Double`	Returns the product of elements that respects the conditions

In next box an example, comprising a quantified assertions and an aggregated function evaluated in AND (`&&`):

Listing 3.7: Quantified Assertions and Aggregated Functions Examples

```
// check if the maximum price is greater than 10
let $prices = /inventory/book/price; // an array with all the prices
max($elem in $prices, $elem > 0) > 10
&&
// check if all the prices are greater than 0
forall($elem in $prices, $elem > 0);
```

3.3.3.2 Numeric Expressions

Use and evaluate numeric expressions, using the canonical operands: sum (`+`), subtraction (`-`), multiplication (`*`), division (`/`) and modulo (`%`). Expressions has left associativity, and priorities between these operators is exactly as they are listed before (`%`, `/` and `*` has greater priority than `+` and `-`). As usual, precedence could be modified with brackets.

Expressions can also be calculated using navigation path and variables, as long as they return numeric values.

Example:

```
// a simple expression using sum, multiplication and modulo
(1 + 1) * 1 % 1 == 0;
```

Chapter 4

Specl Interpreter

The Specl interpreter (also named Specl Analyzer) is developed in Java and use the *XText* productions for parsing and modeling the assertions, and then evaluates and offers a final result on the correctness of the statements.

This represents the result of the thesis work: offer a flexible and extensible tool, with the proper APIs affording simple usability and integrability for the developers' needs.

In this chapter, we will make an overview of its characteristics, its APIs, its fault messages.

4.1 Features

Specl Analyzer is the interpreter of Specl language: this includes, as a jar library (`Specl.jar`), the artifacts generated by XText. In this way, we can use programmatically the provided parser, obtain the EMF model corresponding to the readed tokens and, throught this, evaluate the assertions in Specl.

XText is a language development framework based on Eclipse and thought for DSLs and general-purpose languages. The used internal parser is obtained from the parser generator ANTLR, integrated inside the framework. XText is also based on the EMF framework and let developers build an IDE editor starting from the designed language, extensible and with features like Eclipse. For an overview see AppendixA.

As said in the chapter where we described the Specl language (Chapter 3) we are going to work with SDO in our own implementation. The input file (the one that is going to be interrogated with the language) is translated to an SDO and then processed. The analyzer has already methods for translating XML Document and JSON to SDO, but it is possible, for who

is interested, to develop other methods for mapping other kinds of file.

Now we can retrieve data from the model, declarations and assertions, and pass them to `DeclarationService` and `AssertionService` that starts checking them.

Listing 4.1: Java Snippet for Retrieving Specl's Model

```
Model model = (Model) resource.getContents().get(0);
EList<Declaration> declarations = model.getDeclarations();
Assertions assertionSet = model.getAssertionSet();
```

Specl Analyzer uses *Log4j* as logging and tracing framework, with a console appender for showing proper execution messages. We chose *Log4j* because it is widely used and for give to the developers an easier integration with their works. The severity level of the logger can be setted with an appropriate method; it is also possible to stop log messages.

4.2 Specl Grammar

In this section we will present a simplified version of Specl grammar.

It is purely showed for giving an idea of the substrate upon which we are working. A full version is available in the Appendix B and on the web site.

Listing 4.2: Simplified Specl Grammar

```
Model:
    Declaration* Assertions ';'

Declaration:
    'let' Variable 'in' Assertion ';'

Assertions:
    PrimaryAssertions (('&&' | '||') PrimaryAssertions )*

PrimaryAssertions:
    '(' Assertions ')' | '!(' Assertions ')' | Assertion

Assertion:
    Expression Rop Expression;

Expression:
    Step+ ('.' Function)* | NumericExpression | AssertionQuantified
        | Array | Boolean

Step:
```

```
    '/' ID ('[' Predicate ']')? | Variable

Predicate:
    ID Rop (STRING|NUMBER|Variable) | NUMBER | Variable

Function:
    ID '(' (Values)? ')'

AssertionQuantified:
    Quantifier '(' Variable 'in' Variable, Assertions ')'

Variable:
    '$' ID

Rop:
    '==' | '!=' | '>' | '>=' | '<' | '<='
```

4.3 API

Here we will show the offered API by the analyzer, with a brief description

`void setXMLInput(Document doc)`
: read the passed XML Document `doc` and store its SDO translation;

`void setJSONInput(String json)`
: reads the passed JSON and stores its SDO translation;

`void setMapInput(LinkedHashMultimap map)`
: reads and stores an SDO;

`void setSpeclFilePath(String path)`
: takes assertions from the file in the past `path`;

`boolean evaluate(String assertions)`
: verifies the `assertions`, returns *true* if they are correct, *false* otherwise;

`boolean evaluate()`
: same as the previous, but to use in the case in which the assertions are read from a file (requires that the `setSpeclFilePath` method is previously used);

`Object getVariable(String var)`
: returns the value of a variable used inside Specl assertions;

`void putVariable(String name, Object value)`
: puts a variable in Specl assertions;

`void removeVariable(String name)`
: removes a Specl variable;

`DataObject getInput()`
: returns the used SDO (the representation of the initially past input file);

`void setLoggerLevel(String level)`
: sets the logger to the passed `level` (by default it is set to 'INFO');

`void shutdownLogger()`
: disables the logger;

The constructor method do the variable setup and initialization. The Specl variable are stored in Map and use the singleton pattern, so they will persist until the next `new SpeclAnalyzer()` call.

4.4 Errors

The analyzer lets the user know when errors has been found, in the assertion's syntax (static errors) or at runtime (for example a wrong comparison between different data type). This kind of feedback will be really important for the users of the language and the interpreter.

For spotting syntax errors, we exploits the validator that XText gives us, which checks if the obtained model respects the Specl grammar. Differently, runtime errors are hard-coded and reported by throwing suitable Java exceptions.

4.4.1 Static Errors

With the term *static* we intend those kind of errors related to misspelled assertions in the language.

As just said, we use the validator provided with the other tools by XText that gives accurate information on issues found during the syntax check. For example, let us say we write the following (wrong) assertion:

Listing 4.3: Specl Syntax Error Example

```
let $a = 1 // note that the terminator is missing
$a == 1;
```

The analyzer understands that there is a ';' missing at the end of the first line (the report says that it is expecting it before reading the second line) and responds:

```
INFO - ***** Specl Analyser *****
ERROR - SYNTAX ERRORS: 1 found
ERROR - MSG: missing ';' at '$' - LINE: 2 - TOKEN:'$a'
```

The message format presents at first the logger severity level (`ERROR`), followed by the error message (`MSG`), the line in which the error is found (`LINE`) and the specific erroneous token (`TOKEN`).

The kinds of syntax errors identified spreads from missing characters to unknown/extraneous tokens found.

4.4.2 Runtime

Runtime errors are the ones that are caused during the execution of the interpreter, so during the evaluation of the assertions of *Specl*.

In the table below we list the existing errors and, for each of them, we indicate the description and an example that cause it:

Table 4.1: Runtime Errors and Examples

Error Cause	**Example**
Wrong comparison between the terms of an assertion	`1 == "String";`
Null term	`1 == null;`

Wrong operation related to the type of terms	`// neither makes sense`
	`"StringA" > "StringB";`
	`true > false;`

Undefined variable	`$a == 1;` `//...and $a is not defined`
Variable already in use	`let $a = 1;` `let $a = "String";`

Quantified assertion variable not an array	`// $a is a single number`
	`let $a = 1;`
	`forall($b in $a, $b.length() > 1);`

Quantified assertion alias variable already in use	`let $a = 1;` `forall($a in $array, $a == 1);`

Incorrect function for a certain data type	`let $a = 1;`
	`$a.length() > 1;`
	`// unavailable for numbers`

Wrong number of parameters for a function	`$a.length("arg1",$arg2);`
Key not found in a DataObject	`let $b = $a/non_exist_key;`
Index out of bound in attributes evaluation	`$a/key[5];` `//...where max index is 4`

Error messages provide a textual description of the cause of the error as well as report the wrong token.

When the error is due to a conflict between different types of data, in addition to the information above, are also shown value and current state of each token involved.

Chapter 5

Examples

As said, *Specl* is a very flexible language and for this reason, it could be used in various contexts. In the following chapter, we continue our pragmatical approach to understanding and shows up some nice application.

We start from a simple example based on an XML document representing a bookstore's books. It is going to be presented in a didactic way, with easy statements and various alternatives; the goal is to show the elasticity of *Specl* and the straightforward use.

Then we illustrate the use of the language with two real world examples: inside a SOAP Handler as pre- and post-conditions validator, and for data management and data retrieval from response messages from a REST service.

These two cases are well explained with dedicated tutorials, available online at the site of Specl. Of course, projects' source code is freely accessible. The idea is to give all possible incentives and help to whom is approaching Specl. We really encourage to following this opportunity and play with tutorials and even with the language.

5.1 Didactic example: Bookstore

We use an imaginary small library described by the succeeding XML file, *book.xml*. Then a list of examples, touching all the aspects and functionality of the language, will show how to make declarations and assertions on the SDO translation of this file.

5.1.1 Book.xml

Listing 5.1: Main Example File Book.xml

```
<inventory>
    <name>InventoryABCD</name>
    <book>
        <year>2000</year>
        <title>Snow Crash</title>
        <authors>
            <author>Neal Stephenson</author>
        </authors>
        <publisher>Spectra</publisher>
        <isbn>0553380958</isbn>
        <price>15</price>
    </book>
    <book>
        <year>2005</year>
        <title>Burning Tower</title>
        <authors>
            <author>Larry Niven</author>
            <author>Jerry Pournelle</author>
        </authors>
        <publisher>Pocket</publisher>
        <isbn>0743416910</isbn>
        <price>6</price>
    </book>
    <book>
        <year>1995</year>
        <title>Zodiac</title>
        <authors>
            <author>Neal Stephenson</author>
        </authors>
        <publisher>Spectra</publisher>
        <isbn>0871131811</isbn>
        <price>7.50</price>
    </book>
</inventory>
```

The previous file is acquired as SDO.

Listing 5.2: SDO Tranformation - Book.xml

```
{
  Inventory =
    {
      name = InventoryABCD,
      book=
        [
          {
```

```
            year = 2000.0,
            title = Snow Crash,
            authors =
              {
                author = Neal Stephenson
              }
            publisher = Spectra,
            isbn = 0553380958,
            price = 15.0
          },
          {
            year = 2005.0,
            title = Burning Tower,
            authors =
              {
                author =
                  [
                    Larry Niven,
                    Jerry Pournelle
                  ]
              },
            publisher = Pocket,
            isbn = 0743416910,
            price = 6.0
          },
          {
            year = 1995.0,
            title = Zodiac,
            authors =
              {
                author = Neal Stephenson
              },
            publisher = Spectra,
            isbn = 0871131811,
            price = 7.5
          }
        ]
    }
}
```

5.1.2 First Assertion

Now we have to predicate our *Specl*'s assertions to verify certain conditions. The first assertion is the next one:

Listing 5.3: First Assertion

```
/inventory/book[1]/title == "Snow Crash";
```

The assertion says that the title of the first book (note the use of the predicate in square brackets) is equal to "Snow Crash" and, obviously, the result is true. The expression returns a string and correctly compare it with the string on the right side that we have written. Due to the flexibility of *Specl*, the same assertion can be written in different manners:

Listing 5.4: First Assertion - Alternative 1

```
/* Alternative 1 */
let $a = /inventory/book[1]/title;
let $b = "Snow Crash";
$a == $b;
```

In this case the result of the expression is stored in the variable `$a`, while our string is in `$b`. Then the assertion simply test the equality of the two.

Listing 5.5: First Assertion - Alternative 2

```
/* Alternative 2 */
let $a = /inventory/book[1];
$a/title == "Snow Crash";
```

Instead, in the alternative two, the value assigned to the variable `$a` is an SDO, i.e. an object containing other objects (as you can see from the XML file, a *book* holds nodes like *title*, *authors*, *publisher*, etc...). A representation of the SDO assigned to `$a` is

```
$a = {
  year = 2000,
  title = "Snow Crash",
  authors = {
    author = "Neal Stephenson"
  },
  publisher = "Spectra",
  isbn = "0553380958",
  price = 15
}
```

With the expression `$a/title` we get the string contained by the property title.

Navigational expressions are useful and can be defined with a good flexibility for reaching almost all the user's needs.

However, there are some exceptions: for example, you cannot apply a predicate to a variable or use a variable in steps that is not the first.

Just for example, the statement `$a[key==1]` is not allowed, because predicates are syntax to use in the context of path expressions. The correct way to resolve this is to use the predicate during the variable definition, otherwise, another solution is to use the function `get` when the object is an SDO or an array: `$a.get("key")`.

An example of the second erroneous use of navigational expression could be `/inventory/$a`: the variable could be used only in the first step (so when it could be treated as an SDO, from which the navigation path could start), but not in succeeding steps. Even in this situation, if the user wants to extract a certain value given the key, the solution could be the `get` function (`/inventory.get($a)`).

Note that the use of function `get` is possible when you apply it to an SDO or to an array; it has no sense when the cardinality of the object is one. The function is not intended to substitute the standard navigation, so it has to be used properly.

Listing 5.6: First Assertion - Wrong Predicates

```
/* Wrong use of predicate */
let $a = /inventory/book;
$a[1]/title == "Snow Crash";

/* Correct way */
let $a = /inventory/book;
$a.get(1).get('title') == "Snow Crash";
```

Another possible alternative to the first assertion is the next one, this time using the `get` function:

Listing 5.7: First Assertion - Alternative 3

```
/* Alternative 3 - Array */
let $a = /inventory/book/title;
$a.get(1) == "Snow Crash";
```

In this situation, the result of the declaration is an array with the titles of all the books found: `$a = ["Snow Crash", "Burning Tower" "Zodiac"]`.

Then, with the `get` function, you obtain the first element of the array: `"Snow Crash"` .

5.1.3 Second Assertion - Aggregated Functions

Listing 5.8: Second Assertion

```
let $books = /inventory/book;
```

```
let $authors = $books/authors/author;
exists($author in $authors,
       numOf($book in $books, $book/authors.contains($author)) > 1
);
```

In this case, we are looking for the authors that has written more than one book. The declarations returns, respectively, the array containing all books and the array with all the authors.

The assertion is the result of the nesting of two quantified assertions.

The `exists` assertion starts iterating over the elements of the `$authors` array, and, for each author, checks the condition. The condition is a comparison between another quantified assertion (this time a `numOf`) which returns a number that as to be greater than one to prove the condition of the `exists`.

The `numOf` iterate over all the books, obtains the authors of each book and, with the `contains` function, check if the actual author we are searching is inside that book's authors.

The external assertion has to be true; as already said, this is implicitly deducted from the grammar.

5.1.4 Third Assertion - Negated Assertion

Listing 5.9: Third Assertion

```
/* Negated Assertion */
let $books = /inventory/book;
!(
   exists($book in $books,
        $book/title.startsWith($book/publisher.substring(0,1)))
 );
```

This one assert that there is no books which title starts with the same letter as its publisher.

As you can see, the `exists` assertion is negated (inside brackets and preceded by the exclamation mark `!(...)`).

The `startsWith` function bears out the prefix of the book's title using as parameter the result of another expression: after getting the book's publisher, it takes the first letter by the use of the `substring` function.

5.1.5 Fourth Assertion - Numeric Aggregate Functions

Listing 5.10: Fourth Assertion

```
/* Test for arithmetic expressions (min, max, avg...) */
let $prices = /inventory/book/price;
let $minimum = min($price in $prices, $price > 0);
let $maximum = max($price in $prices, $price > 0);
let $average = avg($price in $prices, $price > 0);
let $product = product($price in $prices, $price > 0);
let $summation = sum($price in $prices, $price > 0);
$minimum == 6.00 && $maximum == 15 && $average == 9.5 ||
       $product > 675.0 && $summation < 28.5;
```

Here we declare many variables, each of them is containing a number, related to the relative expression: for example, `$minimum` contains the minimum price, in the range of the array `$prices`, that is greater than zero (this condition is very weak, so the expression simply means 'search the lowest price'). The same reasoning can be done for the other statements.

The assertions section presents logical operators AND (`&&`) and an OR operator (`||`): according to the priorities, the AND operators comes first, so the OR is applied as showed:

```
($min == 6.00 && $max == 15 && $avg == 9.5) || ($prod > 675.0 &&
    $sum < 28.5)
```

Different orders are indicated with the use of bracket, as for example in the next snippet:

```
$min == 6.00 && $max == 15 && ($avg == 9.5 || $prod > 675.0)  &&
    $sum < 28.5
```

5.1.6 Fifth Assertion - Numeric Expressions

Specl gives also the possibility to use and evaluate numeric expressions (see section 3.3.3.2)

```
let $exp = 1 + 2 * 4 % 5;
exp == 4;
```

The next is an example upon the bookstore example:

Listing 5.11: Fifth Assertion

```
let $first_price = /inventory/book[1]/price;
$first_price + /inventory/book[2]/price > /inventory/book[3]/price;
```

5.2 SOAP Handler Example

In this part, we develop a simple web service based on the example explained in the previous section, we named it *BookstoreWS*. In addition to this, we will build a client communicating with the service.

We are going to use *Specl* for validating conditions on the operations of the web service: on server side we check the incoming messages and evaluate the methods's preconditions; instead, on client side, we extract and manipulate data from response messages sent by the web service.

This is one of the possible use of *Specl* language: predicating assertions on a message inside a handler. A handler is an interceptor of SOAP messages, from which we can read and verify conditions over the content of the passing messages.

At first, we make a short overview of SOAP, then we start with the explanation of the example and of how utilize the language.

5.2.1 SOAP

SOAP stays for Simple Object Access Protocol and is the foundation of a web service protocol stack, a specification for exchanging structured information. It relies on XML. We introduce briefly, what a SOAP message and a handler are.

A SOAP message is a one-way transmission from a sender to receiver, it is composed as a mandatory extensible envelope expressing what features and services are represented in the message. The envelope contains an optional header and a required body (that, for instance, could be empty); the whole message is sent via HTTP protocol [8].

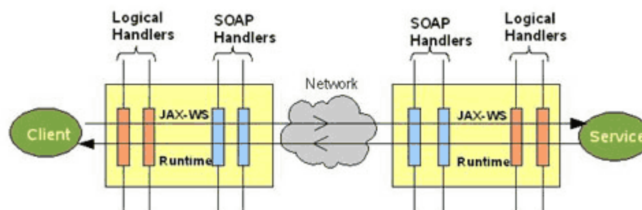

Figure 5.1: SOAP Architecture

The SOAP message architecture allows for SOAP intermediaries, which are nodes along the route from sender to receiver. An intermediary may inspect and even manipulate an incoming message before sending the message on its way toward the ultimate receiver. Java API for XML Web Services (JAX-WS a Java programming language API for creating web services in

XML format, i.e. SOAP) provides a handler framework that allows application code to modify incoming and outgoing SOAP messages.

JAX-WS gives two `Handler` interfaces, the `LogicalHandlers` and the `SOAPHandler`: former is protocol-neutral and has access only to the message payload in SOAP body, the latter is SOAP-specific and access to the entire SOAP message. Handlers are placed within a *handler chain*: a series of interceptors that, in a certain sequence, manipulate the message. It could be defined both on server side than on client side.

5.2.2 Server Side

5.2.2.1 Web Service Methods

It is a simple WS, offering information on books: title, authors, publisher, isbn and price.

BookstoreWS's methods are the following; they will not be discussed because their names are self-explanatory:

- `getBooksByAuthor(String)`
- `getBookByIsbn(String)`
- `getBooksByIsbnList(List<String>)`
- `getAllBooksTitle()`
- `getBooksNumberPerAuthor()`
- `getBooksByPublisherAndYearRange(String, int, int)`

5.2.2.2 Server SOAP Handler

The server side SOAP Handler we are building it is going to use *Specl* for checking constraints on the parameters passed to *BookstoreWS* from service's clients, for every incoming message.

We declare and initialize the analyzer with `SpeclAnalyzer analyzer = new SpeclAnalyzer();` command, then we obtain the XML Document of the SOAP message body and pass it to the analyzer that will use it as input file and parse it as an SDO (`analyzer.setXMLInput(soapBody.getOwner-Document());`).

Listing 5.12: BookstoreWS Handler Snippet

```
// handleMessage is a SOAPHandler interface method
//  for accessing to the underlying message
```

```
@Override
public boolean handleMessage(SOAPMessageContext context) {

        //...

        SpeclAnalyzer analyzer = new SpeclAnalyzer();
        SOAPMessage soapMsg = context.getMessage();
        SOAPBody soapBody = soapMsg.getSOAPBody();

        // the analyzer parse the body of the SOAP Msg to an SDO
        analyzer.setXMLInput(soapBody.getOwnerDocument());

        Node requestNode = soapBody.getFirstChild();
        String name = requestNode.getLocalName();

        if (name.equals("getBooksByAuthor")) {
                //...
        } else if(name.equals("getBookByIsbn")) {

        // ...and we check the called method's name and, for each
        // of them, execute the corrensponding verifications
```

At this point, we check the called method and evaluate *Specl* assertions on the passed arguments: this is equal to checking pre-conditions.

We tested every incoming message, for each method of the web service, with the subsequent assertions:

getBooksByAuthor(String):

```
let $author = /Envelope/Body/getBooksByAuthor/arg0;
$author.cardinality() == 0;
```

getBookByIsbn(String):

```
let $isbn = /Envelope/Body/getBookByIsbn/arg0;
$isbn.length() == 10;
```

getBooksByIsbnList(List<String>):

```
let $isbns = /Envelope/Body/getBooksByIsbnList/arg0;
forall($isbn in $isbns, $isbn.length() == 10);
```

getBooksByPublisherAndYearRange(String, int, int):

```
let $publisher =
       /Envelope/Body/getBooksByPublisherAndYearRange/arg0;
```

```
let $startYear =
       /Envelope/Body/getBooksByPublisherAndYearRange/arg1;
let $endYear =
       /Envelope/Body/getBooksByPublisherAndYearRange/arg2;
$publisher.cardinality() != 0 && $startYear > 1900
       && $endYear <= 2013;
```

Note that `getAllBooksTitle` and `getBooksNumberPerAuthor` methods has no parameters to verify.

5.2.3 Client Side

We developed a web service client for accessing to the *BookstoreWS*, and attached a handler for using *Specl* as post-condition validator and for manipulate SOAP messages.

5.2.3.1 Client SOAP Handler

The handler that we built, on the client side, has the task of intercepting the `getBooksNumberPerAuthorResponse` web service response message (that returns a Map with entries where keys are the authors name and values the number of book in the store written by that author).

With Specl, we find the author with the maximum number of book and we modify the SOAP message removing all entries of other authors, such that the client will receive only the name and the books count of the author with the greatest number of written book in the *BookstoreWS*.

The next piece of code contains the conditions we apply for obtaining the result just explained above (it is wrapped inside an handler method with a structure similar the one presented for the server side handler).

Listing 5.13: BoostoreWS Client Handler - Conditions

```
let $map_entries =
       /Envelope/Body/getBooksNumberPerAuthorResponse/return/map;
let $values = $map_entries/entry/value;
let $maximum = max($num in $values, $num > 0);
let $author = $map_entries/entry[value==$maximum]/key;
1==1; // we are only interested in variables
```

After the evaluation is done by calling the `evaluate(String)` method (where the passed argument is a string containing all the code in the previous Listing 5.13), we obtain the name of the author, with the maximum number of written book. Then, the command `analyzer.getVariable("$author")` returns the value of the `$author` variable of *Specl*.

When the author's name is retrieved, we can delete all the others from the SOAP message and deliver the altered message to the final destination.

```
Node map = soapBody.getFirstChild().getFirstChild().getFirstChild();
for(int i=0; i < map.getChildNodes().getLength(); i++){
  Node child = map.getChildNodes().item(i);
  if(!child.getFirstChild().getTextContent().equals(author)){
    map.removeChild(child);
    i -= 1;
  }
}
```

5.2.4 Results

SOAP Handler is a perfect context in which *Specl* could express its potentiality:

- it allows to navigate and obtain data from the SOAP message in a simpler way than getting lists of nodes, its children or siblings;
- handlers becomes easier to adopt and more useful;
- the user could predicate every assertion he/she wants, and verify the correctness of the requests to the web service.

Another point of view is that web service performance could be improved: handlers are exploited because, when *Specl* assess that a pre-condition is not satisfied, the web service would not consume time for a bad request and the requestor will receive a SOAP Fault message directly generated from the handler.

5.3 Twitter API's Example

The second real world example that we thought, it to use *Specl* for retrieving data from the JSON obtained via Twitter's REST API v1.1.

Data extraction and handling from JSON suffer lack of valid and suitable tools: solutions are few and poor.

This is one of the situations in which the high-level abstraction and general approach of *Specl* comes out: heterogeneous sources are mapped to service data objects and their complexity will be hidden. The simple navigation notation is the other advantage for reduce the effort.

As for the past example, we will introduce this case starting from an overview of the context and technologies in which we are involved. Later the example will be presented and explained.

5.3.1 REST

REST (Representation State Transfer) is ≪a style of software for distributed hypermedia systems; that is, system in which resources (text, graphics, and other media) are stored across a network and interconnected through hyperlink≫[8]. Requests (from clients) and responses (from servers) are built around the transfer of representations of resources. A representation of a resource is typically a document that captures the current or intended state of a resource.A RESTful web API is a web API implemented using HTTP and REST principles. It is a collection of resources, with four defined aspects:

- the base URI for the web API, such as http://example.com/resources/
- the Internet media type of the data supported by the web API. This is often JSON but can be any other valid Internet media type provided that it is a valid hypertext standard.
- the set of operations supported by the web API using HTTP methods (e.g., GET, PUT, POST, or DELETE)
- the API must be hypertext driven

5.3.2 Twitter

Twitter is an on-line social networking and microblogging service that enables users to send and read "tweets", which are short text messages. Twitter offers REST API [19].

The example is based on an application-only authentication which does not need a program to login as a specific user. In this way, the set of accessible information are the ones that are public (publicly accessible tweets, lists or user information), in a read-only manner (for example is useful for widgets and similar). For this sample it is enough, but for a deeper use of Twitter's capabilities is also offered access to OAuth signed requests. See the developers site for more information [19].

5.3.3 Code

In this section, we illustrate portions of code related to the various aspects covered.

5.3.3.1 Authentication Process

The procedure for Application-only authentication goes through the next steps:

1. Register with Twitter a new application to get a consumer key and consumer secret;
2. Combine the key and secret together and encode it with a base64 encoding;
3. Use that new encoded key to ask Twitter for a bearer token;
4. Get the token back from Twitter, save it and then supply it in the headers of additional requests.

At this point, we can require a bearer token from Twitter using the encoding keys inside `encodedCredentials`, obtained via a defined method `encKeys` (this encodes the consumer key and secret to create the basic authorization key).

This will be supplied in the Authorization header and passed in a SSL connection to Twitter's authentication URL. The token will be returned as a JSON object.

Listing 5.14: Twitter's Bearer Code Request Snippet

```
// Constructs the request for requesting a bearer token and
// returns that token as a string
private static String requestBearerToken(String endPointUrl)
       throws IOException {

  HttpsURLConnection connection = null;
  String encodedCredentials = encKeys(consumerKey,consumerSecret);
  try {

    // ...
       // connection and POST request to Twitter passing
       // the encodedCredentials
       // ...

       // Specl Analyzer extracts data from the response
       String response = readResponse(connection);
       String assertion =
         "let $token = /access_token;
         $token.cardinality() != 0 && /token_type == \"bearer\";";

       // ...
       // we call the analyzer for the evaluation,
       // then asks for $token variable
       bearerToken = analyzer.getVariable("$token");
```

The assertions simply check the existence of the object containing the token and its type. Note that we have used *Specl* and *SpeclAnalyzer* for handle the JSON and subsequently recall a *Specl* variable, using the `getVariable` method.

5.3.3.2 Fetch Data

After getting the *bearer token* we could supply it in the header of additional requests, according to our rights (remember that we use application-only authentication, so not all the possible requests are available).

We selected four different API endpoints and wrote many methods to make the appropriate calls to the service and to manage the returned data:

GET statuses/user_timeline [1]

≪Returns a collection of the most recent Tweets posted by the user indicated by the screen_name or user_id parameters.≫.

In addition to *screen_name* there's another argument for the method referring to the number of Tweets to analyze, *count*. Here is the method.

Listing 5.15: Getting Timelines Tweets Snippet

```
private static Object fetchTimelineTweet(String screen_name,
    int count) throws IOException {

  HttpsURLConnection connection = null;

  String baseURL = "https://api.twitter.com/1.1/statuses/
      user_timeline.json";
  String endPointUrl = baseURL + "?screen_name=" + screen_name
      + "&count=" + count;

  try {
    // connection and message sending

    // Parse the JSON response into a JSON mapped object
    // to fetch fields from.
    String response = readResponse(connection);
    String assertion =
      "let $tweets = /root[text != \"\"]/text;
      /root.cardinality() > 0;";

        // ...
```

[1] Details at `https://dev.twitter.com/docs/api/1.1/get/statuses/user_timeline`

```
        Object texts = analyzer.getVariable("$tweets");

  }

  // ...
```

With the method `readResponse` we obtain the string containing the response to the previously request, sent through the connection and with the earned *bearer token*.

Then we write the *Specl* assertion we want to check: we get the text of all the tweets that has non-empty text and store them in `$tweets`, verifying that the response has at least one object (`/root.cardinality() > 0`).

Looking more closely at the assertions we can see that there is the step `/root`: this is introduced because the value reported by `response` is a JSON array, in this cases *Specl* assign that array to a generic key, precisely `root`, leading to the a default SDO. If the evaluation gives positive result, the value corresponding to variable `$tweets` will be assigned to `texts` (the obtained object is an array).

GET search/tweets [2]

≪Returns a collection of relevant Tweets matching a specified query≫.

With this request, we are going to obtain some sort of tweets and return the *screen_name* of the authors of the tweet localized in Italy.

Listing 5.16: Tweets Search Snippet

```
private static Object fetchSearchedTweet(String query,
        int count) throws IOException {

  HttpsURLConnection connection = null;

  String baseURL = "https://api.twitter.com/1.1/search/tweets.
      json";
  String endPointUrl = baseURL + "?q=" + URLEncoder.encode(
      query, "UTF-8") + "&count=" + count;

  try {
    // connection and message sending

    String response = readResponse(connection);
```

[2]Details at `https://dev.twitter.com/docs/api/1.1/get/search/tweets`

```
String assertion =
  "let $italian_user = /statuses[lang==\"it\"]/user/
      screen_name;
  /statuses.cardinality() > 0;";

// ...

Object itu = analyzer.getVariable("$italian_user");

// ...
```

Note that the search is done according to the keywords passed as method's parameter `query`. Also the number of results could be set with the parameter `count`.

The flow is as in the previous case: get the response, fetch data with *Specl* and use them. The expressed assertions select the statuses where the language is Italian (`/statuses[lang=="it"]`), then, for each status, gets the *screen_name* of its user. The result is an array with the list of user's names. The variable we extract is `$italian_user`.

GET trends/place [3]

≪Returns the top 10 trending topics for a specific WOEID, if trending information is available for it.≫.

WOEID is the Yahoo! Where On Earth ID of the location to return trending information for.

In the next snippet, we obtain the trending topics for a passed WOEID; the program is expecting the one for Milan (it can be seen in the *Specl* assertions).

Listing 5.17: Trends in Milan Snippet

```
private static Object fetchTrendsByWOEID(String woeid)
       throws IOException {

  HttpsURLConnection connection = null;

  String baseURL = "https://api.twitter.com/1.1/trends
    /place.json";
  String endPointUrl = baseURL + "?id=" + woeid;

  try {
    // connection and message sending
```

[3] Details at https://dev.twitter.com/docs/api/1.1/get/trends/place

```
  String response = readResponse(connection);
  String assertion =
    "let $trends = /root/trends/name;"
    "exists($elem in $trends, $elem.startsWith(\"#\")) &&
      /root/locations/name == \"Milan\";";

  // ...

  Object trends = analyzer.getVariable("$trends");

  // ...
```

The assertions checks the existence of at least an hashtag (`exists ($elem in $trends, $elem.startsWith("#"))`) and that the name of the city is 'Milan' (`/root/locations/name == "Milan"`).

As for the *user_timeline*, the response is a JSON array, so the `root` key will be generated. This method returns `$trends` array.

GET application/rate_limit_status [4]

«Returns the current rate limits for methods belonging to the specified resource families.».

This GET method is related to the limitations of the use of APIs applied by Twitter.

Here we retrieve the state of our limits and calculate the percentage with respect to the thresholds.

Listing 5.18: Twitter's API Limits Snippet

```
private static Map<String, Object> fetchRateLimitStatus()
    throws IOException {

  HttpsURLConnection connection = null;

  String endPointUrl = "https://api.twitter.com/1.1/application
    /rate_limit_status.json?resources=search,statuses,trends";

  try {
    // connection and message sending

    // Parse the JSON response into a JSON mapped object
```

[4] Details at `https://dev.twitter.com/docs/api/1.1/get/application/rate_limit_status`

```
    // to fetch fields from.
    String response = readResponse(connection);
    String assertion =
      "let $status_remain =
        /resources/statuses/statusesuser_timeline/remaining;
      let $status_limit =
        /resources/statuses/statusesuser_timeline/limit;
      let $status_available =
        $status_remain / $status_limit * 100;
      let $search_remain =
        /resources/search/searchtweets/remaining;
      let $search_limit =
        /resources/search/searchtweets/limit;
      let $search_available =
        $search_remain / $search_limit * 100;
      let $trends_place_remain =
        /resources/trends/trendsplace/remaining;
      let $trends_place_limit =
        /resources/trends/trendsplace/limit;
      let $trends_place_available =
        $trends_place_remain / $trends_place_limit * 100;

      !($status_available <= 0 || $search_available <= 0 ||
        $trends_place_available <= 0);";

        // ...

    Object statusPercentage =
      analyzer.getVariable("$status_available");
    Object searchPercentage =
      analyzer.getVariable("$search_available");
    Object trendsPercentage =
      analyzer.getVariable("$trends_place_available");

        // ...
```

In this final case, we made simple arithmetic expressions for calculating the percentage of use of each resource, after getting the data from the JSON.

We calculate and assign to different variables the use percentage for each resource family (`statuses, friends, trends, help`). At this point, the assertions verifies that no limits has been exceeded.

```
!(\$status\_available <= 0 || \$search\_available <= 0 ||
       \$trends\_place\_available <= 0)
```

There's another little thing: the JSON returned by Twitter has some keys with non-alphanumeric characters (for example the `/` character in `/statuses/user_timeline`, conflicting with the navigation steps), in this situation *Specl* strips away not allowed characters and, in case of blank spaces, replace them with a '`-`' (so `/statuses/user_timeline` has become `statusesuser_timeline`).

The *Specl* interpreter tells, with a log message, the user when one of the previous cases is observed and signals the result of the replacement, but the developers has to keep this in mind when making assertions.

5.3.3.3 Result

Even in this context *Specl* express its value: the notation and the queries are simple and gives back easily the result requested, as XPath and XQuery does for XML.

This example supports the fact that *Specl* could be an optimal tool for retrieving and manipulating data from different sources thanks to SDOs and by using a simple and common syntax.

5.3.4 Web Site

We remember that all *Specl* sources are freely accessible at `http://github.com/samguinea/specl`. The site is intended to be a starting point for the language users. It presents *Specl* and its features, properly associated with examples supporting the understanding, and a full didactic example covering almost every aspect and capability.

The web site gives also references to the dedicated tutorials built and quickly explained in this text. The SOAP Handler WS, SOAP Handler Client and Twitter REST API projects' code are fully available and viewable on GitHub, at the links indicated on the main site. Each repository has also a detailed tutorial, guiding developers through the growth of the work.

Chapter 6

Conclusion

This thesis work is born from the needs of restart and extend WSCoL (Web Service Constrain Language).

The original version of the language was "web service-oriented"; with this work, we enriched and generalized the language to a higher level of abstraction. This gives to the language the capabilities for adaption in many different contexts.

This result is mainly imputable to the use of service data objects, which simplifies and unifies programming across different data sources and facilitates applications to handle and query data easily.

We have seen the flexibility of the language by applying it in examples of different natures and by obtaining a positive result in each of them.

A special mention must be made to XText and its community: the components generated using XText (like lexer, parser and validator) that were put inside the interpreter during its building, are very simple to integrate and configure. Furthermore, all the complexity and the produced tools used are hidden by few lines of code, which lets you obtain the AST of the parsed source file.

Specl and its interpreter could really be a convenient tool for developers, and we tried to push its use through the website, which is not only a place where you can find information about the language, but also a place where you can find guided examples, real-world application tutorials and the all source code of each project.

Appendix A

XText

XText is a highly configurable language development framework, i.e. a technology supporting the activity of developing language.

Given the XText grammar of the language, it provides for that language

- a parser
- an AST builder
- a validator
- a scoping and linking framework
- a code generator
- a serializer and code formatter
- an Eclipse editor with
  - quick fixes
  - template proposals
  - outline
  - hyper-linking
  - syntax coloring
  - project wizard
  - content assistance

Due to its flexibility, these tools, generated for supporting the IDE, are independent of Eclipse IDE or OSGi and can be used externally in any Java environment, in a standalone-way; as done in this thesis. XText provides a full implementation of a DSL running on the JVM.

We chose XText for these reasons:

- **powerful**: lets you build a simple DSL with few efforts (and also gives you an Eclipse-based IDE almost for free);
- **flexible and extensible**: features like the quick-fix, content assist and many others for the generated IDE, are all manageable and customizable;
- **reusable tools**: produced tools for the IDE can be used outside an Equinox configuration (called '*standalone*' setup), like parser, lexer and AST builder;
- **community**: wide and active support community [22];

XText is based on EMF (Eclipse Modeling Framework), and use ANTLR Parser Generator and Google Guice.

A.1 EMF

EMF is ≪a modeling framework and code generation facility for building tools and other applications based on a structured data model≫[18].

Starting from a defined model, this framework generates a set of the corresponding Java classes. In XText it is widely used, but mainly for the grammar representation and for the abstract syntax tree of the language.

An EMF model is in essence a UML class diagram, with similar abstractions of classes and methods. The meta-model used for describe EMF models is called Ecore, that is defined in itself (so it is a meta-meta-model too). Exists four type of Ecore class:

- **EClass**: represents a class, with zero or more attributes and zero or more references.
- **EAttribute**: represents an attribute which has a name and a type.
- **EReference**: represents one end of an association between two classes
- **EDataType**: represents the type of an attribute.

Summing up, an EMF model is an instance of an Ecore model (and then called core model) and is made up of instances of EObjects which are connected. A set of EClasses is contained in a so-called EPackage, and inside EClasses there is a set of EAttribute, EReference and EDataTypes. All these model elements (such as EPackage, EClasses, EDatatype, EReference. and

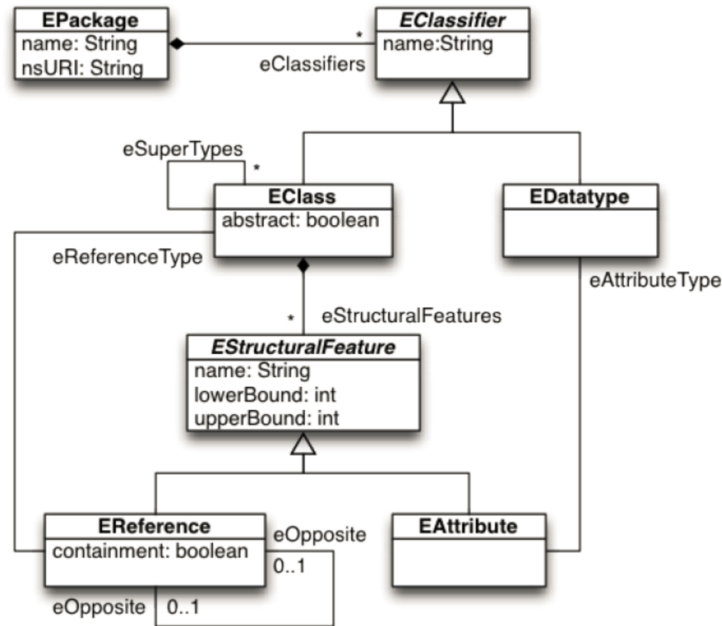

Figure A.1: Ecore Classes

EAttribute) extend EObject. EObject is the base of all model elements in the same way that all java classes extend Object.

Core models use XMI (XML Metadata Interchange) for its representation, and could be defined using annotated Java, XML Schema or UML class diagram. Once the EMF meta-model is specified you can generate the corresponding Java implementations classes from this model. EMF provides the possibility that the generated code can be safely extended by hand.

In XText the ECore model is generated from the user defined DSL as a list of rules of the language. The grammar file is a plain text file with *.xtext* filename extension, and the grammar within is defined with a BNF-like syntax. The grammar file not only defines the syntax of a language, but also how to map it to an ECore model and how to build the AST (the XGL - XText Grammar Language - is used for this purpose). There are four type of rules specified with XGL:

Table A.1: XGL Grammar - EMF Mappings

Rule type	Returns	Used by
Parser Rules	EClass	Parser
Enum Rules	EEnum	Parser

Datatype Rules	EDatatype (EString is implied if none has been explicitly declared)	Parser
Terminal Rules	EDatatype	Lexer

Each of them are mapped as follow

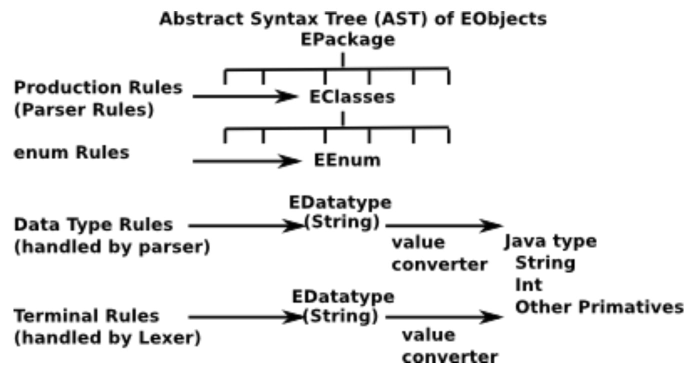

Figure A.2: XGL-Ecore Mapping

For a deeper explanation please read the XText Documentation.

A.2 ANTLR

ANTLR is used for generating the top-down LL(k) parser for the user's language [1].

During the generation phase of the language, the XText grammar (XGL) is converted to an ANTLR grammar and then executed for achieve lexer and parser to use into the framework. As previously said, in the grammar conversion are also passed all the information for construct the AST with EMF.

A.3 Google Guice

A dependency injection framework, that use code annotation for provide a functionality from another component without resolve all its dependencies [4].

XText use it for the composition of the various component of the language, giving move flexibility, reuse of code and maintainability.

DI is at the base of the POJO "standalone" use of XText: with the `Setup`-implementation of the language and its method `createInjectorAndDoEMFRegistration()` the initialization of the infrastructure is done. The setup method return an `Injector`, by which can be obtained the parser

and the components of the language; so it let use the EMF's APIs for load and store model of languages.

A.4 XText Generator

XText uses the MWE2 (Modeling Workflow Engine 2) DSL to configure the generation of its artifacts and components; tweaking the default .mwe2 file, provided by XText, could generate support for further features.

A workflow is a series of components that interact with each other (handles EMF resource, and builds every artifact needed). During its execution, will be generated artifacts related to the UI editor for the DSL but also, and most important, derives an ANTLR specification from the user's grammar, including actions for the AST (the classes for its nodes will be produced using the EMF framework).

The generator infrastructure relies on the Generation Gap Pattern which deals with the problem of separating the code that can be customized from the one generated. In XText the latter is placed in the src-gen folder, while everything could be modified inside src folder.

The main component is *Generator* that contains a list of fragments, each of them corresponds to a language element:

GrammarAccessFragment
: the grammar file (*Specl.xtext*) is transformed in an EMF, from this will be generated `SpeclGrammarAccess.java` that offers API for grammar access, useful for the next fragments;

EcoreGeneratorFragment
: the most important fragment, builds the grammar meta-model for the construction of the AST;

ParseTreeConstructorFragment
: used for the serialization process (the opposite of the parsing, from AST to tokens);

ResourceFactoryFragment
: generates a Factory of `XTextResource` (same concept of `Resource` and `ResourceSet` of EMF) that gives APIs for using all the XText's functionality (it is fundamental for a in Standalone configuration, as in this thesis);

XtextAntlrGeneratorFragment
: produces the ANTLR grammar, lexer and parser of the language;

JavaValidatorFragment
the corresponding class allows to configure a model validator, that signals warnings and errors in the model specification;

InportURIScopingFragment and SimpleNameFragment
generates components for the scope handling;

FormatterFragment
code formatter policy;

LabelProviderFragment
label provider (a stick and a name for each grammar element);

TransformerFragment
manage the outline visualization;

QuickfixProviderFragment
supports quickfixes in the IDE;

JavaBasedContentAssistFragment
generates provision for proposal provider and content assist in the IDE;

Etc...
(refer to XText Documentation for details)

The execution of the workflow produces the code of the components that forms the infrastructure of the language and the tools for its IDE.

A.5 Interact with XText models programmatically

As said before in this thesis we are not interested in the generation of the Eclipse-based IDE for our language, but to export the component of the language architecture and use them in a standalone way inside a Java project for make an interpreter.

EMF models can be persisted by the means of a so called `Resource`. XText languages implement the `Resource` interface which is why you can use the EMF APIs to load a model into memory (and also save them):

A.5.1 Loading and using a model

The first step initializes the language infrastructure to run in standalone mode.

```
new SpeclStandaloneSetup().createInjectorAndDoEMFRegistration();
```

In fact, EMF is designed to work in Eclipse and therefore makes use of Equinox extension points in order to register factories and the like. In a vanilla Java project there is no Equinox, hence we do the registration programmatically. The generated `SpeclStandaloneSetup` class does just that. You do not have to do this kind of initialization when you run your plug-ins within Eclipse, since in that case the extension point declarations are used. The other thing the `StandaloneSetup` takes care of is creating a Guice injector.

A.5.2 Create a ResourceSet

Now that the language infrastructure is initialized and the different contributions to EMF are registered, we want to load a Resource. To do so we first create a `ResourceSet`, which as the name suggests represents a set of Resources. If one `Resource` references another `Resource`, EMF will automatically load that other `Resource` into the same `ResourceSet` as soon as the cross-reference is resolved. Resolution of cross-references is done lazy, i.e. on first access.

```
ResourceSet rs = new ResourceSetImpl();
```

A.5.3 Loading a resource

The next step loads the `Resource` using the resource set. We pass in a URI which points to the file in the file system.

```
Resource resource =
       rs.getResource(URI.createURI("./test.specl"), true);
```

EMF's URI is a powerful concept. It supports many different schemes to load resources from file system, web sites, jars, OSGi bundles or even from Java's classpath. In addition, if that is not enough you can come up with your own schemes. Also, a URI can not only point to a resource but also to any EObject in a resource. This is done by appending a URI fragment to the URI. The second parameter denotes whether the resource should automatically be loaded if it was not already before. Alternatively, we could have written:

```
Resource resource =
       rs.getResource(URI.createURI("./test.specl"), false);
resource.load(null);
```

The load method optionally takes a map of properties, which allows defining a contract between a client and the specific implementation. In

XText, for instance, we use the map to state whether cross-references should be eagerly resolved. In order to find out what properties are supported, it is usually best to look into the concrete implementations. That said, in most cases you do not need to pass any properties at all.

A.5.4 Referencing the root

The last step assigns the root element of the model to a local variable.

```
EObject eobject = resource.getContents().get(0);
```

Actually, it is the first element from the contents list of a Resource, but in XText a `Resource` always has just one root element.

A.5.5 Working with the Grammar

Furthermore, the grammar is represented as an EMF model and can be used in Java. In fact, each node of the node model references the element from the grammar which was responsible for parsing or lexing that node:

```
Model model = (Model) eObject;
CompositeNode node = NodeUtil.getNode(model);
ParserRule parserRule = (ParserRule) node.getGrammarElement();
assertEquals("Model", parserRule.getName());
```

Appendix B

Specl Grammar

In this appendix we present the whole grammar of Specl, written in XGL (XText Grammar Language). XGL have a syntax similar to ANTLR, for details on XText read the Appendix A or take a view of its documentation [6].

Listing B.1: Specl Grammar

```
/*****************************************************************
  Specl is a special-purpose monitoring language, by which it's
      possible to express directives on both functional and non-
      functional properties; which are essentially pre- and post-
      conditions on the interaction with external partner services.
  It is suitable to express general dependability properties as
      safety, integrity, availability and reliability.
*****************************************************************/
grammar it.polimi.specl.Specl with org.eclipse.xtext.common.
    Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate Specl "http://www.polimi.it/specl/Specl"

Model:
  declarations+=Declaration*
  assertionSet=AssertionOr ';';

// Assign to the variable a value (a String, Double, Boolean or SDO)
Declaration:
  'let' var=Variable '=' assert=(Assertion |
      AssertionQuantifiedBoolean) ';';

// Assertion results in logical operation OR
AssertionOr returns Assertions:
```

```
  AssertionAnd ({AssertionOr.left=current} '||' right=AssertionAnd)
      *;

// Assertion results in logical operation AND
AssertionAnd returns Assertions:
  AssertionHP ({AssertionAnd.left=current} '&&' right=AssertionHP)*;

// Assertion results in logical operation with highest priority (NOT
     and parentheses)
AssertionHP returns Assertions:
  ( => AssertionForm | AssertionNot | AssertionBraced);

// Assertion with parentheses
AssertionBraced:
  '(' innerFormula=AssertionOr ')';

// Assertion with logical operator NOT
AssertionNot:
  '!' '(' innerFormula=AssertionOr ')';

// Kinds of Assertion: with comparison and without (2nd and 3rd case
     are for boolean comparison and their result is implicitly 'true
    ')
AssertionForm:
  (=> AssertionStdCmp | AssertionQuantifiedBoolean |
      AssertionBoolean);

// Assertions making a standard comparison between values
AssertionStdCmp:
  leftAssert=Assertion op=Rop rightAssert=Assertion;

// Assertion
Assertion:
  Expression | AssertionQuantifiedNumeric | '[' values=Values ']' |
      boolean=BOOLEAN;

// Assertion with boolean result
AssertionBoolean returns Assertion:
  steps+=Step+ ('.' functions+=Function)* '.' functions+=
      FunctionBoolean;

// Assertion with quantifiers for boolean results
AssertionQuantifiedBoolean returns AssertionQuantified:
  quantifier=BoolQuantifier '(' alias=Variable 'in' var=Variable ','
       conditions=AssertionOr ')';
```

```
// Assertion with quantifiers for numeric results
AssertionQuantifiedNumeric returns AssertionQuantified:
  quantifier=NumbQuantifier '(' alias=Variable 'in' var=Variable ','
       conditions=AssertionOr ')';

// Steps describing the navigation path through an SDO
Step:
  '/' name=ID ('[' predicate=Predicate ']')? | placeholder=Variable;

// Attribute for SDO navigation: according to the operation checks
    the searched value, if possible
Predicate:
  property=ID (op=Rop numberValue=NUMBER | op=StringRop strValue=
      STRING | op=Rop varValue=Variable) | number=NUMBER | var=
      Variable;

// Function
Function:
  name=ID '(' (params=Values)? ')';

// Function that gives boolean result
FunctionBoolean returns Function:
  name=('contains'|'startsWith'|'endsWith') '(' (params=Values)? ')
      ';

Expression:
  Addition;

Addition returns Expression:
  Multiplication (({Plus.left=current} '+' | {Minus.left=current}
      '-') right=Multiplication)*;

Multiplication returns Expression:
  PrimaryExpression (({Multi.left=current} '*' | {Div.left=current}
      '/' | {Rest.left=current} '%') right=PrimaryExpression)*;

PrimaryExpression returns Expression:
  Value | '(' Addition ')';

// List of values
Values:
  value+=Value (',' value+=Value)*;

// Value could be a constant (alphanumeric or numeric) or obtained
```

```
    by an SDO's property
Value:
  Constant | steps+=Step+ ('.' functions+=Function)*;

Constant:
  number=NUMBER | string=STRING;

// Variable (NumbQuantifier is added for give the possibility to use
     'max', 'min', etc. as variables name)
Variable:
  '$' ID;

BoolQuantifier:
  'forall' | 'exists';

NumbQuantifier:
  'numOf' | 'sum' | 'avg' | 'min' | 'max' | 'product';

// Relational Operation for Strings
StringRop:
  '==' | "!=";

// Relational Operation
Rop:
  StringRop | '<' | '<=' | '>' | '>=';

terminal BOOLEAN returns ecore::EBoolean:
  'true' | 'false' | 'TRUE' | 'FALSE';

terminal NUMBER returns ecore::EDouble:
  ('-')? ('0'..'9')* ('.' ('0'..'9')+)?;

terminal INT returns ecore::EInt: // override of the default rule
  'this one has been deactivated';
```

Bibliography

[1] ANTLR. ANTLR. `http://www.antlr.org/`.

[2] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML, 2004.

[3] Gary T. Leavens's JML group. JML - Home Page. `http://www.eecs.ucf.edu/~leavens/JML//index.shtml`.

[4] Google. Guice. `http://code.google.com/p/google-guice/`.

[5] Itemis. XText - Project Home Page. `http://www.xtext.org/`, November 2013.

[6] Itemis. XText 2.4.3 - Documentation, September 2013.

[7] Jordi Cabot and Martin Gogolla. M.: Object Constraint Language (OCL): A Definitive Guide. In *SFM 2012. LNCS*, pages 58–90. Springer, 2012.

[8] Kalin, M. *Java Web Services: Up and Running*. O'Reilly Media, 2009.

[9] L.Baresi and S.Guinea. Toward Dynamic Monitoring of WS-BPEL Processes. In Springer, editor, *ICSOC 2005: 3rd International Conference on Service Oriented Computing*, volume 3826 of *Lecture Notes in Computer Science*, pages 269–282. Springer, 2005.

[10] L.Baresi and S.Guinea. A Dynamic and Reactive Approach to the Supervision of BPEL Processes. In *ISEC 2008: 1st Indian Software Engeneering Conference*, pages 39–48, 2008.

[11] L.Baresi and S.Guinea. Self-Supervising BPEL Processes. *IEEE Trans. Software Eng.*, 37(2):247–263, 2011.

[12] Marlo Verdesca and Jaeson Munro and Michael Hoffman and Maria Bauer and Dinesh Manocha. Object Management Group. Object Constraint Language Specification (in UML 1.4). In *Interservice/Industry*

Training, Simulation and Education Conference (IITSEC, page URL-http://www.omg.or, 2005.

[13] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.

[14] Bertrand Meyer. Applying Design by Contract. *IEEE Computer*, 25:40–51, 1992.

[15] Microsoft Research. Spec#. http://research.microsoft.com/en-us/projects/specsharp/.

[16] OASIS Open CSA. OASIS Open CSA - SDO. http://www.oasis-opencsa.org/sdo.

[17] Robert Bruce Findler and Matthias Felleisen. Contract Soundness for Object-Oriented Languages. In *OOPSLA '01 Conference Proceedings, Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–15. ACM, 2001.

[18] The Eclipse Foundation. EMF - Home Page. http://www.eclipse.org/modeling/emf.

[19] Twitter. Twitter - Dev Page. https://dev.twitter.com/.

[20] W3C. XPath - Documentation. http://www.w3.org/TR/xpath20/.

[21] W3C. XQuery. http://www.w3.org/TR/xpath20/.

[22] XText. XText - Community Forum. http://www.eclipse.org/forums/index.php?t=thread&frm_id=27.