Mauno Rönkkö | Markus Stocker | Mats Neovius |
Mikko Kolehmainen | Luigia Petre

Programming by Construction

Turku Centre *for* Computer Science

TUCS Technical Report
No 1092, November 2013

Programming by Construction

Mauno Rönkkö
University of Eastern Finland, Department of Computer Science
Yliopistonranta 1E, 70211 Kuopio, Finland
mauno.ronkko@uef.fi

Markus Stocker
University of Eastern Finland, Department of Computer Science
Yliopistonranta 1E, 70211 Kuopio, Finland
markus.stocker@uef.fi

Mats Neovius
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
mneovius@abo.fi

Mikko Kolehmainen
University of Eastern Finland, Department of Computer Science
Yliopistonranta 1E, 70211 Kuopio, Finland
mikko.kolehmainen@uef.fi

Luigia Petre
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
luigia.petre@abo.fi

TUCS Technical Report
No 1092, November 2013

Abstract

Modern cloud services use standard interfacing technologies, such as SOAP and REST. What is common to all these technologies is that they separate the service implementation and internal data structure from the service invocation and parameter structure. To support interoperability, the service invocation is often described by using XML and it is made available to the client. Because of this, the same concepts are expressed in at least three different contexts. For instance, the parameter structure that is expressed with XML has to be replicated and represented with a programming language, such as Java, that is used to implement the service. The client is then free to choose the language for calling the service. Needless to say, this means that there is an inherent synchronization issue; the XML description, the service implementation, and the client implementation must speak of the same concepts. There are obviously tools that translate one representation to another, such as JAXB for Java and XML. In this way, however, the translation processes not only the concepts of interest, but also other semantic artifacts found in the source representation. Consequently, there is a significant semantical distance between the concepts, their source representation, and their target representation. This distance is a major source of errors in the development process. Here, we approach this problem from a novel viewpoint. In short, we separate the concept from its semantic context, whereby we can express the concept directly in any given semantic context. In this way, there is no translation as described above. To achieve this, and as the main contribution, we formalize here this approach of programming by construction. The basis of the formalization is a grammar for a simple language that describes the definition of concepts and contexts. We then define the interpretation for the language, including the actual construction, which is formalized as a term rewriting process. We also provide here an open-source implementation of an interpreter for the language in Java. The interpreter is called Mill, and we illustrate its use with a simple case study involving design patterns and Java.

Keywords: Formal language, Service oriented programming, Cloud services, Design patterns

TUCS Laboratory
Distributed Systems Laboratory

1 Introduction

Standard interfacing technologies, such as SOAP [1] and REST [2, 3], have largely contributed to the emergence of web services [1] and cloud computing. These interfacing technologies allow platform independent invocation of services with well defined parameters and data structures. To achieve this, services and their parameters are typically described by using XML. XML provides a standard, structured representation for information. In particular, with XML, one can define element types in a standard way by using XSD descriptions [4].

Although the use of XML, and XSD, provide the means for platform independent descriptions, the services and the client applications need eventually be implemented. Technically speaking, this means that the same concepts expressed in XML and XSD documents need to be represented by using some programming language. Clearly, both representations need to be synchronized. As this synchronization is the heart of the development process, there are tools that translate one representation to another. For instance, JAXB [5] supports translation of typed structures expressed as XSD documents into classes and class structures expressed in Java language. As synchronization of the two presentations is required, the translation process occurs whenever the XSD representation changes. In addition, as XSD describes only the structure of the data, there is usually a need for some manual programming each time a translation occurs.

There are two main issues with the translation processes. The first issue has to do with the semantic distance between the different representations, and the second issue has to do with the actual intention of the translation. Together these two issues constitute the main source for problems in the development process. From these two, the core issue is the semantic distance. For instance, although XSD is used for defining data structures, it differs both intentionally and methodologically from how structures are constructed in Java. This means that translation becomes ambiguous. Even though the XSD document defines a specific structure, it does not define the intent of the structure that would help in determining what would be the corresponding structure in Java. Because of this, there are alternative representations of the structure in Java, and the translation process picks one. Sometimes the mismatch of the automatic choice is revealed only afterwards, when the system is already in use. Then, an error reveals that the structure intended in an XSD document did not fully match the semantics of the Java structure obtained with the automatic translation. In the worst case, this results in manual programming of a partial data structure, which then breaks also the synchronization between the XSD description and the Java implementation.

Here, we approach this problem of semantic distance and lack of intention from a alternative viewpoint. Rather than translating a concept from one representation to another, we separate the concept from its semantic context and express them both in a language and representation independent manner. In this way, there can be multiple semantic contexts for any defined concept. Furthermore, the con-

cept can then be expressed directly in a given context. Consequently, there is no such translation process as discussed above, and we also avoid the translational issues discussed above.

To achieve this, and as the main contribution, we formalize here this approach that we call "Programming by Construction". The basis of the formalization is a grammar [6] for a simple language that describes the definition of concepts and contexts. This language does not restrict the target language. Consequently, constructs can be tailored not only for any programming languages, but also for any definition and modeling languages, such as XML and UML. With respect to the grammar, we then define the interpretation for the language, including the actual construction process. The construction is technically formalized as a term rewriting process [7]. We also provide here an example implementation for the interpreter, called Mill. Mill is written in Java, and the source code is listed as open-source at the end of this report.

Programming by construction is related to parameterized batch processing and code generation. Both of these have existed for a very long time, and they have been used extensively [8]. Typically, however, code generators are designed and implemented for fixed tasks, such as constructing and accessing databases, performing unit testing, constructing user interfaces, creating specific software layers, etc. With programming by construction, there is no specific task. Instead, contexts fully define the use case and semantics for given concepts. Programming by construction is, thus, a radically different approach. It is, in a way, a declarative programming approach[9].

Programming by construction is also related to use of ontologies, as they both speak of terms, concepts, and semantics. There is, however, a clear separation between the two in terms of intentions and use cases. Ontologies are used to formally capture concepts and their relations with the intent of algorithmic deduction and consistency checking. With ontologies, there is no intent of code generation initially. In contrast, programming by construction solely focuses on construction by expressing concepts in well defined contexts. Thus, the application may be program generation, schema generation, etc. Consequently, with programming by construction, there is no intent of deduction initially. From this point of view, use of ontologies and programming by construction are two, very complementary technologies that could greatly benefit from each other.

We illustrate the use of programming by construction here with a simple case study. It involves representation of person data in two formats, XML and CSV. Furthermore, the case study shows how the same concepts used to define these two representations, can also be used to construct a Java program capable of translating data between the two formats.

This technical report is organized as follows. In Section 2, we discuss in detail, what we mean by concepts, contexts, and construction. In Section 3, we present a grammar for the language that is used to define concepts and contexts as well as to construct representations with them. We also define the interpretation

for the language in that section. In Section 4, we discuss the implementation of interpreter for the language, called Mill. In Section 5, we present the case study. In Section 6, we provide a conclusion with discussion. Finally, the open-source listing for the interpreter in provided in Appendix.

2 Concepts, contexts, and construction

Programming by construction is in a way a declarative programming approach[9]. We define concepts and contexts, where contexts describe the semantic interpretation for the concepts. Construction occurs, when the concepts are expressed in some context. To make all this concrete, we discuss each of these elements in detail, next.

2.1 Concepts

A concept is a term that describes an element of interest. It usually has some inner structure that can be understood to be composed of sub-concepts. Thus, a concept defines a binding between outer and inner concepts; however, this binding does not define or imply any further semantics.

Definition 1 Concept. *A concept is expressed as:*

$$\alpha : T$$

where α is a label denoting the name of the concept, and T denotes the content of the concept. T is delimited by a white space, or it can be a string delimited by braces, "{" and "}".

Thus, we use ":" to express a relation between a name and a content. In the simplest case, a concept relates a name to a content. For instance, the following concept relates the name "age" to the content "43":

```
age:43
```

Clearly, one could understand the content of the above concept as "43 years", but this interpretation is not mandated by the concept. The relation above could equally denote "43 months". Moreover, the concept does not express whose age it is, if it at all refers to a human. Strictly speaking, we cannot even infer that, as the name of the concept, "age", has no semantics.

As mentioned in Definition 1, the content of a concept may also be delimited by braces. Then, the content of a concept can have more structure to it. For instance, the following example relates several concepts to a concept called "person":

```
person:{ age:43 hobby:reading }
```

In this case, the only implication that is valid is that the concepts "age" and "hobby" are related to the concept "person". Nothing else can be said about them. For instance, the above could be understood that some person is of age 43 and his/her hobby is reading. It could also refer to a class of persons, who are of age 43 and their hobby is reading, and so on. Again, the proper interpretation, the semantics, not part of the concept, it belongs to a context.

2.2 Contexts

A context is a term that describes the semantics for concepts. Thus, a single context may refer to many concepts. As there can be many ways to express semantics, we do not address the validity of the semantics represented by the contexts. Instead, the binding of concepts to contexts is done by using term rewriting. For this purpose, the context must indicate in the content where specific concepts apply. This is done by using tags in the content. Thus, the term rewriting occurs only within the tagged part of the content of the context.

A context may, thus, contain one or more tags. A tag consists of a label, a separator, and a body. The label is used for matching the tag with the concept to be applied. The separator defines how the body of the tag changes, if the rewriting occurs more than once. The body of the tag is the content subject to rewriting with the content of the concept.

Definition 2 Tag. *A tag is expressed as:*

$$<:\tau<\#C\#>B:>$$

where τ is a label denoting the name of the tag, C is any string denoting the separator in the tag, and B is any string denoting the body of the tag. Both C and B may also contain matching separators of the tag denotation, "$<:$", "$:>$", "$<\#$", and "$\#>$".

For instance, the following tag is identified by the name "ATTRIBUTE". It has a white space as a separator string, and its body string is "private TYPE NAME;".

```
<:ATTRIBUTE<# #>private TYPE NAME;:>
```

It should be noted that anything in the body string can be subject to rewriting. In this case, however, the intent is that "TYPE" and "NAME" are placeholders and to be rewritten with the content of similarly named concepts.

Definition 3 Context. *A context is a concept containing one or more tags.*

For instance, the following context is called “immutable-class”, and it has two kinds of tags: “CLASS” tags and “ATTRIBUTE” tags. In short the context defines the semantics for an immutable Java class for concepts “CLASS” and “ATTRIBUTE”.

```
immutable-class : {
  class <:CLASS<##>NAME:>
  {
    <:ATTRIBUTE<##>private TYPE NAME;
    :>
    public <:CLASS<##>NAME:>(<:ATTRIBUTE<#, #>TYPE NAME:>)
    {
    <:ATTRIBUTE<##>  this.NAME=NAME;
    :>}

    <:ATTRIBUTE<##>  public TYPE NAME() {return NAME;}
  :>}
}
```

Below is another example of a context that is using the same tags as the example above. However, the following context, called “xsd-type” provides an XML type semantics for the exact same concepts as the “immutable-class” context above does for Java.

```
xsd-type : {
  <complexType name="<:CLASS<##>NAME:>Type">
    <sequence>
<:ATTRIBUTE<##>
      <element name="NAME" type="TYPE"></element>
:>    </sequence>
  </complexType>
}
```

2.3 Construction

Construction occurs, when concepts are expressed in contexts. We formalize the construction by using term rewriting. For this purpose, we need to define first, what we mean by textual substitution, sequential textual substitution, tag rewriting, contextual rewriting, and tag removal.

Definition 4 Textual substitution. *Consider a concept $\alpha : T$ and a string S. Then, we denote the textual substitution of all the occurrences of α in S with T by*

$$(\!|S\ \alpha : T|\!)$$

The substitution takes places from left to right so that the substitution value T is never reconsidered.

Thus, the textual substitution, as defined above, terminates always. As an example, consider the concept $A{:}aA$ and a string A_B_A. Then, $(\!|A\_B\_A \;\; A : aA|\!)$ yields aA_B_aA.

Definition 5 Sequential textual substitution. *We denote the sequential textual substitution by*

$$(\!|S\ \alpha_1 : T_1\ ...\ \alpha_n : T_n|\!)$$

It is computed as $(\!|...(\!|S\ \alpha_1 : T_1|\!)\ ...\ \alpha_n : T_n|\!)$.

Note that we cannot define the sequential textual substitution with Kleene star. Such a definition would implicate the substitution taking place with respect to all possible concepts in some given set of concepts. In our case, there is no such set of concepts. Instead, in sequential textual substitution, each concept for substitution must be explicitly stated, and the substitution is performed only with respect to those concepts in the given order. This also means that substitution may occur multiple times with respect to the same concept. In such a case, the order of appearance makes a difference.

Definition 6 Tag rewriting. *Consider a tag* $<{:}\tau<\#C\#>B{:}>$*, where* τ *is the name of the tag,* C *is the separator string, and* B *is the body string as defined earlier. Also, consider a list of concepts* $\alpha_1 : T_1\ ...\ \alpha_n : T_n$*. Then, we denote tag rewriting by*

$$([\![<{:}\tau<\#C\#>B{:}>\;\;\alpha_1 : T_1\ ...\ \alpha_n : T_n]\!])$$

and it is computed as $(\!|B\ \alpha_1 : T_1\ ...\ \alpha_n : T_n|\!)\ <{:}\tau<\#\#>CB{:}>$.

Thus, a tag rewrite performs a textual substitution on the body of the tag with respect to given concepts. In addition, tag rewriting does not remove the tag; rather it duplicates and modifies it by concatenating the separator string with the body string. In this way, the tag rewriting may take place multiple times, and the separator string becomes active for all subsequent tag rewrites. As an example, consider the tag:

```
<:ATTRIBUTE<#/**/ #>private TYPE NAME;:>
```

and a list of concepts (in the given order):

```
NAME:age  TYPE:int
```

Then, the tag rewriting yields the outcome

```
private int age;<:ATTRIBUTE<##>/**/ private TYPE NAME;:>
```

Thus, after the rewriting occurs the modified tag remains as part of the content.

Definition 7 Contextual rewriting. *Consider a string* S*, a tag* $<{:}\tau<\#C\#>B{:}>$*, and a list of concepts* $\alpha_1 : T_1\ ...\ \alpha_n : T_n$*. Then contextual rewriting of tags* τ *in string* S *with concepts* $\alpha_1...\alpha_n$ *is denoted by*

$$[\![S(\tau)\;\;\alpha_1 : T_1\ ...\ \alpha_n : T_n]\!]$$

and it is computed as follows:

1) If S contains no tag, the result is S as is.

2) If S contains a tag, S is of form $P <: \tau < \#C\# > B :> U$, where P is the prefix string not containing the tag, and U is the suffix string that may contain the tag. Then, the result is computed as

$$P(\![<:\tau<\#C\#>B:> \;\; \alpha_1 : T_1 \; ... \; \alpha_n : T_n]\!)[\![U(\tau) \;\; \alpha_1 : T_1 \; ... \; \alpha_n : T_n]\!]$$

Thus, the contextual rewriting defines a sequential, but terminating application of tag rewriting for a string. In effect, it describes the rewriting of all tags of the given form in the string from left to right.

Definition 8 Tag removal. *The removal of all tags from a string S is denoted by*

$$\|\!| S |\!\|$$

and it is computed as follows:

1) If S contains no tag, the result is S as is.

2) If S contains a tag, S is of form $P <: \tau < \#C\# > B :> U$, where P is the prefix string not containing the tag, and U is the suffix string that may contain the tag. Then, the result is computed as $P\|\!|U|\!\|$.

For instance, consider a string with one tag:

```
class Person {
  private int age;<:ATTRIBUTE<##> private TYPE NAME;:>
}
```

After tag removal the string becomes:

```
class Person {
  private int age;
}
```

Definition 9 Construction. *Consider a context of form $\gamma \; : \; S$, and concepts of form $\tau_1 : \{\alpha_1 : T_1 \; ... \; \alpha_n : T_n\} \; ... \; \tau_t : \{\beta_1 : T_1 \; ... \; \beta_m : T_m\}$. Then the construction of the concepts $\tau_1 ... \tau_t$ by using γ is denoted by*

$$[\gamma \;\; \tau_1 : \{\alpha_1 : T_1 \; ... \; \alpha_n : T_n\} \; ... \; \tau_t : \{\beta_1 : T_1 \; ... \; \beta_m : T_m\} \,]$$

It is computed as

$$\|\!| \; [\![... [\![S(\tau_1) \;\; \alpha_1 : T_1 \; ... \; \alpha_n : T_n]\!] ... (\tau_t) \;\; \beta_1 : T_1 \; ... \; \beta_m : T_m]\!] \; |\!\|$$

Consider for instance a context “class” defined as:

```
class MyClass {
  <:ATTRIBUTE<##> public TYPE NAME;
  :>
}
```

Note that the construction considers tags literally so that if we wish to construct each concept to its own line, we need to include a line break within the tag. Therefore, above, the tag actually extends over two lines. This causes the construction to generate also a line break. Now, consider the following construction:

```
[ class
    ATTRIBUTE:{ TYPE:String  NAME:name }
    ATTRIBUTE:{ TYPE:String  NAME:address }
]
```

By Definition 9, it unfolds to

⟦⟦ ⟦class MyClass {
<: ATTRIBUTE<##>public TYPE NAME;
:>
} (ATTRIBUTE) TYPE : String NAME : name
⟧(ATTRIBUTE) TYPE : String NAME : address
⟧⟧

After unfolding the innermost tag rewrite, it becomes:

⟦⟦class MyClass {
public String name;
<: ATTRIBUTE<##>public TYPE NAME;
:>
} (ATTRIBUTE) TYPE : String NAME : address
⟧⟧

After unfolding the remaining tag rewrite, it becomes:

⟦class MyClass {
public String name;
public String address;
<: ATTRIBUTE<##>public TYPE NAME;
:>
}
⟧

Finally, after tag removal, the construction result is:

```
class MyClass {
  public String name;
  public String address;
}
```

Here, each concept is cleanly constructed on its own line, due to including the line break in the tag, in the definition of the “class” construct.

3 Formalization of programming by construction

We formalize the concept of programming by construction by using a formal grammar [6]. More specifically, we use a Backus-Naur form [10] to represent the grammar that describes definition of concepts and contexts as terms as well as construction of concepts by using them. In the grammar, we wish to use construction as a "first class citizen", so that it can determine not only the content for some concept or context, but also the entire definition of new concepts and contexts. We shall start by defining the grammar. As the grammar defines only the syntax used in programing by construction, we then define how each statement in the grammar interpreted. Lastly, we give some examples illustrating the interpretation.

3.1 Formal grammar

Definition 10 Formalization of programming by construction. *The concept of programming by construction is captured by using the following formal grammar, where each row is numbered for later reference:*

```
<corpus>       ::= <element> <corpus>          (1)
                 |  <element>                  (2)
<element>      ::= <construction>              (3)
                 |  <definition>               (4)
<construction> ::= "[:"<label> <corpus>":]"    (5)
                 |  "["<label> <corpus>"]"     (6)
<definition>   ::= <label> ":" <value>         (7)
<value>        ::= <label>                     (8)
                 |  <construction>             (9)
<label>        ::= "{" <text> "}"              (10)
                 |  <word>                     (11)
```

The grammar of Definition 10 determines a higher-order term (string) rewriting system [7], because it allows definition of new terms on-the-fly. Thus, in a way, it is similar to for instance λ-calculus [11]. Furthermore, the grammar is defined recursively. This means, for instance, that not only can a construction determine a definition, but also a definition can determine a construction. This makes the language more expressive, but then we have to define how a term definition behaves during recursion.

Although the grammar above defines, what kind of representations form well defined statements in the concept of programming by construction, it does not define how to interpret those statements. In particular, there is no link between the grammar and the definitions for construction given in Section 2. Therefore, we shall now define the interpretation to all statements of the grammar in a top-down manner. In doing so, we also formally define an interpreter for the grammar.

3.2 Interpretation of the grammar

Before we can speak of the interpretation of grammar, we need to define a dictionary. It is the heart of the interpretation process.

Definition 11 Dictionary. *All defined terms, concepts and contexts, form a dictionary. Dictionary can be modified by redefining a term. Changes in the dictionary are visible only at the current level of interpretation recursion or below it.*

Thus, the result of the interpretation (of the whole corpus) is a dictionary. Because the grammar is recursive, changes in the dictionary are also recursive. This means, for instance, that the terms defined within the construction element do not become visible to the upper level recursion. This becomes apparent later, in the interpretation for the construction.

Interpretation 1 Corpus. *Corpus is a string to be interpreted. As defined in the rows 1 and 2 of Definition 10, a well-formed corpus can consist only of sequential, well-defined elements. Each element is fully interpreted before considering the next element in the corpus.*

In other words, corpus is interpreted element by element until there are no elements left, or an ill-formed element is encountered. In the latter case, the interpretation stops with a partial result.

Interpretation 2 Element. *As defined in the rows 3 and 4 of Definition 10, a well-formed element can either be a construction result or a term definition.*

Note that the construction result and the term definition have their own interpretations that differ significantly. We shall define the interpretation for the contextual rewriting and construction element next.

Interpretation 3 Contextual rewriting. *Consider a corpus* $[: label < corpus >:]T$*, where* T *is the remainder of the corpus. Also, let* R *denote the result of the contextual rewriting of the form* $[: label < corpus >:]$ *as defined in the row 5 of Definition 10. Then, the effect of the contextual rewriting is that* R *is prefixed to the remainder of the corpus,* T*; thus, resulting in corpus* RT*. Consequently, a well-formed contextual rewriting is always of the form*

$$[\![\gamma\ \ \tau_1 : \{\alpha_1 : T_1 \ldots \alpha_n : T_n\} \ldots \tau_t : \{\beta_1 : T_1 \ldots \beta_m : T_m\}\ ]\!]$$

and its result can be computed as contextual rewriting by using Definition 7. However, although the current dictionary is fully available for the contextual rewriting, the concepts $\tau_1..\tau_t$*,* $\alpha_1..\alpha_n$*, and* $\beta_1..\beta_m$ *are visible only during the contextual rewriting, because the contextual rewriting enters a deeper state of recursion.*

Thus, technically speaking, the contextual rewriting only modifies the corpus at the current location of interpretation. The contextual rewriting does not modify the current dictionary in any way. That functionality is solely reserved for the term definition. Moreover, as defined in Definition 7, contextual rewriting preserves all tags in the reconstructed result.

Interpretation 4 Construction. *Consider a corpus of the form* $[label < corpus >]T$*, where* T *is the remainder of the corpus. Also, let* R *denote the result of the construction of the form* $[label <corpus>]$ *as defined in the row 6 of Definition 10. Then, the effect of the construction is that* R *is prefixed to the remainder of the corpus,* T*; thus, resulting in corpus* RT*. Consequently, a well-formed construction is always of the form*

$$[\gamma \ \tau_1 : \{\alpha_1 : T_1 \ ... \ \alpha_n : T_n\} \ ... \ \tau_t : \{\beta_1 : T_1 \ ... \ \beta_m : T_m\} \ ]$$

and its result can be computed by using Definition 9. However, although the current dictionary is fully available for the construction, the concepts $\tau_1..\tau_t$*,* $\alpha_1..\alpha_n$*, and* $\beta_1..\beta_m$ *are visible only during the construction, because the construction enters a deeper state of recursion.*

Thus, the difference, between the contextual rewriting and construction is that construction removes all tags from the result, as it was defined in Definition 9.

Interpretation 5 Term definition. *As defined in the row 7 of Definition 10, a well-formed term definition is of the form* $<label>:<value>$*. Thus, a term can be either a concept or a context depending on its content in accordance to earlier definitions 1 and 3. A well-defined term is added to the dictionary. In case the dictionary has already a term with the same name, it is replaced with the most recent term definition.*

Note that dictionary update is visible only at the current level of recursion or below it. This is mandated and enforced by the interpretation of the construction elements. Note also, that the grammar of 10 does not enforce construction to take place only as value for the term definition. In fact, the grammar does allow construction to take place as an element of corpus. Then, if the result of the construction is a set of term definitions, those terms will be added to the dictionary at the current level of recursion. Moreover, if the result of the construction is merely a label, that label can be used as the name of a new term definition. All this becomes apparent later on in this section, as we illustrate the use of the defined language.

Interpretation 6 Value. *As defined in the rows 8 and 9 of Definition 10, a well-formed value is either a label or the result of a construction.*

Note that this means if the construction appears as a value in a term definition, that the entire result of the construction is used as the value for the term definition and none of it flows into the corpus as new elements.

Interpretation 7 Label. *As defined in the rows 10 and 11 of Definition 10, a well-formed label is either delimited by braces, "{" and "}", or it is a word. The delimiters are not considered to be part of the label, and a word cannot contain white spaces or colons.*

Note that according to the interpretation, a label may well contain special characters. In fact, as the last interpretation, we specialize the interpretation of the label to refer to files and to the command shell.

Interpretation 8 @-label. *A label that starts with "@" refers to a file identified by the rest of the name of the label. A single "@" refers to the command shell.*

This interpretation enforces with the following side effects:

- $@file$:$<value>$ causes writing of $<value>$ to a file named $file$.
- $<label>$: $[@file <corpus>]$ reads the content of a file named $file$ and uses it as the context for the construction.
- $@$:$<value>$ causes execution of command shell with $<value>$.
- $<$ $label$ $>$: $[@ < corpus >]$ uses the most recent result of the command shell execution as the context for the construction.

3.3 Examples of interpretations

We shall now illustrate the interpretation of the defined grammar by using simple examples. A more comprehensive example is shown later as the case study.

As the first example, we consider a simple case of defining a context and using it in constructing two concepts. Consider the following corpus:

```
greeting : {Hello <:person<#, #>name:>!}
@stdout : [greeting
    person:{name:Mauno}
    person:{name:Mats}
]
```

The corresponding parse tree is shown in Figure 1. As the tree shows, the corpus parses into two definitions, from which the latter definition is determined by the construction result. As the result is stored to a term called "@stdout", it effectively prints the result to the standard output:

```
Hello Mauno, Mats!
```

To illustrate the recursive nature of construction, and the fact that the construction is a "first-class citizen" in the language, we restructure the same example differently. We also modify the corpus so that it illustrates the recursive nature of dictionaries and term definitions. Consider now the following corpus:

```
person:whoever...
people :{person:{name:Mauno} person:{name:Mats}}
greeting : {Hello <:person<#, #>name:>!}
@stdout : [greeting [people]]
@stdout : [person]
```

This corpus consists of five definitions. Although the term "person" is redefined in the construction, it does not alter the original definition for the term "person", as construction uses a recursive dictionary. Hence, the above corpus yields the following result on standard output:

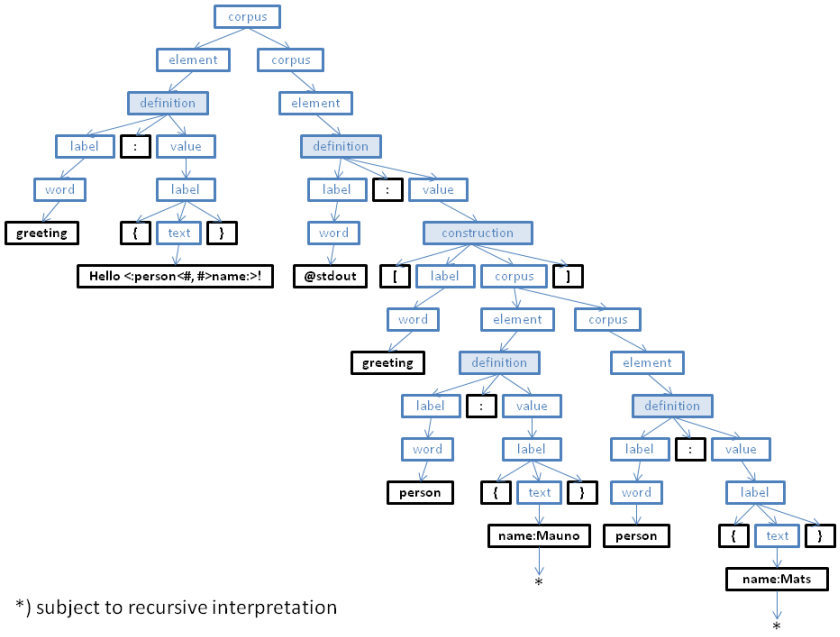

Figure 1: Parse tree for the corpus of the first example.

```
Hello Mauno, Mats!
whoever...
```

As the next example, we illustrate how the result of a construction element is really prefixed to the corpus. In this case, consider the following corpus:

```
person:whoever...
people :{person:{name:Mauno} person:{name:Mats}}
[people]
@stdout : [person]
```

Now, as the result of the construction "[people]" is prefixed to the corpus, it effectively causes the redefinition of the term "person" twice. Since only the latter definition lasts, the corpus above yields the following output:

```
name:Mats
```

As the last example, we illustrate how the result of a construction element can be used as a label to define a new term. Such a possibility is implicitly in the grammar as the construction result is prefixed to the corpus. As an example, consider the following corpus:

```
label:programming
[label]:{by construction}
@stdout : [programming]
```

This yields the following result to the standard output:

```
by construction
```

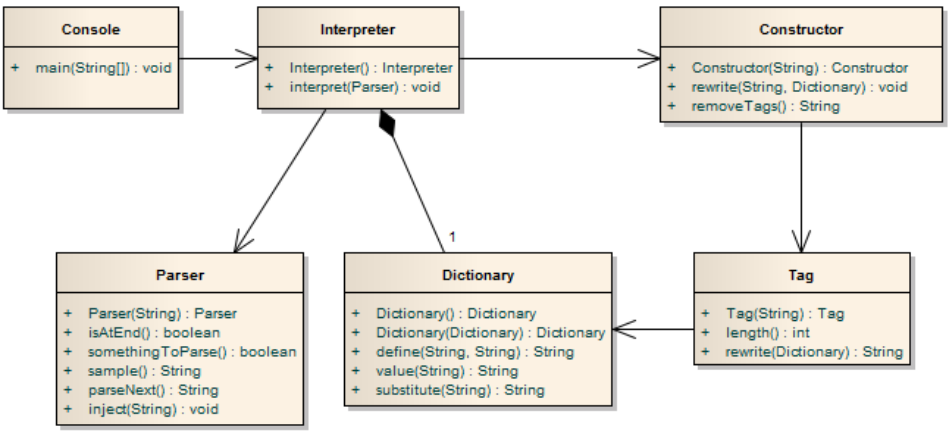

Figure 2: Class digram of Mill implementation.

4 Example Java implementation of programming by construction

We have implemented an example interpreter for the language described in Section 3 in Java. The interpreter is called Mill. We shall now describe the implementation on a general level. The actual Java source code is provided in Appendix I as open-source.

As the grammar is simple, the implementation of Mill consists of only six classes. The classes are *Console*, *Interpreter*, *Parser*, *Dictionary*, *Constructor*, and *Tag*. In short, the *Console* is the main class using the *Interpreter* to interpret corpus elements. The *Interpreter* uses the *Parser* to parse elements from corpus. The *Interpreter* uses the *Constructor* to construct elements and it stores well-founded term definitions are stored into the *Dictionary*. The *Constructor* uses instances of *Tag* for the rewriting processing.

Figure 2 shows the classes and their dependencies as a UML class diagram [12]. It also shows the public methods of the classes that provide the core functionality. We shall next discuss each class in more detail, and explain in particular the role of the public methods in the classes. Note that each class consists of also private attributes and private methods. We do not, however, discuss here the private attributes and methods to keep the presentation concise.

Console. The main class is called *Console*. It has only one public method, *main*, that is used to run Mill. The *main* method creates an instance of the *Interpreter* class. The *main* methods then feeds the data from the standard input as a corpus to the *Interpreter*. To do that, the *main* encapsulates the corpus in an instance of a *Parser* which is then passed to the *Interpreter*. The *main* method terminates either when all data is processed from the standard input or when the *Interpreter* encounters an error. The *Console* also supports running Mill in an interactive mode by passing a command line parameter *interactive*. Then, the *main* method reads one line at a time from the standard input, encapsulates it in a *Parser*, and passes it to the *Interpreter*. In the interactive mode, the *main* method never terminates; it has to be terminated forcefully.

Interpreter. The *Interpreter* class has only one constructor that takes no parameters. The constructor creates an instance of a *Dictionary* and attaches it to the constructed instance of the *Interpreter*. As for the methods, the *Interpreter* class consists of only one public method, *interpret*. It takes one parameter, a *Parser*, that contains the corpus. The *Interpreter* uses the *Parser* to pick one element of the corpus at a time and to process it. If the element is a construction, the *Interpreter* uses an instance of the *Constructor* class to obtain the construction result. The result is then injected back to the beginning of the *Parser*. If, on the other hand, the element is a term definition, the *Interpreter* stores it to its internal *Dictionary*. Prior storing it, any construction takes place by using an instance of the *Constructor* class. If an element if ill-formed, the *Interpreter* stops interpreting and returns back an error as an exception. The *Interpreter* stops also once all elements are parsed. In all cases, the *Interpreter* maintains the dictionary of terms interpreted thus far. Consequently, when the *Interpreter* is called the next time, the terms defined before are available.

Parser. The *Parser* class has only one constructor that takes a string as a parameter. The constructor attaches the string to the constructed instance of the *Parser*. The *Parser* class provides three methods to inspect the state of the corpus. The *isAtEnd* method returns true, if the corpus is empty. The *somethingToParse* method returns true, if the corpus still contains non-white-space characters. The *sample* method returns a clip of at most 50 characters from the beginning of the remaining corpus. In addition to these methods, the *Parser* class consists of the main method to handle the actual parsing. The *parseNext* method parses and returns the first syntactic element from the corpus, leaving the remaining corpus intact. Although the element is syntactically well-formed, it may still be semantically ii-formed for the *Interpreter*. If the element is also syntactically ill-formed, the *parseNext* methods returns an error as exception. Lastly, the *Parser* class has also one method to inject a string back to the beginning of the corpus. That method is called *inject* and it takes as parameter the string to be injected back to the corpus.

Dictionary. The *Dictionary* class has two constructors: one that creates an empty dictionary, and the other that creates an empty dictionary with a reference to another, existing dictionary. The latter of the two methods is used in construction, to cerate a recursive dictionary. In technical terms, if a dictionary is constructed with the latter method, the newly created dictionary inherits all terms defined in parameter dictionary. In particular, the newly created dictionary can access all the values of the parameter dictionary, but is cannot modify any of their values. In addition to the constructors, the *Dictionary* class provides three public methods. The *define* method defines a new term with given label and value. It returns the previous value for the term, or null if it was not defined before. The *value* method returns the value of the term with given label. If the term is not defined, this method returns an empty string. Note that this method also considers any terms inherited during the construction of the dictionary. Lastly, the *substitue* method performs the sequential textual substitution, as defined earlier in Definition 5, on a string provided as parameter by using the terms defined in the dictionary. Note that this method considers only the terms that are defined within the current dictionary. Any terms that are inherited during the construction of the dictionary are omitted unless they are redefined.

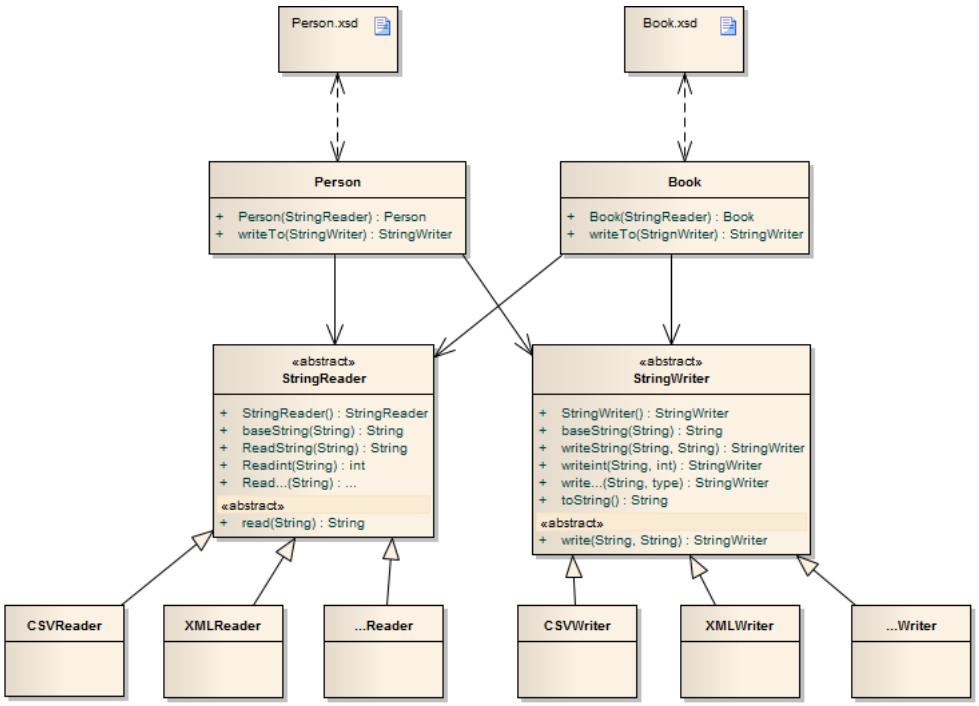

Figure 3: Original design pattern.

Constructor. The *Constructor* class has only one constructor with one parameter that is the context as a string. The *Constructor* class has also two public methods. The *rewrite* method performs the contextual rewriting on the context string, as defined earlier in Definition 7, with respect to a label and dictionary provided as parameters. To do this, the *rewrite* method constructs a *Tag* instance, and calls its *rewrite* method. The *removeTags* method of the *Constructor* performs the tag removal on the context string, as defined earlier in Definition 8, and returns the resulting context as a string.

Tag. The *Tag* class has only one constructor with one string parameter. The string must start with a tag represented in the form as defined earlier in Definition 2. Note that the string can contain multiple tags, but it must start with a tag. The *Tag* class has also two public methods. The *length* method returns the number of characters in the tag, including delimiters "<:" and ":>". The *rewrite* method performs the tag rewriting as defined earlier in Definition 6 with respect to a dictionary provided as a parameter. To do this, the *rewrite* calls the *substitute* method of the parameter dictionary. It then returns the results as a string as defined earlier in Definition 6.

5 Case study: independence of data representation

In the case study, we refactor a design pattern as depicted in Figure 3. The pattern is a combination of a bridge pattern, a template method pattern and a factory method pat-

tern. The problem that the pattern solves is independence of data representation. The way that the pattern solves the problem is that it provides two interfaces, one for reading data elements from a string, and the other for writing data elements to a string. These interfaces can then be specialized by classes managing a specific data representation. The specialized classes are then used by the constructor and a writer method of the data domain class. To keep the presentation short, we do not go into details of the methods of classes shown in the design pattern. Perhaps the only method that needs some further explanation is the *baseString* method that appears in the *StringReader* and in the *StringWriter* classes. Its function is merely to convert a string between Java representation and some other representation, such as XML. Its base class implementation returns the string as is, so for instance the *XMLReader* class needs to override it for converting XML string representation into Java string representation.

Although the design pattern solves the intended problem of representation, it does not solve the synchronization issue between the domain object classes *Person* and *Book* and the corresponding XML schema descriptions *Preson.xsd* and *Book.xsd*. With respect to this issue, there are no applicable design patterns, because the schemas are not in the domain of the programming language, Java. There is also another problem that is not directly visible in the design pattern. Namely, even the attributes of the domain object classes need to be repeated in multiple places with proper types within each domain object class. This is a synchronization problem internal to the domain object classes. In theory, one could construct a factory design pattern to solve this, but then the domain object classes would have to be dynamically typed and constructed during run-time, which undermines the whole compile-time type checking of Java.

Both of these problems fall to the domain of programming by construction. The problem of representing the domain objects as classes and XML schemas fall clearly to the case of representing some specific concepts in some specific contexts. This is also the case with the repeated representation of the attributes within the class. To solve both of these problems, we design a generic construct that supports construction of both domain object classes and their XML schema representations from the same concept descriptions.

The designed construct is shown in Figure 4. The domain objects are expressed as concepts *Person* and *Book*. Similarly, the attribute types are expressed as domain specific concepts *JavaTypes* and *XSDTypes*. The actual contexts for constructing the Java class and the XSD description are expressed as *JavaDomainObject* and *XSDDomainObject*, respectively. All these are glued together by the meta context *Construct*. We shall now present all these concepts and contextx in detail and then illustrate the use of the meta context, *Construct*.

5.1 Concepts for the domain objects

Although Java and XSD are “typed”, their type definitions are not fully compatible. This is typically a source for problems for the beginners. In our case study, we use just two types, an integer type and a string type. From these two types the integer type is comparable between Java and XSD. The string type, however, is not. For one, the representation of characters in a string differs significantly between Java and XSD. For instance, the “&” character is represented in XSD as “&”. Another problem is the name of the type is different in Java and in XSD. In Java it is called “String” and in XSD it is called “string”.

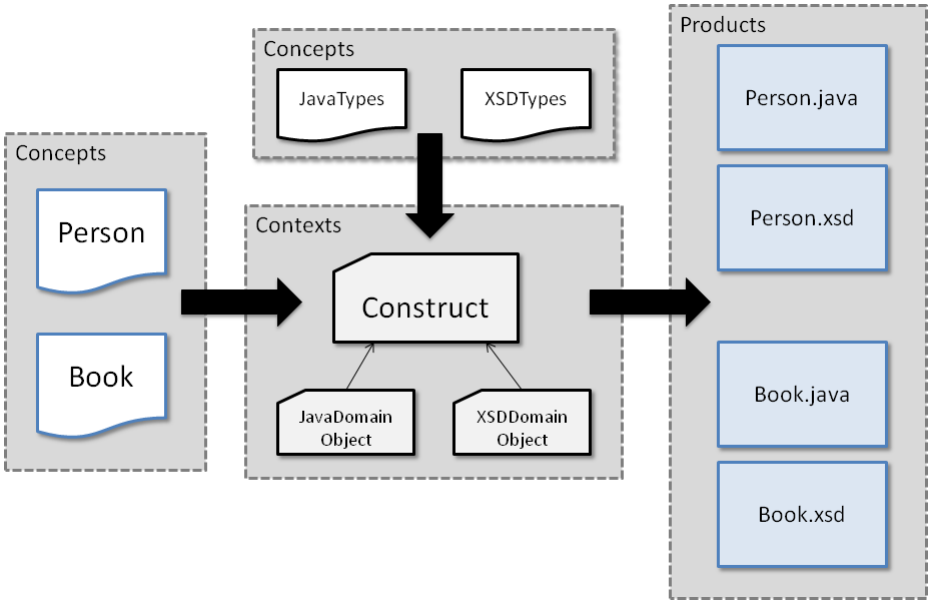

Figure 4: Constructing the domain objects.

The former of these two issues is naturally handled by the design pattern, by overriding the implementation of the *baseString* function. The latter issue with the type naming, however, is addressed naturally by concepts. For this purpose, we define the needed Java types as:

```
JavaTypes : {
  STRING:String
  INT:int
}
```

Similarly, we define corresponding XSD types as:

```
XSDTypes : {
  STRING:string
  INT:int
}
```

These concepts can then be passed to whichever context that needs type information.

In addition to the type concepts, we also define the actual domain objects as concepts. In our case we have two such concepts, one for capturing person details and the other for capturing book details. We define the person details as:

```
Person : {
  CLASS : {NAME:Person}
  ATTRIBUTE : {NAME:name TYPE:[STRING]}
  ATTRIBUTE : {NAME:address TYPE:[STRING]}
  ATTRIBUTE : {NAME:age TYPE:[INT]}
}
```

Similarly, we define the book details as:

```
Book : {
  CLASS : {NAME:Book}
  ATTRIBUTE : {NAME:title TYPE:[STRING]}
  ATTRIBUTE : {NAME:author TYPE:[STRING]}
  ATTRIBUTE : {NAME:publisher TYPE:[STRING]}
  ATTRIBUTE : {NAME:year TYPE:[INT]}
}
```

Thus, both of these concepts assume that types are defined as concepts. For instance, the *Person* model defines an attribute with name "address" to be of whichever type is obtained by constructing the concept "STRING". In case of *JavaTypes* this evaluates to "String" and in case of *XSDTypes* this evaluates to "string". Note that neither of the concepts define what is meant by "CLASS" or by "ATTRIBUTE". These are concepts whose meaning is determined by given contexts.

5.2 Constructs for the domain objects

There are two constructs, one for the Java domain object and the other for the XSD domain object. The construct for the Java domain object for Java is defined as an immutable [13] Java class:

```
JavaDomainObject : {

 public class <:CLASS<##>NAME:>
 {<:ATTRIBUTE<##>
    private TYPE NAME;:>

    public <:CLASS<##>NAME:>(<:ATTRIBUTE<#, #>TYPE NAME:>)
    {<:ATTRIBUTE<##>
        this.NAME=NAME;:>
    }

    public <:CLASS<##>NAME:>(StringReader reader)
    throws Exception
    {<:ATTRIBUTE<##>
        this.NAME=reader.readTYPE("NAME");:>
    }
<:ATTRIBUTE<##>
    public TYPE NAME()
    { return NAME; }
:>
    public StringWriter writeTo(StringWriter writer)
    throws Exception
    {
        return writer<:ATTRIBUTE<##>
            .writeTYPE("NAME",NAME):>;
    }
 }
}
```

Note that in the *JavaDomainObject* context, both the class and the attribute concepts appear in many places with varying semantics. For instance, the first occurrence of a class concept is at the very definition of the Java class. Then, the semantics of the class concept is to give a name for the Java class. The next time the same class concept appear is in the constructor of the Java class. Then, the class name is used to identify a constructor method. This latter instance of the class concept is specific to Java language. The attribute concept is used even more often than the class concept and its semantics is, thus, even more versatile. Naturally, the attribute concept appears in the attribute definition with the intention to define class attributes. It then appears as constructor parameter with the intention to relate and assign parameter values to class attributes. Thus, the attribute concept must also appear in the actual assignment statements as well. Lastly, the attribute concept appears in method declaration, to define getter methods for the attributes. Note that there are no setter methods, as the constructed class is to be immutable.

The context for the XSD domain object differs significantly from the Java domain object. The biggest reason for this is the XSD deals with structural definition, whereas Java considers also the dynamics or functionality in addition. The XSD domain object is defined as:

```
XSDDomainObject : {
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.org/<:CLASS<##>NAME:>"
  xmlns:tns="http://www.example.org/<:CLASS<##>NAME:>"
  elementFormDefault="qualified">
  <complexType name="<:CLASS<##>NAME:>Type">
    <sequence><:ATTRIBUTE<##>
            <element name="NAME" type="TYPE"></element>:>
    </sequence>
  </complexType>
</schema>
}
```

Thus, in case of the XSD domain object, the class concept appears in many places, but the attribute concept appears only in one place.

5.3 Meta context

The last missing context is the one that glues together the type concepts and the domain object concepts with the domain object contexts. As this context is effectively a context on contexts, it is technically speaking a meta context. We define the meta context as:

```
Construct : {
  <:OBJECT<##>
  [JavaTypes]
  @JAVA/NAME.java : [JavaDomainObject [NAME]]
  [XSDTypes]
  @XSD/NAME.xsd : [XSDDomainObject [NAME]]
  :>
}
```

The meta context practically binds the Java type concepts and the domain object concept referred to as “NAME” with the Java domain object context, and stores the result into a “JAVA” sub-folder based on the name of the domain object concept. Similarly, it binds the XSD type concepts and the domain object concept referred to as “NAME” with the XSD domain object context, and stores the result into a “XSD” sub-folder based on the name of the domain object concept.

This meta context shows how the separation of concepts and contexts supports formation of higher level idioms in a unified and coherent manner. In fact, the meta context could have been split into a sub-context, where the binding of the type concept and the domain object context would have been determined by a domain concept, thus avoiding the repetition of the binding and construction phases for Java and XSD now apparent in the meta context.

5.4 Construction of “Person” and “Book”

With the presented concepts and contexts, the construction of both the person and the book domain objects, the Java files and the XSD files, can now be achieved with a single construction line:

```
[Construct OBJECT:{NAME:Person} OBJECT:{NAME:Book}]
```

Naturally, the construction of yet another domain object can be done adding one more “OBJECT” concept to the construction element. The constructed Java file for the person object is:

```
public class Person
{
  private String name;
  private String address;
  private int age;

  public Person(String name, String address, int age)
  {
    this.name=name;
    this.address=address;
    this.age=age;
  }

  public Person(StringReader reader)
  throws Exception
  {
    this.name=reader.readString("name");
    this.address=reader.readString("address");
    this.age=reader.readint("age");
  }

  public String name()
  { return name; }
```

```
  public String address()
  { return address; }

  public int age()
  { return age; }

  public StringWriter writeTo(StringWriter writer)
  throws Exception
  {
    return writer
      .writeString("name",name)
      .writeString("address",address)
      .writeint("age",age);
  }
}
```

The corresponding reconstructed XSD file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.org/Person"
  xmlns:tns="http://www.example.org/Person"
  elementFormDefault="qualified">
  <complexType name="PersonType">
    <sequence>
        <element name="name" type="string"></element>
        <element name="address" type="string"></element>
        <element name="age" type="int"></element>
    </sequence>
  </complexType>
</schema>
```

The Java and XSD files for the book object are analogous.

6 Conclusion

In this technical report we discussed programming by construction. The idea in to separate concepts from their semantic contexts. We formalized this approach by defining a simple grammar that captured the definition of concepts and contexts. We then gave an interpretation for the grammar, including the construction, by formalizing it as a term rewriting process. We also presented an example implementation of the grammar in Java. The interpreter, called Mill, is also listed as open-source in Appendix.

We illustrated the use of programming by construction in constructing a design pattern typical to cloud service implementation, where data needs to be represented in different formats, causing translation from one representation to another. The simple case study showed how effective the separation of concepts from contexts is in supporting the construction of the same concepts in very different contexts. Furthermore, by defining the

construction process as term rewriting system, we can define meta contexts that support automation of construction processes, where specialization is critical.

With the programming by construction approach we can achieve a uniform design and development process that covers use of multiple, heterogeneous technologies and services. As we can now separately express concepts and their semantic contexts, we are free to integrate new contexts in an technology independent manner. Furthermore, contexts can be subjected to formal reasoning, which further supports reliability studies. Note that programming by construction is fully complementary with current representation technologies for metadata, including use of XML and ontologies. In fact, there is a synergy advantage of mixing the use of, for instance, ontology descriptions with programming by construction. We plan to explore and research the possibilities and potential of such mixed use in future studies.

Although the grammar presented in this technical report is very expressive, it is still very primitive. As potential future work, we plan on extending the grammar based on needs that arise in further case studies. We plan to apply the programming by construction approach in designing and implementing future resilient control systems. Programming by construction bridges the gap between formal methods, ontologies, service interfaces, and data representations. In particular, it provides a formal framework for construction of objects based on concepts and their semantic contexts. In this respect, as a future work, we plan to study use of programming by construction in construction of machines in Event B and RODIN platform.

Acknowledgements.

This research is funded by the Academy of Finland project "FResCo: High-quality Measurement Infrastructure for Future Resilient Control Systems" (Grant numbers 264060 and 263925).

References

[1] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*, Prentice Hall, 2005.

[2] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*, Doctoral dissertation, University of California, Irvine, 2000.

[3] C. Pautasso, O. Zimmermann, F. Leymann. RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision, Proceedings of 17th International World Wide Web Conference (WWW2008), 805-814, 2008.

[4] P. Walmsley. Definitive XML Schema, 2nd Edition, Prentice Hall, 2012.

[5] D. Vohra. *Pro XML Development with Java Technology*, Apress, 2006.

[6] J. E. Hopcroft , R. Motwani, J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, 3rd Edition, Prentice Hall, 2006

[7] Terese. *Term Rewriting Systems*, Cambridge University Press, 2003.

[8] J. D. Herrington. *Code Generation in Action*, Manning Publications, 2003.

[9] M. L. Scott. *Programming Language Pragmatics*, 3r Edition, Morgan Kaufmann, 2009.

[10] J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference, *Proceedings of the International Conference on Information Processing*, 125-131, 1960.

[11] A. Church, A set of postulates for the foundation of logic, *Annals of Mathematics*, Series 2, 33, 346-366, 1932.

[12] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd Edition, Addison-Wesley, 2003.

[13] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*, 2nd Edition, Addison-Wessley, 1999.

Appendix I: Example implementation, Mill.


```
public class Console
{
  public static void main(String[] args) throws Exception
  {
    Interpreter interpreter=new Interpreter();
    if (args.length>0)
    {
      System.out.println("**** Mill, v1.0 (C) Mauno Ronkko 2013 ****");
      System.out.println("**** "+new File("").getAbsolutePath()+" ****");
      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
      String line="";
      while (true) try
      {
        System.out.print("\n: "); System.out.flush();
        line=br.readLine();
        interpreter.interpret(new Parser(line));
      } catch (Exception e) {
        System.out.println("\n**** ERROR:\n"+e.getMessage());
      }
    }
    else
    {
      int available=System.in.available();
      if (available>=0)
      {
        byte b[]=new byte[available];
        System.in.read(b);
        interpreter.interpret(new Parser(new String(b)));
      }
    }
  }
}
```

```
public class Interpreter
{
  private Dictionary dictionary;

  private Interpreter(Dictionary dictionary)
  { this.dictionary=new Dictionary(dictionary); }

  public Interpreter()
  { this.dictionary=new Dictionary(); }

  private String cleanLabel(String label)
  { return label.startsWith("{") ? label.substring(1,label.length()-1) : label; }

  private String readFromFile(String filename) throws Exception
  {
    File file=new File(filename);
    int length=(int)file.length();
    if (length==0) return "";
    byte[] buffer = new byte[length];
    FileInputStream fis = new FileInputStream(filename);
    fis.read(buffer);
    fis.close();
    return new String(buffer);
  }

  private String interpretBlock(String block) throws Exception
  {
    block = block.substring(1,block.length()-1);
    boolean retain=block.startsWith(":") && block.endsWith(":");
    Parser parser=new Parser(retain ? block.substring(1,block.length()-1) : block);
    String sample=parser.sample();
    String label=cleanLabel(parser.parseNext());
    String body=(label.startsWith("@")  && label.length()>1)
                ? readFromFile(cleanLabel(label.substring(1)))
                : dictionary.value(label);
    Constructor constructor=new Constructor(body);
    try
    {
      while (parser.somethingToParse())
      {
        String element=parser.parseNext();
        if (element.startsWith("["))
        {
          parser.inject(interpretBlock(element));
          //this method does not modify current dictionary
        }
        else
        {
          Interpreter interpreter=new Interpreter(dictionary);
          if (!parser.parseNext().equals(":"))
            throw new Exception("MISSING \":\", AFTER LABEL {"+element+"}\n");
          interpreter.interpret(new Parser(cleanLabel(parser.parseNext())));
          constructor.rewrite(element, interpreter.dictionary);
        }
      }
    } catch (Exception e) {
      throw new Exception(e.getMessage()+"WHILE RECONSTRUCTING ["+sample+"]\n");
    }
    return retain ? constructor.withTags() : constructor.removeTags();
  }
```

```
  private String execute(String command) throws Exception
  {
    Process p=Runtime.getRuntime().exec(command);
    int status=p.waitFor();
    InputStream is=p.getInputStream();
    byte[] bytes=new byte[is.available()];
    is.read(bytes, 0, bytes.length);
    String output=new String(bytes);
    is=p.getErrorStream();
    bytes=new byte[is.available()];
    is.read(bytes, 0, bytes.length);
    if (status!=0 || bytes.length>0)
      throw new Exception("FAILED EXECUTING "+command
                          +"\nERROR OUTPUT: "+new String(bytes));
    System.out.println("OUTPUT OF "+command+": "+output);
    return output;
  }

  private void writeToFile(String filename, String value) throws Exception
  {
    if (filename.equals("stdout"))
      System.out.println(value);
    else
    {
      FileWriter writer = new FileWriter(filename);
      writer.write(value);
      writer.close();
    }
  }

  private void interpretTerm(String label, Parser parser) throws Exception
  {
    if (!parser.parseNext().equals(":"))
      throw new Exception("MISSING \":\", AFTER LABEL {"+label+"}\n");
    String value=cleanLabel(parser.parseNext());
    if (value.startsWith("[")) value=interpretBlock(value);
    if (label.startsWith("@"))
      if (label.equals("@"))
        dictionary.define("@", execute(value));
      else
        writeToFile(label.substring(1),value);
    else
      dictionary.define(label,value);
  }

  public void interpret(Parser parser) throws Exception
  {
    String sample=parser.sample();
    try
    {
      while (parser.somethingToParse())
      {
        String element=parser.parseNext();
        if (element.startsWith("["))
          parser.inject(interpretBlock(element));
        else
          interpretTerm(cleanLabel(element), parser);
      }
    } catch (Exception e) {
      throw new Exception(e.getMessage()+"WHILE PARSING "+sample+"\n");
    }
  }
}
```

```
public class Parser
{
  private String corpus;

  public Parser(String corpus)
  { this.corpus=corpus.trim(); }

  public boolean isAtEnd()
  { return corpus.isEmpty(); }

  private char next()
  { return isAtEnd() ? 0 : corpus.charAt(0); }

  private boolean nextIsWhitespace()
  { return Character.isWhitespace(next()); }

  private char getNext()
  { if (isAtEnd()) return 0;
    char c=corpus.charAt(0);
    corpus=corpus.substring(1);
    return c;
  }

  private String getBlock(char start, char end) throws Exception
  { if (next()!=start) throw new Exception("MISSING "+start+"\n");
    String block="" + getNext();
    int depth=1;
    while (!isAtEnd())
    {
      if (next()==start) depth++;
      if (next()==end) {if (--depth<=0) break;}
      block += getNext();
    }
    if (depth>0) throw new Exception("MISSING "+end+"\n");
    return block+getNext();
  }

  private String getLabel()
  { String label="";
    while (!isAtEnd() && !nextIsWhitespace() && next()!=':')
      label +=getNext();
    return label;
  }

  public boolean somethingToParse()
  { while (!isAtEnd() && nextIsWhitespace()) getNext();
    return !isAtEnd();
  }

  public String parseNext() throws Exception
  { if (!somethingToParse()) throw new Exception("NOTHING TO PARSE!\n");
    if (next()=='[') return getBlock('[',']');
    if (next()=='{') return getBlock('{','}');
    if (next()==':') return "" +getNext();
    return getLabel();
  }

  public void inject(String string)
  { corpus=string+corpus; }

  public String sample()
  { int cut=corpus.length();
    return "\"..."+corpus.substring(0,cut>50 ? 50 : cut)+"...\"";
  }
}
```

```
public class Constructor
{
  private String construct;

  public Constructor(String construct)
  { this.construct=construct; }

  public void rewrite(String label, Dictionary dictionary) throws Exception
  {
    String header="<:"+label;
    String old=construct;
    construct="";
    while (!old.isEmpty())
    {
      int at=old.indexOf(header);
      if (at<0)
      {
        construct+=old;
        old="";
      }
      else
      {
        if (at>0) {construct+=old.substring(0,at); old=old.substring(at);}
        Tag tag=new Tag(old);
        old=old.substring(tag.length());
        construct+=tag.rewrite(dictionary);
      }
    }
  }

  public String removeTags() throws Exception
  {
    String old=construct;
    construct="";
    while (!old.isEmpty())
    {
      int at=old.indexOf("<:");
      if (at<0)
      {
        construct+=old;
        old="";
      }
      else
      {
        if (at>0) {construct+=old.substring(0,at); old=old.substring(at);}
        old=old.substring(new Tag(old).length());
      }
    }
    return construct;
  }

  public String withTags()
  { return construct; }
}
```

```
public class Tag
{
  private String label;
  private String separator;
  private String body;

  private String getBlock(String text, String start, String end) throws Exception
  {
    String block="";
    if (!text.startsWith(start)) throw new Exception("MISSING \""+start+"\"\n");
    int depth=0;
    while (!text.isEmpty())
    {
      if (text.startsWith(start)) depth++;
      if (text.startsWith(end)) {if (--depth<=0) break;}
      block += text.charAt(0);
      text = text.substring(1);
    }
    if (depth>0) throw new Exception("MISSING \""+end+"\"\n");
    return block+end;
  }

  public Tag(String string) throws Exception
  {
    String tag=getBlock(string,"<:",":>");
    tag=tag.substring(2,tag.length()-2);
    int sep=tag.indexOf("<#");
    if (sep<0) throw new Exception("MISSING \"<#\"\n");
    label=tag.substring(0,sep);
    tag=tag.substring(sep);
    separator=getBlock(tag,"<#","#>");
    body=tag.substring(separator.length());
    separator=separator.substring(2,separator.length()-2);
  }

  public int length()
  { return label.length()+separator.length()+body.length()+8; }

  public String rewrite(Dictionary dictionary)
  {
    String result=dictionary.substitute(body);
    body=separator+body;
    separator="";
    return result+toString();
  }

  public String toString()
  { return "<:"+label+"<#"+separator+"#>"+body+":>"; }
}
```

```
public class Dictionary
{
  private Dictionary original;
  private Hashtable<String,String> dictionary;

  private void construct(Dictionary original)
  {
    this.original=original;
    dictionary=new Hashtable();
  }

  public Dictionary()
  { construct(null); }

  public Dictionary(Dictionary original)
  { construct(original); }

  private boolean isDefined(String label)
  {
    if (dictionary.containsKey(label)) return true;
    if (original==null) return false;
    return original.isDefined(label);
  }

  public String value(String label)
  {
    String value=dictionary.get(label);
    if (value!=null) return value;
    if (original==null) return "";
    return original.value(label);
  }

  public String define(String label, String value)
  { return dictionary.put(label,value); }

  private HashSet<String> labelSet()
  {
    HashSet<String> termSet=new HashSet<String>();
    termSet.addAll(dictionary.keySet());
    return termSet;
  }

  private String[] sortedLabels()
  {
    String l;
    String[] labels=labelSet().toArray(new String[0]);
    for (int i=0; i<labels.length-1; i++)
      for (int j=i+1; j<labels.length; j++)
        if (labels[i].compareTo(labels[j])>0)
          {l=labels[i]; labels[i]=labels[j]; labels[j]=l;}
    return labels;
  }

  private int firstIndex(String string, String[] sortedLabels)
  {
    for (int i=sortedLabels.length-1; i>=0; i--)
      if (string.startsWith(sortedLabels[i]))
        return i;
    return -1;
  }
```

```
  public String substitute(String string)
  {
    String result="";
    String[] sortedLabels=sortedLabels();
    while (!string.isEmpty())
    {
      int index=firstIndex(string,sortedLabels);
      if (index<0)
        result += string.charAt(0);
      else
        result += value(sortedLabels[index]);
      string=string.substring(index<0 ? 1 : sortedLabels[index].length());
    }
    return result;
  }
}
```

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 A, 20520 TURKU, Finland | www.tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics

Turku School of Economics
- Institute of Information Systems Sciences

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

ISBN 978-952-12-2978-7
ISSN 1239-1891