Automated Support for Data Exchange via XML

Tom Tourwé
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussel - Belgium
tom.tourwe@vub.ac.be

Luk Stoops
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussel - Belgium
luk.stoops@vub.ac.be

Stijn Decneut
IMEC
Kapeldreef 75 - 3001 Leuven - Belgium
stijn.decneut@imec.be

Abstract

XML has recently emerged as a standard for exchanging data between different software applications. In this paper, we present an approach for automatic code generation to interpret information in an XML document. The approach is based on a user-defined mapping of the XML document's structure onto the application's API. This mapping is declarative in nature, and thus easy to specify, and is used by code generator that applies advanced code generation and manipulation techniques to generate the appropriate code. The approach relieves developers from the time-consuming and error-prone task of writing the interpreter themselves, and complements existing XML technologies such as XSLT.

1. Introduction

In recent years, XML (eXtensible Markup Language) has become the de facto standard for data exchange between software applications. The number of applications that are based on or make use of XML technology is simply overwhelming. The popularity of XML is due to many reasons:

- It defines a standard format for exchanging information, as opposed to an application-specific format. This makes it easier to exchange data between different applications;
- It is open, so it can be used to model any sort of information;
- It comes with a large tool set, consisting of many tools that can be used to handle and manipulate XML documents in an easy and straightforward way;
- It rigorously defines the syntax and structure of a document. This allows us to check whether a particular document is lexically and syntactically well formed.

XML can be held responsible for a major breakthrough in the standardisation of the way applications exchange data with one another. Before XML, each application had it its own proprietary way of representing and storing data. Therefore, data exchange between applications was a challenging and difficult task. Nowadays, however, applications send and receive XML documents containing the appropriate information in a cleanly structured way.

Naturally, XML isn't all roses. Exchanging information between two applications via an XML document requires that the first application packages the information into a well-structured document and the second application interprets the contents of the document in the appropriate and correct semantic way. Developers responsible for implementing such functionality are faced with the problem of developing the appropriate parsers and interpreters. Although such technology is well understood and some (limited) support for it exists in the XML toolkit (e.g. DOM or SAX), implementing it still remains a difficult and error-prone task:

- a developer should have very detailed knowledge about an application's API in order to know how data can be extracted from it, or how data should be fed to it;
- writing an interpreter is a form of meta programming, which is considered difficult by most developers. For this very same reason, the process is extremely vulnerable to errors;
- the process is not robust with respect to evolution. Changes to either the application's API or the XML document's structure require change propagation to the

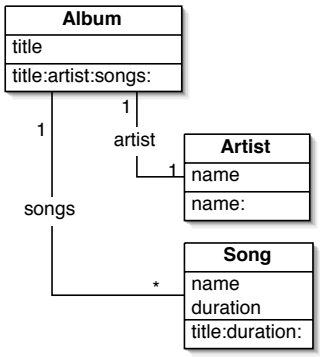


Figure 1. The music organiser's class diagram

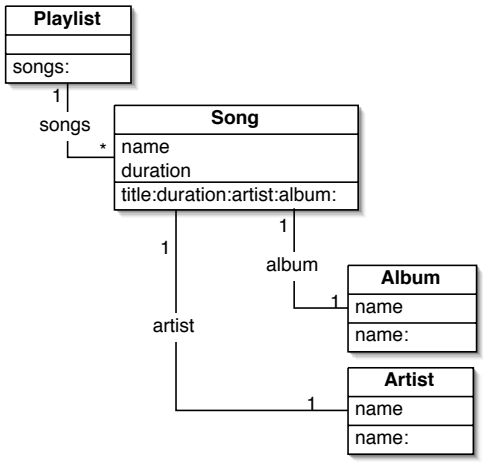


Figure 2. The playlist organiser's class diagram

parsers and interpreters. Since applications, and thus their APIs, evolve at a rapid pace, this is a serious problem;

- the process does not support easy application integration. Integrating a new application and allowing it to communicate with already existing applications may require the document structure to be changed, since it may be inadequate for containing all appropriate data. Given that many applications need to communicate these days, this is a serious problem as well.

These problems can be alleviated up to a large extent, if we unleash the full power that is offered by standardised data exchange over XML. The fact that XML documents are well structured and that various tools exist that allow advanced navigation through such documents, enables us to generate appropriate parsers and interpreters automatically, based on descriptions of an application's API and the structure of the XML documents containing the data. This would solve the problems mentioned above because the interpreter would be generated automatically, which is certainly less time consuming then writing it manually, and which is also less prone to errors. Moreover, when the application's API or the XML document's structure changes, only the corresponding descriptions need to be adapted, and the interpreter will automatically be changed accordingly.

In the remainder of this paper, we will propose an approach that does exactly this. The next section introduces the running example that will be used throughout this paper. Section 3 then discusses the approach we propose in more detail and shows how it can be used in practice. Section 4 then discusses the results of applying our approach to the running example, by showing the code that is generated for the example. Section 5 identifies work that remains to be done, Section 6 presents related work, and finally Section 7 concludes.

2. Running Example: Managing Music Files

Throughout the paper, we will make use of one single running example to illustrate the main ideas of our approach. Consider two multimedia applications, the first of which is a music organiser, that keeps track of music albums in a database, while the second is a playlist organiser, that stores playlists containing songs gathered by a user. These two applications need to communicate with each other, because they handle the same data that should remain synchronised. The data is represented in XML format. For example, the representation of a music album is represented as follows:

```
<?xml version="1.0"?>
<album>
   <title>Album 1</title>
   <artist>Artist 1</artist>
   <song>
      <title>Song 1</title>
      <duration>4:10</duration>
   </song>
   <song>
      <title>Song 2</title>
      <duration>2:50</duration>
   </song>
   ...
</album>
```

It represents an album called *Album1* by an artist *Artist1* containing a number of songs. Each song in its turn has a title and a duration.

Both applications have a class diagram that is used to store the information internally. The music organiser's diagram is represented in Figure 1. It shows a class `Album` that has associations with classes `Artist` and `Song`. The `Artist` class represents an artist and simply contains his or her name. The `Song` class represents a single song, and

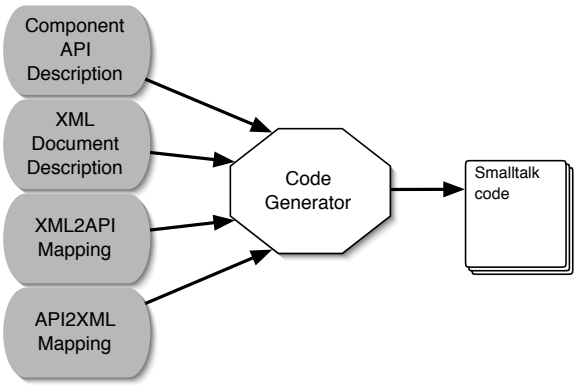


Figure 3. Our Code-Generation Approach

contains the name of the song and its duration. We assume an album is performed by only one artist, and contains multiple songs.

The playlist organiser's class diagram is shown in Figure 2. It consists of a class `Playlist` that contains references to a number of songs. These songs are represented by the `Song` class, which contains the name and the duration of the song. Moreover, the class has references to the `Artist` and `Album` classes, that represent artists and albums respectively.

3. Approach: Generating XML Interpreters Automatically

In this section, we discuss our approach to generate code for handling XML data automatically. We first present a general overview, and discuss the most important aspects in more detail afterwards.

3.1. Overview

The approach to generate code for feeding XML data to an application's API automatically is depicted in Figure 3. As can be seen, we make use of a code generator, that takes various descriptions as input, and generates appropriate Smalltalk code as output. Smalltalk is a class-based, object-oriented programming language, similar to Java. We choose it as the target language for our experiments, since it contains some advanced features that allow us to rapidly produce a working prototype and adapt it to our needs whenever necessary.

The descriptions that are passed to the code generator are the following:

application API description : a description of the classes and methods that make up the class diagram of the application's components. This description mainly contains the names of the classes, their constructor methods and the arguments that should be passed to them;

XML document structure description : a description of the XML documents that can be exchanged between the applications. This is merely the XML schema that defines the class of documents;

XML → API and API → XML mapping . The first describes how information from the XML document should be interpreted and fed to the application's API. The second describes describes how the data can be extracted from a component and packaged into an XML document.

All descriptions take the form of an XML document, and are thus declarative in nature. The code generator needs to generate code that transforms an XML document into an object structure appropriate for the application. Therefore, it needs to know how such an object structure can be constructed, which is described by the application's API description. In order to know how the information from the XML document should be mapped onto the classes, the code generator uses the XML → API description.

In what follows, we will discuss the application's API description and the XML → API description in more detail. The XML document structure description merely consists of the XML Schema, and is thus not explained any further. The API → XML is similar to the reverse mapping, but will not be considered in detail in this paper, due to space restrictions.

3.2. Application API Description

The application API description documents an application's API in terms of the classes that it offers, how these classes should be instantiated and how their objects should be structured. For example, the following description documents the `Song` class from the music organiser application:

```
<?xml version="1.0"?>
<api>
   <class>
      <name>Song</name>
      <constructor>
         <name>title:duration:</name>
         <param>String</param>
         <param>Integer</param>
      </constructor>
   </class >
</api>
```

As can be seen, an instance of the `Song` class is constructed by using the `title:duration:` constructor method, that initialises the title and the duration of the song. These are instances of class `String` and `Integer` respectively.

The `Song` class of the playlist organiser application is different, which is reflected in its description:

```
<?xml version="1.0"?>
<api>
   <class>
      <name>Song</name>
      <constructor>
         <name>title:duration:artist:album:</name>
         <param>String</param>
         <param>Integer</param>
         <param>Artist</param>
         <param>Album</param>
      </constructor>
   </class>
</api>
```

Objects of this class are instantiated by means of the `title:duration:artist:album:` constructor. The arguments passed to this constructor are of types `String, Integer, Artist` and `Album` respectively. The `Artist` and `Album` classes are then described as follows:

```
<?xml version="1.0"?>
<api>
   <class>
      <name>Artist</name>
      <constructor>
         <name>name:</name>
         <param>String</param>
      </constructor>
   </class>

   <class>
      <name>Album</name>
      <constructor>
         <name>name:</name>
         <param>String</param>
      </constructor>
   </class>
</api>
```

Three important issues need to be stressed. First of all, note how the description of the API not only documents the classes and their constructors, but also describes the runtime object structure for representing information. For example, the description of the `Song` class of the playlist organiser application includes that a `Song` object contains both a `Artist` and `Album` object. Second, the API description is purely declarative in nature. As a consequence, such descriptions are easy to understand and specify, which also makes them easy to change. Last, we require the developer to explicitly specify such a description, which can be considered as a burden. Apart from the fact that this description is used by an automatic code generator that takes over a large part of the developer's task, it can also be considered as an excellent form of documentation. Note, for example, how this description closely resembles a JavaDoc-style of documentation, be it in the form of XML. We can thus easily imagine extracting the information contained within the description from a JavaDoc comment, for example.

3.3. XML → API Mapping

The XML → API mapping describes how information in the XML document should be mapped onto the application's API. Basically, this boils down to describing how the XML document should be traversed and which objects should be constructed during this traversal. For traversing the XML document, we use XPath, which is part of the standard XML processing toolkit. To describe how objects should be created and combined while traversing the document, we introduce a number of special *commands*. Each command is responsible for specifying how an object should be created. The following commands are used in the remainder of this paper:

- the *Simple* command: this command consists of a single XPath expression and optionally a class whose constructor should be called with the value returned from evaluating this expression. If no class is present, the value is a simple string;
- the *Repeat* command: this command consists of an XPath expression, a command and optionally a class. Its purpose is to continuously repeat applying the command for every value of the XPath expression, and optionally call the constructor of the specified class with a collection of all the values obtained by executing the command;
- the *Compound* command: this command is a simple container that contains an aggregation of other commands and the name of the class whose constructor should be called with the value obtained by applying the commands.

More advanced commands exist as well, which are used to convert complex XML documents that require complex traversal strategies. A discussion of these commands is outside the scope of this paper however.

An Example Consider how the XML document of Section 2 should be mapped onto the music organiser's API described in Section 3.2. This mapping is quite straightforward, as the object structure and the XML document structure match quite nicely:

```
<?xml version="1.0"?>
<compound>
   <simple>
      <xpath>/album/title</xpath>
   </simple>
   <simple>
      <xpath>/album/artist</xpath>
      <class>Artist</class>
   </simple>
   <repeat>
      <xpath>/album/song</xpath>
      <compound>
          <simple>
            <xpath>title</xpath>
         </simple>
         <simple>
            <xpath>duration</xpath>
         </simple>
         <class>Song</class>
      </compound>
   </repeat>
   <class>Album</class>
</compound>
```

As can be seen, we define one compound command that consists of two simple commands and one repeat command, and that is responsible for instantiating objects of the `Album` class. Both simple commands use an XPath expression to gather the information they require. The first command retrieves the title of the album, and since it does not specify any class that should represent this title, it is left in the form of a string. The second simple command uses an XPath expression to retrieve the name of the artist, and specifies that this should be used to construct an `Artist` object. Note that we do not specify how this instance should be constructed, as this information is present in the application API description and should not be repeated here. The repeat command will execute its specified compound command for each song in the XML document, as specified by the XPath expression. The compound command merely consists of two simple commands, that retrieve the title and the duration of the specified song and construct an instance of class `Song` with these parameters.

In a similar way, we can specify how the information in the XML document should be mapped onto the API of the playlist organiser as described in Section 3.2. This mapping is less straightforward as the previous one, since the object structure and the document structure do not map exactly. Nonetheless, we can specify the mapping relatively easily as well:

```
<?xml version="1.0"?>
<repeat>
   <xpath>/album/song</xpath>
   <compound>
       <simple>
          <xpath>title</xpath>
       </simple>
      <simple>
         <xpath>duration</xpath>
      </simple>
      <simple>
         <xpath>/album/title</xpath>
         <class>Album</class>
      </simple>
      <simple>
         <xpath>/album/artist</xpath>
         <class>Artist</class>
      </simple>
      <class>Song</class>
   </compound>
   <class>Playlist</class>
</repeat>
```

This mapping specifies a repeat command that will execute a compound command for each song in the XML document, and will instantiate an object of the `Playlist` class with the result. The compound command itself consists of four simple commands, and will instantiate an object of the `Song` class. The simple commands use XPath expressions to gather information from the XML document, and instantiate the appropriate classes with this information.

It should once again be stressed that these descriptions are entirely declarative in nature and entirely specified in an XML format. As such, they nicely complement the other already existing tools in the XML family.

3.4. Code Generator

Our code generator is implemented in SOUL, a logic meta programming environment developed at our lab. SOUL is based on a tight symbiosis between an object-oriented base language and a declarative meta language. This makes it possible to reason about and to manipulate object-oriented programs in a straightforward, declarative and intuitive way [1]. The technique has already been used extensively to implement advanced code generators for diverse application domains such as aspect-oriented programming [2] and domain-specific languages [3]. This strengthens our belief that it is an excellent medium for our purposes as well.

Our code generator thus consists of a number of logic rules, that consult the XML specifications provided by the developer, reason about these specifications and generate appropriate Smalltalk code based on these specifications, A detailed description of this code generator, together with an illustration of how the logic rules are implemented and how they work is beyond the scope of this paper however. We refer the interested reader to [2] and [1].

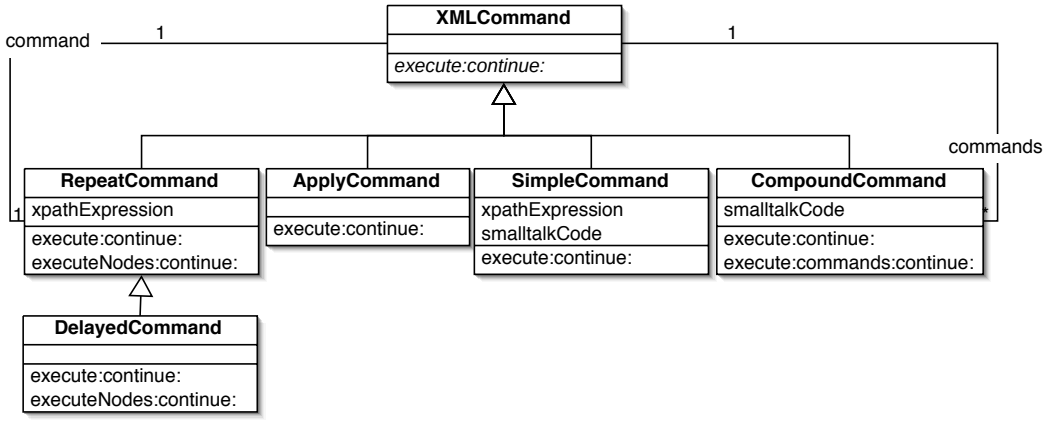
4. Experimental Results

This section discusses the results of applying our approach to the running example of Section 2. We will show the code that is generated by our code generator based on the descriptions provided in the previous section. Before we can do that, we need to elaborate upon the implementation of the XML commands that are used by the code generator.

4.1. XML Command Library

The commands as specified in the XML → API description are implemented as Smalltalk classes in an `XMLCommand` class hierarchy. This hierarchy is depicted in Figure 4. There are a number of peculiarities about each command that should be discussed:

- the `SimpleCommand` and `RepeatCommand` are both parameterised with an XPath expression that gathers the necessary information from the XML document;
- the `SimpleCommand` and `CompoundCommand` are both parameterised by a snippet of Smalltalk code that describes how an object should be instantiated. This code merely calls the appropriate constructor of the appropriate class, as described by the application API and the XML → API mappings;
- the `CompoundCommand` and `RepeatCommand` are both parameterised by other commands that need to



be executed. In the case of the `CompoundCommand`, there can be many such extra commands, whereas for the `RepeatCommand` only one command needs to be specified.

Each command should implement the `execute:continue:` method, which is responsible for the actual traversal of the XML document. The first argument of this method is an XML node to be processed, while the second argument represents a *continuation*, e.g. it specifies where the traversal should continue when the node has been processed. For example. the `execute:continue:` method in the `SimpleCommand` class applies the XPath expression to the specified node, and then executes the snippet of smalltalk code to instantiate the appropriate object. In the `CompoundCommand` class, the method executes all commands upon the specified node, whereas in the `RepeatCommand` the method applies the XPath expression to the specified node, to gather all appropriate nodes, and applies the command to each of these. The `execute:context:continue:` method is mainly used by the `ApplyCommand` and `DelayedCommand` classes, which we will not discuss here.

4.2. Generated Code

Given as input the descriptions presented in Section 3, our code generator generates the following Smalltalk code for mapping information from an XML document to the music organiser's API:

```
sc3 := SimpleCommand xpath: 'title'.
sc4 := SimpleCommand xpath: 'duration'.
cc1 := (CompoundCommand
       new: [ :coll |
                   Song
                       perform: #title:duration:
                       withArguments: coll asArray ])
            add: sc3;
            add: sc4.
rc1 := RepeatCommand
xpathExpression: '/album/song'
command: cc1.
sc1 := SimpleCommand xpath: '/album/title'.
sc2 := SimpleCommand
       xpath: '/album/artist'
       resultBlock: [:res |
                    Artist
                        perform: #name:
                        withArguments: res asArray ].
^(CompoundCommand
         new: [:coll |
                    Album
                        perform: #title:artist:songs:
                        withArguments: coll asArray ])
 add: sc1;
 add: sc2;
 add: rc1.
```

Note how in Smalltalk, strings are delimited by single quotes (e.g. '/album/song' is a string) and square brackets delimit a *block*: a piece of Smalltalk code that is not evaluated immediately, but whose evaluation is deferred until the `value` message is send to it. Blocks are thus the Smalltalk way of implementing closures. Furthermore, we use an advanced reflective feature of Smalltalk to instantiate the objects: instead of invoking the constructor of the class directly, we use the `perform:withArguments:` method. This method will invoke the first argument (which denotes a method) with the second argument, represented as a collection, as arguments. For example `Album perform: #title:artist:songs: withArguments: coll asArray` will invoke the `title:artist:songs:` constructor of class `Album` with as arguments all elements of the collection in `coll`. It is implicitly assumed that this collection only contains three elements, corresponding to the three arguments of the constructor.

This code presented above is a straightforward translation of the mapping specified in Section 3.3. It instantiates four simple commands: `sc1, sc3` and `sc4` simply gather strings representing the title and duration of a song and the title of the album, the `sc2` command retrieves the artist from the XML document, and instantiates an object of the `Artist` class using its `name:` constructor. The compound command `cc1` will instantiate a `Song` object, each of which contains the title and the duration of the song. The `rc1` command will make sure this happens for every song present in the XML document. The `cc2` command at the

end will make sure an `Album` object is instantiated with the appropriate title, artist and song objects.

The code generated for the playlist organiser application is similar:

```
sc1 ;= SimpleCommand xpath: 'title'.
sc2 := SimpleCommand xpath: 'duration'.
sc3 := SimpleCommand
              xpath: '/album/artist'.
              resultBlock: [ :res |
                        Artist
                           perform: #name:
                           with: res asArray ].
sc4 := SimpleCommand
              xpath: '/album/title'.
              resultBlock: [ :res |
                        Album
                            perform: #title:
                            with: res asArray ].
cc1 := (CompoundCommand
       new: [ :coll |
                Song
                    perform: #title:duraction:artist:album:
                    withArguments: coll asArray ])
            add: sc1;
            add: sc2.
            add: sc3;
            add: sc4.
^RepeatCommand
        xpath: '/album/song'
        command: cc1.
```

Once again, this code is a straightforward translation of the XML → API mapping given in Section 3.3 and extensively uses the components offered by the `XMLCommand` hierarchy.

5. Future Work

Although the initial experiment reported upon in the previous section shows good results, in the future, we need to experiment with more complex application APIs and more complex mappings. Up until now, the object structures we considered were very simple, consisting of a few objects that are easy to combine and straightforward to instantiate. We can easily imagine real-world applications using more complex object structures, consisting of many more objects, that are more complicated to instantiate and that should be combined in more complex ways than we considered here. Moreover, the XML → API mappings we considered are also quite simple. This is a result of the fact that the XML document we considered is simple and straightforward to traverse. The more complex the XML document's structure, the more complex the traversal will become. However, we strongly believe our approach will be able to cope with more complex traversals, since it is implemented by means of continuations and delayed evaluation, and it has been proved many times that this greatly facilitates implementing traversals. We do feel that more XML commands may be required, but we consider this only a minor extension to our approach.

Another issue that deserves some more attention is the reusability of the XML ↔ API mappings. Although not explicitly considered in this paper, as it is now, we require a developer to specify both this mapping and the reverse API → XML mapping. We feel this puts to much burden upon the developer, and we therefore want to investigate whether it is possible to use one single mapping to specify both directions of the translation. Since we use a purely declarative approach, this should pose no problem. However, the mappings as they are represented now would presumably not fit this purpose and should therefore be changed.

To further reduce the burden of specifying all necessary descriptions, we would like to provide a graphical user interface that allows a developer to specify the descriptions in a more intuitive way. For example, the application's API could be specified by clicking on the appropriate class and constructor methods, and the tool could then automatically generate a corresponding specification, perhaps based on some JavaDoc-style comments that are already present. It would even be possible to identify the necessary argument classes automatically and construct a specification for these as well. Furthermore, the XML ↔ API mappings could also be constructed in a more intuitive way, by implementing some drag and drop operations from a application's API to the XML document structure and vice versa.

6. Related Work

Model Driven Architecture [4] also starts from a declarative (UML) description to produce running code. MDA is a new way of writing specifications and developing applications, based on a platform-independent model (PIM). A complete MDA specification consists of a definitive platform-independent base UML model, plus one or more platform-specific models (PSM) and interface definition sets, each describing how the base model is implemented on a different middleware platform. A complete MDA application consists of a definitive PIM, plus one or more PSMs and complete implementations, one on each platform that the application developer decides to support. MDA separates implementation details from business functions. Thus, it is not necessary to repeat the process of modelling an application or system's functionality and behaviour each time a new technology (e.g., XML/SOAP) comes along. In terms of products, MDA will be implemented by tools - or suites of tools - that integrate modelling and development into a single environment that carries an application from the PIM, through the PSM, and then via code generation to a set of language and configuration files implementing interfaces, bridges to services and facilities, and possibly even business functionality.

In order to express the interaction between the higher phases of software development and the implementation level better, our lab is experimenting with declarative meta

programming [5]. The underlying idea is that abstract information can be expressed on top of the actual programming language concepts by means of logic facts and declarative rules [6]. By explicitly reasoning with these facts and rules, it becomes possible to provide automated support for a variety of tasks in the software development process in a uniform way. e.g.

- check source against certain constructs, conventions, patterns
- search source for certain constructs, conventions, patterns
- extract certain constructs, conventions, patterns from source
- enforce certain constructs, conventions, patterns in source
- transform source based on high-level declarative description
- generate source from high-level declarative descriptions

A number of successful experiments with declarative meta programming have already been carried out in the context of code generation in soul [2].

Other approaches of code generation for specific environments as the UML CASE Tool is described by Park [7]. Milosavljevi describes a tool that follows simple rules about mapping JavaBean components to a database schema and generates components, as well as a set of standardised JSP pages. The mapping is specified as an instance of an XML Schema document [8]. Cleaveland [9] describes how to use XML to describe the programs you need and then write a Java program template to generate them automatically. Different techniques are described to generate code with DOM, JSP and the combination of XPATH and XSLT.

7. Conclusion

This paper presented an approach for automatically generating code that retrieves information from an XML document and translates it to an application's API. The approach is based on declarative descriptions of the application's API, the XML document's structure and the way the information from the XML document should be mapped onto the application's API. These descriptions form the input for a code generator, that used advanced code generation and manipulation techniques to generate executable code that translates the XML document to the application's API. We reported upon an initial experiment with this approach in a multimedia context, which showed promising results and thereby illustrates that the approach is both useful and feasible.

8. Acknowledgements

This work is a result of a joint collaboration between Vlaamse Radio en Televisie (VRT, public broadcaster of Flanders), IMEC and Vrije Universiteit Brussel (VUB). This is one of the e-VRT projects funded by the Flemish government. We especially thank Steven Van Assche for reviewing the paper.

References

[1] Roel Wuyts, *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*, Ph.D. thesis, Departement Informatica, Vrije Universiteit Brussel, 2001.

[2] Kris De Volder, *Type-Oriented Logic Meta Programming*, Ph.D. thesis, Departement Informatica, Vrije Universiteit Brussel, 1998.

[3] Johan Brichau, Kim Mens, and Kris De Volder, "Building composable aspect-specific languages," in *Proceedings of the Conference on Generative Programming and Component Engineering*. 2002, Springer-Verlag.

[4] OMG Group, "www.omg.org/mda/," 2003.

[5] Theo D'Hondt, Kris De Volder, Kim Mens, and Roel Wuyts, "Co-evolution of object-oriented design and implementation," in *Int. Symposium on Software Architectures and Component Technology: The State of the Art in Research and Practice*, 2000.

[6] Roel Wuyts, "Declarative Reasoning about the Structure of Object-Oriented Systems," in *Proc. TOOLS USA'98, IEEE Computer Society Press*, 1998, pp. 112–124.

[7] D.H. Park and S.D. Kim, "Xml rule based source code generator for uml case tool," in *Eighth Asia-Pacific Software Engineering Conference*, 2001.

[8] Branko Milosavljevi, Milan Vidakovi, and Konjovi Zora, "Applications of java programming: Automatic code generation for database-oriented web applications," in *Proceedings of the inaugural conference on the Principles and Practice of programming*, 2002.

[9] J. Craig Cleaveland, *Program Generators with XML and Java*, Prentice Hall, 2001.