

XML-based Programming Language Modeling: An Approach to Software Engineering

Christian Reichel
University of Heidelberg
Heidelberg, Germany
christian.reichel@informatik.uni-heidelberg.de

Roy Oberhauser
Corporate Technology, Siemens AG
Munich, Germany
roy.oberhauser@siemens.com

ABSTRACT

Today's software faces escalating technical and business difficulties, yet it continues to be coded in static, inflexible structures that are not prepared for automation and agility. xApproach is an XML-based approach to software engineering that leverages XML-based language representations and pipeline transformations to provide a consistent and flexible solution for number of issues, including complexity, comprehension, and automation aspects. A core framework (FXLF) in combination with the editor (FXLE) supports role-based activities while removing the burden of programming in XML. Usage scenarios evaluated include customization, separation of concerns, domain-oriented languages, and technology mapping, with results showing a range of benefits, applicability, and prospects.

KEY WORDS

Software Engineering, Automation, Modeling, Programming Languages, Domain-Specific Languages, XML

1. Introduction

In the face of escalating technical difficulties (e.g., complexity, distribution, integration) and business forces (e.g., development time, maintenance costs), software agility and flexibility properties become increasingly important. Current software code is primarily expressed in static structures that constrain agility; they are often difficult and time-consuming to change and not prepared for future automation (software factories, generative programming, etc.). Additionally, views of these structures (e.g., documentation, architecture, operation) are not based on an interoperable, unified model, and various tools have their own internal representation (Eclipse, Sun Java Studio Creator, e.g., based on Abstract Syntax Trees (ASTs). Analogous to plain-text documents, these representations are missing a standardized transformation capability (e.g., XSL Transformations (XSLT)) and must rely on proprietary mechanisms to make model modifications, thereby limiting interoperability and automation. These static

structures exacerbate solutions to issues such as complexity, comprehension, and automation, and thus constrain further improvement possibilities in software engineering.

For example, it is difficult to optimize the presentation of software source code in support of the various roles (programmer, operator, etc.) and phases (e.g., development, operation, and maintenance) involved in the production and use of software. Regarding both complexity and comprehension, available abstraction mechanisms do not adequately integrate these aspects in a consistent fashion as to how software artefacts are presented. Thus, viewers of software are often confronted with distracting and irrelevant information for their current interest, and difficulties exist in keeping the various related pieces in sync. Standardized and generic modeling approach such as Model Driven Architecture (MDA) with XML Metadata Interchange (XMI), address some of the aforementioned aspects. Yet in many cases, bottom-up strategies with domain-specific models and views can provide simpler, lightweight, less error-prone, and more agile solutions.

Concerning the underlying code structures, current programming languages restrict flexibility in their keywords and grammar and have limited customization and modification mechanisms. E.g., while data transformation is quite common in daily Enterprise IT activities, no common standards for code transformation have been established. Although XML has a proven record for interoperability, XML-based source code representations (e.g., JavaML [1] and CppML [2]) in the context of a unified and interoperable model that address the software engineering aspects of complexity, comprehension, and automation have not received sufficient attention.

This paper investigates the applicability of XML-based source code representations for a unified and interoperable model approach to the above issues in software engineering. In section 2, the basic principles involved are elucidated. Section 3 validates the fundamental approach with a reference implementation, and presents applicable tooling that enhances its usage.

Section 4 analyzes the overall solution results with measurements, possible areas of application, and mentions future work. In Section 5 related work is discussed, followed by a conclusion.

2. Solution Approach

An approach addressing the aforementioned aspects within software engineering involves certain key requirements and objectives. In the first step, all basic mechanisms are expressed by two main principles: the *representation principle* and the *transformation principle*. As shown in Fig. 2.1, the *representation principle* introduces central XML-based models for programming languages, whereby the conversion between the *model* and all related *views* is specified by type T_a transformations. XML was chosen due to its standardization, ubiquity, and tool support.

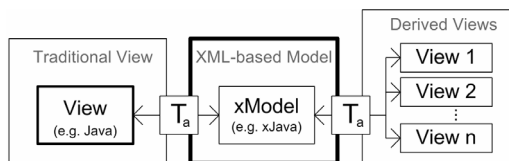


Figure 2.1: Representation Principle

In this context, the term *view* includes common, e.g., plain text, and derived, e.g., hierarchical or graphical, representations, yet all based on the same XML-based model. This includes every traditional non-XML-oriented source code document of a given programming language Lang (e.g., Java) that can be alternatively represented in an XML-based document *xModel* (e.g., *xJava*), which forms the model for all derived views (e.g., a Java syntax view with custom keywords).

Given these central models, the *transformation principle* defines two additional transformation types that allow the modification of the underlying models (T_b) and the conversion between models (T_c). In Fig. 2.2, T_c can be illustrated with an example wherein *Model A* can be a high-level description (e.g., a Service Language (SL) that is a Domain Specific Language (DSL)) and *Model B* can be a technology-specific (e.g., uses platform-specific APIs) or a language-specific model (e.g., *xJava*).

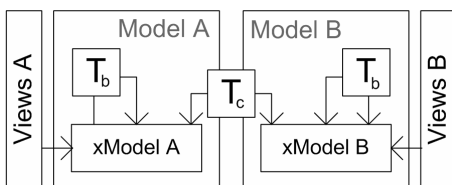


Figure 2.2: Transformation Principle

The combination of these principles are united in an XML-based approach called *xApproach*. The inherent transformation capability of *xApproach* yields dynamism

and flexibility, and addresses the following problem aspects:

- Type T_a transformations enable the creation of different views, and, via the adaptive presentation of information, various abstraction levels can be supported → addresses complexity and comprehension aspects,
- Type T_b transformations allow the modification (e.g., insertion, removal of blocks) of the underlying model → addresses agility aspects as well as language and software personalization/customization,
- Type T_c transformations provide the conversion of models → addresses certain technology mapping and integration aspects.

All transformation types enable an improved automation capability and in combination with XML-based models provide the foundation for a unified approach.

In this context, specific *xModels*, instead of common plain-text views and their associated ASTs, form a central component to software engineering. T_a related approaches for producing XML-based representations of source code have been described in research work, e.g., JavaML and srcML [3]. In the case of srcML, a data view and document view are supported, thus, e.g., it can be used as an exchange format as well as an independent layer of representation between the developer and the source code. While *xApproach* encompasses these efforts, its additional and explicit focus on derived views (e.g. DSLs) and the exploitation of standardized transformation mechanisms for XML-based model documents enables flexible capabilities in many fields of application. In this way, e.g., workflow graphs or XML pipelines [4] execute chains of transformations (e.g., via XSLT or the XML Query Language (XQuery)) on a given *xModel* document.

3. Solution Realization

To provide a reference implementation of *xApproach* and realize its objectives, a core framework (FXLF) was developed. The interaction of the framework functionality with the human roles involved in software is supported with an editor (FXLE) that presents the graphical interfaces and views, and thereby enables the application of the principles in various problem domains.

3.1 Core Framework (FXLF) Implementation

FXLF is responsible for the transformation of models and the generation of specific views. To realize the representation principle, two directions of type T_a transformations are considered:

- View → xModel. In the case of a traditional view (e.g., a programming language such as Java) the parser generator ANTLR [5] provides the front-end for the *xModel* creation (see *xJava* example in Fig. 3.1); standard grammars are slightly modified (e.g., by

activating AST generation, etc.) and used as the basis for the automatic lexer/parser generation which builds the subsequent AST representation. A visitor is then used to traverse the tree and to generate XML nodes (element, attribute, etc.) based on the current AST-node type. Special mapping properties (name, formatting, etc.) are specified in a XML-based visitor configuration file.

In all other cases, the transformation principle can be applied with intermediate models (including type T_b and T_c transformations) when necessary.

```
<method>
  <modifiers><public/><static/></modifiers>
  <type><void/></type>
  <identifier>main</identifier>
  <parameter>
    <type>
      <identifier>String</identifier>
      <array_declarator/>
    </type>
    <identifier>args</identifier>
  </parameter>
  <statements> ... </statements>
</method>
```

Figure 3.1: xJava snippet for Java main method

b) xModel→View. The XML form of xModel supports the direct usage of XML-based transformation languages like XSLT and XQuery. The XML framework for Java DOM4J is used for XSLT, where templates allow the pattern-based definition of rules (e.g., for every node type). These can generate traditional plain-text views such as formatted Java, filtered views (e.g., middleware-only), or graphical output formats (e.g., SVG). Each file encompasses all the rules necessary for generating that specific view (see Fig. 3.2).

```
<xsl:template match="method">
  <xsl:apply-templates select="modifiers"/>
  <xsl:apply-templates select="type"/>
  <xsl:apply-templates select="identifier"/>
  <xsl:text></xsl:text>
  <xsl:apply-templates select="parameter"/>
  <xsl:text></xsl:text>
  <xsl:apply-templates select="throws"/>
  <xsl:apply-templates select="statements"/>
</xsl:template>
```

Figure 3.2: xJava→Java XSLT method template

The combination of a) and b) enables bidirectional mapping between models and views.

To realize the transformation principle, type T_b and T_c transformations are applied for **xModel**→**xModel*** and **xModel A**→**xModel B** transformations respectively. A description of the modification processing steps (chain or graph) is done using XML Pipeline utilizing *processes* that have XML I/O. The XML Pipeline specification was implemented separately because no Ant-task-independent implementation was available. Various process plugins are supported (XSLT, XQuery, etc.). Usages include xModel modification (e.g., weaving of data, removal of

code, etc.). These process descriptions can be reused as shareable library entries, such as an entry for supporting tracing capabilities. For type T_c transformations, model conversions are accomplished in a similar fashion as type T_b but have a different purpose and result (e.g., SL to Web Service Description Language (WSDL), SL to Java, etc.). Validation mechanisms for achieving a certain grade of syntactical correctness of xModel documents rely on XML Schema validation or generated model parsers.

3.3 FXL Editor (FXLE)

The GUI-based FXLE application (see Fig. 3.4) uses the aforementioned FXLF implementation. It consists of a basic project editor with access to xApproach core functionality components. Realized are: various views of source code documents (traditional, XML-based, filtered, SVG, etc.), a graphical pipeline management and execution (graph-based, intermediate result views, debugging with stepping, etc.), varying libraries (predefined transformations, processes, pipelines, language converters, etc.), as well as automatic xBuild handling of projects. Roles are supported with perspectives that combine available views and that address their individual needs. Additionally, programming languages can be integrated via the flexible import of ANTLR grammar files.

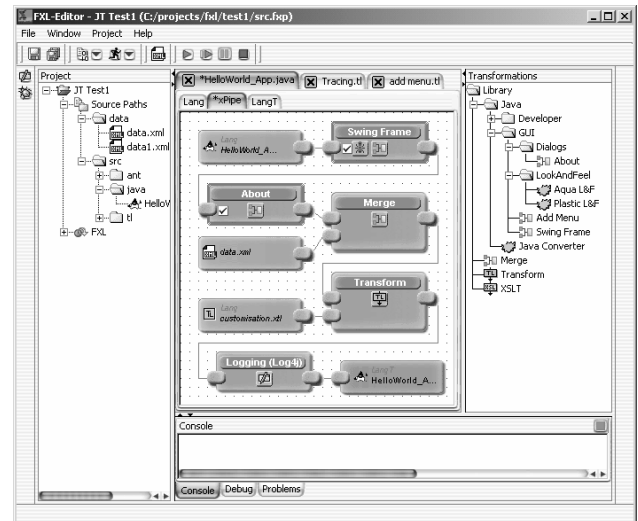


Figure 3.3: FXL Editor

4. Solution Results

The solution will now be assessed using selected metrics of FXLF, software engineering usage scenarios with FXLE, and an evaluation of the current implementation. This is followed by identified future work.

4.1 FXLF Metrics

To determine if the FXLF implementation is suitable for a realistic usage scenario, performance and size were initially measured. In this context, an xBuild task encompasses the typical processes of Java→xJava conversion (XJC), xJava→Java conversion (JC) via

XSLT, and the compilation Java→bytecode (C) via javac from J2SDK 1.4.2. The results shown in Fig. 3.3 were made for a project example consisting of 425 Java classes and 27,637 LOC. The graph on the left shows time measurements which were the average of 20 cycles across all project classes for different systems. On each system the normal Build time C of the project (see the left 100% bar) was compared to the xBuild time (on its right) which consisted of the aforementioned conversion times for xJC, JC, and C. This shows that the complete xBuild takes approximately 8-9 times longer than the C build and that JC takes the largest portion (78% of the total). With increasing system performance nearly linear reductions in xBuild times are observed. The graph on the right shows that xJava representations of Java code require approximately 6 times more space.

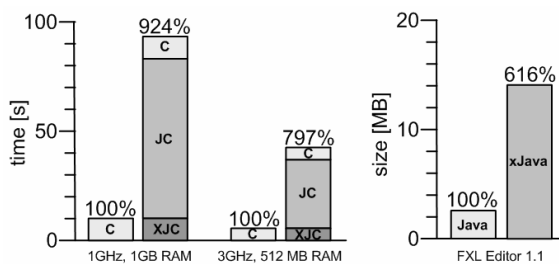


Figure 4.1: FXLF Measurements

The pipeline execution time should be considered as an additional factor in the xBuild time, but was not included in these measurements due to the wide spectrum and complexity in possible pipeline structures. However, as an indication of the potential magnitude, a simple XSLT copy of an xJava document (duplication) takes 4 times longer than the Java→xJava conversion time of that source file.

4.2 Use Case Tests using FXLE

To demonstrate the applicability and benefits of xApproach/FXLE in the context of software engineering tasks, various usage scenarios were successfully applied and tested. These included:

DSLs and Technology Mapping

If the description and development of software is moved to a high(er)-level of abstraction, the developer must not categorically choose concrete low(er)-level technologies and implementations and can thus concentrate on coding the functionality. FXLE can address both this abstraction aspect as well as the process of lower-level technology mapping, although issues such as correctness and completeness still need to be solved.

Test scenarios that illustrate comprehension, flexibility and time improvements, comprised:

- Several customized Java syntaxes for personalized language wishes (that are transformed to valid Java syntax).

- The creation of DSLs to support different abstraction levels; a Transformation Language (TL) and a SL were specified with xModels xTL and xSL and several views.
- Model conversions for TL to XSLT and to a limited extent TL to XQuery were defined and validated.

Customization/Personalization

FXLE can help to adapt, extend or create customized software. The XML-based technologies can be used to encapsulate changes in special files and apply them as transformations in different scenarios. The term customization of software generally refers to the activities:

- Removal or modification of existing code blocks, e.g., to replace customer-bound algorithms
- Insertion of new code blocks, e.g., to add extra-functionality a customer paid for
- Activation/deactivation of existing code blocks, e.g., to activate inherent functionality

Tests that show reusability, adaptation and time improvements consisted of the customization of FXLE (e.g., the editor layout and dialogs with external data sets, updates for additional features, Swing GUI generator, etc.) via TL transformations.

Variation Points (VP)

VPs enable the ability to change pre-defined areas of the code after development time. xModel documents can represent variation points by both coupling objects and XML code (e.g., avoiding runtime parsing of configuration files by setting initialization variables appropriately in the code) and enabling transformations to be carried out at various points in time. For instance, an xJava document that holds the initialization values can be dynamically loaded as a Java class without the usage of runtime XML frameworks such as JAXB and Castor [6].

Simple VP tests allowed operator modification of xModel-based code on predefined points in the software (e.g., choice of security mechanisms) and exemplify related simplification and maintenance improvements.

Separation of Concerns (SOC)

Many approaches to modularizing crosscutting concerns exist (e.g., Aspect-Oriented Software Development (AOSD)). FXLE can be used to address SOC. Critical aspects like security policies or error handling are stored in transformation files (e.g., in an Aspect Language (AL)) and, when desired, weaved into the source code via transformations.

Flexibility and reusability tests consisted of xModel modifications to achieve SOC, specifically for logging/tracing (an XSLT-based transformation “transparently” added tracing code to every method).

4.3 Solution Evaluation

Practical usage of xApproach in combination with appropriate tooling such as the FXLE has shown that it is viable for addressing the problem areas considered. The system performance results show that its application in current development and operational systems is feasible. From this point of view, it appears promising that xApproach can be the basis for a unified solution approach to various software engineering problems such as complexity, comprehension, and automation.

Regarding measurements of FXLF, the following aspects should be considered: the xBuild process had no logic, e.g., to compare timestamps and avoid unnecessary Java→xJava→Java conversions and compilations, thus all files were regenerated. Separately, incremental builds could aid the perceived performance. The XSLT transformation was shown to be a significant bottleneck of 80% of the xBuild time (without pipelines), and thus other or highly optimized XSLT implementations can reduce these times. For typical “best practice” programming files that are not horrendously large, the memory footprint on the systems in Section 3.1 was not shown to be an issue, even when 50 concurrent xBuild threads were executed. The 6x size increase of xJava vs. Java files can be improved by reducing tag name sizes, however, once standards for xModels become prevalent, the amount of reduction achievable will be limited.

Since xApproach transformations are dependent on an xModel, to achieve wide applicability and reusability of the transformation libraries and entries a standardization of these xModels (e.g., for programming languages) is necessary. In the context of complex scenarios, xModel transformations and modifications require additional attention as well as solution approaches due to issues that can occur in areas such as correctness, completeness, and efficiency. As these transformations become more widespread and available, mechanisms must be included to assure the appropriate quality and trust for the user.

The XML Pipeline engine was shown to be very flexible, even if open issues remain in the pipeline execution such as addressing order dependencies between pipelines, versioning, conditional branching, etc. Extensions to the XML Pipeline standard are possible, however, other solutions are feasible, e.g., usage of the XL platform [7] as a workflow engine.

FXLE has shown that the xApproach principles had no negative impact for the user since the tools can hide the approach from the user, yet offer access when desired. Different usage scenarios were applied and shown to be flexibly supportable across a wide spectrum without requiring significant changes to the tooling. The application of this degree of flexibility can positively affect role activities, e.g., programming via personalized language extensions, optimized views, etc. Furthermore,

the use of xApproach in conjunction with state-of-the-art XML-based software engineering frameworks (e.g., XMI) can deliver additional benefits.

One noticeable aspect in the evaluation of FXLE was that there is a need for more flexible transformation and language description/creation, e.g., via graphical solutions that enable automated grammar definition. Here, DSLs such as SL and TL (which will be described in separate papers) can be helpful and more efficient.

4.4 Future Work

Areas for future work were identified and include the following activities:

- Improving xBuild performance times: results showed that the transformation engine (XSLT) is a significant factor, and therefore optimized implementation alternatives need to be considered.
- Practical and efficient DSL expression and creation: Existing XML-based transformation and service languages are cumbersome; in this area TL and SL are being completed and graphical views are being considered.
- Correctness, completeness, and validity checks for model transformations.
- Pipeline/workflow improvements and versioning issues.
- Greater integration with common developer tools by porting of the xApproach/FXLF to IDEs such as Eclipse.
- Distributed pipeline support: xApproach would enable a new paradigm of distributed software build processing across systems. The overhead cost versus the benefit needs further investigation.
- Synchronization of documentation with code: the power of XML-based code representation enables the cross-referencing with XML-based documentation artefacts.

Generally, the lack of standardized xModel language representations inhibit the ability to address current and future software engineering issues in the aforementioned manner, and usage of xApproach in more complex and widespread scenarios would require this standardization for true interoperability and exchangeability, yielding significant benefits for the community.

5. Related Work

In this paper, xApproach, its realization, and selected fields of application were presented. The section below mentions related and representative work in the fields of XML-based source code representation and domain-oriented programming.

With regard to source code representations, JavaML [1] provides an XML-based representation of Java source code. JavaML uses the Jikes Java compiler framework to transform the source code from a plain-text representation into JavaML, and then XSLT for the reverse transformation from JavaML into Java. This representation does not preserve the original structure of

the source code (e.g., formatting information) and has certain weaknesses in comparison to semantically enhanced grammars [8]. Alternatively, the Source Markup Language (srcML) [3] creates a multi-layered document where the original source code (including formatting) forms the base-layer and the added XML-based markup layer reveals, e.g., the syntactic information and the underlying structure. Thus, the original source code remains intact and can be easily extracted by removing all XML tags from the srcML format. While xApproach goes beyond these source code representation aspects, JavaML and srcML can be used as xModels for Java within the *representation principle*.

In the area of SOC and Aspect Oriented Programming (AOP), the operator approach [9] proposes XML-based operators (enclosed by the `transform` tag) as an extensible aspect language. The prototype has been realized using JavaML for xJava creation, XML tools for xJava modification and (in one case) XSLT for Java source code generation. This approach addresses a subset of the xApproach software engineering goals in the context of SOC, such as the XML-based creation of an Aspect Language (AL) and the improvement of the code structure. Generally, the operator approach focus is restricted to the AOP problem domain (e.g., aspects, join points).

SmartTools [10] presents a software generator for domain-specific or programming language creation based on XML and object-oriented technologies (e.g., visitor design pattern, aspect, etc.). Its modular architecture and generic visualization tools (concerning Xpp, lmltree, BML, etc.) form the basis for different views on documents. The SmartTools approach also addresses a number of xApproach goals, such as the creation of DSLs and the development of a higher-level transformation language. Basically, its approach differs in some essential parts (e.g., framework generator, AST visitors, message controller) and its focus is narrower than xApproach.

6. Conclusion

The escalating technical difficulties and business forces in software engineering require a new approach to achieve the flexibility and agility properties necessary to enable enhanced automation and comprehension in the onslaught of ever more code, aggregation, and integration. New forms of programming languages will play an essential part, and good reasons exist for future programs to be stored as XML documents (e.g., so that programmers can represent, process, and transform code and meta-data uniformly) [11].

Starting with the theoretical foundation of *xApproach* as expressed in the *representation* and *transformation principles*, a core framework was realized to evaluate its applicability to a number of software engineering challenges. Automation capabilities are improved through the unifying *xModel* that enables standardized type T_a , T_b and T_c transformations, and the evaluation

of FXLF demonstrated that a general usage of xApproach is feasible. Additionally, an editor FXLE that utilizes FXLF was implemented to address several complexity and comprehension aspects. It supports the creation of new abstraction levels (customization, DSLs, etc.) as well as the inclusion of role-based and activity-based views throughout the software lifecycle. Hereby, for instance, programmers, testers, or operators are able to work with a suitable language that they can comprehend and with views that present the essential concerns of the software applicable to their current interest. The combination of xApproach with FXLE has shown that various software engineering scenarios can be uniformly addressed (e.g., customization, DSLs, technology mapping, generative programming) while providing a tremendous untapped potential for addressing current and future agility and automation issues.

7. Acknowledgements

We would like to thank Ulrich Dinger and Martin Saler for their implementation work.

References:

- [1] Greg J. Badros, JavaML: A Markup Language for Java Source Code. *Proceedings of 9th International World Wide Web Conference (WWW9)*, Amsterdam, The Netherlands, May 13-15, 2000.
- [2] Mamas, Even, Kontogiannis, Kostas, Towards Portable Source Code Representations Using XML. *Proceedings of WCRE'00*, Brisbane, Australia, November 2000, pp. 172-182.
- [3] M.L. Collard, etc., Supporting Document and Data Views of Source Code. *Proceedings of the 2nd ACM Symposium on Document Engineering (DocEng 2002)*, McLean, VA, November 8-9, pp. 34-41, 2002.
- [4] N. Walsh, E. Maler, XML Pipeline Definition Language Version 1.0. W3C Note 28 February 2002. Available from <http://www.w3.org/TR/xml-pipeline>.
- [5] Terence Parr, ANTLR: Another Tool for Language Recognition. Available from <http://www.antlr.org>.
- [6] The Castor Project: An open source framework for Java. 2004, Available from <http://www.castor.org>.
- [7] D. Florescu, A. Grünhagen, D. Kossmann, XL: a platform for Web Services. *CIDR 2003*, CA, USA.
- [8] H. Simic, M. Topolnik, Prospects of encoding Java source code in XML. *ConTel 2003: 7th International Conference on Telecommunications*, June 11 -13, Zagreb, Croatia, 2003.
- [9] S. Schonger, et al., Aspect oriented programming and component weaving: Using XML representations of abstract syntax trees. *Workshop Aspektorientierte Softwareentwicklung*, Universität Bonn, Germany, 2002.
- [10] I. Attali, et al., Aspect and XML-oriented Semantic Framework Generator: SmartTools. *Second Workshop on Language Descriptions, Tools and Applications, LDTA'02*, Grenoble, France, 2002.
- [11] Gregory V. Wilson, Extensible Programming for the 21st Century. January 2004. Available from <http://pyre.third-bit.com/~gwwilson/xmlprog.html>.