Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science 94 (2004) 71–79

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Three-Layered Source-Code Modelling

Anthony Cox[1]

Faculty of Computer Science
Dalhousie University
Halifax, NS, Canada

Charles Clarke[2]

School of Computer Science
University of Waterloo
Waterloo, ON, Canada

Abstract

Models of software must serve many roles; storage management, information representation and information interchange. A model tailored to a specific role is often inadequate for, and incapable of, supporting other roles. For this reason, we advocate a multi-layered approach, where each successive layer fulfils a specific role. Our current experience with a 3-layered implementation indicates that the use of a storage layer decreases the demands upon conceptual and interchange representation models. Furthermore, we believe that a sufficiently general storage layer can support the simultaneous use of multiple conceptual models. In this approach, the storage layer and its associated database is an enabling technology for the integration of multiple representation models.

Keywords: Source-Code, Storage Model, Conceptual Model

1 Introduction

Models and schemas provide the representations needed to express information extracted from software systems. Each model is subject to tension between the needs of the model's users and the properties of the data to be modelled. Models feel pressure from above, in support of various user applications, and

[1] Email: amcox@cs.dal.ca
[2] Email: claclarke@plg.uwaterloo.ca

doi:10.1016/j.entcs.2004.01.003

from below, in expressing a wide variety of potentially unrelated information. It is unreasonable to expect any model to be able to support all tasks, or to express all information. Furthermore, as indicated by Blaha [2], tasks such as data storage and data interchange require fundamentally different models. Interchange models tend to be smaller, easier to parse and have higher level schemas.

As an example, consider XML. As a meta-model, XML permits the expression of schemas defined for a specific role. To express the syntactic structure of code, SrcML [13] and JavaML [1] both define XML compliant schemas. However, the former views the source-code as the foundation and adds markup to the code while the latter views the model as the foundation and adds the code, as attributes, to the representation. In this case, XML is used in two ways to accomplish the same purpose but to support differing higher level pressures. SrcML is designed to assist programmers working with the source-code while JavaML is designed to be easily manipulated by other tools for XML.

GXL [7], an XML schema, can be used to express graph-based software structures. When used for this role, GXL has little relationship to SrcML or JavaML. In this case, pressure from below, through the desire to express unrelated concepts, is exhibited.

To further complicate matters, XML is designed to express hierarchically structured information. However, as Cordy *et al.* [4] have indicated, source-code contains information, such as source-code factors, which is not hierarchically structured. As well, source-code's syntax and format (line/file structure), while both hierarchically structured, can not be simultaneously represented using a single hierarchy. Although the use of SGML and its `concur` construct addresses the latter issue, SGML is still not capable of representing non-hierarchical information. This complication demonstrates the inability of one meta-model, XML, to satisfy all representational requirements. The use of alternative meta-models, such as the Dagstuhl Middle Model (DMM) [11], does not provide a solution as each meta-model exhibits its own representational limitations resulting from design choices.

We believe that a solution to these issues is to support multiple schemas and representation models. The key to this response is to utilize a multi-layered approach, where each layer minimizes the constraints it imposes on higher layers and decouples higher layers from the constraints imposed by lower layers. In this article a three-layered implementation and its elements are detailed.

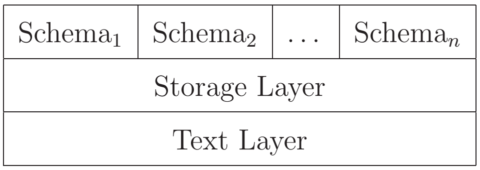

Fig. 1. A System for Representing Software

2 A Multi-Layered Approach

Simplistic one-dimensional models are limited in their ability to satisfy the demands placed upon them. In response, we advocate a multi-layered approach, where each layer supports a specific role. While there are many configurations for a multi-layered system, we propose a 3-layer, tree-structured system, as shown in Figure 1. Future research may show that additional layers are necessary, and it is likely that having fewer layers will lack the flexibility our system provides.

The approach is source-code based since extracted information must be obtained from the source-code, or from a model generated from the source-code. The common origin, in source-code, of all information suggests that the code can be used to integrate and relate extracted information. Low-level models, such as abstract syntax trees (AST), provide an alternative basis but at the cost of losing information only obtainable from the source file. For example, ASTs are generated by a parser and require code to be processed in preparation for parsing. Preprocessing directives such as macros must be removed, and consequently, are not expressed by an AST. The textual nature of source-code permits information that is not based on source-code to be represented and integrated using unstructured text, similar to comments within the code.

2.1 Text Layer

The lowest layer is the text layer. In most software systems, this layer is easily represented using ASCII or Unicode, however, in historic and unique systems, this layer can be replaced with EBCDIC or some equivalent encoding. As well, the layer abstracts operating system specific functionality for accessing the textual elements of a software system. For the most part, this layer is of little importance with modern computer systems. The primary role of the layer is to decouple low-level text management issues from higher layers and provide support for working with legacy software, developed for alternative operating systems and character representation formats.

In our present system [5], the text layer consists of a few simple functions to translate non-printing and extended ASCII characters into a storable format. Complementary routines return these characters to their original format on retrieval. This ad hoc interface is sufficient, but in future work, we hope to develop a more defined interface.

2.2 Storage Layer

Layer 2 is the key to the system's flexibility. The decoupling of storage and conceptual code-modelling issues permits the use of a common storage system (database or repository) for all conceptual schemas. Users and tools can access every schema using a single interface, as provided by the storage system. The layer also provides a bridge between the unrelated information contained in the various schemas. In our source-based approach, schema integration is achieved through the mapping of each schema entity to the region of source-code from which it was extracted.

The storage layer is implemented using a database or repository system. For this reason, the storage model is database specific. Examples of storage models include the relational model, object models and text models. It is not anticipated that there will be significant scalability issues as modern databases are used routinely to store data-sets in the gigabyte range.

To support the representation of multiple conceptual and interchange formats, the storage model must be capable of implicitly representing any specific schema. In the terms of Jin *et al.* [9], the storage model, with respect to the stored conceptual models, should utilize an implicit external schema. That is, the storage model must have sufficient expressibility to represent the elements described by each schema without imposing constraints upon these elements. External storage schemas delegate tasks, such as checking schema conformance, to the tools that manipulate the database. This flexibility permits the storage layer to store multiple models, but prevents query tools from exploiting the schema's attributes. In a sense, this approach permits a conceptual level schema to be considered as a specific database view.

It is known that integrating schemas, and consequently models, is a difficulty task [14]. As Moise and Wong suggest, one approach is to develop a comprehensive global schema that encompasses all information to be represented. While this is a significant challenge at the conceptual level, it is much easier to accomplish at the storage level. Database storage models are designed to represent a wide range of information and thus offer much promise for combining schemas.

2.3 Conceptual Layer

The conceptual layer is where extracted information is expressed. Each stored model is an instance of a specific conceptual schema that is expressed using an appropriate meta-model. For example, a source-code model can be expressed using the SrcML schema, which itself is XML compliant. Since each model is mapped to a common storage layer, there is no limit to the number of models and schemas that can be used. Tools can access any information using the interface of the lower layer. The only requirement of a schema is that it is mappable to the storage layer.

It is also possible to consider the storage layer as providing an additional schema. For example, if there is information for which there is no need for a higher-level conceptual model, or for which a conceptual model does not exist, the information can be stored directly using the storage model. This practise is not desirable, as the storage model interface usually lacks a strong set of features for manipulating information. In response to this lack, Paul and Prakash [15] have developed an algebraic query language specifically designed for manipulating source-code.

3 Discussion

In recent research, relational database systems have been examined for the storage of XML compliant data [6,17]. In this approach, XML data is stored in tables and accessed using SQL. However, to abstract the storage model, tools use a query language such as XML-QL, which is then translated into the appropriate SQL query. We feel that mapping tools for each schema to the storage layer is an effective technique for their integration into a single environment.

While it is possible to store multiple XML schemas in the same database, there is no documented research in this area. Specifically, there needs to be further research on the implications of normalization to remove redundancy. As well, the use of a relational database to store source-code is known to be inefficient [12] when the code is decomposed into grammatic units. Queries must reconstruct code segments recursively using sub-queries to access each sub-element. In modern systems, this reconstruction can be performed efficiently, but it is not known whether this performance is maintained when multiple low-level conceptual models are integrated in a single database

Jarzabek [8] in his development of PQL (Program Query Language), uses Prolog predicates as a storage model, but states “PQL queries are written in terms conceptual program design models that are independent of the way the design models are actually stored.” His separation of the conceptual and

$^{<item>}<^{<tag>}\textbf{var-dec}^{</tag>}>^{</item>}$

  $^{<item>}<^{<tag>}\textbf{type}^{</tag>}$

    $^{<attr><name>}\textbf{type}^{</name>} = {}^{<val>}“^{<code>}\textbf{int}^{</code>}”^{</val></attr>}$

  $>^{</item>}$

  $^{<code>}\texttt{temp;}^{</code>}$

$^{<item>}<^{<tag>}\textbf{/var-dec}^{</tag>}>^{</item>}$

Fig. 2. Example of Multi-Layer Modelling Using 2-Level Markup

storage models is another example of the multi-layered approach to software modelling.

In Jupiter [5], we are exploring a multi-level markup approach. For conceptual modelling, traditional markup, as seen in JavaML, SrcML and GXL, is used. To store the conceptual models, a second, lower level of markup is used. The second level of markup is highly simplified and is specifically tailored for the MultiText structured text database system [3]. While similar to XML-style markup, the storage level markup exhibits several key differences in support of retrieval efficiency. First and foremost, MultiText markup is index-based to support algebraic query solution. Relationships are represented as pairs of indices and have no semantic constraints. As well, apart from relationships, markup items are atomic and attributeless tokens that have no imposed limitations on their locations. The similarity of the conceptual and storage models permits conceptual models to be easily mapped to the storage model.

Figure 2 provides an example of our two-level markup approach. In the figure, the C code “`int temp;`” is shown with SrcML markup in **bold** text and MultiText storage markup in $^{\textit{superscripted}}$ text. Storage markup is used to identify the elements of both the conceptual representation (SrcML) and of the source-code (C). Using the query language of the storage model, it is possible to retrieve SrcML marked-up code constructs, unmarked source-code or SrcML markup items.

It should be observed that figure 2 is an illustrative device only. It is expected that any tool interfacing to the database, via the query language, will suitably remove markup tags to avoid overwhelming users. When several models are integrated, only the markup for the feature of interest, in the model of interest, should be presented.

The storage layer must provide the flexibility needed to represent unrelated conceptual models. In Jupiter/MultiText, this flexibility is a product of the storage model's simplicity. The lack of any restrictions on the location and content of a storage markup item permits any conceptual model to be mapped to an arbitrary set of source-code regions. The major limitation of the approach is the requirement for conceptual models to be explicitly mapped into the source-code. However, this limitation enables multiple models to be integrated by focusing on the models derivation, although perhaps indirectly, from the source-code.

A 3-layered approach stacks conceptual models upon a storage model. Extension of this approach will likely entail the splitting of the conceptual layer into layers focused on a specific concept. For example, a 4-layered implementation could have a low level AST-based layer on top of the storage layer and below a higher level DMM-based layer.

4 Future Directions

Multi-layered approaches have yet to have a significant impact on the field of reverse engineering. Though they offer improved flexibility, they are more complex to implement and have not been well researched. It is not known how well a storage model, such as the relational model, will integrate multiple conceptual models. More research on the representation of the various conceptual models using storage specific models is needed. Similarly, it is known that relational query languages are less desirable for use in some maintenance tasks [10]. Hence, there is also a need for development of techniques to map abstract graphical languages to database retrieval languages such as SQL.

There is some discussion on the development of common conceptual models for information interchange [16] but this discussion has not been carried out with respect to software storage models. Existing repository systems utilize relational, object-oriented, text, logical (Prolog) and customized database systems. There is now a need to develop a consensus on the most appropriate storage models for representing conceptual models.

The use of a multi-layered approach unifies much of the work that has been occurring in the development of reverse engineering tools. Research on information extraction and representation is focusing on the schema layer while research on repository systems usually examines the storage layer. These topics should not be viewed as disjoint, but instead as two elements of a more complete layered implementation approach.

Multi-layered approaches improve upon single element models by reducing the demands placed upon each layer. Layered design permits lower layers to abstract storage and management details, lessening lower level pressures on conceptual models. As well, the use of multiple conceptual models, disperses pressure from above by permitting each model to focus on a specific demand. The key to the successful application of this approach is the use of a common storage model that integrates multiple higher-level models, where each may conform to a different schema. When one is not restricted to a single conceptual schema, reverse engineering is facilitated through the ability to use the most appropriate model for each extracted information entity.

Acknowledgements

We would like to thank Andreas Winter for his assistance and helpful comments. His insights have been highly beneficial and have greatly improved this article.

References

[1] Badros, G., *JavaML: A markup language for Java source code*, Computer Networks **33** (2000), pp. 159–177.

[2] Blaha, M., *Data store models are different than data interchange models*, Electronic Notes in Theoretical Computer Science (2004).

[3] Clarke, C., G. Cormack and F. Burkowski, *An algebra for structured text search and a framework for its implementation*, The Computer Journal **38** (1995), pp. 43–65.

[4] Cordy, J., K. Schneider, T. Dean and A. Malton, *HSML: Design directed source code hot spots*, in: *Ninth International Workshop on Program Comprehension*, IEEE, Toronto, Canada, 2001, pp. 145–154.

[5] Cox, A. and C. Clarke, *Representing and accessing extracted information*, in: *International Conference on Software Maintenance*, IEEE, Florence, Italy, 2001, pp. 12–21.

[6] Florescu, D. and D. Kossman, *Storing and querying XML data using an RDBMS*, IEEE Data Engineering Bulletin **22** (1999), pp. 27–34.

[7] Holt, R., A. Winter and A. Schürr, *GXL: Towards a standard exchange format*, in: *Seventh Working Conference on Reverse Engineering*, IEEE, Brisbane, Australia, 2000, pp. 162–171.

[8] Jarzabek, S., *Design of flexible static program analyzers with PQL*, IEEE Transactions on Software Engineering **24** (1998), pp. 197–215.

[9] Jin, D., J. Cordy and T. Dean, *Where's the schema? a taxonomy of patterns for software exchange*, in: *Tenth International Workshop on Program Comprehension*, IEEE, Paris, France, 2002, pp. 65–74.

[10] Lange, C., H. Sneed and A. Winter, *Comparing graph-based program comprehension tools to relational database-based tools*, in: *International Conference on Software Maintenance*, IEEE, Florence, Italy, 2001, pp. 209–218.

[11] Lethbridge, T., *The dagstuhl middle model: An overview*, Electronic Notes in Theoretical Computer Science (2004).

[12] Linton, M., *Implementing relational views of programs*, in: *Symposium on Practical Software Development Environments*, ACM SIGSOFT, Pittsburgh, Pennsylvania, 1984, pp. 132–140.

[13] Maletic, J., M. Collard and A. Marcus, *Source code files as structured documents*, in: *Tenth International Workshop on Program Comprehension*, IEEE, Paris, France, 2002, pp. 289–292.

[14] Moise, D. and K. Wong, *Issues in integrating schemas for reverse engineering*, Electronic Notes in Theoretical Computer Science (2004).

[15] Paul, S. and A. Prakash, *A query algebra for program databases*, IEEE Transactions on Software Engineering **22** (1996), pp. 202–217.

[16] Sim, S. and R. Koschke, *WoSEF: Workshop on standard exchange format*, Software Engineering Notes **26** (2001), pp. 44–49.

[17] Zhang, C., J. Naughton, D. DeWitt, Q. Luo and G. Lohman, *On supporting containment queries in relation database management systems*, in: *International Conference on Management of Data*, ACM-SIGMOD, Santa Barbara, California, 2001, pp. 425–436.