Scheme on the Web and in the Classroom

A Retrospective about the LAML Project

Kurt Nørmark

Department of Computer Science,
Aalborg University, Denmark
normark@cs.aau.dk

Abstract

LAML is a software system that brings XML languages into Scheme as a collection of Scheme functions. The XML languages are defined by XML document type definitions (DTDs). We review the development of LAML during more than a decade, and we collect the experiences from these efforts. The paper describes four substantial applications that have been developed on top of the LAML libraries.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Documentation, Languages, Experimentation.

Keywords Scheme, LAML, XML.

1. Introduction

This is a short paper about the use of LAML for creation of simple web pages, complex collections of web materials, and interactive web systems – many of which support my teaching in the areas of programming and programming languages. The paper summarizes the basic ideas behind LAML, and it gives an overview of the different kinds of applications that are supported by LAML. The paper systematically refers to all published papers about LAML, or LAML related works. The readers are recommended to consult these papers for detailed exposition of the LAML work, and for references to similar work.

The work on LAML was started in 1999 [2]. LAML stands for "Lisp Abstracted Markup Language". The initial idea was to use Lisp syntax instead of "HTML syntax" for the web source files. More specifically, we wanted to have a Lisp function for each "tag" in HTML, and more generally for each element in an XML language.

The initial LAML ideas could be supported by different Lisp languages. Emacs Lisp was considered as a candidate, mostly because I use Emacs for all my text editing and text management needs. Common Lisp was also a candidate, of course. But Scheme became the natural choice. Scheme was attractive because it was a small, powerful and elegant language supported by several different implementations. In comparison with Scheme, Emacs Lisp would not have been practical for our purposes, because it is difficult to run an Emacs Lisp program outside the scope of the interactive text editor. Common Lisp felt too complex for the task.

I decided not to bind LAML to a single Scheme system. LAML was intended to work on any R4RS/R5RS Scheme system – on both Linux, Windows, and Macs. Missing pieces, especially related to operating system interfaces, were organized in *compatibility library files*. Compatibility files exist for many combinations of Scheme implementations and operating systems.

The use of S-expressions for representation of HMTL/XML documents has been explored in several Lisp contexts, and in other functional languages as well. It is possible to go in at least two different directions:

- **The data direction.** An HTML/XML fragment is an S-expression, which can be processed in different ways by passing it to different functions.
- **The program direction.** An HTML/XML fragment corresponds to a Lisp expression where each XML tag/element is represented by a function. The Lisp expression is hereby processed in a fixed way, as defined by the HTML/XML functions that are involved.

I took the program direction in LAML. As a consequence, each element of the markup language gives rise to a Scheme function – a so-called *mirror function* (see Section 3). This is important, because these Scheme functions can be used as "building blocks" in a functional program. It is, for instance, possible to map the functions over a list of subdocuments, and more generally to pass the functions around as parameters in a Scheme program.

Clearly, the mirror functions must be generated automatically. In the first couple of years of the LAML development process I used considerable efforts to "get this right". I ended up with functions, with specialized parameter passing conventions, that generate an internal document form (a LAML AST). This is discussed in Section 3. The LAML AST can subsequently be processed in different ways, as described in Section 4.

2. Markup language versus Programming Language

In contexts where program fragments and markup fragments are used side by side – such as in server side programs – I realized how unattractive such combined documents may appear. It led to the identification of four different situations [8]:

- **Program hosting:** The outer structure is a program in which pieces of markup appear in the program details.

Scheme'12 September 9, 2012, Copenhagen, Denmark
Copyright © 2012 ACM [to be supplied]. . . $10.00

- **Markup hosting:** The outer structure is a web document (in HTML, or in an XML language) in which pieces of program appear.
- **Program subsumption:** The markup fragments are eliminated, and instead expressed in the programming language
- **Markup subsumption:** The program fragments are eliminated, and instead expressed in the markup language.

Program hosting can be achieved by use of formatted printing (`printf` in C, for instance). Markup hosting is common in frameworks such as ASP and JSP. I consider these solutions as very problematic, because two entirely different languages are combined "in strange ways". Neither of the two languages have been designed with any consciousness about the other. This is, at best, a temporary ad hoc solution, waiting for a better solution to appear.

LAML falls in the category of program subsumption. Markup subsumption (in XML languages) is rare in web server contexts, and probably bound to be problematic. The reason is that XML syntax and programming language syntax do not fit nicely together. In my opinion, XSL and XSLT are good examples of this observation.

Experience: A uniform syntax, which covers both a programming notation and XML, is desirable instead of mixing programming notation and XML notation.

3. LAML Basics

After a couple of initial "quick and dirty attempts" I decided to develop a more solid base level of LAML. As explained already, the idea is still that each element in an XML languages gives rise to a *mirror function* in Scheme.

The mirror functions are generated from XML document type definitions (DTDs). The first step parses the XML DTD. After that comes a synthesizing phase that generates a file with Scheme definitions (at the level of text). A part of a generated Scheme definition embeds a finite automata (compactly represented in the Scheme code) which validates the grammatical composition of the web document. The validation is done at document generation time – at Scheme run time. This is a natural choice in Scheme, because Scheme is dynamically typed (type error are found at run time). In statically typed functional programming languages the LAML approach is considered as a primitive solution. The reason is that in these languages, comprehensive error checking takes place before the program is executed.

XML Schema soon became an attractive alternative to XML DTDs. XML Schema is more complex (and more powerful that DTDs). LAML does not support XML Schema.

Mirror functions in Scheme have a number of properties:

- ***Rule 1.*** An attribute name is a symbol in Scheme, which must be followed by a string that plays the role as the attribute's value.
- ***Rule 2.*** Parameters which do not follow a symbol are content elements (strings, numbers, or instances of elements).
- ***Rule 3.*** All content elements are implicitly separated by white space.
- ***Rule 4.*** A boolean false value (which we conveniently bind to a variable named underscore _) suppresses white space at the location where the boolean value appears.
- ***Rule 5.*** Every place an attribute or a content element is accepted we also accept a list, the elements of which are processed recursively and spliced into the result.
- ***Rule 6.*** A Scheme procedure represents a *delayed content item*. The delayed content item is a Scheme procedure of two parameters: the root AST and the immediate parent AST. The procedure is called, in a post processing phase, when the document AST has been constructed. The resulting content items and/or attributes are spliced into the document, as a replacement of the procedure.

The five initial rules are taken from the JFP paper about LAML [11].

Experience: The recursive list flattening (the splicing of lists into their contextual lists), as carried out by all mirror functions, is extremely convenient.

Without automatic splicing a certain amount of explicit flattening, or list appending, must be done. In approaches that go in the data direction (instead of the program direction) where no automatic list flattening is arranged for, this gives rise to use of quasiquotation and unquote-splicing.

4. LAML Processing and Transformations

When a LAML expression is evaluated, an abstract syntax tree (AST) is returned. The AST is the internal representation of the document.

The AST can be linearized, and written to a file, or rendered in to a client browser. In other situations (typically if the AST does not belong to HTML or SVG) we want to transform the AST to a form that can be rendered. In LAML, a so-called *action procedure* may grab the AST and initiate the transformation process. Action procedures typically exist for top-level forms only.

When a LAML document is processed, some contextual information is passed to the LAML processor:

- The name of the source file, where the LAML expression(s) reside.
- The absolute directory path of the source file.
- A number of additional and optional LAML program parameters.

The LAML program parameters can be used to control the course of the action procedures, and hereby facilitate more than one possible kind or LAML document processing.

The transformation of LAML ASTs from one XML language to another is facilitated by a small set of Scheme functions, such as

- `(find-asts ast element-name [ast-transformer-fn])`
- `(transform-ast transf-spec-list source-list-ast)`

The function `find-asts` searches for all sub-ASTs of `ast` named `element-name`, and it applies an optional AST transformation function on these before they are returned. A similar function uses a predicate for selection, instead of comparing element names. The function `transform-ast` applies a transformation from the transformation specification on each element in the source AST list. A transformation specification is a pair of a predicate and an AST-transformer. The list of transformation specifications are tried out sequentially until one of the predicates succeeds.

Experience: For all the AST transformation needs we have encountered, a small repertoire of simple AST transformation functions have been adequate.

With the power of Scheme, and with the ease of defining simple recursive transformations functions in Scheme, there seems to be no good reason to introduce a transformation framework from the outside (such as XSLT).

5. LAML Applications

In this section we will give an overview of the most substantial applications of LAML, and we will highlight our experiences with these applications.

5.1 The LENO Lecture Notes

One of the first LAML applications was targeted at production of lecture notes represented as interlinked HTML pages [3]. The system should replace my use of Powerpoint^{TM} for authoring of teaching materials (annotated slides). I found it attractive to represent slides as ordinary HTML pages, which could be smoothly integrated with other resources on the web.

In addition to hosting the slides as HTML pages on the web, I wanted a solution to a particular problem that is encountered if programs are presented on slides: Two copies of the program exist – one in the slides and one in a source file; Eventually, you change one of them without changing the other. Changing a program in a slide typically leads to errors in the program, because the slide version of the program is never checked by a compiler.

In the LENO system, a source program is included in the teaching material by transclusion. In practical terms, the program is inserted when the HTML pages are generated (upon LAML processing of the LENO document). In LENO, the insertion of the source program also involves (1) selection of a part the program and (2) superimposition of colors on those program details which are important in the context of the surrounding slide.

Experience: The support of program source file inclusion in lecture notes, without creating a copy of the program, is one of the major assets of the LENO system.

The first version of the LENO system was developed as a function library on top of some early HTML mirror functions. Later on, I reengineered the system based on a LENO XML DTD, and generation of Scheme mirror functions for that language. In order to preserve backward compatibility (because a body of existing lecture notes) I decided to transform a document in the new LENO language to the old internal format.

Experience: In the long run it became too complicated to maintain both the new LENO front end language and the old kernel of the system. In some situations, it is better to skip the first version of a system entirely, instead of building a bridge from new to old software.

The LENO system supports four different views on lecture notes. The *slide view* is intended for presentation in an auditorium. The *annotated slide view* adds comments to the individual elements on a slide, in a two column setup. The *aggregated view* shows all slides on a single, long HTML page (supporting a format with "margin notes"). The final, and most ambitious view, is envisioned as a classical *text book* based on the slides. The main idea is to create a text book source document on top of the lecture note (slide) source document, called the secondary and primary source respectively [13]. The secondary source refers to slide elements in the primary source – and includes them by transclusion.

Experience: Despite several countermeasures, it is problematic to keep both the text book representation and the lecture note (slide) representations consistent with each other.

LENO has been used to write teaching materials for Java, Scheme, C#, C, and C++.[1] Since 1999, hundreds of students at Aalborg University have been exposed to teaching materials in LENO.

5.2 LAML SchemeDoc

The most fundamental parts of LAML are made up of libraries with Scheme definitions. Some libraries are coded by hand – in the conventional way. Others are derived from XML document type definitions. In both cases, it is crucial for effective long term use that the interfaces of the libraries are well-documented.

Soon, therefore, it became necessary to extract API information from the LAML libraries. The extraction is based on some simple conventions of specially marked documentation comments in Scheme, similar to conventions used for many other programming languages. A Scheme function, `extract-documentation--from-scheme-file`, is able to extract API documentation from a Scheme library file.

LAML SchemeDoc [10] relies on an XML documentation languages, which is brought into Scheme via XML mirror functions. The XML language allows manual authoring of documentation.[2] It also handles extracted API documentation, extracted XML DTD documentation, and hybrids of extracted and manually authored documentation. In the typical case, it only requires a tiny XML-in-LAML document to setup a few parameters (such as author information, CSS presentation style, and some interlinking details) for control of the extracted documentation.

Experience: LAML SchemeDoc has been an essential tool for the use and organization of LAML software.

On top of LAML Schemedoc it is possible to create an index – a documentation browser – of a number of libraries. This includes documentation of the functions and syntactical forms of the Scheme language as such, via links into the R5RS Scheme report.

SchemeDoc makes API documentation available to other tools as well, through list structures in so-called *manual Lisp files*. The Emacs editor can read these file and utilize the documentation for name completion, and for bringing up tooltip documentation boxes. The major mode of an Emacs buffer is mapped to a list of desirable manual Lisp files, from which documentation is brought into the editor.

As the last element of SchemeDoc, it is possible to capture examples of function calls via *interactive unit testing* [14]. In a Scheme REPL, it is possible to collect test cases very easily by stating that the entered expression (a function call of *f*) and the calculated value are regarded as being a correct example of the use of *f*. The collected test cases can be fused into an external Scheme Unit testing tool, hereby enabling regression testing. The test cases also serve as examples, which can be presented in SchemeDoc manual pages. This is very attractive, because good examples are often easier to grasp than textual explanations in English. Finally, the examples can also be shown in Emacs, in the same style as the more conventional documentation.

[1] Links the these teaching materials appear on my homepage `http://people.cs.aau.dk/∼normark/`

[2] The manual authoring facility is only used in special situations (such as for documentation of some aspects of SchemeDoc itself).

Experience: Concrete examples of function applications, and their results, are effective parts of API documentation. It is attractive to gather the examples from test cases.

5.3 The Scheme Elucidator

Elucidative Programming (EP) [4] grew out of a fascination of Literate Programming (LP), but it was also based a number of concerns [5].[3] LP and EP are both targeted at internal program documentation, in contrast to interface documentation (as supported by SchemeDoc). EP relies on relations between entities in the documentation, and entities in the source programs. In contrast to LP, the documentation and the program source files are separate in EP, and the source files are not affected by the documentation efforts. An elucidative program is presented in two frames of an internet browser, with mutual navigation in between the documentation frame and the program frame.

With the development of large Scheme programs, as parts of LAML, it was attractive to implement a Scheme Elucidator. The Scheme Elucidator processes a documentation file together with a number of Scheme source files (together called a *documentation bundle*), and it generates the necessary HTML pages. As a characteristic of an elucidative program, the documentation and the presentations of the source files are heavily interlinked.

The first version of the Scheme Elucidator was based on documentation files that used simple and special-purpose markup. In the next (and current) version, the documentation file was based on XML-in-LAML markup, just like all the other LAML tools. This turned out to be a better solution, because the availability and power of the Scheme programming language makes a noticeable difference for authoring of complex documentation.

The processing of (Scheme) source files in a documentation bundle was the major technical challenge during the implementation of the Scheme Elucidator. The Scheme source files are decorated with a lot of links. Some links go to relevant places in the documentation (where a definition is discussed). Other links are cross references to other places in the program, links into SchemeDoc API documentation, or links to the R5RS Scheme Report.

Internally, a Scheme source file is first pre-processed, the same way as done by LAML SchemeDoc. This turns comments into syntactical constituents. Next, the source file is traversed simultaneously at the lexical level and at the syntactical level (where also the comments are syntactical entities). This procedure is crucial for creation of all the links, and for preservation of the original program layout.

Processing of program source files, in preparation for comprehensive linking of source file names to surrounding entries, is useful and valuable in other contexts than the Elucidator. LAML Schemedoc can be set up to activate such a processing of the Scheme source programs. Hereby the interface documentation links to the underlying Scheme source file, which is cross-linked as much as possible, and which in turn links back to SchemeDoc pages.

Experience: The creation of *richly linked* program source files is useful for a broad spectrum of program documentation needs.

In retrospect, the Scheme Elucidator has been much less important than SchemeDoc in the development process of LAML. The by-product of the Elucidator – "the Scheme source program docorator" – has turned out to be useful in its own right.

[3] `http://people.cs.aau.dk/ normark/litpro/issues-and-problems`.

Experience: Independently of tool support it remains hard to convince programmers (and yourself) to invest time and efforts in writing internal documentation.

5.4 MIDI LAML

The last LAML application is completely different from the three other applications discussed in this paper. This application is rooted in a hobby activity. A few years ago, I decided to understand the MIDI format for electronic music instruments – all the way to the bottom. Recalling the quote of Kristen Nygaard – "to program is to understand" – I decided to write a MIDI parser and an unparser. Given my background in LAML and XML languages, I decided to parse MIDI files to an XML format. By incidence, the MIDI Manufacturers Association already had developed a MIDI XML DTD, which I elaborated and extended for my purposes.

Based on the outcome of the MIDI parser it is possible do carry out many systematic music transformations task in Scheme. I have developed a rather comprehensive library of Scheme functions for MIDI music transformation. On a regular basis, I use this library for "real life" music transformation purposes [15].

As a functional programmer, it is attractive to work with lists of MIDI events via the classical higher-order functions `map`, `filter`, and `reduce`. I observed, however, that it was often more useful to apply these functions on sublists of MIDI events than on individual MIDI events. This prompted the work on mapping and filtering of *bites of lists* [16]. A bite is a non-empty prefix of a list. Successive "biting" results in a disjoint partitioning of a list, which turned out to be useful as the starting point for many music related MIDI transformation tasks. I have developed a collection of bite-mapping and bite-filtering functions, together with a number of higher-order functions that generates "biting functions".

Experience: MIDI music transformation revealed an application area where mapping and filtering is more useful on disjoint sublists than on individual list elements.

Half of the work on the MIDI LAML system has been directed towards the programming of an operational MIDI LAML environment – a so-called MIDI sequencer – in the Emacs text editor. Although the result is primitive compared to contemporary music sequencers and "DAWs", the Emacs-based tool can be used for real music-related problem solving. Most important, the MIDI LAML system supports a music production process where Scheme programming is used for non-trivial editing of your music.

6. Status

As of 2012 I still use LAML for all my web work (simple as well as more complex web pages). The LENO lecture note system is probably the part of LAML which I use most frequently. I also regularly program CGI applications in LAML that support teaching activities, along the lines of the system for submission of programming assignments [17] and the one that supports peer assessment of coursework [1]. My own use of LAML is based on a legacy version of PLT MzScheme.

A lot of text has been authored in languages derived from XML DTDs, in Scheme syntax. In other words, the Scheme programming language has been used as a textual markup language in a number of different application domains. Good editor support is crucial for this to be successful. This includes support of a few generic editing commands (embedding, nesting, string splitting, and their inverse commands), as well as template support for the various applications areas.

Experience: Scheme syntax can be used for textual markup purposes, if supported by a few appropriate editing commands. Authoring text in Scheme gives uniform access to programming power, at any time, and at any place in a document.

The core LAML system consists of 75,000 lines of Scheme program. The LAML environment support in the Emacs editor includes 25,000 lines of Emacs Lisp program. 20 papers have been published about LAML [1–17, 19–21].

Until late 2011, downloadable LAML distributions have been derived as a subset of the development version. The latest is LAML version 38. The distribution comes in three variants: The full version with comprehensive documentation, a slim version where most of the documentation is stripped, and a version with only LAML SchemeDoc. In the early years I kept track of who downloaded the system, by asking for registration before the actual download. The number of downloads was quite low, typically around 20 pr. month. Later, I gave up on the registration, and I do not longer keep any record or statistics about the download frequency.

7. Conclusions

At the personal level, and in relation to the teaching activities that I take care of, LAML has been a success.

In the beginning, I envisioned LAML to be a side activity, mostly for personal enjoyment. Later on it became clear that an array of different activities could take place on top of the basis LAML libraries and tools. Seen in the light of the early expectations, it is satisfactory that 20 scientific papers have been published in the slipstream of this work. The external use of LAML, as of 2012, is very low (if existing at all).

After software, like LAML, has been developed, a transition is taken in the direction of maintenance. It is difficult to justify time for software maintenance in a university job. You can spend time on software development as long as it supports your research activities (as long as you can write papers based on the developed software). But you cannot afford to use lots of hours, just for keeping the software alive.

For several reasons it is a burden to maintain 75,000 lines of Scheme code. First, and most important, the R5RS Scheme system on which I primarily depend (PLT/Racket) has changed a lot the last few years.[4] I have not been tempted to adapt LAML to R6RS Scheme. Operating systems and Emacs also come in new versions every now and then. The Department Computer Science at Aalborg University shifted from Linux to Windows a couple of years ago (forcing users from a Linux file server to a Windows file server). On a broad scale (not least in relation to LAML) it caused a lot of extra work. In the slipstream of this, our Apache Web server was separated from our file server (due to legal issues).

Experience: Almost every contextual software element around LAML has changed over a decade.

Eventually I expect to lose the battle against the changes of the contextual software of LAML. Instead of continued patching of old software, I anticipate the need for "a new beginning". When that happens, I hope that I can base the work on a modern and high quality Scheme system which supports the old virtues of being *small, powerful and elegant*.

[4] At some point in time (in the transition to PLT 400) it was too time demanding to update LAML relative to the changes in PLT Scheme. As part of this, it would also be very difficult to keep LAML running in other Scheme systems as well (based on a simple "compatibility library").

Acknowledgments

I want to thank Olivier Danvy for encouraging me to write this paper.

References

[1] H. Hüttel and K. Nørmark. Experiences with web-based peer assessment of coursework. In *Proceedings of the 4th International Conference on Computer Supported Education - CSEDU*, April 2012.

[2] K. Nørmark. Using Lisp as a markup language—the LAML approach. In *European Lisp User Group Meeting*. Franz Inc., 1999. URL `http://people.cs.aau.dk/~normark/laml/papers/-lugm-laml.pdf`. Available via [18].

[3] K. Nørmark. A suite of WWW-based tools for advanced course management. In *Proceedings of the 5ht annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*. ACM Press, July 2000. Available via [18].

[4] K. Nørmark. Elucidative Programming. *Nordic Journal of Computing*, 7(2):87–105, 2000. URL `http://www.cs.helsinki.fi/njc/-References/normark:87.html`.

[5] K. Nørmark. Requirements for an elucidative programming environment. In *Eight International Workshop on Program Comprehension*. IEEE, June 2000. Also available from `http://people.cs.aau.-dk/~normark/elucidative-programming/`.

[6] K. Nørmark. An elucidative programming environment for Scheme. In *Proceedings of NWPER'2000 - Nordic Workshop on Programming Environment Research*, May 2000. Also available from `http://people.cs.aau.dk/~normark/-elucidative-programming/`.

[7] K. Nørmark. Programmatic WWW authoring using Scheme and LAML. In *The proceedings of the Eleventh International World Wide Web Conference - The web engineering track*, May 2002. URL `http://www2002.org/CDROM/alternate/296/`.

[8] K. Nørmark. The duality of XML markup and programming notation. In *The proceedings of the IADIS International Conference on WWW/Internet*. IADIS, November 2003. Available via [18].

[9] K. Nørmark. XML transformations in Scheme with LAML - a minimalistic approach. In *The proceedings of the International Lisp Conference, ILC 2003*. Association of Lisp Users, October 2003. Available via [18].

[10] K. Nørmark. Scheme program documentation tools. In O. Shivers and O. Waddell, editors, *Proceedings of the Fifth Workshop on Scheme and Functional Programming*. Department of Computer Science, Indiana University, September 2004. Technical Report 600.

[11] K. Nørmark. Web programming in Scheme with LAML. *Journal of Functional Programming*, 15(1), January 2005. URL `http://people.cs.aau.dk/~normark/laml/papers/-web-programming-laml.pdf`.

[12] K. Nørmark. A graph library extension of SVG. In *Proceedings of SVG Open 2007, Tokyo, Japan*, September 2007. URL `http://people.cs.aau.dk/~normark/laml/papers/-svg-open-2007/paper.html`.

[13] K. Nørmark. Deriving a comprehensive document from a concise document - document engineering in Scheme. In D. Dubé, editor, *The 8th Workshop on Scheme and Function Programming*. Départment d'informatique et de Génie Logiciel, Université Laval, Canada. Technical Report DIUL-RT-0701, September 2007. URL `http://-sfp2007.ift.ulaval.ca/procPaper11.pdf`.

[14] K. Nørmark. Systematic unit testing in an read-eval-print loop. *Journal of Universal Computer Science*, 16(2), 2010.

[15] K. Nørmark. MIDI programming in Scheme - supported by an Emacs environment. Proceedings of the European Lisp Workshop 2010 (ELW 2010), June 2010. URL `http://www.cs.aau.dk/~normark/-laml/papers/midi-laml-paper.pdf`.

[16] K. Nørmark. Bites of lists - mapping and filtering sublists. In *The proceedings of the 4th European Lisp symposium*,

March 2011. URL `http://people.cs.aau.dk/ normark/-laml/papers/bites-paper.pdf`.

[17] K. Nørmark. A web support system for submission and handling of programming assignments. In *The proceedings of E-Learning'11 - E-Learning and the Knowledge Society*, August 2011. URL `http://people.cs.aau.dk/ normark/-programming-assignments-paper.pdf`.

[18] K. Nørmark. The LAML home page, 2012. `http://people.cs.-aau.dk/∼normark/laml/`.

[19] T. Vestdam and K. Nørmark. Aspects of internal program documentation - an elucidative perspective. In *10th International Workshop on Program Comprehension*. IEEE, June 2002.

[20] T. Vestdam and K. Nørmark. Maintaining program understanding - issues, tools, and future directions. Presented at 11th Nordic Workshop on Programming and Software Development Tools and Techniques - NWPER'2004, May 2004.

[21] T. Vestdam and K. K. Nørmark. Towards documentation of program evolution. In *Proceedings of the 21st International Conference on Software Maintenance*, September 2005.