Toward Documentation of Program Evolution

Thomas Vestdam and Kurt Nørmark
Department of Computer Science
Aalborg University
Denmark
{odin,normark}@cs.aau.dk

Abstract

The documentation of a program often falls behind the evolution of the program source files. When this happens it may be attractive to shift the documentation mode from updating the documentation to documenting the evolution of the program. This paper describes tools that support the documentation of program evolution. The tools are refinements of the Elucidative Programming tools, which in turn are inspired from Literate Programming tools. The version-aware Elucidative Programming tools are able to process a set of program source files in different versions together with unversioned documentation files. The paper introduces a set of fine grained program evolution steps, which are supported directly by the documentation tools. The automatic discovery of the fine grained program evolution steps makes up a platform for documenting coarse grained and more high-level program evolution steps. It is concluded that our approach can help revitalize older documentation, and that discovery of the fine grained program evolution steps greatly help the programmer in documenting the evolution of the program.

1 Introduction

The starting point of this paper is a situation where a program is documented by some written explanations. The documentation of concern in our work is targeted at both the overall and detailed design, as well as program details. Along another dimension, we focus on *internal documentation* [**?**] as a contrast to interface documentation and end-user documentation. At the outset [**?**] we were inspired by Literate Programming [**?**] in which the program is contained within a program essay. Since 1999 we have developed another, but related, documentation style called Elucidative Programming [**?**, **?**, **?**], which is based on a relation between the documentation and the source program files. Elucidative Programming is briefly explained in Section 2.

Over some time a program is likely to undergo an evolution through a number of versions. In the ideal case the documentation should be evolved in parallel with the program. Often, however, the evolution of the documentation ceases at some point in time. As a typical scenario, the initial version of the program is documented, but subsequent program versions are not. We will refer to this scenario as *initial documentation only*.

Figure 1(a) illustrates the ideal case, where the program and the documentation evolves in parallel. D_i documents P_i for i running from 0 to n. Figure 1(b) illustrates the situation where the initial documentation D_0 falls behind the evolution of the program. Thus, D_0 documents P_0 (let us here presume in a perfect match). To some degree, D_0 can also be seen as documentation of P_i ($i > 0$), but the precision and the completeness of D_0 decreases when i grows. It may even be expected that some aspects in D_0 are erroneous relative to P_i.

There may, of course, be several reasons behind the *initial documentation only* scenario. For example, the evolution of the documentation may *deliberately be cut off*, for instance due to economical concerns. Another example is when the program develops through a *large number of very small steps*. The evolution of P_i to P_{i+1} may be so small that the developers do not feel obliged to update D_i. However, the net evolution of the program from version i to $i + d$, for some relative large value of d, may be significant. In this case, D_i will not accurately document P_{i+d}. Finally the awareness of the initial documentation may simply *fade away*. This is most likely to occur if the documentation and the programs are not in close proximity [**?**, **?**].

At some point in time it may be decided to revitalize the documentation efforts relative to the programming efforts. If the latest versions of the program and the documentation are P_n and D_m respectively ($n > m$), it will be a natural to update D_m. This corresponds to an updating of the documentation to the latest version of the program. As another option, it may be decided to reorient the documentation toward the evolution of the program P. With this approach, D_n

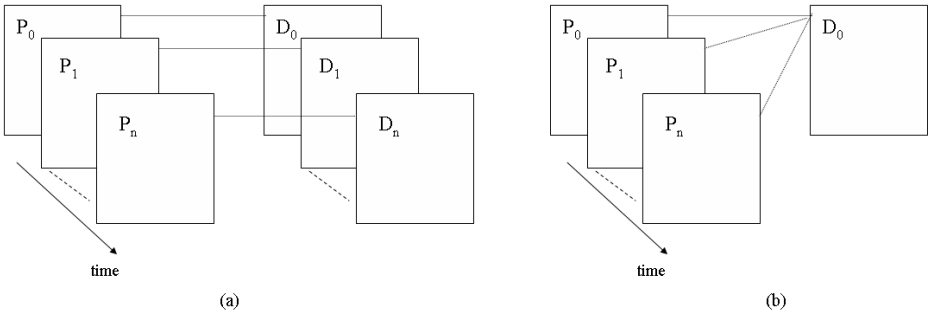

Figure 1. The ideal case (a) and initial documentation only (b).

explains the evolution of P from version 1 (or m) to n.

Studies indicate that it may not necessarily be an acute problem if the documentation is not up-to-date with the latest program version. This is especially the case if the documentation contains high-level abstractions that remain valid [**?**, **?**]. However, knowledge about a program will be lost over time. This makes it more and more difficult to evolve the program, and it will most likely result in *code decay* [**?**]. Hence, knowledge about the program—the program understanding—should ideally be maintained in the documentation of the program [**?**, **?**]. This suggests that documentation tools that support programmers in communicating knowledge about a program or its evolution can provide valuable assistance during all phases of software development.

In this paper we will discuss documentation tools which are aware of multiple program versions. References from the documentation to program entities automatically point at the most recent version of the program entity. In addition, source program abstractions are automatically linked to preceding and succeeding versions, and new or updated abstractions are automatically identified and marked. Renamed abstractions, and abstractions moved from one source file to another are also dealt with. The tools that we discuss in this paper are extensions of our Elucidative Programming tools [**?**, **?**, **?**, **?**]. We have developed two versions of the tools: One for Scheme [**?**] and another for Java [**?**].

The contributions of this paper are especially oriented toward support of the *initial documentation only* scenario. We hypothesize that the tools improve the immediate value of the initial documentation, relative to a sequence of program versions developed at a later point in time. In addition, the tools provide a good starting point for developers who decide to revitalize some older documentation, either with the purpose of a documentation update, or with the purpose of documenting the evolution of the program.

In the remaining parts of the paper, Section 2 describes the main ideas of Elucidative Programming. Following that, in Section 3, the new version-oriented elucidative programming tools are described. In Section 3.1 and 3.2 we distinguish between fine grained and coarse grained program evolution, and we discuss the support of the tools that can be expected in these two areas. The description in Section 3.1 and 3.2 is generic in the sense that it covers the essentials in both the Java and the Scheme tools. In Section 4 we describe related work, and finally in Section 5 we conclude on our finding so far, and we outline future work.

2 Elucidative Programming Background

Literate Programming [**?**] is based on the idea of representing fragments of the source program within an essay, which serves as program documentation. The conventional source program can be extracted from the essay by a tool called *tangle*. In addition, the essay can be pretty printed together with the embedded program fragments by a tool called *weave*.

Elucidative Programming emerged because Literate Programming was found to be unfit for modern software development [**?**]. The main problem of Literate Programming, as envisioned by Knuth, is the need to reorganize the program sources into fragment, which are hosted inside the documentation. Elucidative Programming can be seen as a practical variant of Literate Programming, building on both inspiration and experience gained from Literate Programming. Details on the background of Elucidative Programming can be found in [**?**, **?**, **?**, **?**, **?**, **?**]. The core facility of an Elucidative Program makes it possible to address program abstractions, as located in arbitrary organizations or the source programs, from phrases within the documentation. In other words, an Elucidative Program enables *writing about the program* without affecting the program.

The essential ideas and concepts of Elucidative Programming can be defined in an Elucidative Programming model. In this model, documentation and program are defined in separate entities called *documentation nodes* and *program nodes*. A documentation node represents a coherent doc-

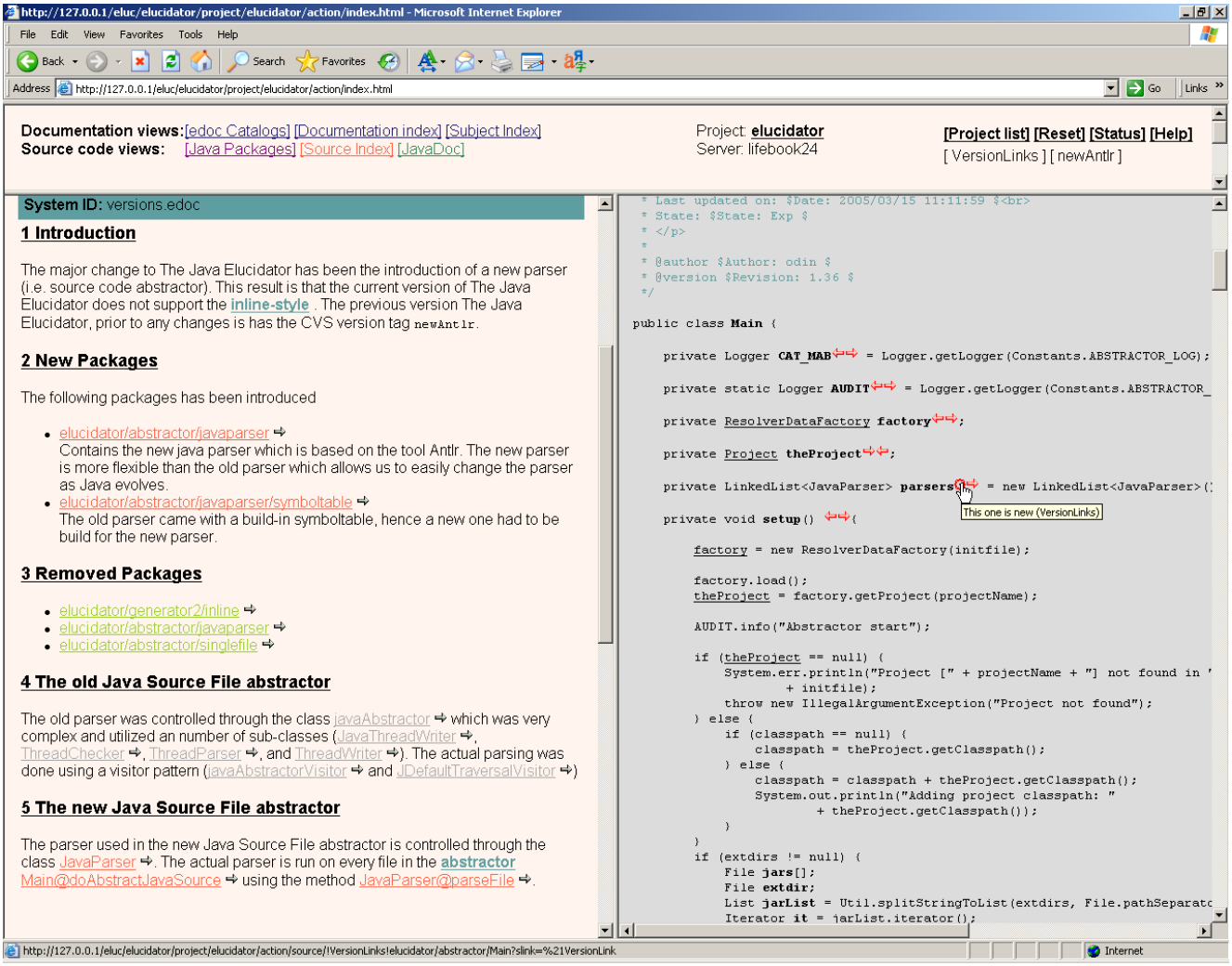

Figure 2. An example of the presentation of an elucidative program in a web-browser.

umentation fragment of varying granularity. A program node represents a conventional program source file. The set of documentation nodes and program nodes is called a *documentation bundle*, or an *elucidative program*. An elucidative program is tied together by a `doc-prog` relation, which connect documentation nodes and named entities (abstractions) in the program nodes. When used in *lenient addressing mode*, the `doc-prog` relation does not specify the exact location (source file or class, for instance) of named abstractions. Normally, however, the `doc-prog` relation is specified using an *exact addressing mode*. In case the `doc-prog` relation is too coarse grained, a notion of *source markers* can be used to address specific locations or regions within the program nodes.

The `doc-doc` relation supports cross references among documentation nodes. At the program side, the `prog-prog` relation represents the connection between applied and defined names, within a single program node or in-between program nodes. The program source files can be organized in the way demanded by either the programming language, or by the particular design behind the program. The documentation nodes can be hierarchically organized in *catalogues*.

An elucidative program is intended to be presented in a two-framed browser. This setup is illustrated in Figure 2, where a documentation node is presented in the left frame, and a program node is presented in the right frame. The `doc-prog` relation is represented as bidirectional hyperlinks, anchored within the documentation text and within the program source text respectively. When a hyperlink is activated in the documentation frame, the program fragment in question is scrolled into view in the program frame. When activating a hyperlink in the program frame, the user is presented with a list of places in the documentation,

where the program fragment is explained. Activation of one of these will scroll the documentation into view in the documentation frame. In addition to the documentation and program frames, various navigational features are available in the top frame (e.g. an index of documentation files, an index of source code files, and an identifier index).

Both the Scheme Elucidator tool [**?**, **?**, **?**] and Java elucidator tool [**?**] are supported by Emacs [**?**, **?**] at the editor-level. The main purpose of the editor support is to assist the programmer when creating contributions to the `doc-prog` relation. In order to do this, the Elucidator must *abstract* both source code and documentation, in order to gather information about program abstractions such as functions, methods, and classes. Hence, an Elucidator is a language-aware tool.

As the result of an abstraction the Scheme Elucidator produces the two-framed browser as a collection of static HTML pages. The Java Elucidator stores information about abstractions in a database, and the information in the database is used by a web-server to present the elucidative program on a more dynamic basis. The web-server behind the Java Elucidator is implemented as Java Servlets [**?**] running on an Apache Tomcat server [**?**]. The database is also used by Emacs to provide various lookup features when creating links.

The Java Elucidator also supports an *in-line style* [**?**] which allows embedding of specific parts of source code in the presentation of the documentation. Previous experiments also include embedding both the editor-level and the presentation-level of the Java Elucidator in the development environment TogetherJ [**?**, **?**].

3 Version-oriented Elucidative Programming tools

As a fundamental premise, the documentation tools described in this section are based on *unversioned documentation* and *versioned source programs*. Thus, the tools do not support more than one version of the documentation as such. The main rationale behind unversioned documentation is that we, as an end goal, aim at documentation which documents the evolution of the source program (from version 1 to the latest version). There should, quite naturally, only be one version of this account. As a secondary rationale, we are also interested in facilitating augmentation of up-to-date documentation with historical sections that address the affairs in earlier versions of the program.

Versioning an elucidative program will introduce all the structure, cognition and interface related problems known from versioning hypertext documents. Although promising and pragmatic tools exists (see for example [**?**, **?**]) we do not find a need for complicating matters further by introducing full-fledged versioning of entire elucidative programs.

The Elucidative Programming tools are aware of the specific versions of the source files that together form an aggregated source program. Hence, an elucidative program (see Section 2) actually consists of a set of documentation nodes and a number of *sets* of program nodes. Each set of program nodes contains all the source program files that constitute specific versions of the program—a *version-set*. The documentation tools enumerate the version-sets according to their respective ages going from the oldest version (the original) to the newest version (the current version). Thus, the version-sets of a source program will be enumerated from 1 (the original) to n (the current version). Source program files that exist in different versions are labeled according to the enumeration of the version-sets.

The Elucidative Programming tools are now able to provide bi-directional navigation between abstractions in adjacent version-sets. In Section 3.1 we describe the heuristics the tools use in order to detect which kind of fine grained evolution abstractions in a given version-set has undergone since the previous version-set. At the presentation-level, each abstraction *a* in the program is labeled according to the evolution it has undergone since the previous version-set, and if *a* can be found in the previous version-set navigation hereto is provided. The labels signal whether *a* has been *introduced*, *modified*, *deleted*, *renamed* or *moved* since the previous version-set. If *a* has not changed in any way, *a* is labeled as "unchanged" and navigation to *a* in the previous version-sets is also provided. In order to provide bi-directional navigation, a label is also placed at the destination (i.e., at *a* in the previous version-set) and navigation is provided to *a* in following version-set. These labels signal whether *a* is modified, renamed, moved, or unchanged in the following version-set.

Implicitly, links from documentation to program abstractions will per default always address the newest version of a given program abstraction. In case an addressed abstraction *a* is not part of the newest version of the source program (say, version n) the tools will, in succession, look for *a* in version $n-1$, $n-2$, ..., 1. Thus, program references automatically *falls through* the versions of the source programs. At the presentation-level, links that fall through to older versions are dimmed, by showing them on a grey background. This enables the documentation reader to immediately identify links that address older versions of a program. Phrases that contain several links to older versions can quickly, by use of these means, be identified as explanations that potentially are outdated or irrelevant in relation to the current version of the program.

At the discretion of the documentation writer, it is possible to address program abstractions in a specific version of a source file. To this end, the documentation writer makes use of a *version locked reference* to *a*. Use of version locked references makes it possible to discuss *historical aspects* of

the program, and to keep this discussion intact relative to future developments of the program.

At the source program side there are *backlinks* to the documentation from the documented program abstraction *a*. In case *a* exists in several versions, of which the documentation refers to the most recent one, all versions of *a* backlinks to the same location in the documentation. Thus, the particular location in the documentation serves as generic documentation of a number of different versions of *a*. This is convenient if the documentation of *a* happen to be accurate across the versions. If, however, *a* is changed during the program development process, the documentation of *a* may need to be updated. Quite naturally, version locked references are only backlinked from the program version in question.

At the source program side, different versions are extensively crosslinked. At top level, version n of a source program is linked to version $n-1$ and $n+1$, if these program versions exist. Individual abstractions in the source file are crosslinked similarly. As a consequence, it is easy to navigate from the documentation to the newest version of an abstraction *a*, and from this starting point back to older versions of *a*.

3.1 Fine grained program evolution.

Let us now consider a source program P which is developed through version 1 to n. Let us furthermore assume that version 1 of P holds a number of abstractions, say a_1, a_2, ... a_k. We will now identify different types of *fine grained program evolutions* and explain to what extent our documentation tools detect the changes induced by such evolution steps. In this context, a fine grained program evolution is a modification that only affects a single abstraction in the source program. Focusing on a single abstraction, which we here call *a*, the following kinds of fine grained programs evolutions may appear in the transition from version m to $m+1$.

- **Introduction**. *a* is introduced in version $m+1$. Thus *a* does not exist (in the given source file) in version m.
- **Modification**. *a* exists in both version m and $m+1$ (and in the same source file), but *a* in version $m+1$ has been modified (edited) somehow relative to version m.
- **Deletion**. *a* exists in version m, but not in version $m+1$.
- **Renaming**. *a* is renamed in version $m+1$. Thus, *a* from version m exists in version $m+1$, but under another name, say *b*. Renaming occurs within a given source file.
- **Moving**. *a* in version m also exists in version $m+1$, but it is moved to another source file. As a contrast to renaming, moving involves two different source files.
- **Internal reorganization**. *a* is moved from one location to another in the same source file. Thus, both version m and $m+1$ hold *a*.

In case of **introduction**, the documentation tools will identify the new abstraction in version $m+1$, and it will mark it as "New". In this way, the reader of the source program can easily spot new elements in the source program.

Similarly, in case of **modification**, the modified abstraction is marked as "Updated". An abstraction is said to be modified if the body of *a* in version m is different from the body of *a* in version $m+1$. The updated abstraction is linked to its older version, and vice versa. Thus, this provides bi-directional navigation between different versions of abstractions.

In case of **deletion**, the abstraction *a* in version m is labeled with a special *dead end icon* which signals that the following version of the source file does not contain *a*. Ideally, an abstraction marked with this icon has been removed entirely from the source program, not just renamed or moved.

The documentation tools are also able identify **renaming**, and to mark the new version as "Renamed". The identification of renaming is based on the heuristics that a renamed abstraction preserves its body and parameter list, but not its name. In languages with overloaded abstractions, such as Java, the signature (including the parameter list) is allowed to be changed as part of a renaming.

In case we use a lenient addressing mode (see Section 2) **moving** of *a* will not invalidate the documentation. Thus, the fact that *a* has been moved from one source file to another is discovered by the search mechanism of the Elucidator tool. The documentation will always, per default, address *a* in one of the most recent source program versions. If we use a specific addressing mode (see again Section 2) the moving of *a* to another source file will affect the documentation, because in this case the address of *a* in the documentation is strongly bound to the location of *a*. The documentation tools are able to track and label moved abstractions based on the heuristics that a moved abstraction preserves its body, parameter list, and its name. The tracking is represented as a link from the moved abstraction to its previous location, and vice versa.

An **internal reorganization** is not discovered by the tools. Internal reorganizations are not expected to affect the documentation.

We claim that the set of steps from above are complete in the sense that any program evolution process can be *modeled as a sequence of fine grained program evolution steps*.

Our modeling is not necessarily in full accord with the actual source file changes that take place in a program evolution process. We claim, however, that the model presented above is a good starting point for the understanding and description of more coarse grained evolutions, which we will discuss next.

3.2 Coarse grained program evolution.

As a natural contrast to the fine grained program evolution steps, discussed in Section 3.1, we will now introduce the concept of *coarse grained program evolution*. A coarse grained program evolution can be modelled as an aggregation of a sequence of fine grained program evolution steps, as they were described in Section 3.1.

In principle we can imagine an understanding of a coarse grained program evolution in terms of the understanding of a (potentially large) number of fine grained program evolution steps. It is, however, difficult to attach a high-level semantic understanding to such a set of fine grained program evolution steps. Consequently, we find it necessary to document coarse grained program evolution explicitly, in terms of some explanations that address the high-level understanding of the changes that have taken place.

We illustrate coarse grained program evolution with a very simple, stylized program written in Scheme, see Figure 3. The very simple setup of the example allows us to focus on the essential ideas and observations. In section 3.3 we will bring the discussion back to more realistic software.

The leftmost program in Figure 3 shows the original version (1) and the one to the right shows the most recent program version (2). Our documentation tools will report the following fine grained evolution steps:

- The function `f` is marked as updated
- The function `g` is not marked at all, indicating that `g` has not been changed.
- The function `hhh` is marked as a renaming of `h`.
- Both the functions `i` and `j` are marked as being new relative to version 1.

As mentioned above, the automatically supplied clues of the fine grained program evolution are useful, but they are not sufficient for a high-level understanding of development process that has taken place. There may be existing documentation of version 1 of the program, and it may be considered to bring this documentation in accord with version 2 of the program. We will, however, focus on documenting the evolution from version 1 to version 2. The following text represents an attempt to capture this evolution:

> *The function* `f` *is generalized with an extra optional parameter* `y`. *This ensures that* `f` *is backward compatibility with the earlier version of* `f`. *The generalization of* `f` *prompted the development of the function* `i`, *which in turn uses the function* `j`. *The function also uses* `hhh`, *which in the earlier version was named* `h`. *The name* `h` *turned out to be unnatural in the new context.*

Figure 4 illustrates a snapshot of the Elucidative Program. The documentation in the leftmost frame corresponds to the evolution described above. The program in the rightmost frame represents version 2 of the program from Figure 3.

We are able to write the documentation in a succinct style, because we have referential access to both the original version and the most recent version of the involved program abstractions. The automatically supplied clues shown within version 2 of the source program (such as "New", "Updated", and "Renamed") reinforce the explicitly written documentation. However, the clues cannot stand alone, because they do not carry any understanding of the evolutionary intentions behind the program changes. If we want to keep *the real understanding* of these intentions, this documentation must be written explicitly by the programmer, who carried out the development process.

3.3 Observations about real-life software evolution

It is evident that the example from Section 3.2 is far away from real-life software evolution. We have used the Elucidators for (partial) documentation of the Java and Scheme Elucidator tools themselves. After some years of development we basically found ourselves in *initial documentation only* situations (see Section 1). More specifically, there exists substantial documentation of the original tools, but this documentation has not been systematically maintained during the subsequent phases of the program evolution. Rather than striving for bringing the initial documentation up-to-date, we have gone for documenting the evolution of the tools.

The possibility of processing the initial documentation together with a number of different program versions (including the most recent version) is a definitive advantage compared to the situation where the initial documentation is processed with only the most recent program version. We have noticed the following advantages:

- **Highlighting of dated documentation.**
  Sections of the documentation which refer to a number of abstractions, which are not present in the new version of the program, are visually easy to spot, because they contain a lot of dimmed references. Such

```
(define (f x)
  (h x x))

(define (g a)
  (* 2 a))

(define (h a b)
  (fun (g b) (g a))
)
```

```
(define (f x . optional-parameters)
 (let ((y (optional-parameter 1 optional-parameters #f)))
   (if y
       (i x x (hhh y y))
       (hhh x x))))

(define (g a)
  (* 2 a))

(define (hhh a b)
  (fun (g b) (g a))
)

(define (i x y z)
  (j z y x))

(define (j x y z)
  (fun x y))
```

Figure 3. The original (left) and the updated (right) version of a program.

documentation sections are possible candidates for demotion to historical sections. More interesting in our context, it will be relevant to add a new documentation node which explains the reasons behind the dated aspects, and which points out possible newer program counterparts. In addition, by studying the program side, markings of "Dead ends" can help the documentation writer to identify other abstractions that are no longer present in the program. This may reflect changes that has a potential relevance as a historical aspect in the documentation.

- **Groups of new abstractions.**
  Groups of abstractions, which have been added at some recent point of time, are also easy to spot due to use of the "New" markings at the program side. Based on such a section of "New" abstractions, it is often relevant to add a documentation node which explains the evolution that led to these abstractions.

- **Groups of unaffected abstractions.**
  Regions of the program, which are not marked as "New" or "Updated", represent the stable aspects of the program. As the starting point for documenting the program evolution, it is valuable to be able to spot such regions, mainly because the documentation efforts should be directed at the complementing regions. These complementing regions can also be spotted as regions of the program where many abstractions are marked as "New" or "Updated". Furthermore, abstractions that are marked as "Updated" through several versions may represent more unstable aspects of the program.

- **Signs of reorganizations.**
  Sets of abstractions which are marked as "Renamed" or "Moved" are likely to reflect reorganizations of the software. Some of these reorganizations may play a role in the understanding of the software evolution. As such, these markings may serve as reorganization hints for the documentation writer.

In some cases it might be possible to spot parts of the program marked with the "Updated" tag. This can be taken as a hint of some kinds of updating which ought to be addressed in the documentation of the program evolution. It is, however, our experience that substantial parts of the program may be marked as updated, especially if only a few program versions are in play. As another difficulty, we are not currently able to distinguish minor and major updates. Major updates are likely to play a significant role in the understanding of the program evolution, whereas minor updates are more likely to be ignored.

As already discussed in Section 3.2 there is no hope for automating the creation of high-level documentation of the program evolution. But equipped with the means described above, it is possible to kick-start the documentation of the program evolution.

4 Related Work

Several researchers have proposed tools that somehow utilize information about previous versions of a program in order to assist programmers maintaining the program or in order to support further development of the program. We will here discuss a selection of these in relation to our own work. The selected tools represents four different approaches to supporting programmers in gaining an understanding of an existing program.

GEVOL extracts information about the evolution of a Java program stored in a CVS repository. GEVOL express the information as graphs, and it uses a temporal graph drawing system to visualize the information [**?**]. GEVOL

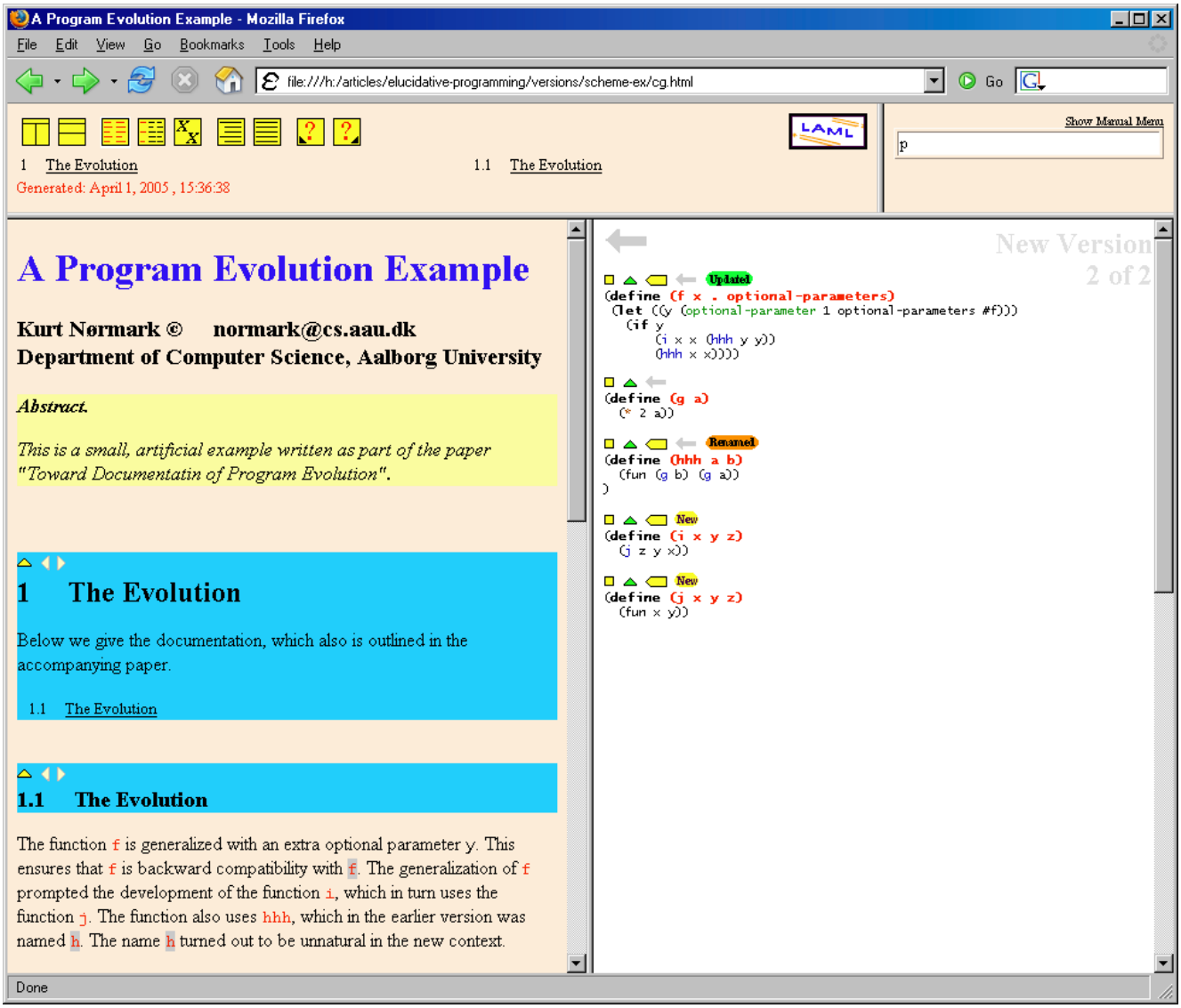

Figure 4. A presentation of the documentation of the evolution of the program in figure 3.

divides time into slices (for example one day) and creates inheritance-, control-flow-, and call-graphs for each time-slice. In the visualization of the graphs color-codes are used to denote the author of nodes, and how long nodes have been left unchanged. The programmer can now browse through the evolution of the program by stepping through the different graphs from each time-slice. Using GEVOL, programmers can visually identify parts of the program that are unstable for a long period of time, which programmers has modified which parts of the program and when, which parts of the program have been heavily modified over time, and identify growth in complexity throughout the program. Hence, GEVOL provide programmers with a visual overview of the entire program and its evolution, but it is up to the programmers to interpret and put meaning into this overview.

Our elucidative tools does not provide such an elaborate overview of the changes a program has undergone. The fine grained evolution-hints provided by the Elucidators do, however, provided the programmer with detailed information about evolution at specific locations in a specific version of the program.

The ROSE plug-in for the programming environment Eclipse [**?**] provides programmers with more direct information when modifying a given part of a program by indicating which other parts of the program have been changed

in the past in conjunction with the program part the programmer is currently changing [**?**]. The information is gathered using data mining techniques on change data from a CVS repository, resulting in a large set of rules which are queried whenever a programmer alters a given entity (e.g. a method, field, class or file) in the program. The queries result in a number of suggestions weighted with a confidence level. Each suggestion points to a location in the program that previously has been changed as a consequence of changing the specific part of the program which is currently being altered. Suggestions with a high confidence level issues warnings if the programmer commits changes without adhering to the suggestions. Even changes to files containing documentation as a consequence of a specific program change can be predicted. The precision of the suggestions generated by ROSE is fairly high (approx. 40% of related files and 28% of related entities) for stable programs with a long history (e.g. GCC). However, this precision seems to decrease drastically with more rapidly evolving programs. Others have experimented with similar tools, such as in [**?**] where the data mining is performed in update records from legacy systems (i.e. resolved bug reports) in order to find relationships between files that are changed together. Common for both data mining tools are fairly complex models and heavy processing requirements of the source files from different versions of a given program.

The elucidative tools focus on the specific change from one version of a program to the next version of the program in order to support revitalization of documentation. The ROSE tool could prove a very useful supplement when programmers are "digging out" the history of program.

Theme-based Literate Programming (TBLP) [**?**] has been introduced as a means for creating different paths through the chunks of a literate program. These paths through the chunks are called *themes*. In contrast to traditional Literate Programming [**?**], chunks are not limited to code or documentation. New chunk types can be defined as needed (e.g. chunks for figures, diagrams, or unit tests). Theme-based literate programs are explored and created in a special tool. The tool provides a facility for browsing themes and for editing the contents of the individual chunks of a given theme. All available chunks are stored in a common repository, and from the repository chunks can be assigned to relevant themes. Hence, an occurrence of a given chunk in any theme is a reference to the stored chunk in the repository. Furthermore, the tool contains an integrated version control system allowing themes to refer to specific versions of chunks. Hence, programmers can create themes that explain a given historical aspect by including different versions of relevant chunks of code (or documentation).

Themes explaining historical aspects can also be created using Elucidative Programming by mixing explaining text and version locked references. A concern about the TBLP model is that both program and documentation need to be divided into chunks. Just dividing a program and its documentation into chunks is not difficult, but figuring out the *right* chunks from the beginning is very difficult. "Chunking" of program and documentation is not needed if using Elucidative Programming.

Partitioned Annotations of Software (PAS) [**?**] is a hypertext based notebook in which programmers record their understanding of a program. This understanding stems from the programmer's work with and observations of an existing program. Hence, PAS supports programmers in incremental *re-documentation* of programs. The resulting documentation is studied in a web-browser. PAS divides a program into components (i.e. classes, functions, dependencies, and function arguments). Each component is associated with an *annotation* that explains that particular component. Each, annotation is divided into a number of *partitions*. The partitions for classes are for example, domain annotation, class dependencies, dependency annotation, authors comments, and member functions. Other partitions can be created as needed. The division of a program into components and partitions is automatically generated. The job of the programmers is then to fill the partitions and annotations with explaining text.

Although the approach presented in [**?**] does not, as such, deal directly with the actual changes a program has undergone, the approach serves as an interesting contrast, and alternative to our approach — especially when dealing with undocumented legacy programs.

5 Conclusion

In this paper we have presented an extension of the Elucidative Programming model that takes multiple program versions into account. The version-aware Elucidative Programming tools are able to process a set of source program files of different versions together with an unversioned documentation file. Due to the *version fall through feature* of the `doc-prog` relation, and due to the presentation of clues about the fine grained evolution steps, our approach can help revitalize older documentation in relation to more up-to-date program source files. In addition, the automatic discovery of the fine grained program evolution steps greatly help the programmer in documenting the evolution of the program.

We intend to continue our work on documentation of program evolution. We hypothesize that the understanding of the program evolution is important in a long running program development process. It is, of course, also important to understand the newest version of a program, but it is not always realistic to develop the program and the documentation in strict parallels. We find it interesting to explore documentation styles that explain the evolution of the pro-

gram from the starting point to the most recent version. In future work we therefore wish to explore *documentation of program evolution* in additional details.

References