TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

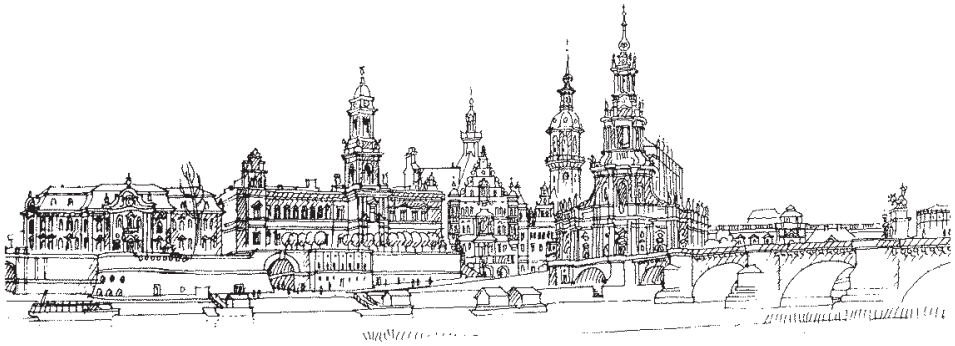
Technische Berichte
Technical Reports
ISSN 1430-211X

TUD-FI12-01-Januar 2012

**C. Wilke, A. Bartho, J. Schroeter,
S. Karol, U. Aßmann**

Institut für Software- und Multimediatechnik

Extended Version of Elucidative Development for Model-Based Documentation and Language Specification



Technische Universität Dresden
Fakultät Informatik
D–01062 Dresden
Germany
URL: http://www.inf.tu–dresden.de/

Extended Version of Elucidative Development for Model-Based Documentation and Language Specification

Claas Wilke, Andreas Bartho, Julia Schroeter, Sven Karol, and Uwe Aßmann

Institut für Software- und Multimediatechnik
Technische Universität Dresden
D-01062, Dresden, Germany
{claas.wilke|andreas.bartho|julia.schroeter|
sven.karol|uwe.assmann}@tu-dresden.de

Abstract. Documentation is an essential activity in software development, for source code as well as modelling artefacts. Typically, documentation is created and maintained manually which leads to inconsistencies as documented artefacts like source code or models evolve during development. Existing approaches like literate/elucidative programming or literate modelling address these problems by deriving documentation from software development artefacts or vice versa. However, these approaches restrict themselves to a certain kind of artefact and to a certain phase of the software development life-cycle. In this paper, we propose elucidative development as a generalisation of these approaches supporting heterogeneous kinds of artefacts as well as the analysis, design and implementation phases of the software development life-cycle. Elucidative development allows for linking source code and model artefacts into documentation and thus, maintains and updates their presentation semi-automatically. We present DEFT as an integrated development environment for elucidative development. We show, how DEFT can be applied to language specifications like the UML specification and help to avoid inconsistencies caused by maintenance and evolution of such a specification.

Key words: Elucidative development, elucidative programming, literate programming, literate modelling, automated documentation, automated specification, UML.

1 Introduction

To ensure comprehensibility and reusability, documentation is an essential activity in software development. Source code that belongs to frameworks and shall be reused by other developers has to be documented, as developers have to understand how to instantiate its classes or to invoke its operations. Besides, development models intended for reuse or explanatory reasons have to be documented as well. Finally, modelling languages or frequently used metamodels

have to be documented (typically as specifications) to explain their concepts and intentions.

Today, documentation is mostly created and maintained manually. Textual documents are maintained using text processing software, code listing and diagrams are created and pasted into these documents manually. This leads to problems, once documented software or development artefacts evolve: the documentation has to be maintained and the changes done to the artefacts have to be reflected in the documentation. Manual maintenance can cause inconsistencies, as sections requiring a revision can be overlooked. Furthermore, it is possible that evolved artefacts are updated at only some of their occurrences in the documentation, leading to further inconsistencies and contradictions. A good example for a documentation containing many inconsistencies caused by evolution and maintenance is the Unified Modeling Language (UML) specification [1], which does not document a tool or a framework but a modelling language used by a large community of software developers. Since its first revision, the UML specification documents have been maintained manually, and as a matter of fact, current versions contain many errors and contradictions [2–7].

To solve such kinds of problems, approaches such as literate programming (LP) [8], literate modelling (LM) [9] and elucidative programming (EP) [10] emphasise a documentation-centric style of programming or modelling. However, these approaches only cover small parts of the software life cycle restricting themselves to source code documentation during the implementation phase or the documentation of UML diagrams during the early stages of software analysis. Furthermore, to support documentation in model-driven software development (MDSD) processes, a holistic approach would have to consider documentation of artefacts from textual and graphical domain-specific modelling languages (DSMLs), metamodels and general purpose modelling languages, which is not the case for the current approaches. In this paper, we propose *elucidative development (ED)* as a holistic approach to documentation creation and maintenance which covers multiple phases of the software development life cycle and also supports documentation in MDSD processes. In fact, ED is a generalisation of LP, LM and EP. Furthermore, we present the Development Environment For Tutorials (DEFT), a tool supporting ED. We show how documentation can be created and maintained with DEFT and apply ED to a short excerpt from the UML specification to show, how inconsistencies within specifications caused by maintenance and evolution can be avoided.

The remainder of this paper is structured as follows. First, we introduce ED and DEFT in Sect. 2. Afterwards, in Sect. 3, we present an excerpt from the UML superstructure as a usage example for ED and show, how DEFT can be used to maintain its evolution. Subsequently, we discuss our approach in Sect. 4 and present related work in Sect. 5. Finally, we conclude this paper in Sect. 6.

2 From Literate Programming to Elucidative Development

In this section we introduce ED as a paradigm for a consistent documentation of arbitrary software and modelling artefacts that occur in software development projects. ED is a generalisation of the EP paradigm, proposed by Nørmark [11], which itself is a variant of LP by Knuth [8]. Both, EP and LP put strong emphasis on supporting developers in writing and maintaining program source code and its documentation in parallel during the implementation phase of the software life-cycle.

In the following, we first briefly introduce LP, EP and LM, a further variant of LP for documenting UML analysis models. Afterwards, we discuss the ED approach and compare it to the aforementioned documentation approaches. Afterwards, we present DEFT,[1] an Eclipse-based tool that can be used for ED.

2.1 Literate Programming and Related Documentation Approaches

LP is an integrated approach for writing documentation and programming within the same file format. Code and text are intertwined in the same document by embedding the source code into the documentation files. Hence, programming takes place in the documentation environment, e.g., a TEX [12] editor. Consequently, before the program can be executed or the final documentation is rendered, pre-compilers (called weave and tangle) have to extract printable TEX documentation and compilable source code from the documentation files. This way, LP completely avoids inconsistencies between source code and code listings in the rendered documentation. However, LP has drawbacks in large software projects: The program is scattered across the documentation files and every code detail has to be described textually. As a result, the program is fragmented and intertwined with pieces of text which makes it harder to understand its real structure for average programmers who expect programs to be organised along a certain structure determined by the concepts of the programming language.

EP tries to overcome these problems by strictly separating documentation and source code artefacts. The connection between them is maintained within an integrated elucidative programming environment. As a result, programming language semantics such as name analysis can be reused for consistency checks in the documentation files. Furthermore, the granularity of the documentation is adjusted to its actual purpose, e.g., abstract interface descriptions as well as complete source code descriptions are possible. In comparison to LP, in EP consistency between code listings and the actual program code is ensured by adding so-called relations between locations in the documentation and elements of the source code. The entirety of documentation, source code, and relations between those two is called an *elucidative program* [10]. If the source code evolves, the final documentation can be regenerated. It is possible to identify inconsistencies to some extent, e.g., relations which refer to removed or renamed source code.

[1] http://deftproject.org/

	documentation format	**artefact support**	**artefact location**	**tool support**	**operations**	**software dev. phases**
literate programming (LP)	typesetting language (e.g. TEX)	homogenous (source code)	integrated	pre-compiler (e.g. CWEB [12])	weave, tangle	implementation
elucidative programming (EP)	typesetting language	homogenous (source code)	separate	elucidative IDE (e.g. Java Elucidator [14])	embed	implementation
literate modelling (LM)	WYSIWYG format	homogenous (UML diagrams)	separate	literate model editor (e.g. LiMonE [13])	embed	analysis
elucidative development (ED)	WYSIWYG format or typesetting language	heterogenous (models, source code, XML ...)	separate	elucidative development environment (e.g. DEFT [15])	hot update, transconsistency	analysis, design, implementation

Table 1. Comparison of different advanced documentation approaches.

The LM [9] approach applies concepts of LP to high-level UML analysis models. The main focus of LM lies in improving the communication between developers, requirements engineers and other stakeholders who are not educated in UML and, thus, have difficulties in interpreting UML diagrams. Similar to LP, models and documentation are intertwined within the same document – the *literate model*. However, recent efforts also move LM in the direction of separating documentation and the documented artefacts: The *Literate Modelling Editor (LiMonE)* [13] implementation keeps both separate and combines textual model documentation with Object Constraint Language (OCL) consistency constraints derived from natural language descriptions.

The first three rows of Tab. 1 contrast the documentation approaches discussed above with each other. As an essence, it can be seen that each of them is restricted to one single phase in the software life-cycle and to one single type of artefact, i.e., source code in a certain (implementation dependent) programming language or artefacts in a certain modelling language.

2.2 Elucidative Development

ED generalises the aforementioned documentation approaches in two ways (cf. last line of Tab. 1). First, it covers the analysis, development and implementation phases in software development. Hence, programmers, designers and other stakeholders can share their views on the system at different levels of abstraction. Second, ED provides a conceptual grounding for the documentation of heterogeneous kinds of software artefacts, e.g., formalised requirements specifications, models, or source code. This is essential for model-driven software development processes, where many different metamodel-based languages are used to implement a system by transformation and code generation.

As a consequence, an *elucidative development environment (EDE)* has following basic requirements which go beyond the requirements known from EP, LP and LM tools:

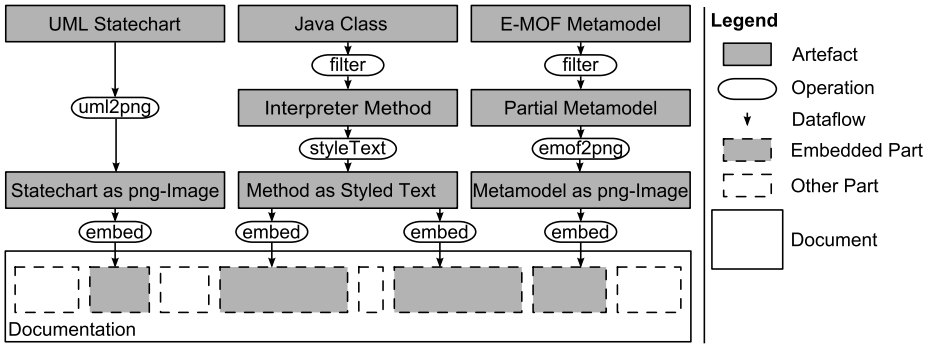


Fig. 1. Dataflow diagram of an ED document.

Support for model transformations. Different kinds of languages require different kinds of transformations to prepare artefacts to be displayed in a documentation file. This includes model-to-model transformations (e.g., operations to filter elements that should not be included in the documentation), model-to-text transformations (e.g., for deriving textual artefact representations) and model-to-image transformations (e.g., for converting a diagram into an image).

Composition of model transformations. The aforementioned transformations need to be composable to produce images or code listings that integrate with the surrounding hand-written text and other parts in the documentation file. Valid compositions are determined by the types of input and output ports of the participating transformations. Consequently, the compositional relation of transformations, data and the documentation text form a directed bigraph, which represents the documents architecture [16]. Fig. 1 shows the architecture of an hypothetic ED document. Assume a project that implements an interpreter for a DSML in Java. The corresponding Java source code is extracted from a Java class model and transformed into a styled code listing which is finally embedded in the documentation file. To support the documentation, a statechart image is generated from a UML statechart diagram. The statechart specification originates from the design phase. Finally, the documentation includes parts from the DSML's metamodel specification which are relevant for the documented part of the interpreter.

Hot update and immediate invalidation. As the documented system artefacts and the documentation itself evolve, frequent updates have to be triggered over time. ED documents are *active documents* [16]. An active document triggers an update operation as soon as a change in a source artefact is observed. Due to its explicit architecture, the document (or the EDE, respectively) is aware of all places where artefact representations to be recomputed occur. Since the required transformations may contain complex computations, the corresponding invalid parts of the document are marked until the recomputation is finished and the document becomes consistent

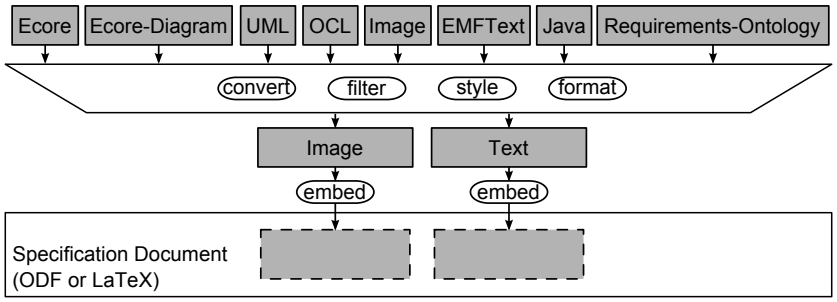


Fig. 2. The Development Environment for Tutorials (DEFT).

again. In [16], this kind of update is called a *hot update* while an active document with hot update is called a *transconsistent* document. *Transconsistency* is closely related to the terms *transclusion* and *transclude*, which both originate from the early hypertext systems [17]. Nelson defines *transclusion* as "the same content knowable in more than one place" [18].

In the following, we introduce our tool DEFT. DEFT supports the features discussed above to a large extent and, thus, is a good candidate for supporting an ED processes.

2.3 The Development Environment for Tutorials (DEFT)

An EDE supports the documentation author in creating and maintaining elucidative documentations. It also provides automatic notifications and further support for hot updates of the included source artefacts once the documented concepts evolve. DEFT is an implementation of such an EDE. It was originally designed to keep the documentation of whole software systems, tutorials up to date. As we show in this paper, it is also feasible for writing and maintaining large language specification documents.

Out of the box, DEFT supports the documentation of artefacts that occur in usual software development or MDSD, such as Java source files and Eclipse Modeling Framework (EMF) artefacts (cf. Fig. 2, top). Besides, DEFT's integration with EMF allows the documentation of arbitrary languages based on Ecore, which is the de-facto Essential MOF (EMOF) implementation for Java. Hence, DEFT supports the documentation of the UML metamodel based on EMF. For textual modelling, DEFT is integrated with EMFText [19]. Therefore documentation support for EMFText-based languages is available. Hence, OCL constraints can be documented by using the EMFText-based OCL implementation of Dresden OCL [20]. For graphical modelling, the Ecore diagram editors of the EMF are supported. Integration for UML models and diagrams is current work in progress. As documentation formats, DEFT supports LaTeX and Open Document Format (ODF) documents (cf. Fig. 2, bottom). A documentation file produced with DEFT can contain manually written parts like continuous text,

as well as transcluded elements like code listings or images. These elements can be derived from all the input formats described above and can be converted or formatted before embedded into the specification document (cf. Fig. 2, center). For example, OCL constraints can be transformed into code listings, UML diagrams can be rendered as images, or enumerations can be generated from UML metamodel elements (e.g., of a class' properties and operations). If a modification of an artefact requires a modification of a transcluded element in the documentation, DEFT updates the artefact representation automatically.

The user interface of DEFT is divided in multiple areas[2]. A *project explorer* presents documentation chapters, source artefacts, and their relations. The largest part of the screen is covered by the *writing area*, where the documentation text can be edited. Relations to artefacts can be added to the document using a wizard. By default, DEFT does not display the relations directly. Instead, the computed representation is transcluded. Finally, DEFT provides a *task view* which tells the author where changes in the source artefacts took place, where the documentation has been updated and, thus, where proofreading is necessary.

3 The UML Specification as a Use Case for Elucidative Development

To demonstrate the advantages of ED in contrast to other documentation approaches, we decided to use an excerpt from the UML 2 specification. In this section, we first identify different kinds of consistency problems and give examples for them within the current UML standard. Afterwards, we discuss the Object Management Group (OMG)'s specification process and identify error-prone steps causing consistency issues in current OMG specifications. Finally, we apply ED to the example and show, how ED can avoid the current inconsistency problems of the UML specification.

3.1 Inconsistencies in UML 2.4.1

Since its first specification, the UML has been extended and revised multiple times. A major change in the UML was the specification of UML 2.0 in 2005 which contained many new concepts. However, further revisions of UML 2 added many inconsistencies to the specification document. As a small example, we compare Sect. 7.3.37 of the UML 2.4.1 specification [21, p. 108–110] with the same section of the UML 2.0 specification document [22, p. 103–105]. It specifies the class `Package` within the UML `Kernel` package as shown in Fig. 3. Changes between UML 2.0 and UML 2.4.1 are highlighted, as well as inconsistencies that have not been revised, yet. The example section contains a short description of the `Package` class, its inheritance relationships, attributes, associations, constraints, additional operations, and semantics.[3]

[2] A screenshot will be presented in Sect. 3.3, in context of the case study.

[3] For complexity reasons, the graphical notation, presentation options, and examples following in the specification are not considered as part of our example.

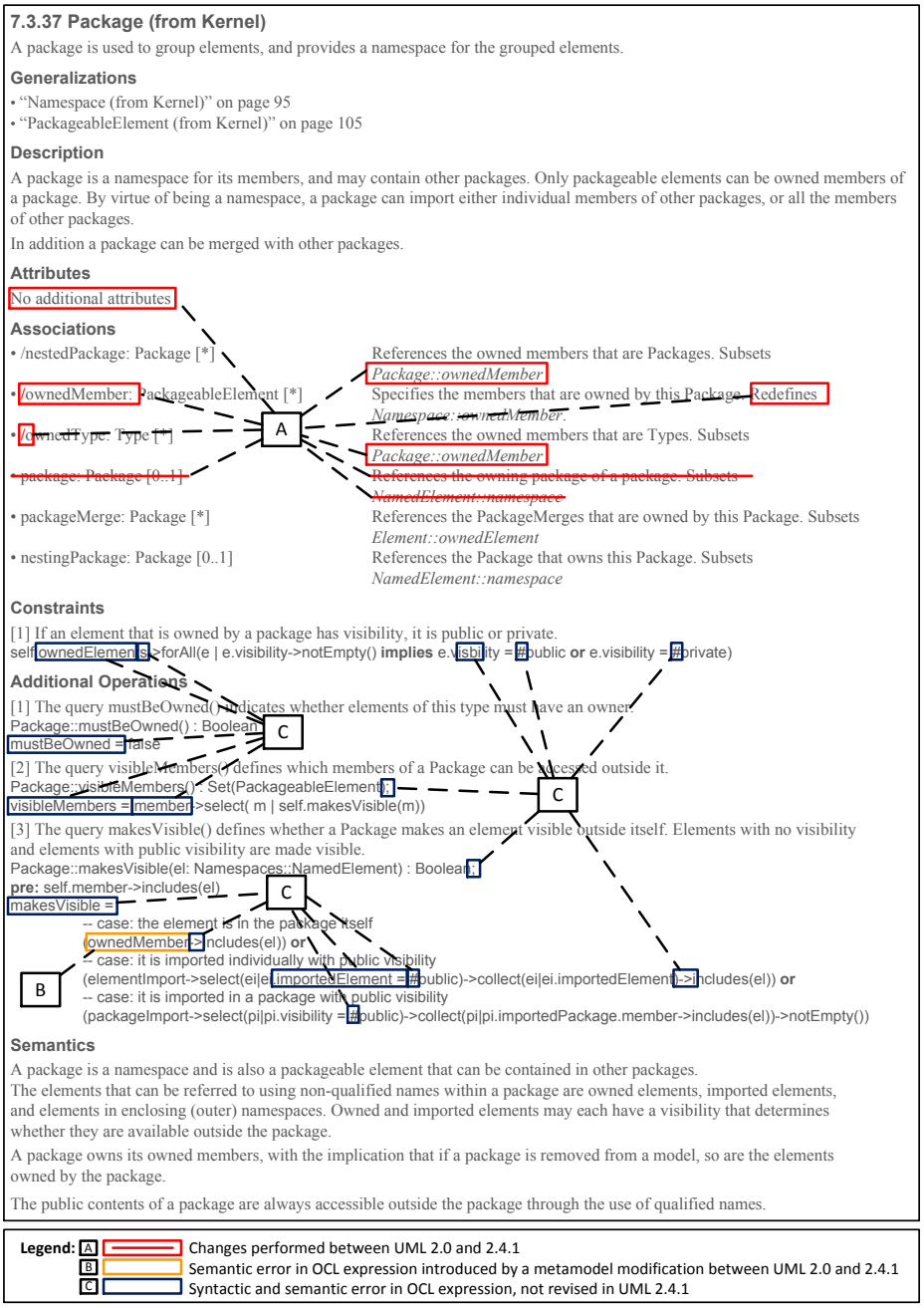


Fig. 3. Excerpt from UML 2.0 and its modification until UML 2.4.1 (cf. [22, p. 103f]).

The major changes of the example section are located within the *Attributes* and the *Associations* subsections, as shown in Fig. 3, `[A]`. A new attribute `URI` is introduced. The association `ownedMember` is renamed to `packagedElement`. This association and the `ownedType` association are marked as `derived`, which is indicated by a leading backslash. Furthermore, the `package` association is removed. References to the renamed element are revised as well (e.g., the `subsets` relationship from `nestedPackage` to `ownedMember`). However, these references are sources of potential errors as the complete specification has to be inspected to check whether other references to the modified element exist that must be updated as well. For example, the renaming of the `ownedMember` association leads to an inconsistency within the *Additional Operations* subsection where an OCL expression references this association (cf. Fig. 3, `[B]`). This is not surprising since obviously the OCL expressions used within the UML specification have not been revised since their original definition in UML 2.0 and have been specified without using any OCL tooling such as a parser checking their syntax and static semantics [7]. Thus, the expressions contain various syntactical and semantic inconsistencies (cf. Fig. 3, `[C]`). Summarising, we identified four different kinds of consistency problems[4] that occur during specification maintenance:

(P1) Textual Representation: Modification of elements (e.g., rename, remove, insert) specified in the language, entails the update of enumerations in the specification containing those elements. For example, the renaming of the association `ownedMember` must be performed in the *Associations* subsection. Neglecting this leads to inconsistent documents.

(P2) Continuous Text: Missing updates of continuous text that documents and clarifies specification elements. For instance, if the class `Package` is renamed, the introduction of the section and the *Description* subsection need to be revised accordingly as they describe the `Package` class.

(P3) Graphical Representation: The concepts of UML are specified graphically as class diagrams. Thus, if any property or association of the `Package` class is modified, all diagrams containing this class must be updated as well. For example, the specification contains the *Fig. 7.14* documenting the `Package` class and its relations [22, p. 31] that must be revised after modifications of this class.

(P4) External References: Other content referring to the specified model elements (e.g., the OCL expressions) must be updated and probably modified as well. This is a task that is obviously too complicated to be performed manually, as the UML specification contains many inconsistency problems of this category [7].

3.2 The OMG Technology Adoption Process

To understand the reasons for all the inconsistency problems, we now shortly elaborate the OMG technology adoption process, used by the OMG to manage

[4] Three out of the four problem kinds exist within our small example.

its various specifications and their evolution (cf. Fig. 4). It is defined by a series of strict rules and clearly distinguished responsibilities [23]. The process consists of five stages[5], that are shortly presented in the following:

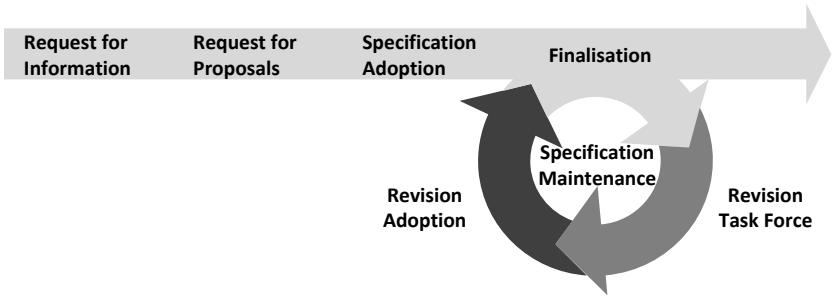


Fig. 4. The OMG Technology Adoption Process.

First, a *request for information* is issued to collect information about the requirements for a new standard from industry. Afterwards, a *request for proposals* calls for proposals that are collected and reviewed. This stage results in one or more major proposals for a new OMG specification. During *specification adoption*, the proposals have to pass votes of several committees, before they become an *OMG adopted specification*. During *finalisation*, the adopted specification is maintained. The result of this stage is the final specification document. After another series of votes, the specification obtains its official release number and becomes a formal OMG specification. The fifth stage, *specification maintenance* is directly initialised once the specification has been published. It is performed iteratively, each iteration resulting in a new revised specification. In detail a revision task force (RTF) is responsible to maintain change requests, which can be issued by everybody. Those requests result in an RTF report or a proposal for a modified specification. Thereby, each issue can be resolved, merged, transferred, closed, or deferred. Afterwards, the proposal for a revised specification has to pass votes of the same committees as during *specification adoption*. Each maintenance cycle results in a new *finalisation*. In this stage the passed proposals are introduced into the final specification document, resulting in a new OMG formally released version. A summary of the activities of the maintenance cycle can be found in [24].

Generally, the process works well as it describes how to adopt a specification and defines a maintenance cycle that leads to revised specifications. However, inconsistencies occur as shown in Sect. 3.1. The major reason for these inconsistencies is that the process only handles textual specification documents. During the first *finalisation* stage, a textual document is created and released. Afterwards, change requests are formulated w.r.t. this textual specification. Once the

[5] A sixth stage, handling *specification retirement* is not considered in this exploration.

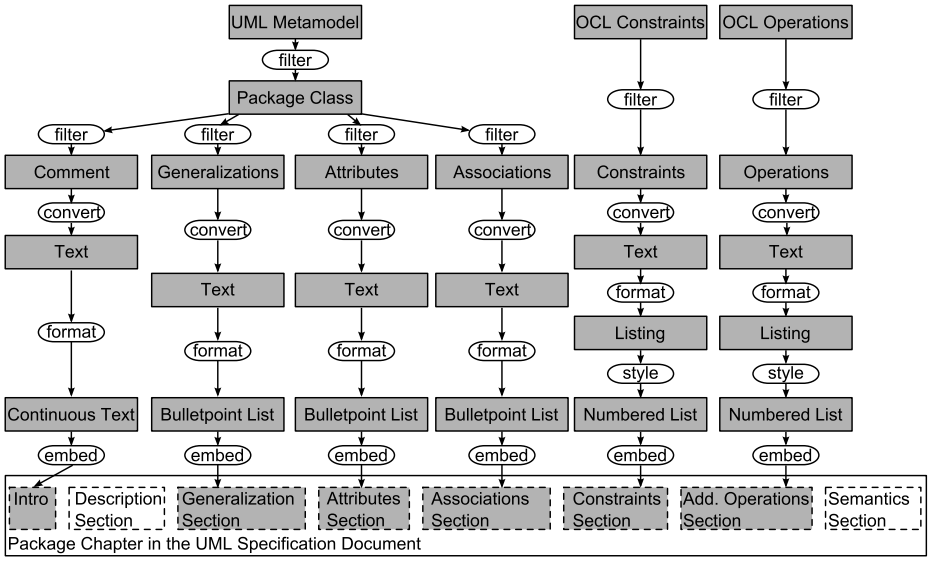


Fig. 5. Transclusion graph for an elucidative UML specification.

issues have been discussed by the RTF, they result in a change proposal that documents all pages, paragraphs etc. that have to be modified within the textual specification document. Identifying all element occurrences in the text manually is complicated and time-consuming and hence, inconsistencies are very likely to be introduced.

To overcome the inconsistency problems identified above the OMG has issued a new request for proposal that shall result in a new revised, and inconsistency-free version of the UML, as well as up-to-date implementations of the metamodel and its OCL constraints [25]. Besides this specification revision, we propose the application of ED. Instead of focusing on textual specification documents only, change requests focus on modifying elements in the formal representation of the specification (i.e., the specified artefacts such as models and constraints). Tooling helps to derive textual specifications and to identify regions within them that have be revised and updated semi-automatically.

3.3 UML Language Specification with DEFT

We now show, how DEFT supports ED exemplified by the UML specification excerpt presented in Sect. 3.1. The example is realised using the EMF-based UML metamodel of Eclipse Eclipse Model Development Tools (MDT). For the specification of derived operations and OCL well-formedeness rules (WFRs) we use Dresden OCL. Finally, a diagram representation of the `Package` class and its relations to other classes is created using the graphical EMF Ecore editor of EMF.

For the UML example we transclude diagrams and OCL files in a specification document. This includes the selection which parts (e.g., which lines from an OCL file) shall be presented in the document. That way, almost all sections from the given example can be transcluded into the document (cf. Fig. 5). The introductory paragraph is derived from an annotated comment of the `Package` class. The *Generalizations*, *Attributes*, and *Associations* sections are taken from the `Package` class and its relations to other classes. The content of those sections is directly derived from the metamodel and formatted using rendering templates. The *Constraints* and *Additional Operations* sections are transcluded from the linked OCL files. Only the sections *Descriptions* and *Semantics* are not present in the artefacts and must therefore be written directly into the document.

Fig. 6 shows a snippet of the elucidative UML specfication in DEFT according to the transclusion graph in Fig. 5. On the left side of the screen the project content is shown in the *project explorer*. It presents specification's chapters, source artefacts like the UML metamodel and OCL constraints, and their relations side by side. The actual specification document is displayed in the *writing area* and can be edited using OpenOffice. As visualised in Fig. 6, the artefact representations in the documentation file are transcluded from the artefacts added to the project explorer.

To solve the identified problems (P1) to (P4), model artefacts and the specification document have to be modified in parallel during the maintenance cycle. During this phase, DEFT supports the author with the revision of the specification. If one of the referenced model elements changes, DEFT will immediately update the corresponding transcluded elements of the documentation and will notify the specification writer about updates. This helps the author to keep track of the changed specification parts as displayed in the *task view* (cf. Fig. 6).

(P1) problems can be avoided by transcluding textual model descriptions. If the models are modified, DEFT will update the descriptions automatically. (P2) problems can be handled similarly. Words within manually written text which relate to names or content of the specified metamodel (such as the terms `package` or `namespace` in Fig. 3) can and should be transcluded as well. If the model elements are renamed later, the specification document is automatically updated and the new names will appear in the document. However, if the semantics of the model changes, it is necessary to update the corresponding text manually. (P3) problems are also solved automatically. The outdated graphical representation will be replaced by the current version of the diagram. Finally, (P4) problems can be avoided if the OCL constraints are re-parsed after metamodel modifications and it is checked whether they are still consistent w.r.t. their static semantics. As they are also transcluded, modifications are immediately reflected in the specification.

Maintenance of the specification is expected to be rerun in multiple iterations (cf. Fig. 4). After each modification of the specified metamodel the documentation can be revised semi-automatically using DEFT. A revised specification can be released and the next maintenance cycle can be performed. This iterative process, using small changes and fast iterations helps to avoid inconsistencies as

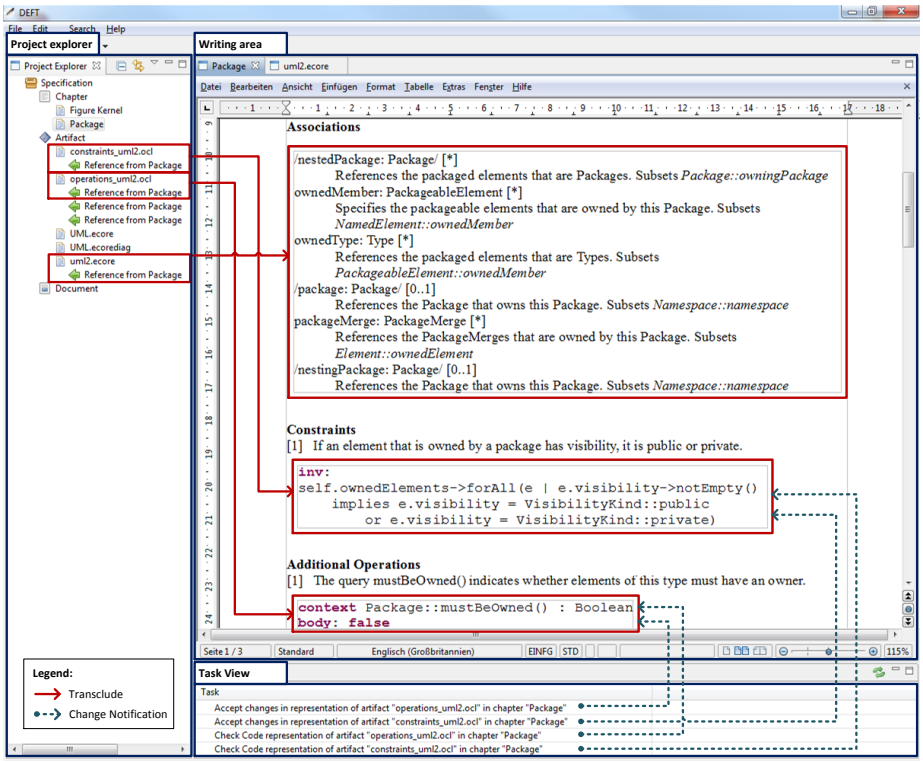


Fig. 6. Screenshot of the specification document in DEFT.

each modification of the metamodel or related artefacts is immediately displayed in the specification via hot update.

4 Discussion

After presenting ED and applying it to an excerpt of the UML specification we now discuss the resulting benefits. In our opinion ED can help to achieve more consistent language documentation and more formal specifications in MDSD processes. As results from our earlier work demonstrated, almost every second OCL constraint of the UML 2.4.1 specification contains errors that can be avoided by using ED [7]. A simple integration of model-based techniques and OCL parsing into the specification process is sufficient to solve problems as syntactical and type checking errors (about 61.6% of all inconsistencies identified in [7]). Besides, the use of transclusion avoids inconsistencies due to overlooked modifications caused evolution of the UML metamodel (another 22.1% of all inconsistencies identified in [7]).

Furthermore, ED allows small maintenance iterations as the maintenance of the source artefact and the specification are more interconnected. Thus, re-

quested changes are realised and newly revised specifications can be released quickly and regularly. Another advantage of using source artefacts during the specification process is that models can be shipped together with the specification. For example, the specified UML metamodel and parsable OCL files can be provided.

However, creating an initial documentation using DEFT may at first be time-consuming and more complicated than using a non-elucidative process. Learning how to use DEFT and how to integrate artefacts into specification documents and implementing transformations for language-specific presentation of content (e.g., as for the *Associations* section of the given example) may be a time-consuming task as well. However, we argue that ED pays off when the documentation (or the system) evolves. Besides, for documents like the UML specification, where similar sections occur for all the different classes defined in the manually—a script could help to generate a first version of the documentation containing similar information transcluded from the various artefacts. Such a script would have to be written once, whereas otherwise, the same content for all metaclasses would have to be created manually.

Further, we argue that ED helps starting a language specification when the language is still in an early stage and many changes occur. Early documentation pays off as documenting a language gives new thoughts and sights onto possible problems in the language design. ED supports specification evolution and thus, encourages to start a language documentation early.

The UML specification process could also be enhanced using a standard template engine. However, instead of using a template, applying ED with DEFT has several advantages. First, the purpose of DEFT is creating and maintaining large documentation files in an author-oriented way. Hence, it offers more appropriate abstractions and a WYSIWYG editor while a template engine is more likely a simple programming language and requires a certain amount of extra learning effort. Second, a template engine is not aware of the involved artefact relations, thus there is no support for hot update. Regeneration has to be triggered manually and involves the whole documentation, in the case of the UML specification this can be time consuming. Third, it is more difficult to identify changes that need to be proofread by a human. Finding differences between new and previous versions would require running a differentiation tool. With standard differentiation tools this can be cumbersome to examine.

5 Related Work

In this section we discuss tools that are related to our proposed tool DEFT and applicable for maintaining documentations and specifications. Furthermore, we present work discussing the soundness and quality of the UML specification and WFRs within the UML specification.

5.1 Documentation in MDSD Processes

Besides DEFT, other tools for documentation in MDSD processes exist.

Topcased[6] is an open-source tool for model-driven development. It is based on the modelling capabilities of the Eclipse MDT but comes together with its own editors for graphical UML modelling and OCL editing support. As part of the tool suite, Topcased Gendoc allows the generation of textual documentations for UML models. Templates consisting of explanatory descriptions of the modelled concepts can be defined, including queries against UML models to derive explanatory figures and diagrams. It is possible to generate reports similar to the documents maintained with DEFT. However, Topcased Gendoc uses a completely generative approach to create model documentations and, thus, suffers from the shortcomings for generative approaches as discussed in Sect. 4.

The Eclipse-based tool suite *Business Intelligence Reporting Tools (BIRT)* can be used to generate business reports from data maintained in databases.[7] It uses a template-based approach for the generation of textual reports, including tables and charts derived from the documented data. Although BIRT allows the creation and maintenance of documents, its major focus is on the generation of database reports and not on the documentation of modelling artifacts, such as EMF models. Furthermore, similar to Topcased Gendoc, BIRT only supports a model-based an no elucidative documentation process.

Intent[8] is a recent documentation project inspired by LP. Documentation is created and maintained in a textual DSML that can express textual documentation as well as EMF-based artefacts (e.g., EMF model elements). The described artefacts and the documentation are derived from this description.[9] Besides models, Intent also supports other artefacts describable using EMF (e.g., source code).

The *LiMonE* (c.f. Sect. 2.1) tool uses natural language processing for improving documentation consistency. For example, a sentence like *"A Class can have multiple Operations"* can be transformed into an OCL query that checks that the association between the classes `Class` and `Operation` has the right multiplicity in the UML metamodel. By combining DEFT and LiMonE, the elucidative process outlined in this paper could be further improved, as consistency checks on explanatory descriptions within the UML specification would be possible. Theoretically, even hints could be derived to inform the user which sentences within the specification have to be modified in which way to update the descriptions w.r.t. the modified metamodel.

5.2 Consistency of UML Specifications

A lot of related work exist that focuses on consistency of the UML specification.

One of the most well-known publications in this domain is an article by Henderson-Sellors [3] that documents the result of a panel discussion of a group of modelling experts documenting their impressions of UML 2.0. Although the

[6] http://www.topcased.org/
[7] http://www.eclipse.org/birt/
[8] http://eclipse.org/intent/
[9] According to http://wiki.eclipse.org/Intent/Architecture, visited in January 2012.

article addresses various kinds of problems in UML, these descriptions include the necessity of future revisions to improve the specification and the finding that many definitions are scattered throughout the specification.

In [2] Selic defines a basis for a formal description of UML 2.0's runtime semantics. Although the paper focuses on the semantics definition, it also documents that semantics of UML concepts is scattered throughout the complete specification and different statements even contradict, which leads to inconsistent semantics definitions and statements such as that UML has "no semantics". Again, this work can be considered a motivation that the UML specification requires techniques such as ED.

France et al. present the UML 2.0 in [26], but criticise its size and complexity. The work states that one of the UML's major challenges is its evolution which cannot be performed manually since concepts are scattered around the complete specification and are strongly interconnected.

A work by Belaunde that focuses on the OCL instead of the UML specification process gives a short insight into the OMG specification process and lists some reasons causing the existing inconsistencies during specification maintenance [24].

Other authors focus on the consistency of OCL rules within the UML specification and also on co-refactoring (or co-evolution) of OCL rules and their constrained (meta)model. Some of these works are outlined below.

In 2003, Fuentes et al. [5] investigated the consistency of OCL rules within the UML 1.5 specification. They identified about 450 errors they categorised into non-accessible elements, empty names and other errors, including about 150 errors w.r.t. inconsistencies between the rules and the UML metamodel. Besides the identification of 450 errors Fuentes et al. also investigated inconsistencies (and even contradictions) between the given OCL rules and their textual documentation.

In 2004, Bauerdick et al. investigated OCL WFRs specified within the UML 2.0 superstructure [6] and detected more than 350 errors within these OCL rules. The errors were structured into categories including syntactical errors, inconsistencies between OCL rules and UML metaclasses and type checking errors. Eearlier work of the same group was performed by Richter et al. in 2000 [4]. An excerpt of UML 1.3 metamodel was investigated containing more than 70 OCL expressions with more than 35 errors.

In earlier work we investigated the consistency of constraints defined within the UML 2.3 specification using OCL [7]. We identified about 320 errors in 442 OCL constraints. Hence, about 26% of all investigated OCL rules contained errors w.r.t. consistency between the rules and the evolved UML metamodel.

Some works exist that focus on the UML/OCL co-evolution or co-refactoring problem. Marković et al. formalised first refactorings of UML class diagrams that affect related OCL constraints and proposed Query/View/Transformation (QVT) rules for OCL co-refactorings [27]. Further work in this area based on existing Eclipse tools was done by Hassam et al. [28]. These results could be used to further improve our approach w.r.t. guidance for semi-automated OCL

co-evolution which could help to keep the OCL WFRs and operation body definitions consistent to the UML metamodel.

6 Conclusion

In this paper we have presented the *elucidative development* approach as a more versatile variant of *literate programming*. ED supports the documentation of source code, model artefacts and language specifications and DSMLs. Source artefacts such as metamodels and OCL constraints are transformed and transcluded into documentation files via hot update.

As a use case, we investigated an excerpt from the UML specification and identified inconsistency problems of different kinds resulting from a manual specification maintenance process. As demonstrated, these problems can be prevented by using an elucidative IDE such as DEFT instead.

Additionally, some support for describing variants of the same specification would be a valuable add-on for ED, since a different group of readers may require different levels of abstraction with regard to the full specification. For example, for a business audience, a specification usually needs to be much more abstract to ease understanding. In [29], we proposed to use *feature models* [30] to model and generate variants from *document families* based on the ODF and OpenOffice. Since DEFT also supports ODF, an integration of both approaches should be easily feasible in the future. Also, a combination with the LiMonE approach seems a promising idea, especially if feature models are used to capture semi-structured text content of specifications. However, these ideas still in an early state of evaluation.

Furthermore, ED could be combined with other techniques for co-evolution. For example, co-evolution of EMF or UML models OCL constraints that allows the propagation of model modifications to their OCL rules would be an interesting task. First works in this domain exist [27, 28] and could provide a basis for such an integration.

Finally, round-trip support that allows the editing of transcluded model representations in the documentation and propagates changes back to the model repository could improve the usability of DEFT and ED.

Acknowledgement

This research has been co-funded by the European Social Fund and Federal State of Saxony within the project ZESSY #080951806, by the European Social Fund, Federal State of Saxony and SAP AG within project #080949335, by the Collaborative Research Center 912 (HAEC), funded by DFG, and by the Federal Ministry of Education and Research within the project CoolSoftware #FKZ13N10782.

References

1. Object Management Group (OMG) Unified Modeling Language. Online available specification. http://www.omg.org/spec/UML/.

2. Selic, B.: On the Semantic Foundations of Standard UML 2.0. In: Formal Methods for the Design of Real-Time Systems. Volume 3185 of LNCS., Springer Berlin/Heidelberg (2004) 75–76
3. Henderson-Sellers, B.: UML – The Good, the Bad or the Ugly? Perspectives from a panel of experts. Software and Systems Modeling **4** (2005) 4–13
4. Richters, M., Gogolla, M.: Validating UML Models and OCL Constraints. In: Proceedings of the 3rd international conference on the Unified Modeling Language: advancing the standard, Berlin/Heidelberg, Springer (2000) 265–277
5. Fuentes, J., Quintana, V., Llorens, J., Génova, G., Prieto-Díaz, R.: Errors in the UML metamodel? ACM SIGSOFT Software Engineering Notes **28**(6) (2003)
6. Bauerdick, H., Gogolla, M., Gutsche, F.: Detecting OCL Traps in the UML 2.0 Superstructure: An Experience Report. In: UML 2004 - The Unified Modelling Language. Volume 3273 of LNCS., Springer Berlin/Heidelberg (2004) 188–196
7. Wilke, C., Demuth, B.: UML is still inconsistent! How to improve OCL Constraints in the UML 2.3 Superstructure. In: Proceedings of the Workshop on OCL and Textual Modelling (OCL 2011). Volume 44 of Electronic Communications of the EASST. (2011)
8. Knuth, D.E.: Literate Programming. In: The Computer Journal. Volume 27(2). (May 1984) 97–111
9. Arlow, J., Emmerich, W., Quinn, J.: Literate Modelling - Capturing Business Knowledge with the UML. In: The Unified Modeling Language UML'98: Beyond the Notation. Volume 1618 of LNCS., Springer Berlin/Heidelberg (1999) 378–392
10. Nørmark, K.: Elucidative programming. Nordic Journal of Computing **7** (June 2000) 87–105
11. Nørmark, K.: Requirements for an Elucidative Programming Environment. In: Proceedings of the 8th International Workshop on Program Comprehension. IWPC '00, Washington, DC, USA, IEEE Computer Society (2000) 119–128
12. Knuth, D.E., Levy, S.: The CWEB System of Structured Documentation: Version 3.0. 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1994)
13. Schulze, G.: Synchronization of UML Models and Narrative Text using Model Constraints and Natural Language Processing. Master's thesis, University of Innsbruck (2011)
14. Nørmark, K., Andersen, M., Christensen, C., Kumar, V., Staun-Pedersen, S., Sørensen, K.: Elucidative programming in Java. In: Proceedings of IPCC/SIGDOC '00, IEEE Educational Activities Department (2000) 483–495
15. Bartho, A.: Creating and maintaining tutorials with DEFT. In: Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on, IEEE (2009) 309–310
16. Aßmann, U.: Architectural styles for active documents. Science of Computer Programming - Special issue on new software composition concepts **56** (April 2005) 79–98
17. Nelson, T.H.: Complex information processing: a file structure for the complex, the changing and the indeterminate. In: Proceedings of the 1965 20th national conference, New York, ACM (1965) 84–100
18. Nelson, T.H.: Literary Machines. 3rd edn. Mindful Press, Sausalito, CA (1981)
19. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: Model Driven Architecture - Foundations and Applications. Volume 5562 of LNCS., Berlin/Heidelberg, Springer (June 2009) 114–129

20. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Thiele, M., Wende, C., Wilke, C.: Integrating OCL and Textual Modelling Languages. In: Workhops and Symposia Proceedings of Models 2010. Volume 6627 of LNCS., Berlin/Heidelberg, Springer (May 2011)
21. Object Management Group (OMG) Unified Modeling Language: Superstructure Version 2.4.1. Online available specification (August 2011)
22. Object Management Group (OMG) Unified Modeling Language: Superstructure Version 2.0. Online available specification (August 2005)
23. Object Management Group (OMG) OMG's Technology Adoption Process. OMG Website Article (Visited in January 2012) http://www.omg.org/gettingstarted/processintro.htm.
24. Belaunde, M.: Evolution fo the OCL OMG Specification. In: Invited talk at the Workshop on OCL and Textual Modelling, collocated with MODELS2010. (2010)
25. Object Management Group (OMG) UML Specification Simplification RFP (December 2009) http://www.omg.org/cgi-bin/doc?ad/2009-12-10.
26. France, R., Ghosh, S., Dinh-Trong, T., Solberg, A.: Model-driven development using UML 2.0: promises and pitfalls. Computer **39**(2) (2006) 59–66
27. Marković, S., Baar, T.: Refactoring OCL Annotated UML Class Diagrams. In: Model Driven Engineering Languages and Systems. Volume 3713 of LNCS. Springer Berlin/Heidelberg (2005) 280–294
28. Hassam, K., Sadou, S., Le Gloahec, V., Fleurquin, R.: Assistance System for OCL Constraints Adaptation During Metamodel Evolution. In: Proceedings of 15th European Conference on Software Maintenance and Reengineering (CSMR 2011), Los Alamitos, CA, USA, Conference Publishing Services (CPS) (2011) 151–160
29. Karol, S., Heinzerling, M., Heidenreich, F., Aßmann, U.: Using feature models for creating families of documents. In: Proceedings of the 10th ACM symposium on Document engineering. DocEng '10, New York, ACM (2010) 259–262
30. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, PA (1990)