submitted to: ESEC/FSE'97 Workshop on Object-Oriented Reengineering Zürich, Friday September 26, 1997

Rewriting poor Design Patterns by good Design Patterns

Position Paper

Jens Jahnke, Albert Zündorf
University of Paderborn, Germany
[jahnke|zuendorf]@uni-paderborn.de

1 Introduction

[GHJV94] proposed a number of *design patterns* that offer good solutions for recurring design situations. Along with the good solutions they also discuss naiv solutions to these design situations and argue why such naiv solutions should be replaced by a sound design pattern. Our position is that this exactly describes the task of OO reengineering: find patterns within a legacy OO program that show a poor solution for a (recurring) problem and replace this poor solution by a sound solution offered by the corresponding design pattern.

By replacing a poor solution with a good design pattern one gains all the advantages offered by design patterns like a well structured and well documented and easy to understand (sub)design that is flexible, extensible and easy to maintain.

To yield this one has to solve two main problems:

1. detecting occurences of (poor) implementation/design patterns in legacy programs
2. replacing the poor code/design by a good implementation/design pattern

We attack the first problem by GFRNs (Generic Fuzzy Reasoning Nets, cf. [JSZ97a]) which offer a general, graphical, very high level language for modelling and applying reverse engineering knowledge. This language enables to define and analyse fuzzy knowledge that deals with incomplete or contradicting analysis results, is able to defer expensive analysis operations until demand and to incorporate interactively (uncertain) user assumptions.

The second problem is attacked by a knowledge base of design patterns that stores the structure of design patterns on a high conceptual level together with a description of a prototypical implementation and rules for the variation and adaptation of a pattern (and its implementation).[1] In addition, this knowledge base will contain rules describing how to replace poor design patterns by corresponding good desgin patterns.

Nevertheless, we consider OO reengineering as an interactive design task. Thus, we plan to offer the proposed mechanisms as semi-automatic tool support within the FUCABA (From Uml to C++ And Back Again) design environment [Rose97]. The FUCABA environment currently supports editing of UML class diagrams with (limited) support of design patterns and transformations of UML class diagrams to C++ including standard implementations for associations and design patterns. The FUCABA environment also supports structure oriented editing of the C++ code and backward propagation of design relevant changes to the UML class diagram.

Section 2 introduces a Singleton situation as a running example. Section 3 shows how GFRNs will be used to identify the poor situation within our example and section 4 shows how this poor situation will be replaced by an instance of the Singleton design pattern.

We assume that the reader is familar with UML (cf. [UML97a]) and C++ ([Strous91]).

1. The representation of design patterns we have in mind is to some extend comparable to the one found in [FMW97].

2 The singleton example

A typical yet simple example of a poor design/implementation that may/should be fixed by an appropriate design pattern is a (number of) global variable(s) that should be replaced by a Singleton design pattern.

Figure 1 shows some code of an example university administration system containing a global variable course_program of type map<string, Course*> that provides global access to all courses via their title. As one would expect, the global variable consists of a definition of the variable on file scope within a certain cc-file and a(n extern) declaration in the corresponding h-file making it available to clients.

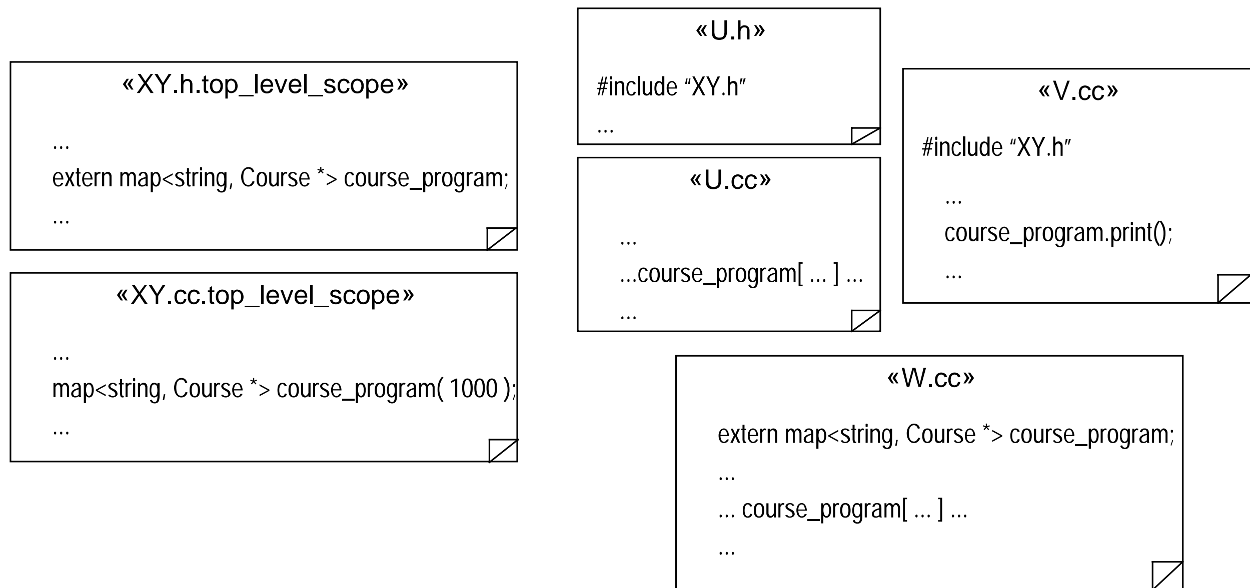

Figure 1: Global variable example code

During reverse engineering we analyse the application code and identify its basic elements and create a logical representation that abstracts from the code details and its variants. Figure 2 shows the logical representation derived from the code fragements of Figure 1 in an (extended) UML-class-diagram-like notation. The global variable is represented as an explicit object using the UML notation for class instances, i.e. an underlined object declaration within a rectangle. Its identification as a poor Singleton pattern is shown in a dashed oval, the UML notation for design patterns.

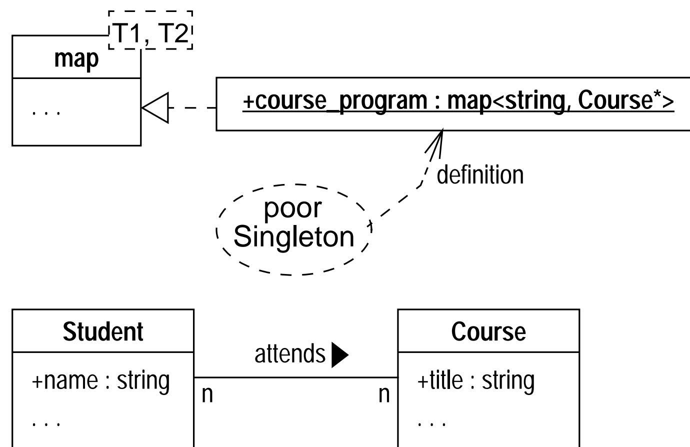

Figure 2: A poor Singleton situation

According to [GHJV94] the example shows a poor situation since everyone has full direct read/write access to this global variable from everywhere. (1) During maintenance of a big system nothing prevents a (freshman) programmer from accidently deleting a course, inserting an inconsistent course, or from destroying the whole map just by assigning it a new value via. (2) Due to the full access to all map operations it is not trivial to replace later on the hash table implementation e.g. by a sorted tree for support of ordered listings.

Such global variables should be replaced by a Singleton pattern, cf. Figure 3. The global variable is changed into a private data member within a (new) class Course_Program. (This private data member closely corresponds to a qualified association shown in grey/green color.) The private access to the constructor prevents clients from creating instances accidently. The provided static[2], public instance method (1) creates the sole instance on demand and (2) provides easy access to this sole instance from everywhere, e.g.: c = (* Course_Program::instance())[“Reengineering”].

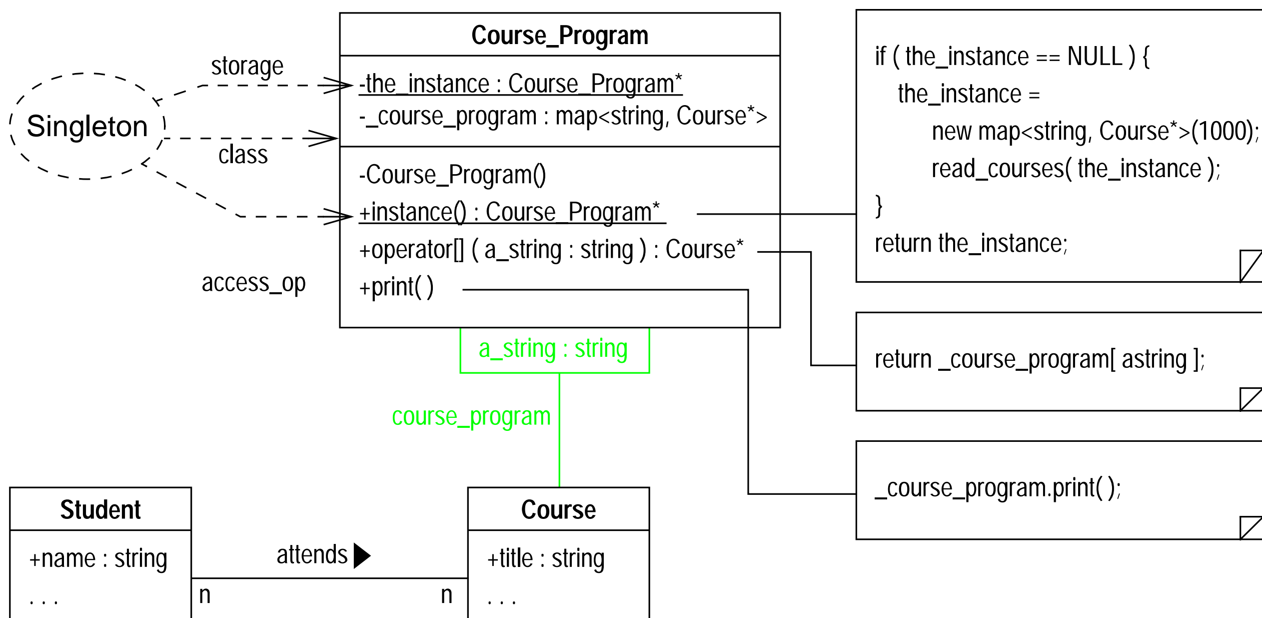

Figure 3: A sound Singleton pattern instance

For all methods that have been used directly on the global variable, class Course_Program provides encapsulating methods, that by dedault call the corresponding method on the private data member. This enables later extension e.g. by consistency checks. In addition, this shows the cutout of methods of class map that is actually used for course_program within the application. Encapsulation of this cutout facilitates to switch to other implementations. Making the used methods visible helps to identify potential misusage, e.g. accidental writes.

Replacing the global variable by a Singleton pattern includes replacing all applied occurences like the direct method invocations by invocations of appropriate methods of the new Singleton class. Obsolete extern declarations or includes should be removed and include statements for the (h-file of the) new Singleton class have to be inserted. Altogether this enhances the design and implementation of the legacy system utilizing the advantages of the introduced (good) design pattern (hopefully) without affecting the applications execution semantics.

2. Static class members are indicated by underlining.

3 Detection of (poor) patterns

The general problem in detecting (poor) design patterns in a legacy OO program is that they seldom appear in their pure, strict form. The fairly simple example of a global variable should consist of a definition of the variable on file scope within a certain cc-file and a(n extern) declaration in the corresponding h-file making it available to clients, cf. Figure 1. A slight variation is a client unit that contains the (extern) declaration itself, instead of including the corresponding h-file. Another case is a static class member variable with public access rights. There may exist also inconsistent situations, e.g. a public global variable that is used locally only.

Figure 4 shows a cutout of a GFRN (Generic Fuzzy Reasoning Net) modelling simple rules for the identification of a poor Singleton pattern. A GFRN consists of states (grey/green ovals) representing certain pieces of knowledge and fuzzy implications (grey/green rectangles) representing knowledge inference rules. For example, implication i2 represents the fuzzy rule that an item declared within a class and with public: access is globally visible (+ ...) with a certainty of 1.0. Negation is indicated by solid arrowheads, e.g. rule i1 says that a file-scope item wich is NOT declared static is a global item, too. Rule i3 concludes that an item v that is globally visible and represents a variable (... : T) and that has storage class static (...) represents a global variable with a certainty of 0.8. Together with some other facts rule i4 identifies an instance of a poor Singleton pattern.

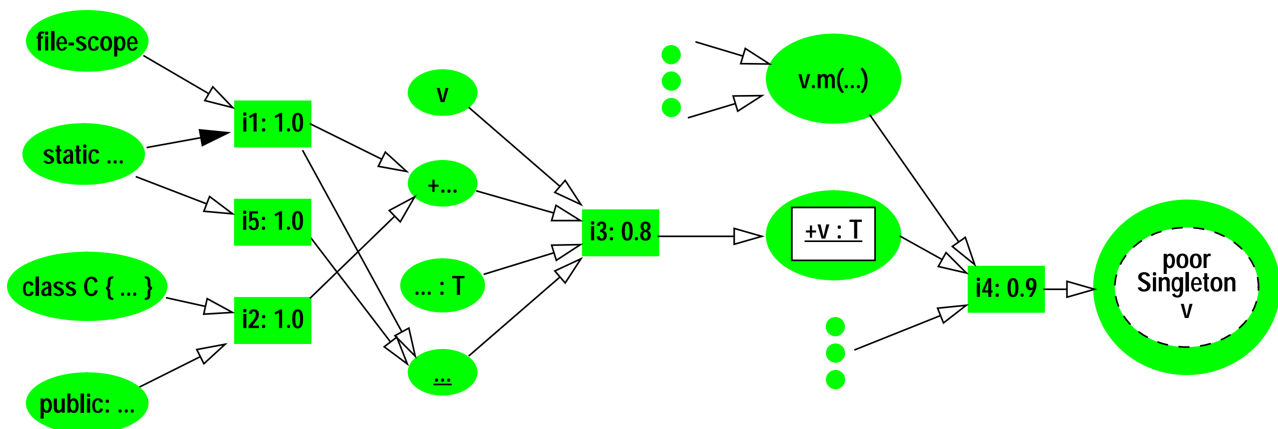

Figure 4: Reverse engineering knowledgebase for poor Singleton patterns

Due to our experiences in analysing relational databases, during reverse engineering one has to deal with inconsistent and incomplete knowledge and with certain very resource demanding analysis operations. Since we consider reverse engineering as an interactive desgin task and plan only a semi-automatic tool support we have to deal with uncertain user assumptions, too. GFRNs allow us to deal with these problems using fuzzy inference mechanisms and demand driven analysis.

Due to limitted space we can not show the full power of this mechanism here, one may have a look in [JSZ97a]. To give an impression how the GFRN of Figure 4 is used for the identification of a poor Singleton, we could start the inference with a user assumption that the variable course_program represents a poor Singleton. To validate this assumption, our inference mechanism tries to instantiate the incomming implications (i4) together with their prerequisites. Thus a fuzzy place for a global variable and at least one applied occurrence[3] is created where the formal parameter v is already bound to course_program. Further analysis checks, whether course_program is actually a variable (and binds parameter T to map<string, Course*>) with public access rights and storage class static. If one of these

3. The inference mechanism searches for all applied occurences of course_program. It generates a fuzzy place for each occurrence where the formal parameters v and m are bound appropriate each time.

facts can not be proved the fuzzy belief of the “poor Singleton” assumption will evaluate to 0, rejecting the interactive hint.

Basic facts (like course_program represents a variable) are derived via source code analysis tools and get a fuzzy belief of 1. A fuzzy implication combines the fuzzy beliefs of its input places with its own certainty using a fuzzy AND (i.e. a minimum function). The fuzzy belief of derived facts (like course_program : map<string, Course*>) is computed by a fuzzy OR (i.e. the maximum) of the fuzzy beliefs computed by its incomming implications.[4] This shows how GFRNs handle uncertain and inconsistent facts and how expensive source code analysis is triggered on demand.

To summarize, we use GFRNs for the combination of basic facts (derived from the source code) to logical items on a conceptual level. These logical items will serve as input for the pattern rewriting process.

4 Pattern rewriting

Beyond detecting poor design situations we plan tool support for the replacement of poor patterns by their corresponding good patterns. Therefore, we need an extensible pattern knowledge base that describes not only the available good patterns but also the corresponding poor patterns and how poor patterns (and its various parts) are to be replaced by (the parts of) good patterns. To set up this knowledge base we again use a very high level, graphical specification language that provides a formal yet intuitive pattern description. The idea is to specify the poor pattern and the corresponding good replacements in an extended UML-class-diagram-like notation.

Figure 5 shows how pattern rewrite rules for the Singleton example look like. The LHS (Left-Hand-Side) of rule 1 refers to elements of the GFRN of Figure 4. It contains a global variable v : T (with formal parameters v (variable name) and T (variable Type) that are bound within the analysis phase. In addition the global variable has been classified as a “poor Singleton” (with a sufficient certainty). By applying rule 1, an occurence of its LHS is replaced by a copy of its RHS (Right-Hand-Side). Thus, the global variable is deleted and an new Singleton Class with the same name is inserted instead. This new class contains the former variable as a private data member _v[5]. In addition it contains static data members for holding the one sole instance of the new Singleton class and providing access to this sole instance.

Rule 1.1 is a subrule of rule 1. The “*” indicates that an application of rule 1 triggers the application of rule 1.1 as often as possible. The LHS of rule 1.1 refers to elements of rule 1 (shown in grey/green color) and to (new) elements of the GFRN. Rule 1.1 looks for methods invocations on v (v.m(...)) where the corresponding method is not yet incorporated in the Singleton class (a crossed out element says “not yet existing”). For each such occurence (i.e. for each used method) rule 1.1 creates the corresponding member function. In addition, rule 1.1 creates a default implementation that forwards the method call to the private data member _v.

Rule 1.1.1 is a subrule of rule 1.1. Again, the “*” indicates that each application of rule 1.1 triggers the application of rule 1.1.1 as often as possible. Thus, rule 1.1.1 replaces all direct invocations of a certain method by calls to the corresponding new Singleton member function where the Singleton object is retrieved via v::instance().

4. See [JSZ97a] for the handling of negations.

5. The _«+»v indicates that the name of the private data member is computed as the concatenation of an _ and the name bound to v.

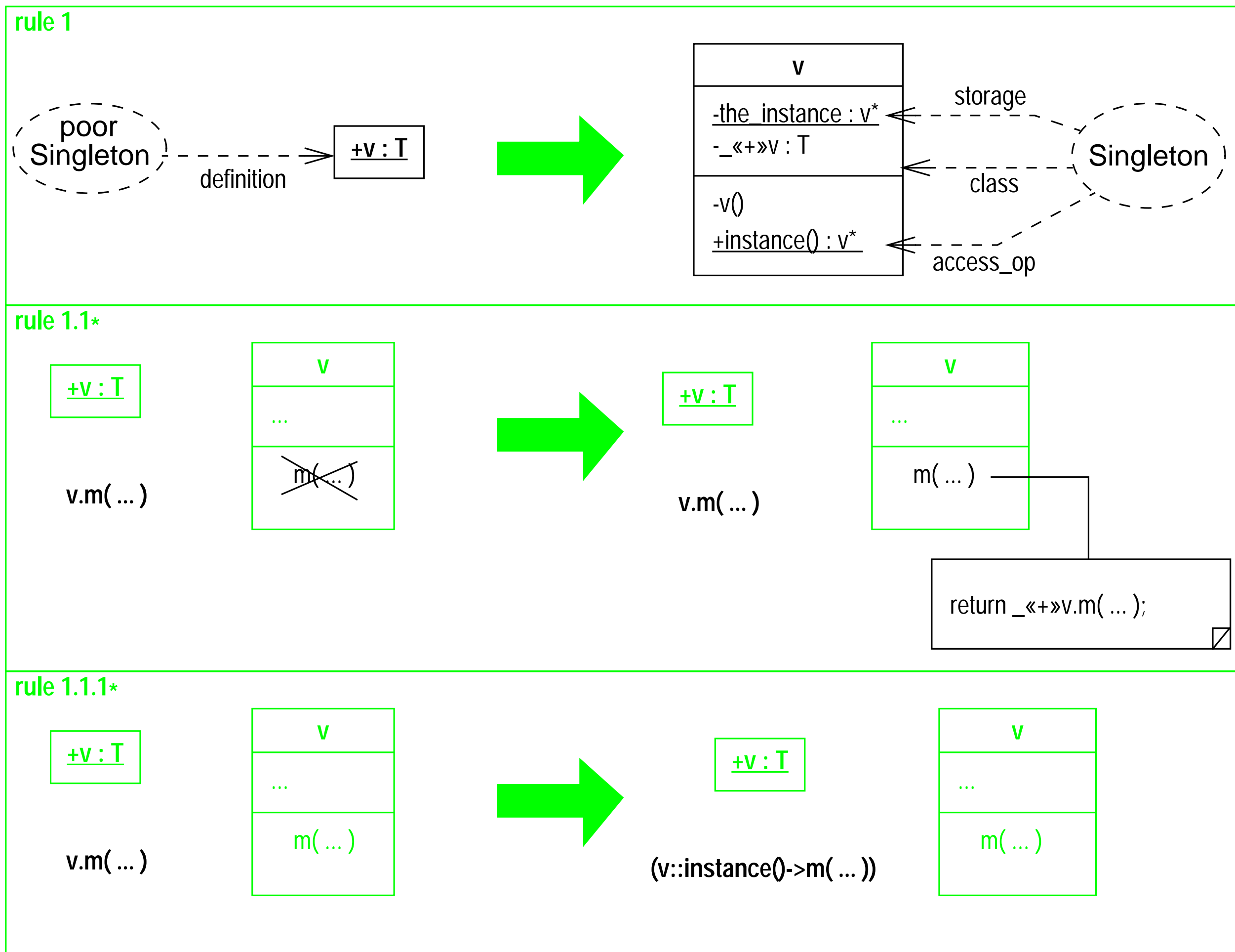

Figure 5: Rewrite rules for the Singleton pattern

Note that these rewrite rules are defined in terms of high level logical concepts. Thus introducing a new class actually results in the creation of the corresponding h- and cc-file containing the class definition as well as the usual #ifndef ... #define ... #endif directives ensuring single inclusion as well as standard constructor and destructor declarations with default implementations as well as other default elements of a C++ class implementation like a header comment template. Creating the Singleton access operation not only declares this function as a public, static member function in the h-file but also provides its default implementation in the cc-file. Removing a method invocation v.m automatically removes no longer used external declarations or include directives, too. Adding the logical concept (v::instance()->m) also adds necessary include directives, etc.. To summarize, each logical element of the pattern rewrite rules represents a rather complex analysis, deletion or insertion of a design level concept that may cause complex text analysis and change operations on many files within the underlying implementation code.

In general, there are multiple ways to fix a certain design problem. In our approach this is reflected by multiple rule sets and rule variants that apply to the same poor pattern. The different rule sets represent alternative choices for good patterns or alternative implementations of one pattern. The FUCABA environment will be able to identify applicable rewrite rules and to execute rules choosen by the reengineer.

5 Conclusions

This paper proposes a semi-automatic tool support for reengineering OO legacy systems by rewriting poor design patterns by good design patterns. The code analysis and reverse engineering phase is supported by GFRNs (Generic Fuzzy Reasoning Nets) offering a general means for dealing with uncertain, inconsistent and incomplete reengineering knowledge. The forward engineering task is supported by a knowledge base of design pattern rewrite rules that specify on a design level how poor code should be replaced by a well designed implementation.

The reverse and the forward engineering tasks are attacked using very high level, graphical specification languages. The authors have a strong background in the definition and implementation of such graphical specification languages. Jens Jahnke is developing and implementing the GFRNs as a general means for complex (program) analysis and reverse engineering tasks as a major part of his PhD thesis. Albert Zündorf partly developed and fully implemented a general purpose pattern rewrite language called PROGRES [Schürr91] during his PhD thesis [Zündorf96]. Using this experiences he is going to realize the design pattern rewriting approach as part of his postdoctoral qualification.

References

[AM97] M. Aksit, S. Matsuoka (Eds.): ECOOP’97 - Object-Oriented Programming; Proc. 11th European Conf. Jyväskylä, Finnland, June 1997, LNCS1241 Springer (1997)

[FMW97] G. Florijn, M. Meijers, P. Winsen: Tool Support for Object-Oriented Pattern; [in AM97] pp. 472-495 (1997)

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns (Elements of Reusable Object-Oriented Software); Addison Wesley, ISBN 0-201-63361-2 (1994)

[JSZ97a] J.-H. Jahnke, W. Schäfer, and A. Zündorf: Generic Fuzzy Reasoning Nets as a basis for reverse engineering relational database applications; to appear in: Proc. of ESEC’97 (to appear)

[Rose97] J. Rose: An Integrated Design Environment for UML and C++; Master Thesis University of Paderborn (In German) (1997)

[Schürr91] A. Schürr: Operational Specification with PROgrammed GraphREwritingSystems; PhD Thesis (in German), RWTH Aachen 1991, Deutscher Universitäts Verlag, Vieweg, ISBN 3-8244-2021-X (1991)

[Strous91] B. Stroustrup: The C++ Programming Language; Addison-Wesley, New York (1991)

[UML97a] The UML Notation Guide; Technical Report, Version 1.0, Rational Software, Santa Clara, http://www.rational.com (1997)

[Zündorf96] A. Zündorf: An Development Environment for PROgrammed GraphREwritingSystems - Specification, Implementation and Application; PhD Thesis (in German), RWTH Aachen 1995; Deutscher Universitäts Verlag, Vieweg, ISBN 3-8244-2075-9 (1996)