
Chapter 9 Domain Engineering Method for Reusable Algorithmic Libraries (DEMRAL)¹⁴⁸

9.1 What Is DEMRAL?

DEMRAL is a Domain Engineering method for developing algorithmic libraries, e.g. numerical libraries, container libraries, image processing libraries, image recognition libraries, speech recognition libraries, graph computation libraries, etc. The characteristics of algorithmic libraries include the following:

- main concepts are adequately captured as abstract data types (ADTs) and algorithms that operate on the ADTs;
- the ADTs often have container-like properties, e.g. matrices, images, graphs, etc.;
- there is usually a well developed underlying mathematical theory, e.g. linear algebra [GL96], image algebra [RW96], graph theory [BM76];
- the ADTs and the algorithms usually come in large varieties. For example, there are many different kinds of matrices (sparse or dense, diagonal, square, symmetric, and band matrices, etc.) and many specialized versions of different matrix factorization algorithms (e.g. the different specializations of the general LU include LDL^T and Cholesky factorizations with specialized versions for different matrix shapes and with and without pivoting);

DEMRAL supports the following library design goals:

- providing the client with a high-level, intentional library interface;
 - the client code specifies problems in terms of high-level domain concepts;
 - the interface supports large numbers of concept variants in an effective way;
 - the client code is able to specify problems at the most appropriate level of detail (of course, which level is most appropriate depends on the problem, the client, and other contextual issues);
- achieving high efficiency in terms of execution time and memory consumption;
 - the large number of variants should not have any negative effect on the efficiency;
 - possibilities of optimizations should be analyzed and useful optimizations should be implemented;
 - unused functionality should be removed and, whenever possible, static binding should be used;
- achieving high quality of library code;
 - high adaptability and extendibility;
 - minimal code duplication;
 - minimal code tangling;

DEMRAL has been developed as a specialization of the ODM method (see Section 3.7.2), while applying ODM in the development of the matrix computation library described in Chapter 10. DEMRAL combines ideas from several areas including

- Domain Engineering (Chapter 3),
- Generators and Metaprogramming (Chapter 6),
- Aspect-Oriented Programming (Chapter 7), and
- Object-Oriented Software Development.

DEMRAL is an ongoing effort. We are currently refining the method based on our experience with analyzing and implementing matrix algorithms. We still need to test DEMRAL on other domains.

This chapter gives a high-level overview of the method and explains its basic concepts. Chapter 10 contains a comprehensive case study of applying DEMRAL in the development of a matrix computation library.

9.2 Outline of DEMRAL

The major activities of DEMRAL are shown in Figure 133. Of course, the development process is an iterative and incremental one.¹⁴⁹ During development, the various activities may be scheduled and rescheduled in arbitrary order. For example, identification of key concepts (1.2.1.) and feature modeling (1.2.2.) usually require many iterations. Also, the domain definition often needs to be

revised based on insights from Domain Modeling, Domain Design, and Domain Implementation. Similarly, feature models are usually refined and revised in Domain Design and Implementation. Furthermore, activities may be revisited because of external changes, e.g. changes of the stakeholder goals, environmental changes, etc. We encourage prototyping at any time since it represents an excellent tool for gaining a better understanding of new ideas and for evaluating alternative solutions.

Thus, the process outlined in Figure 133 should be viewed as a default process and a starting point for the user, who should customize it to suite his needs. However, we use this default outline to structure the documentation of the matrix computations case study in Chapter 10.

Reenskaug et al. give an excellent characterization of the role of process outlines [Ree96]:

“Documentation is by its nature linear, and must be strictly structured. Software development processes are by their nature creative and exploratory, and cannot be forced into the straightjacket of a fixed sequence of steps. In an insightful article, Parnas et al. state that many have sought a software process that allows a program to be derived systematically from a precise statement of requirements [PC86]. Their paper proposes that although we will not succeed in designing a real product that way, we can produce documentation that makes it appear as if the software was designed by such a process.

The sequences of steps we describe in the following sections and in the rest of the book are therefore to be construed as default work processes and suggested documentation structures. We also believe that you will have to develop your own preferred sequence of steps, but you may want to take the steps proposed here as a starting point.”

- | |
|---|
| <ol style="list-style-type: none"> 1. Domain Analysis <ol style="list-style-type: none"> 1.1. Domain Definition <ol style="list-style-type: none"> 1.1.1. Goal and Stakeholder Analysis 1.1.2. Domain Scoping and Context Analysis <ol style="list-style-type: none"> 1.1.2.1. Analysis of application areas and existing systems (i.e. exemplars) 1.1.2.2. Identification of domain features 1.1.2.3. Identification of relationships to other domains 1.2. Domain Modeling <ol style="list-style-type: none"> 1.2.1. Identification of key concepts 1.2.2. Feature modeling of the key concepts (i.e. identification of commonalities, variabilities, and feature dependencies/interactions) 2. Domain Design <ol style="list-style-type: none"> 2.2. Identification of the overall implementation architecture 2.1. Identification and specification of domain-specific languages 3. Domain Implementation (implementation of the domain-specific languages, language translators, and implementation components) |
|---|

Figure 133 *Outline of DEMRAL*

The DEMRAL activities have been derived from the ODM phases and tasks (see Figure 15 and Table 5 in Section 3.7.2.1). However, there are several differences:

- *Different divisions into top-level activities:* DEMRAL subscribes to the more widely accepted division of Domain Engineering into Domain Analysis, Domain Design, and Domain Implementation. This division does not represent any substantial difference to ODM. The ODM phases (see Figure 15) are easily mapped to these top-level activities: Plan Domain and Model Domain correspond to Domain Analysis. Scope Asset Base and Architect Asset Base correspond to Domain Design. Finally, Implement Asset Base corresponds to Domain Implementation.

- *Stronger focus on technical issues:* A unique feature of ODM is its focus on organizational issues. The description of DEMRAL, on the other hand, is more focused on technical issues. Of course, whenever appropriate, DEMRAL can be extended with the organizational tasks and workproducts of ODM.
- *Only a subset of ODM tasks and workproducts:* As a specialization of a very general Domain Engineering method, DEMRAL covers only a subset of the ODM tasks and workproducts. Additionally, the DEMRAL process outline is less detailed than the ODM process outline. In a sense, DEMRAL can be seen as a “lightweight” specialization of ODM.

The special features of DEMRAL as a specialization of ODM include

- focus on the two concept categories: ADTs and algorithms;
- specialized feature starter sets for ADTs and algorithms;
- application of feature diagrams for feature modeling (ODM does not prescribe any particular feature modeling notation);
- focus on the development of DSLs.

In the remaining sections of this chapter, we describe each of the activities of DEMRAL. A comprehensive case study of applying DEMRAL to the domain of matrix computations is presented in Chapter 10.

9.3 Domain Analysis

Domain Analysis involves two main activities: Domain Definition and Domain Modeling. They are described in the following two sections.

9.3.1 Domain Definition

The first activity of Domain Definition is the identification of goals and stakeholders. The complexity of this activity depends on the size and the context of the project. We will not further elaborate on this activity. We assume that the result of this activity are cross-checked, prioritized lists of goals and stakeholders.

The next activity is to determine the scope and characterize the contents of the domain by defining its domain features. Two important sources of domain features are

- the analysis of the application areas of the systems in the domain and
- the analysis of the existing exemplar systems.

For example, if our goal is to define the domain of matrix computation libraries, we need to analyze the application areas of matrix computations and the features of existing matrix computation libraries. The results of these analyses are shown in Table 13 through Table 16. Please note that the format of the application areas tables (i.e. Table 13 and Table 14) and the existing libraries tables (i.e. Table 15 and Table 16) is similar. The conclusion of both analyses is that the main features of matrix computation libraries are different types of matrices and different types of computations they implement.

The results of the analysis of the application areas and the exemplar systems are summarized in a *domain feature diagram*. The domain feature diagram for the domain of matrix computation libraries is shown in Figure 137. This diagram describes which features *are* part and which *can* be part of a matrix computation library. For example, band matrices are optional, but at least dense

or sparse matrices have to be implemented. A matrix computation library can also implement both dense and sparse matrices. This is indicated in the diagram by the fact that dense and sparse matrices have been shown as or-features. Please note that alternative *concept features*, e.g. dense or sparse of the concept *matrix*, emerge in the *domain feature* diagram as or-features.

We found it useful to annotate the domain features with priorities. There are at least three important factors influencing the priority of a domain feature:

- typicality rate of the domain feature in the analyzed application areas,
- typicality rate of the domain feature in the analyzed exemplar systems, and
- importance of the domain feature according to the stakeholder goals.

The priorities are assigned on a rather informal basis. However, they are still very useful. They indicate the importance of the various parts of a domain and will help to decide which parts of the domain will be implemented first. Of course, the priorities may have to be adjusted as the goals evolve and more knowledge about the domain becomes available over time. The domain feature diagram represents a definition of a domain from the probabilistic viewpoint (see Section 2.2.3). It is a concise and convenient style of defining a domain.

Right from the beginning of Domain Analysis, it is essential to establish a *domain dictionary* (see e.g. Section 10.4) and a register of sources of *domain knowledge* (see e.g. Section 10.1.1.2.2). As the analysis progresses, both the dictionary and the register need to be updated.

*Domain dictionary
and domain
knowledge*

Finally, we analyze related domains, such as *analogy* or *support domains*. As you remember from Section 3.6.3, an analogy domain has significant similarities to the domain being analyzed and thus may provide some useful insights about the latter domain. A support domain, on the other hand, may be used to express some aspects of the domain being analyzed. Examples of both types of related domains can be found in Section 10.1.1.2.6.

*Analogy or support
domains*

9.3.2 Domain Modeling

Domain Modeling involves two main activities: *identification of key concepts* and *feature modeling of the key concepts*. We describe these activities in the following two sections.

9.3.2.1 Identification of Key Concepts

By definition, DEMRAL focuses on domains whose main categories of concepts are ADTs and algorithms. Identifying the key ADTs is usually quite simple, e.g. the key ADTs in matrix computation libraries are matrices and vectors and the key ADTs in image processing libraries are various kinds of images.

An ADT defines a whole family of data types. Thus, a matrix ADT defines a family of matrices (e.g. sparse, dense, diagonal, square, symmetric, etc.). Similarly, we usually have whole families of algorithms operating on the ADTs. For example, matrix computation libraries may include factorization algorithms and algorithms for solving eigenvalue problems. Furthermore, each general version of a factorization algorithm may be specialized for matrices of different properties, e.g. there is over a dozen of important specializations of the general LU factorization (see Section 10.1.2.2.2).

We make a distinction between *basic ADT operations* and the *algorithm families* which access the ADTs through accessing operations and the basic operations. Examples of basic operations in matrix computations are matrix addition, subtraction, and multiplication. We analyze basic operations together with the ADTs since they all define a cohesive *kernel algebra*. We usually implement the kernel algebra in one *component* which is separate from the more complex

*Basic ADT
operations and the
algorithm families*

algorithm families. For example, a matrix component would include a matrix type and a vector type and the basic operations on matrices and vectors.

It is important to note that if we model types using OO classes, we usually do not define the basic operations directly in the class interfaces, but rather as free-standing operators or operator templates.¹⁵⁰ This, of course, is only possible if the programming language we use supports free-standing operators (e.g. C++). The class interfaces should include a minimal set of necessary methods, e.g. accessing methods for accessing directly stored state or abstract (i.e. computed) state. This way, we avoid “fat class interfaces” and improve modularity since we can define families of operators in separate modules. Furthermore, it is easier to add new operators. Another reason for defining operators outside the class definitions is that they often cannot be thought of as a conceptual part of just one class. For example, in the expression $M*V$, where M is a matrix and V is a vector, the multiplication operation $*$ is equally part of the matrix interface and the vector interface. If dynamic binding is required, $*$ is best implemented as a dynamic multi-method¹⁵¹, otherwise we can implement it as a free-standing operator or as a (possibly specialized) operator template. Furthermore, if expression optimizations are required, the operators are best implemented as *expression templates* (see Section 8.8). Indeed, for our matrix computation library, we will implement matrix operations using expression templates (Section 10.3.1.7).

The more complex algorithms (e.g. solving systems of linear equations) are often more appropriately implemented as classes rather than procedures or functions (see e.g. [Wal97]). This allows us to organize them into family hierarchies. The algorithms usually call both basic operations and ADT accessing methods.

9.3.2.2 Feature Modeling

The purpose of feature modeling is to develop feature models of the concepts in the domain. Feature models define the common and variable features of concept instances and the dependencies between the features.

We already discussed how to perform feature modeling in Chapter 5. The only addition of DEMRAL to what we said there are the feature starter sets for ADTs and algorithms. These are listed in Sections 9.3.2.2.1 and 9.3.2.2.2.

9.3.2.2.1 Feature Starter Set for ADTs

The following is the feature starter set for ADTs:

- *Attributes*: Attributes are named properties of ADT instances, such as the number of rows in a matrix or the length of a vector. Important features of an attribute are its type and whether it is mutable or not (i.e. if its possible to change its value). Other features concern the implementation of an attribute, e.g. whether the attribute is stored directly or computed, whether its value is cached or not, whether it is *owned* by the ADT or not (see [Eis96]). If an attribute is owned by the ADT, the ADT is responsible for creating and destroying the values of the attribute. Also, accessing a value owned by an ADT usually involves copying the value. Each of the features of an attribute could be parameterized.
- *Data structures*: The ADT may be implemented on top of some complex data structures. This is especially the case for container-like ADTs such as matrices or images. We will discuss this aspect later in more detail.
- *Operations*: Examples of operations are accessing operations and kernel algebra operations (i.e. basic operations used by more complex algorithms). Other operations may be added during the analysis of algorithms. In addition to operation signatures (i.e. operation name, operands and operand types), we need to analyze various possible implementations of

operations. In particular, possible optimizations need to be documented. For example, matrix operations may be optimized with respect to the shape of the operands. Also the optimization of matrix expressions based on the expression structure is possible (e.g. loop fusing). Binding mode (e.g. static or dynamic) and binding time (e.g. compile time or runtime) are other features of an operation (see Section 5.4.4.3). Binding mode and binding time of an operation can be parameterized (see e.g. Section 7.10).

- *Error detection, response, and handling*: Error detection is often performed in the form of pre-, intra-, and post-condition¹⁵² and invariant checking. What is an error and what is not may depend on the usage context of an ADT. Thus, we may want to parameterize error detection (also referred to as *error checking*). In certain contexts, it may be also appropriate to switch off checking for certain error condition. For example, if the client of a container is guaranteed to access elements using only valid indices, no bounds checking is necessary. Once an error condition has been detected, various responses to the error condition are possible: We can throw an exception, abort the program, issue an error report [Eis95]. Finally, on the client side, we need to handle exceptions by performing appropriate actions. An important aspect of error response and handling is *exception safety*. We say that an ADT is exception safe if exceptions thrown by the ADT code or the client code do not leave the ADT in an undefined, broken state. Interestingly, this important aspect has been addressed in the Standard Template Library (STL) only in its final standardization phase.
- *Memory management*: By memory management we mean approaches to allocating and relinquishing memory and various approaches to managing virtual memory. We can allocate memory on the stack or on the heap using standard mechanisms available in most languages. We can also manage memory ourselves by allocating large chunks of memory (so-called *memory pool*) at once and allocating objects within this customarily managed memory. We can also use automatic memory management approaches, e.g. reference counting or some more sophisticated garbage collection approach (see e.g. [Wil92]). In a multithreading environment, it may be useful to manage thread-specific memory, i.e. per-thread memory where a thread can allocate its own objects not shared with other threads. Other kinds of memory are memory shared among a number of processes and persistent store. Memory management interacts with other features, e.g. exception safety: One aspect of exception safety is making sure that exceptions do not cause memory leaks. The memory allocation aspect of container elements in STL is parameterized in the form of *memory allocators* (see [KL98]), which can be passed to a container as a template parameter. Paging and cache behaviors often need to be tuned to the requirements of an application (e.g. blocking in matrix algorithms; see Section 10.1.2.2.1.3.1). An important aspect of memory management in databases is location control and clustering, i.e. where and how the data is stored.
- *Synchronization*: If we want to use an ADT in a multithreaded environment, we have to synchronize the access to shared data. This is usually done by providing appropriate synchronization code (see Chapter 7). Synchronization variability may include not only different synchronization constraints but also different implementation strategies. For example, synchronization can be implemented at different levels: the interface level of an ADT or the internal data level. The interface level locking is, as a rule, less complex than the data level locking. On the other hand, data level locking allows more concurrency. For this reason, data level locking is usually used in large collections, e.g. in databases. If we also want to use an ADT in a sequential environment, it should be possible to leave out its synchronization code entirely, or, in some cases, replace it by error checking code. Thus, there is also an interaction between synchronization and error checking. We already saw an example of this interaction in Section 7.4.4.
- *Persistency*: Some application may require the ability to store an ADT on disk (e.g. in a file or in a database). In such cases, we need to provide mechanisms for storing appropriate parts

of the state of an ADT as well as for restoring them. This aspect is closely related to memory management.

- *Perspectives and subjectivity*: Different stakeholders and different client programs usually have different requirements on an ADT. Thus, we may consider organizing ADTs into subjects based on the different perspectives of the stakeholders or client programs on the ADTs. Some of the perspectives may delineate different parts of an ADT according to their service (e.g. printing, storing, etc.) and other perspectives may require different realizations of the same service, e.g. different attributes and operations, attribute and operation implementations, etc. In this case, we might want to develop a model of the relevant subjective perspectives and define ADT features that correspond to these perspectives.

If the ADT has a container-like character (e.g. matrix or image), we also should consider the following aspects:

- *Element type*: What is the type of the elements managed by the ADT?
- *Indexing*: Are the elements to be accessed through an index (e.g. integral index, symbolic key, or some other, user-defined key)?
- *Structure*: How are the elements stored? What data structures are used? For example, the structure of a matrix has a number of subfeatures such as entry type, shape, format, and representation (see Section 10.1.2.2.1.3).

An ADT may have a number of different interfaces. For example, a matrix will have a base interface including the basic matrix operations as well as a configuration interface, which allows us to select different storage formats, error checking strategies, etc. The configuration interface may be needed at different times, e.g. compile time or runtime. Also, the base interface may be organized into subinterfaces and be configurable (e.g. in order to model subjectivity).

Some features in the feature starter set are aspects in the AOP sense, e.g. synchronization or memory management.

Of course, the starter set should be extended as soon as new relevant features are identified.

9.3.2.2.2 *Feature Starter Set for Algorithms*

The feature starter set for algorithms includes the following feature categories:

- *Computational aspect*: This is the main aspect of an algorithm: the abstract, text-book formulation of the algorithm without the more involved implementation issues. We may specify this aspect using pseudocode. Furthermore, we have to investigate the relationships between algorithms, e.g. specialization and use, and organize them into families. It may also be useful to classify them according to the general algorithm categories such as search algorithms, greedy algorithms, divide-and-conquer algorithms, etc. (see Figure 100, Section 6.4.4). For example, iterative methods in matrix computations are search algorithms.
- *Data access*: Algorithms access ADTs through accessing operations and basic operations. Careful design of the basic operations is crucial in order to achieve both flexibility and good performance. For example, algorithms benefit from optimizations of basic operations. We can use different techniques in order to minimize the coupling between algorithms and data structures, e.g. iterators [GHJV95, MS96] and data access templates (or data accessors) [KW97].

- *Optimizations*: There are various opportunities for domain-specific optimizations, e.g. optimization based on the known structure of the data, caching, in-place computation (i.e. storing result data in the argument data to avoid copying), loop restructuring, algebraic optimizations, etc.
- *Error detection, response, and handling*: We discussed this aspect in the previous section.
- *Memory management*: Algorithms may also make direct calls to memory management services for tuning purposes.

There are also more specialized domain-specific features, e.g. pivoting strategies in matrix computations (see Section 10.1.2.2.2).

An aspect we did not discuss is parallelization, which is relevant in high-performance areas such as scientific computing. Parallelization is a very advanced topic and we refer the interested reader to [GO93].

9.4 Domain Design

The purpose of Domain Design in DEMRAL is to develop a library architecture consisting of a decomposition into packages and the specifications of the user DSLs. The DSL specifications include both abstract syntax and implementation specifications in a form which can serve as a basis for their implementation using some appropriate language extension technology. Domain Design builds on the results of Domain Modeling, i.e. feature models of different aspects of ADTs and algorithm families.

Since DSLs are central to Domain Design in DEMRAL, we first review the advantages of DSLs, the sources of DSLs, and implementation technologies for DSLs in Section 9.4.1. Then we describe the Domain Design activities in DEMRAL in Section 9.4.2.

9.4.1 Domain-Specific Languages Revisited

A domain-specific language (DSL) is a specialized, problem-oriented language. Similarly as in the case of domains, some DSLs are more general modeling DSLs (e.g. a DSL for expressing synchronization)¹⁵³ and others are more specialized, application-oriented DSLs (e.g. DSL for defining financial products).

Compared to conventional libraries, DSLs have a number of advantages. We already summarized them in Section 7.6.1, but let us restate some main points (see Table 12 for other advantages):

- *Intentional representation*: DSLs are designed to allow a direct, well-localized expression of requirements without obscure language details. Also, they enforce the inclusion of all relevant design information which would otherwise be lost if we used a general-purpose language. Thus, programs written in DSLs are easier to analyze, understand, modify, extend, reuse, and maintain.
- *Error checking*: DSLs enable error checking based on domain knowledge.
- *Optimizations*: DSLs allow optimizations based on domain knowledge. Such optimizations are usually much more effective than low-level optimization at the level of a general-purpose language (see Section 6.4.1). They are the key to being able to write cleanly designed, abstract code and still have it compiled into a high-performance executable. For example, if we implement matrix addition naively in an OO language using overloaded binary operators, the performance of such implementation will be unacceptable. Given the expression $M1+M2+M3$, the sum of the matrices $M2$ and $M3$ will be computed first and the intermediate result will be passed as the second argument to the first plus operation in the expression.

Unfortunately, intermediate results can cause a significant overhead, especially if the matrices are large. Therefore, such an expression is often manually implemented as a pair of nested loops iterating through rows and columns and adding the corresponding elements of all matrices at once, which does not require any intermediate results and an extra pair of loops. The manual implementation, despite its better performance, is much more difficult to maintain than the original expression $M1+M2+M3$. Domain specific optimizations offer a solution to this dilemma. In our case, we have to extend the compilation process with the matrix expression optimization computing the efficient implementation from the abstract expression automatically. Thus, we can write abstract expressions and get maximum performance at the same time.

The conclusion of Chapter 7 was that aspectual decomposition requires specialized language constructs for dealing with crosscutting. Also, we concluded that language extensions are well suited for capturing aspects in an intentional way.

Furthermore, we saw in Section 7.6.3 that the distinction between languages and language extensions disappears if we subscribe to the idea of modular language extensions. In this case, instead of using some (possibly extended) fixed language, we rather use configurations of language extensions. DSLs as language extensions have three important advantages (also see Section 7.6.1):

- *Reusability*: Breaking monolithic languages into modular language extensions allows us to use them in different configurations.
- *Scalability*: We can develop a small system using a small configuration of necessary language extensions. When the system grows to encompass new aspects, we can add new language extensions addressing the new aspects to the initial configuration. Modular language extensions avoid the well-known problems of large and monolithic DSLs that are hard to evolve (see e.g. [DK98]).
- *Fast feature turnover*: Modular language extensions are a faster vehicle for distributing new language features than closed compilers. Also, modular language feature extensions have to survive based on their merits since they can be loaded and unloaded at any time. This is not the case for language features of fixed languages, in which case it is usually not possible to get rid of questionable features once they are part of a language (since there may be users depending on them).

An important issue is how to compose the language extensions. We could classify the composition mechanisms into three categories:

- *Embedding*: Small sections of code written in one language extension are “embedded” in the code written in other language extensions, e.g. embedded SQL statements in a general-purpose programming language. The boundaries between the embedded code and the surrounding code can be usually easily identified since both codes use a different style, paradigm, etc.
- *“Seamless” integration*: One language extension “naturally” extends another one. An example of such integration is the extension of Java 1.1 with *inner classes* in Java 1.2.
- *Referential integration*: Different modules are written in different language extensions. The modules reference each other at appropriate join points. A good example of referential integration is the composition of Jcore and Cool modules in AspectJ (see Section 7.5.1).

It is important to note that, from the linguistic viewpoint, conventional libraries of procedures or classes extend a programming language within its syntax since they introduce new vocabulary

(i.e. new abstractions) for describing problems at a higher level. Such extensions are adequate as long as we do not require

- syntax extensions,
- semantic extensions or modifications of language constructs,
- domain-specific optimizations,
- domain-specific error checking, and
- domain-specific type systems.

Some languages allow implementing domain-specific optimizations, error checking, and type systems without leaving the syntax of the language, e.g. C++ thanks to static metaprogramming. Other languages allow us to extend their syntax and semantics by extending the libraries defining them, e.g. Smalltalk and CLOS (see Section 7.4.7).

In general, technologies for implementing language extensions covering domain-specific optimizations, error checking, type systems, syntactic extensions, and modifications of language constructs include the following:

- *Preprocessors*: Preprocessors are popular for extending existing programming languages. They usually expand some embedded macros into the target programming language, which is referred to as the host language. The advantage of preprocessors is that they do not have to understand the host language entirely and thus their development cost can be much smaller than the cost of developing a compiler. Moreover, they can be simply deployed in front of any favorite compiler for the host language. Unfortunately, this advantage is also their disadvantage. Errors in the target source are reported by the compiler in terms of the target source and not in terms of the source given to the preprocessor. Also, debuggers for the host language do not understand the extended language. Thus, preprocessors usually do not adequately support the programmer. Furthermore, if a preprocessor does not completely understand the host language, macros cannot utilize other information contained in the source (e.g. host language constants, constant expressions, etc.) and thus many important kinds of domain-specific optimizations cannot be implemented. This problem could be solved if the preprocessor had access to the internal representation of the compiler. In an extreme case, we could imagine a pipeline of preprocessors, all storing their metainformation in one repository. Indeed, such an architecture can be viewed as a modularly extendible compiler, which we mention in the last point.
- *Languages with metaprogramming support*: Some languages have built-in language extension capabilities. Templates in C++ allow us to implement domain-specific optimizations while staying within the C++ syntax and semantics. Reflective languages such as Smalltalk or CLOS allow us to implement any kinds of extensions since their definition is accessible to the programmer as a modifiable and extendable library. However, the languages do not provide the kind of architecture for modular extensions as modularly extendible compilers or programming environments do.
- *Modularly extendible compilers and modularly extendible programming environments*: An example of a system in this category is the Intentional Programming (IP) environment described in Section 6.4.3. Language extensions in IP are implemented as extension libraries which extend editors, compilers, debuggers, etc.

The latter two technologies allow us to develop active libraries, i.e. libraries which, in addition to the runtime code, also include domain-specific optimizations, or any other kind of compiler or programming environment extensions (see Section 7.6.4).

In the following text, whenever we use the term DSL, we actually mean a domain-specific language extension.

9.4.2 Domain Design Activities

Domain Design in DEMRAL involves the following activities:

- identify packages,
- identify user DSLs,
- identify interactions between DSLs,
- scope DSLs, and
- specify DSLs.

These activities are described in following five sections.

9.4.2.1 Identify packages

We divide the library to be developed in a number of packages. Packages serve several of purposes:

- defining the high-level modules of the library which helps to minimize dependencies and improve understandability;
- assigning different packages to different developers for concurrent development;
- selective import of required packages by different applications.

The usual strategy in DEMRAL is to put each ADT and each algorithm family into a separate package. We can use the UML package diagrams in order to represent the package view on the library under development.

9.4.2.2 Identify User DSLs

User DSLs are the DSLs provided by the library to their users as Application Programmer's Interfaces (APIs). There are two important kinds of user DSLs in DEMRAL:

- Configuration DSLs* • *Configuration DSLs*: Configuration DSLs are used to configure ADTs and algorithms. We discuss configuration DSLs in Section 9.4.3.
- Expression DSLs* • *Expression DSLs*: Expression DSLs are DSLs for writing expressions involving ADTs and operations on them, e.g. matrix expressions. We discuss expression DSLs in Section 9.4.4

Other, more problem-specific DSLs are also possible, e.g. a language extension for expressing pivoting in matrix computation algorithms (see [ILG+97]).

9.4.2.3 Identify Interactions Between DSLs

Next, we need to address the following question: What kind of information has to be exchanged between the implementations of the DSLs? For example, the implementation of a matrix

expression DSL will need access to different properties of matrices (e.g. element type, shape, format, etc.), which are described by the matrix configuration DSL. This information is necessary in order to implement the operations contained in an expression by selecting optimal algorithms for the given matrix arguments. Furthermore, it is used to compute the matrix types for the intermediate results.

9.4.2.4 Scope DSLs

The features to be covered by an implementation of a DSL have to be selected from the feature models based on the priorities recorded in the feature models and the current project goals and resources.

9.4.2.5 Specify DSLs

We specify a language by specifying its syntax and semantics. At this point, we will only specify the *abstract syntax* of each DSL and leave the *concrete syntax* (also referred to as the *surface syntax*) to Domain Implementation. The difference between abstract and concrete syntax is shown in Figure 134. The abstract syntax of a language describes the structure of the abstract syntax trees used to represent programs in the compiler (or another language processor), whereas the concrete syntax describes the structure of programs displayed on the screen. Technologies such as Intentional Programming allow us to easily implement many alternative concrete syntaxes for one abstract syntax. We specify abstract syntax in the form of an abstract grammar (see e.g. Figure 134) and we use the *Backus-Naur Form* (BNF; see e.g. [Mey90]) for the concrete syntax.

*Abstract and
concrete syntax*

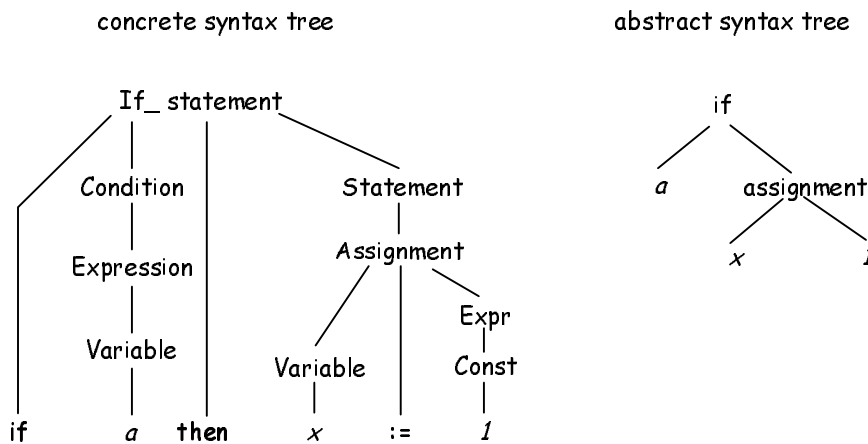


Figure 134 Concrete and abstract syntax tree for the statement *if a then x := 1* (adapted from [WM95])

The specification of the semantics of a language is a more complex task. Meyer describes five fundamental approaches to specifying semantics [Mey90]:

- *Attribute grammars*, which extends the grammar by a set of rules for computing properties of language constructs.
- *Translational semantics*, where the semantics is expressed by a translation scheme to a simpler language.
- *Operational semantics*, which specifies the semantics by providing an abstract interpreter.

- *Denotational semantics*, which associates with every programming construct a set of mathematical functions defining its meaning.
- *Axiomatic semantics*, which for a programming language defines a mathematical theory for proving properties of programs written in this language.

Each of these approaches has its advantages and disadvantages. In any case, however, formal specification of the semantics of a language is an extremely laborious enterprise.

A simple and practicable approach is to specify how to translate DSL programs into pieces and patterns of code in some appropriate programming language or pseudo code.

Finally, we have to specify how the different DSLs should be integrated (e.g. embedding, seamless integration, or referential integration).

Since the two most important categories of DSLs in DEMRAL are configuration DSLs and expression DSLs, we discuss them in Sections 9.4.3 and 9.4.4, respectively.

9.4.3 Configuration DSLs

A configuration DSL allows us to specify a concrete instance of a concept, e.g. data structure, algorithm, object, etc. Thus, it defines a family of artifacts, just as a feature model does. Indeed, we derive a configuration DSL of a concept from a feature model by tuning it to the needs of the reuser.

We usually implement a configuration using a generative component (see Section 6.2) or a configurable runtime component. The model of a generative component is shown in Figure 135. An instance specification in a configuration DSL (e.g. specification of a matrix: complex elements, lower-diagonal shape, stored in an array, and with bounds checking) is given to the generative component which assembles the concrete component instance (e.g. the concrete matrix class) from a number of *implementation components* (e.g. parameterized classes implementing element containers, bounds checkers, adapters, etc.) according to the specification. The implementation components can be connected only in certain ways. This is specified by an *implementation components configuration language (ICCL)*.

*Implementation
components
configuration
language (ICCL)*

A configurable runtime component has the same elements as the generative component. The only difference is that the finished configuration, the translator, and the implementation components are all contained in one configurable runtime component. Of course, a generative component may also generate a configurable runtime component, which implements a subset of its original configuration DSL and thus includes a subset of its implementation components and a smaller translator.

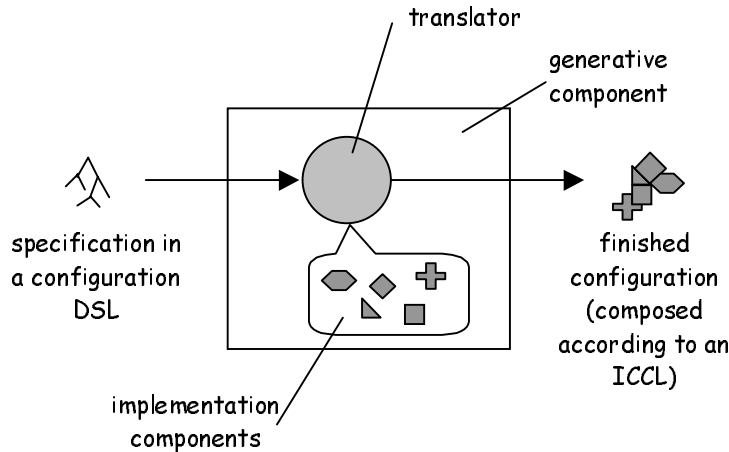


Figure 135 Generative component implementing a configuration DSL

Thus, the complete specification of a configuration DSL consists of a DSL grammar (see Section 10.2.3 and Figure 150), the specification of the ICCL (see Section 10.2.4), and the specification of how to translate statements in the configuration DSL into ICCL statements (see Section 10.2.5).

Why do we need both configuration DSLs and ICCLs? The reason is that both kinds of languages have different foci. The focus of a configuration DSL is to allow the user (or the client program) of a component to specify his needs at a level of detail that suites him best. The focus of an ICCL is to allow maximal flexibility and reusability of the implementation components. Thus, the configuration DSL describes the problem space, whereas the ICCL describes the solution space.

As stated, the configuration DSL should allow the user of a component to specify his needs at a level of detail that suites him best. He should not be forced to specify any implementation-dependent details if he needs not to. This way, we make sure that a client program does not introduce any unnecessary dependencies on the implementation details of the server component. For example, a client might just request a matrix from a generative matrix component. The matrix component should produce a matrix with some reasonable defaults, e.g. rectangular shape, real element values, dynamic row and column numbers, etc. In general, the client should be able to leave out a feature in a specification, in which case the feature should be determined by the generative component as a *direct default* or a *computed default* (i.e. a default determined based on some other specified features and other feature defaults). The client could be more specific and specify features stating some usage profile, e.g. need dense or sparse matrix or need space- or speed-optimized matrix. The next possibility would be to provide more precise specifications, e.g. the matrix density is 5% nonzero elements. Furthermore, the client should be able to specify some implementation features directly, e.g. what available storage format the matrix should use. Finally, it should be possible for the client to contribute its own implementation of some features, e.g. its own storage format. The different levels of detail for specifying a configuration are summarized in Figure 136.

Direct or computed defaults

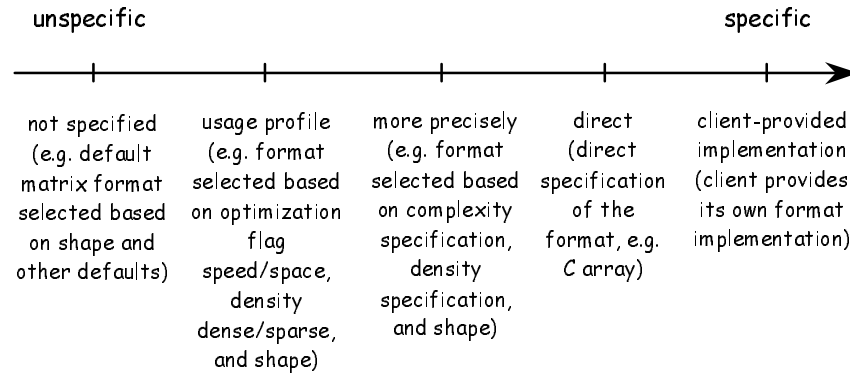


Figure 136 Different levels of detail for specifying variable features

Metainterface

The idea of the different levels of detail was inspired by the Open Implementation approach to design [KLL+97, MLMK97]. In this approach, the configuration interface of a component is referred to as its *metainterface*.

The focus of an ICCL is on the reusability and flexibility of the implementation components. This requirement may sometimes conflict with the requirements on a configuration DSL: We strive for small, atomic components that can be combined in as many ways as possible. We want to avoid any code duplication by factoring out similar code sections into small, (parameterized) components. This code duplication avoidance and the opportunities for reuse of some implementation components (e.g. containers, bounds checkers, etc.) in other servers may lead to implementation components that do not align well with the feature boundaries that the user wants to use in his or her specifications. An example of a library which only provides an ICCL is the C++ Standard Template Library (STL). The user of the STL has to configure the STL implementation components manually (i.e. he has to hardcode ICCL statements in the client code).

The use of a configuration DSL and an ICCL separates the problem space and the solution space. This separation allows a more independent evolution of the client code, which uses a configuration DSL as the configuration interface to a component, and the component implementation code, which implements the ICCL. Indeed, during the development of the matrix computation library described in Chapter 10, there were cases in which we had to modify the ICCL and these modifications had no effect on the DSL whatsoever.

The configuration DSL of a concept can be directly derived from the feature model of the concept. The approach for this derivation involves answering the following questions:

- *Which features are relevant for the DSL?* Obviously, a configuration DSL will consist of variation points and variable features only. Other features of a feature model are not relevant for the configuration DSL.
- *Are any additional, more abstract features needed?* For example, during Domain Design of the matrix computation library described in Section 10.2.3, we added the *optimization flag* with the alternative subfeatures *speed* and *space* to the original matrix feature model created during feature modeling. The optimization flag allows us to decide which matrix storage format to use based on the matrix shape and density features (e.g. a dense triangular matrix is faster in access when we store its elements in an array than in a vector since we do not have to convert the subscripts on each element access; however, using a vector requires half the storage of an array since only the nonzero half of the matrix needs to be stored in the vector).

- *Is the nesting of features optimal for the user?* It may be useful to rearrange the feature diagrams of a feature model according to the needs of the users (which may be different). Also, the target implementation technology may impose some constraints on the DSL. For example, the matrix configuration DSL described in Section 10.2.3 uses dimensions as the only kind of variation points. This way we can easily implement it using C++ class templates.
- *Which features should have direct defaults and what are these defaults?* Some features should be selected by default if not specified. For example, the *shape* of a matrix is a dimension and should have the default value *rectangular*. Other dimensions of a matrix which should have direct defaults include element type, index type, optimization flag, and error checking flag (see Table 42).
- *For which features can defaults be computed and how to compute them?* Features, for which no direct defaults were defined should be computable from the specified features and the direct defaults. For example, the storage format of a matrix can be determined based on its shape, density, and optimization flag. The computed defaults can be specified using dependency tables (see Table 24 in Section 10.2.3 and, e.g., Table 48 in Section 10.2.3.4.3).¹⁵⁴

The implementation components may be based on different component architectures. For the matrix package, we have used the GenVoca architecture described in Section 6.4.2. A GenVoca component represents a parameterized layer containing a number of mixin classes (thus, the layers are also referred to as *mixin layers* [SB98]). These layers can be configured according to a predefined grammar. Thus, a GenVoca grammar and a set of GenVoca layers may describe a whole family of OO frameworks since a configuration of a number of layers may represent a framework. In a degenerate case, a GenVoca layer may contain one class only. Configuring such layers corresponds to configuring a number of parameterized classes, some of which are parameterized by their superclasses.

If we use the GenVoca architecture as our implementation components architecture, our ICCL is obviously a GenVoca grammar (see e.g. Figure 173). In this case, we can think of our generative component to produce whole customized class hierarchies based on specifications in a configuration DSL.

The generated component or the configured runtime component should keep the metainformation describing its current configuration in an accessible form since this information may be interesting to other components. For example, the metainformation of a matrix is needed by the generative component implementing matrix expressions, so that it can select optimal algorithm implementations for the given argument matrices.

We prefer to keep this metainformation in a *configuration repository* in the form of name-value pairs for all features (i.e. both the explicitly specified features and the computed defaults) rather than the original DSL description since we do not have to parse the description on each feature access. Each component has such a repository as its “birth certificate” (for the statically generated component) or current “data sheet” (for the dynamically reconfigurable component).

*Configuration
repository*

9.4.4 Expression DSLs

An important class of DSLs in algorithmic domains are expression DSLs. An expression DSL allows us to write algebraic expressions involving ADTs and operations on ADTs. Examples of expressions are matrix expressions, e.g. $M1+M2*M3+M4-M5$, where $+$, $*$, and $-$ are matrix addition, multiplication, and matrix subtraction and $M1$ through $M5$ are matrices. Another example are image algebra expressions, e.g.

$$\left((a \oplus s)^2 + (a \oplus s)^2 \right)^{1/2}$$

where a is an image, s is a template (i.e. an image whose values are images), and \oplus is the right linear product operator. This expression is used in the image algebraic formulation of the Roberts edge detector (see [RW96]).

First, we need to list the operations and the ADTs they operate on. For our matrix case study in Section 10.2.6, for example, we only consider matrix addition, subtraction, and multiplication. The BLAS vector and matrix operations (see Section 10.1.1.2.4) provide a more comprehensive set of basic operations for matrix algorithms. Image Algebra [RW96] defines several dozens of basic operations.

The operations usually have some systematic relationships with the properties of their arguments. For example, adding two lower triangular matrices results in another lower triangular matrix, adding a lower triangular matrix and an upper triangular matrix results in a general square matrix, adding two dense matrices results in a dense matrix, etc.

If we know that a certain property of a matrix does not change during the entire runtime, we can encode this property in its static type. We can then use the static properties of the arguments to derive the static properties of operation results. This corresponds to type inference. In general, we use the configuration repositories of the argument ADTs (a configuration repository contains all the static features of an ADT) to compute the configuration repository of the resulting ADT. Thus, we need to specify the mapping function for computing the features of operation results from the features of the arguments. For this purpose, we use *dependency tables*, which we define in Section 10.2.3. The functions for computing result types for matrix operations are given in Section 10.2.7.

Dependency tables

Finally, we need to investigate opportunities for optimizations. We distinguish between two types of optimizations:

- *Optimizations of single operations*: This kind of optimization is performed by selecting specialized algorithms based on the information in the configuration repository of the participating ADTs. For example, we use different addition and multiplication algorithms depending on the shape of the argument matrices. The specification of such optimizations involves specifying the different algorithms and the criteria for selecting them (see e.g. Section 10.2.6.3).
- *Optimizations of whole expressions*: This kind of optimization involves the structural analysis of the entire expression and generating customized code based on this analysis. Examples of such optimizations are elimination of temporaries and loop fusing. The optimizations may be performed at several levels of refinement. Sophisticated optimization techniques for expression DSLs are described in [Big98a, Big98b].

9.5 Domain Implementation

Different parts of an algorithmic library require different implementation techniques:

- ADTs, operations, and algorithms can be adequately implemented using parameterized functions, parameterized classes, and mixin layers (i.e. the GenVoca model). All of these abstraction mechanisms are available in C++.
- The implementation of configuration generators and expression optimizations require static metaprogramming capabilities. Here we can use built-in language capabilities such as template metaprogramming in C++, custom-developed preprocessors or compilers, or specialized metaprogramming environments and tools such as IP (Section 6.4.3) or Open C++ [Chi95].

- Domain-specific syntax extensions require preprocessors, custom compilers, or extendible compilers (e.g. IP). The C++ vector library Blitz++ (see Section 7.6.4), however, demonstrates that a rich language such as C++ allows us to simulate a great deal of mathematical notations without the need of syntax extensions.

Chapter 10 will demonstrate two implementation approaches:

- implementation in C++ in Section 10.3.1 (including implementation of a GenVoca architecture using C++ class templates, implementation of a configuration generator using template metaprogramming, implementation of the expression DSL using expression templates) and
- implementation in the IP System in Section 10.3.2.

Both approaches are evaluated and compared in Sections 10.3.1.8 and 10.3.2.

9.6 References

- [Big98a] T. Biggerstaff. Anticipatory Optimization in Domain Specific Translation. In *Proceedings of the Fifth International Conference on Software Reuse (ICSR'98)*, P. Devanbu and J. Paulin, (Eds.), IEEE Computer Society Press, 1998, pp. 124-133
- [Big98b] T. Biggerstaff. Composite Folding and Optimization in Domain Specific Translation. Technical Report, MSR-TR-98-22, Microsoft Research, June 1998
- [BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. North-Holland, New York, 1976
- [Chi95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'95)*, *ACM SIGPLAN Notices*, vol. 30, no. 10, 1995, pp. 285-299, <http://www.softlab.is.tukuba.ac.jp/~chiba/openc++.html>
- [Cop92] J. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1992
- [DK98] A. van Deursen and P. Klint. Little Languages: Little Maintenance? In *Journal of Software Maintenance*, no. 10, 1998, pp. 75-92, see <http://www.cwi.nl/~arie>
- [Eis95] U. W. Eisenecker. Recht auf Fehler. In *iX*, no. 6, 1996, pp. 184-189 (in German)
- [Eis96] Ulrich W. Eisenecker: Attribute im Zugriff. In *OBJEKTSpektrum*, no.5, September/October 1996, pp. 98-101 (in German)
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995
- [GL96] G. Golub and C. van Loan. *Matrix Computations*. Third edition. The John Hopkins University Press, Baltimore and London, 1996
- [GO93] G.H. Golub and J.M. Ortega. *Scientific Computing: An Introduction with Parallel Computing*. Academic Press, Boston, 1993
- [ILG+97] J. Irwin, J.-M. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-Oriented Programming of Sparse Matrix Code. XEROX PARC Technical Report SPL97-007 P9710045, February 1997, see <http://www.parc.xerox.com/aop>
- [Kee89] S. Keene. *Object-oriented programming in Common Lisp: a Programmer's Guide to CLOS*. Addison-Wesley, 1989
- [KL98] K. Kreft and A. Langer. Allocator Types. In *C++ Report*, vol. 10, no. 6, 1998, pp. 54-61 and p. 68
- [KLL+97] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy. Open Implementation Design Guidelines. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, 1997, pp. 481-490
- [KW97] D. Kühn and K. Weihe. Data access templates. In *C++ Report*, July/August, 1997
- [Mey90] B. Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, 1990

- [MLMK97] C. Maeda, A. Lee, G. Murphy, and G. Kiczales. Open Implementation Analysis and Design. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, M. Harandi, (Ed.), ACM Software Engineering Notes, vol 22, no. 3, May 1997, pp. 44-52
- [MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, Massachusetts, 1996
- [PC86] D. L. Parnas and P. C. Clemens. A Rational Design Process: How and Why to Fake It. In *IEEE Transactions to Software Engineering*, vol. SE-12, no. 2, February 1986, pp. 251-257
- [Pes98] C. Pescio. Multiple Dispatch: A New Approach Using Templates and RTTI. In *C++ Report*, vol. 10, no. 6, June 1998
- [Ree96] T. Reenskaug with P. Wold and O.A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Manning, 1996
- [RW96] G. X. Ritter and Joseph N. Wilson. *Handbook of Computer Vision Algorithms in Image Algebra*. CRC Press, 1996
- [SB98] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proceeding of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, E. Jul, (Ed.), 1998, pp. 550-570
- [Wal97] R. van der Wal. Algorithmic Object. In *C++ Report*, vol. 9, no. 6, June 1997, pp. 23-27
- [Wil92] P.R. Wilson. Uniprocessor garbage collection techniques. In *Memory Management*, Y. Bekkers and J. Cohen, (Eds.), Lecture Notes in Computer Science, no. 637, Springer-Verlag, 1992, pp. 1-42
- [WM95] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995