The Architecture Description Language MoDeL

Peter Klein

pk@pk-1.de

Abstract. This paper is devoted to the topic of architecture modeling for software systems. The architecture describes the structural composition of a system from components and relationships between these components. Thereby, it provides a basis for the system's realization on technical as well as on organizational level.

We present some key concepts of the architecture description language MoDeL (Modular Design Language). By selecting and combining modeling elements which proved to be helpful for the design of software systems, this approach is integrative and pragmatic: It allows the definition of "clean" logical structures as well as adaptations necessary due to implementation constraints. Both the logical architecture as well as concrete architectures reflecting respective modifications are considered as individual results of architecture modeling. Even more, the transformation steps describing the changes induced by a particular realization constraint contain valuable modeling knowledge as well.

1 Introduction

The observation that the structure of a software system as defined by its architecture is a crucial aspect in the development and maintenance process is almost as old as the software engineering discipline itself. The reasons for this are obvious: The main expenditure during the development process concerning manpower, time, and money is still Programming in the Small (PiS). A good design makes PiS easy in that the programmer can concentrate on a problem with a comprehensible complexity. Errors made in the implementation can be found and eliminated more easily because realization details are encapsulated. For the same reason, software systems are more adaptable, reusable, and portable than before. On the other hand, errors made in the design may lead to an enormous waste of implementation efforts. One may argue that the same dependency holds for requirements specification and architecture, but a good and adaptable architecture always represents a set of similar requirements. To a certain extent, changes in the requirements are readily integrated into a good design.

Specifically with the advent of object-oriented specification approaches, however, there has been an overall tendency to focus more strongly on analysis activities and consider architecture modeling more as a mapping of "classes and objects found in the vocabulary of the object domain" [1] to an implementation view – something which might or might not make sense from a structural perspective. Although languages like UML [2], [3] most certainly can be used to describe software architectures in the above sense, they do not specifically encourage architects to design a robust framework structure for the PiS phase.

As a language naturally influences the way the speaker thinks when he communicates in that language, it should neither restrict nor overtax the speaker with its vocabulary and rules. For an architecture description language (ADL), this means that it should noticeably …

1. … be easy to use and to understand.
2. … provide the necessary detail to allow the definition of independent working packages for implementation, documentation, and testing, but not more.
3. … not impose a certain style or methodology by preferring or neglecting certain kinds of abstractions.
4. … be independent of the programming language to be used for implementation.
5. … allow for different levels of abstraction.

Based on a long history of preceding work (cf. e.g. [4], [5], [6], [7], [8], [9]), [10] suggests an ADL called MoDeL (Modular Design Language) developed adhering the above requirements. Although retaining the general approach that an architecture, in the first place, needs to define structure, it provides additional views allowing the architect to communicate design decisions. This paper summarizes the main concepts of this language; examples of its use can be found in [10].

2 Architecture Views

One of the basic ideas of the MoDeL language is the distinction of two dimensions with respect to what and how the architecture is modeled, cf. fig. 1. In the top-left corner, a system's static structure is defined with its components and their interfaces. To describe the dynamic behavior of the system, one or more interaction diagrams (top-right corner) may be used. Both specifications are restricted to the logical level, i.e. they strictly adhere to the concepts of modularity and encapsulation. Declarative semantics may be defined formally in the static part and informal operational semantics in the dynamic part as considered appropriate by the architect.

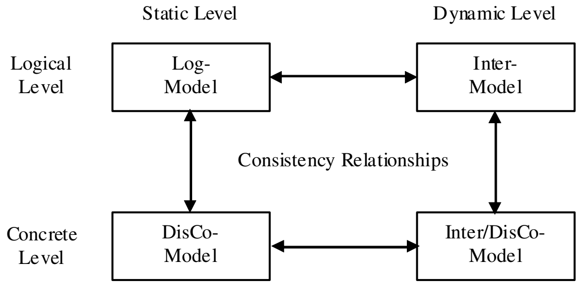

Fig. 1. MoDeL Views

However, there are many potential reasons why an architecture cannot be implemented exactly the way it is specified on logical level. Furthermore, there might be additional information the architect wants to specify beyond the logical structure. Some examples are:

1. Annotation of concurrency properties of components, like which components comprise a process, synchronization semantics of interface operations etc.
2. Introduction of components to handle distribution, e.g. for parameter marshaling, finding a service provider etc.
3. Extension or adaptation of the architecture in order to integrate components with a different architectural structure, e.g. if external libraries or components generated by external tools are used.
4. Specification of the implementation of usability relationships, e.g. via (remote) procedure calls, exceptions, interrupts, event-triggering, or other forms of callback mechanisms.
5. "Opening up" an abstract data type to increase efficiency.

In some cases, there are dependencies between the corresponding realization decisions: As an example, the architect might want to separate a data object (e.g. a database) from the rest of an application to make it remotely accessible by different users. He could implement the incoming usability relationships as RPCs (cf. 4). This also requires corresponding components as those mentioned in 2 to be introduced. These components might use existing data conversion libraries and generated stubs, so he may have to make some changes to the logical architecture as in 3. Consequently, he might want to denote the synchronization semantics necessary due to the concurrent access to the data object as in 1 etc.

All of these activities require changes to the architecture of the system as interfaces change, new components are introduced, or implementation details are added. The resulting architecture though has a different quality than the logical architecture: It does not aim at the best possible structure with respect to maintainability etc., instead it describes a step towards a concrete implementation of the system. In this sense, we call it a concrete architecture in the following.

To some extent, this idea is similar to the distinction between Platform Independent and Platform Specific Models in Model Driven Architecture [11] and its evolutions. However, logical architectures in MoDeL are not necessarily platform independent – this will depend, on a case by case basis, on the nature of the system to be described (cf. section 6).

It should also be noted that different concrete architectures for one logical architecture can exist. These may reflect a sequence of possibly interdependent decisions as sketched above, a set of independent decisions, or different realization variants. Even more, in the context of re- and reverse engineering, the existing system to be analyzed can be considered an implementation of some concrete architecture which has to be derived from the source code or other documentation. Then, the logical architecture can be distilled from the concrete architecture which, in turn, will probably be the basis for restructuring the system and respective new concrete architectures.

Although logical and concrete architectures of a system are naturally related, they should be treated as individual results of the design process. The logical architecture is necessary to understand the system's structure and it remains the central document for implementation and maintenance activities. The aspects described by the concrete architecture are an important step towards the realization of the system, but they should not undermine its logical structure. In other words, each of the architectures represents a set of orthogonal design decisions which should be made, described, and maintained separately. The integration of these architecture views and checking their consistency can be supported by tools.

And finally, also the transformation steps leading from one architecture to another contain important design knowledge. For many existing systems, only either the initial (and typically logical) architecture is kept or the latest instance of the concrete architecture is maintained. However, both the original design decisions as well as how and why modifications were made are necessary to understand a system's structure, and an explicit transformation step offers a convenient place to document differences between the logical and concrete level. Furthermore, if a specific transformation occurs frequently, it can be possible to formalize it and provide tool support for its application. In this sense, the design knowledge concerning how to modify an architecture to meet some certain purpose can be formally specified and, in consequence, communicated, reused, and possibly supported automatically by tools. Generally formalizing and even automating such transformations, as in [11], has however not been our ambition.

From the wide range of possible uses of concrete architectures, we particularly focus on the specification of concurrency and distribution. These important aspects of a software system require some additional architecture language elements to be described properly; we summarize the respective extensions in the Dis(tributed)Co(ncurrent)MoDeL sublanguage. At the bottom of fig. 1, we have therefore introduced a concrete level which allows such specifications. As on the logical level, static (left box) and dynamic (right box) properties can be defined.

Though fig. 1, at first sight, closely resembles approaches like the 4+1 view model as described in [12], the distinction between the views in MoDeL follows a different pattern: It is not our aim to serve different stakeholder's requirements, but to facilitate communication between the architectural and the "programming" level of a software project. So, for example, the logical view in [12] would remain, in our approach, in the analysis part of the project, whereas the physical view might or might not be described in a concrete MoDeL architecture – depending on whether the distribution of processes across nodes is part of the design or part of the implementation, i.e. whether it is relevant for the overall structure of the system or not.

3 Abstraction Types

As far as its basic abstractions are concerned, MoDeL is mostly motivated by programming language concepts. Apart from being well-known to architects and programmers, this particularly facilitates implementing a design in some programming language. However, all of these concepts have to be reconsidered on architectural level. Of course, implementing a design is more or less problematic depending on the expressiveness of the programming language to be used, but generally always possible. As an example, a module or package concept as can be found in languages like Modula-2, Modula-3, and Ada makes the translation of architectural modules into programming language constructs easier, but a module/package in one of these languages is not necessarily a module on architecture level. Vice versa, an architectural module can be translated into programming languages without explicit support for such a concept as well.

On component level, MoDeL distinguishes module types in two dimensions:

- Functional and Data Abstraction: Whereas traditional design methods like Structured Design [13] tend to produce hierarchies of functional components in a top-down fashion, object-oriented methods like Object-Oriented Design [1] focus on a loosely coupled set of abstract data types/classes. Meanwhile, it is commonly agreed upon (cf. e.g. [14]) that both kinds of abstractions are necessary. This conclusion is essentially based on Parnas' salient observation that each design decision should be encapsulated by a module and vice versa [15]. Such a design decision can be expressed in MoDeL, according to its nature, either as an abstraction from operation or from state.
- Type and Instance Abstraction: Unlike most other approaches, MoDeL allows modules encapsulating a state or control flow and modules offering a template/type to dynamically create a state/control flow at runtime in one architecture description. Using modules on instance level, the designer can introduce global (i.e. system-wide) state in the architecture without the need to define a data type and additionally assuring that exactly one instance of that type will be instantiated (cf. the Singleton pattern in [16]). Furthermore, this allows a clean model of situations close to the hardware boundaries of the system where concrete devices (e.g. “the keyboard”, “the heat sensor”) are involved.

Concerning component relationships, the following concepts are supported:

- Locality: Derived from block-structured programming languages as can be found e.g. in the ALGOL family, the concept of locality together with a corresponding set of visibility/usability rules is equally important on architecture level. It denotes that some component is designed to fulfill its purpose in a special context only and that it should not be accessible from outside this context.
- Generality: To introduce components with a general character into an architecture, a corresponding general usability relationship may be used. As an unspecific way to make interface resources of one component usable by another component, this relationship is comparable to an import or use construct in programming languages with a module concept like Modula-2, Modula-3, or Ada.
- Specialization: Object-oriented programming languages like C++, Smalltalk, or Eiffel offer inheritance between data types/classes as a special notion of similarity. Although some of the potential of inheritance depends on using a programming language which supports inclusion polymorphism (in the sense of [17]) in its type system, the concept of modeling some data type as a specialization of another data type is generally useful on architecture level.

Finally, MoDeL supports parametric polymorphism (as defined in [17]) in the form of generic components. Although genericity can be simulated using inheritance on programming language level, these concepts are rather different on architecture level. Whereas specialization is used to model the similarities and differences between data types, genericity allows arbitrary parameterization of arbitrary components.

4 The Static View

MoDeL's key elements can be informally defined as follows:

- **Interface:** A collection of resources like operations, types, constants etc. Basically, we use the term as known from programming languages with a module concept. In contrast to other approaches, we do not assign descriptions of dynamic aspects to operations; consequently, there is no execution model for (a collection of) interfaces.
- **Module:** A module is a logical unit of a software system with a clearly defined purpose in a given context. It consists of an export interface defining which resources the module offers to the rest of the system, an import interface defining which resources from other modules the module may use to realize its export interface, and an implementation in some programming language. The term interface of a module, without further qualification, refers to the export interface.
- **Subsystem:** A collection of components (see below). Subsystems have, like modules, interfaces. The import interface of a subsystem is the union of all import interfaces of the contained components, minus the resources defined by components in the same subsystem. The export interface of a subsystem is an explicitly defined subset of the union of all export interfaces of the contained components.
- **Component:** A module or a subsystem.
- (Component, Module, Subsystem) **Relationship:** A dependency between components resulting from the fact that some resource contained in the export interface of one component (the resource provider) is usable by another component (the resource employer) by appearing in its import interface.
- **Architecture:** The structural building plan for a software system. It defines all of the system's components and their relationships in the form of their import and export interfaces, but not their implementations.

MoDeL distinguishes between different module and relationship types based on the different abstractions introduced in section 3.

4.1 Functional Abstraction

In general, functional abstraction is at hand if a module has some kind of transformation character. This means that an interface resource transforms some kind of input data into corresponding output data. Functional abstraction facilitates the hiding of algorithmic details of this transformation.

An important property of functional modules is that they may not contain memory unless some code inside the module's body is executed. In other words, as soon as the execution of an interface resource of the module is finished, the module has no knowledge about previous calls. The reason for this restriction is that a functional module with a state commonly contains two implementation decisions in one module: one for the actual functionality of the module and one for the state of the module, which should be modeled as a separate data abstraction module instead. This is no conceptual restriction because any module with an internal memory can be made stateless if the state is stored elsewhere and either read by the functional module or passed to respective operations by the resource employer.

This restriction does not mean that the body of a functional module may not have any global variables: it is acceptable if e.g. a module computing trigonometric functions uses an internal table of key values, or if a text-processing module uses a buffer of cached characters. This is no violation of the principle that the mapping of input to output values is independent of the runtime "history" of the module (though it may depend on the history of data abstraction modules used by the functional module). In fact, as the architect defines only interfaces, he cannot keep the implementation from using global variables for whatever purpose as long as the semantic restrictions mentioned above are adhered to.

Regarding the distinction between type and instance level abstraction, we note that the logical level only requires function object modules. Function type modules come into the picture in the context of concurrency, cf. section 6.2.

4.2 Data Abstraction

Data abstraction is present if the module encapsulates the access to some kind of "memory" or "state". The module hides the realization of the data representation. The module's interface only shows how the data can be used, not how it is mapped onto the underlying storage.

In MoDeL, data abstraction is supported by two module types, namely data object and data type modules.

Data object modules represent global state in the architecture which can be implemented by global variables inside the module, i.e. directly mapped onto some structures of the programming language, and/or other data object modules or instances of data type modules. The interface of a data object module exports operations to manipulate the state of the module.

Data type modules export exactly one type identifier (although trivial "helper" types are allowed as well) and access operations for instances of this type. Other modules may use the interface to create instances of the data type and manipulate it with the given operations. Data type modules are templates for the creation of memory and may not contain a global state: The execution of an interface resource on an identical instance of the type always modifies this instance in the same way. As with function object modules, this does not mean that the body of the module may not contain global information, but operations on one instance of the type may not have visible side-effects on other instances.

For reasons of clarity, we always treat types introduced by abstract data type modules as reference types. This does not necessarily require such a type to be mapped onto a "pointer" in the programming language: A CORBA object identifier, an integer-valued handle etc. have reference semantics as well. Important consequences of this restriction are:

- Instances of the type always have to be created (and possibly destroyed) explicitly, be it on the heap provided by the programming language runtime system or within some other component of the software system. Noticeably, assignment (explicit or by parameter passing) or variable declaration never create a new instance of the type. Means to copy instances, if required, have to be specified as an operation. This allows the architect to specify whether copying is possible at all and, if so, what semantics (deep, shallow, or anything intermediate) is used. Multiple copy operations with different semantics can be specified as well.

- Comparison is reference comparison. As above, the specification may define if comparison of values is possible and what semantics apply by defining respective operations.
- Using reference semantics makes it sensible to think of `self`, the parameter denoting the object on which the operation is applied, as an input parameter: Whether or not the operation modifies the instance's state, `self` as a reference is never changed by the operation.

4.3 Interface Extensions and Export Control

Frequently, the necessity arises to give some employers of a module more visibility on its realization than others. A common example is that an employer defining a subtype of some data type requires more control over the attributes of its supertype than "ordinary" employers of that module. Exactly this situation is covered in many object-oriented modeling and programming languages by introducing a separation between public, protected, and private resources of a data type.

Although the public/private/protected scheme meets many basic requirements, it has some major drawbacks:

- The separation is only usable in the context of data types and inheritance. There is no good reason why a similar mechanism should not be available for other module types or other module relationships as well. In object-oriented literature, this is typically solved by using the "friend" concept. However, when using friend relationships, export control is given up altogether as complete access to the realization details is granted.
- Even under the above restriction, there are situations that cannot be described appropriately with only three layers of accessibility. This results from the approach to define the accessibility of a feature as an inherent property of the feature itself: In the interface of the module, the feature is generally classified as being public, protected, or private – regardless of the employer. Conceptually, this is not quite correct as its accessibility has something to do with how the feature is used in the overall design of the system, i.e. the actual pair of provider and employer.

Consequently, we take on a more general approach that allows one or more views on a module to be specified. Some views might provide more control over the module's internals than others by relieving certain abstractions. To provide such views, we use the concept of interface extensions: the architect may define an arbitrary number of additional so-called view interfaces for the module. A view interface may contain additional operations and helper types for data object and functional modules. Furthermore, for data object modules, some or all of the internal representation of the module's state may be exposed. For data type modules, additional features (including attributes) may be defined.

4.4 Module Relationships

After presenting the different kinds of basic design units, we now introduce the means MoDeL provides to describe interactions between these units. First of all, some modules want to make use of resources offered by other modules. We distinguish three different logical levels on which such dependencies can be discussed:

1. The first prerequisite for the interaction between an employer and a provider module is that the design allows the employer to access the resources offered by the provider. We call this the usability level.
2. If the architecture allows some module to access another module (on the usability level), the employer may make use of this by actually using some of the provider's resources (e.g. if the employer's implementation contains a call of a procedure offered by the provider). This use is static, i.e. it can be determined by looking at the implementation of the employer. Therefore, it is on the static uses level.
3. If a static use of some provider resource is executed during the runtime of the program, we say that this is on the dynamic uses level.

Since we are considering the static view on the architecture, it cannot be determined whether the employer's implementation makes use of a resource or not. It can only allow some module to use another module's resources. So, if we talk about some module importing another module, this is always on usability level. The uses level, however, plays a role in a dynamic view of the system, cf. section 5.

Besides the usability relationships from above, we also have structural relationships between modules. These are used to express structural design concepts and allow or forbid certain usability relationships.

4.4.1 Local Containment/Usability

We start our discussion with a relationship called local containment. This is a structural relationship describing that some module is contained in another module. From this containment, some rules derived from the block-structuring idea present in many programming languages follow. This relationship forms a local containment tree in the module dependency graph. Placing a module in such a tree means that the module can only be accessed from certain parts inside this tree. In this sense, it introduces information hiding on architecture level.

The following usability relationships are possible in a local containment tree: A module can use itself, its direct successors, its predecessors, and direct successors of all predecessors (especially its brothers). This is completely analogous to the rules of locality and visibility in block-structured programming languages. We say that potential local usability exists between the module and its above mentioned relatives in the containment tree.

One problem with the relationship of potential local usability is that many relationships are made possible between modules which are not necessary. We therefore introduce the local usability relationship. Local usability is a relationship which is explicitly specified in the architecture, although such a relationship may only be defined between modules for which a potential local usability exists. In other words, the local usability relationships are a subset of the potential local usabilities defined by the designer.

4.4.2 General Usability

The local usability relationship introduced so far is not suited for all situations where one module wants to use resources from another module. This is particularly the case if some module should be usable by arbitrary employers, possibly from different containment structures.

With the general usability relationship, the architect has means to express that a module exports general resources which can be accessed from all other modules for which a corresponding general usability relationship exists.

There is no structural relationship directly connected to general usability: General usability edges from any part of the system can end in one module (which, itself, must not be contained in another module). On the other hand, it is not unusual that only one employer uses some provider module through general usability. Whether or not a module is generally or locally usable is a question of the kind of provider module. If a module offers general services to employers, it should be inserted using the general usability. If it offers services which are only useful in a certain context, it should be inserted using the local containment and local usability relationships.

4.4.3 Specialization/Specialization Usability

The other structural relationship in MoDeL besides local containment is the specialization relationship. Although the concept of specialization is influenced by ideas from object-oriented programming languages, it denotes a relationship between modules on the design level here. Whether the actual implementation language's type system supports specialization in some way should not influence the logical architecture (though it might influence a concrete architecture).

The specialization relationship can only exist between data type (and function type, cf. section 6.2) modules. If some data type is a specialization of another data type, this implies that every instance of the special type has at least all the properties of an instance of the general type. As usual, we call the special type a subtype of the general type, which is vice versa referred to as the supertype of the special type. This terminology extends to arbitrary ancestors and predecessors of a data type module in the specialization hierarchy.

An important characteristic of the specialization relationship is that the set of features offered by the subtype is a superset of the set of features of the supertype. We therefore do not have to repeat these features in the subtype's interface.

As with local containment and local usability, the structural specialization relationship is accompanied by a usability relationship, the specialization usability. By definition, a specialized module needs to import the module it specializes.

It should be noted that the structural relationship of generalization implies no usability relationships. In MoDeL, a subtype module may not use some supertype's operation just because it is a subtype: All usability relationships have to be introduced explicitly by the designer. On the other hand, the structural relationship again determines the set of possible usability relationships. This leads to a strong correlation between architectural usability relationships and the common notion of an "import": A subtype module needs to import the supertype module's interface in order to use the corresponding type identifier in the declaration of the subtype identifier. Furthermore, if some subtype wants to call an implementation of an operation as defined for some specific supertype (in contrast to whatever implementation is dynamically bound to the operation, see below), it has to address this module directly. Obviously, every subtype module must have a specialization usability edge to its immediate supertype module and may have additional specialization usability edges to arbitrary predecessors in the specialization hierarchy.

Apart from these definitions on module level, we also have to consider the connection between specialization as a module relationship and the consequences for the types and their operations exported by corresponding modules. Unfortunately, this affects issues which are related to Programming in the Small. On one hand, defining them on architectural level makes a few (possibly unwelcome) assumptions concerning the implementation. On the other hand, in the context of specialization, a data type module's interface cannot be properly defined and understood if these questions remain open.

- As usual in object-oriented modeling approaches, we consider variables of an abstract data type as polymorphic, i.e. any variable or formal parameter of some type can be assigned/passed a value of some subtype of this type.
- A supertype operation's implementation can be redefined for a subtype. Syntactically, this is not visible in the interface, but an informal (comment) or formal (cf. section 4.7) semantics specification can indicate this.
- In the context of polymorphism, a variable's or parameter's dynamic type at runtime can be a subtype of its static type (according to the declaration). If the subtype has redefined operations, we always consider the dynamic type decisive for the selection of the implementation (late binding, dynamic dispatching). In this case, there may be transfers of program control from one module to another module without an explicit usability relationship between them: An employer module can call a data type's operation according to the (supertype) interface it is aware of, whereas at runtime, some code within another provider module (defining a subtype of the type in question) is executed. Conceptually, however, the employer is independent of the subtype interface and only depends on the supertype interface. Therefore, a usability relationship between employer and subtype module is not required.
- We allow operations to be redefined in subtypes, i.e. a subtype may supply a new (new in the sense of a different signature and different semantics) operation with the same name as an inherited operation. It should be noted that redefining an operation is something different than redefining an operation's implementation as discussed above. Although operation redefinition is not unproblematic with respect to polymorphism, it is frequently the case that some subtype operation is "more complex" than a very similar operation in a supertype, consequently, it will want to raise more exceptions, might need more input parameters, produce more complex results etc. Some of these points can be relieved by stating co- and contravariance rules for parameters, but this introduces additional complexity without completely solving the problem.

It should be noted that redefining an operation, in contrast to redefining the implementation, does not replace the existing operation's implementation. As far as the selection of the implementation in the context of polymorphic variables is concerned, we consider the static type of the variable as decisive: This is necessary as an employer can only provide the input and handle the output of an operation as it is specified in the interface it knows and by which it declares the variable/parameter.

Frequently, if a subtype redefines an operation, it also redefines the implementation of the inherited operation of the same name to sensibly handle attempts of operation calls which are not appropriate for the dynamic type of an instance.

4.5 Subsystems

Obviously, the module level introduced so far is too fine-grained for the description of large software systems. We therefore need design units which allow a hierarchical specification of the architecture. Subsystems allow the designer to express such units which are "greater" than modules: they may contain an arbitrary number of modules and other subsystems.

Most of the characterizations given for modules at the beginning of this chapter can be applied to subsystems as well: first of all, subsystems are units of abstraction. They have an interface which describes the resources which can be accessed from the outside. The interface of a subsystem is a composition of explicitly selected modules and/or subsystems inside the subsystem. Furthermore, subsystems are units of work, units of testing, and units of reusability. For this reason, subsystems should – just like modules – obey the rules of low coupling and high cohesion: the modules and subsystems should not interact more than necessary with other units on the same design level. On the other hand, a module or subsystem should only contain logically related resources.

However, it still makes sense to distinguish between modules and subsystems as the former are directly linked to the PiS level in the sense that they involve implementation work, whereas the latter start out purely as a design concept which might not materialize in program code at all.

As stated above, the designer decides explicitly which interfaces in the subsystem contribute to the subsystem's interface. An immediate consequence is that we cannot assign each subsystem a unique type as we could with modules. We therefore do not distinguish different subsystem types. Nevertheless, we sometimes talk about functional or data type subsystems if the subsystem's interface consists of one or more modules of the corresponding type.

Of course, being part of a subsystem is a structural relationship between the corresponding components. Accordingly, we already introduced a natural candidate for subsystems in the previous chapter, namely local containment trees. Containment trees are a special sort of subsystems where the interface is implicitly given by the root of the tree. However, apart from isolating components within the subsystem from components outside, containment trees additionally introduce the locality/visibility restrictions as described above.

4.6 Generics

MoDeL supports a reuse mechanism commonly known as genericity (or parametric polymorphism according to the more precise terminology introduced in [17]). The main idea of genericity is to write (generic) templates for system components. In the template, an arbitrary number of details is not wired into the code, instead the template code refers to these details using formal parameter names. The programmer can then create a concrete component by supplying the missing details in the template.

We call these details (generic) parameters, and the process of creating a component using a generic template and generic parameters (generic) instantiation. Accordingly, the resulting component is called a (generic) instance.

Ideally, the process described above is directly or indirectly supported by the programming language or the development environment. But even if not, it allows the architect to indicate that for different physical modules the same code should be used.

A common example for generic templates is a collection (container) of instances of an entry type. The generic parameter for the template is the entry type for the collection. The reason for this approach is that the implementation of the collection is more or less independent of the type of objects it can store, i.e. neither interface nor implementation of the collection refer to special properties of the entry type. MoDeL therefore allows a generic specification of the container, making the entry type a generic parameter. The designer can then use this template to instantiate arbitrary concrete containers by providing respective entry types from data type modules in the architecture.

We can readily extend the concept of generics from modules to subsystems. Conceptually, generic subsystems are just as important as generic modules, e.g. in a situation where some collection is built on top of one or more other collections. For example, a generic module offering a table data structure might be implemented using a generic data type module for dynamic arrays. In this case, it would be natural to instantiate a generic subsystem consisting of a generic table and a generic array module.

Of course, a generic subsystem may also contain non-generic components. From the viewpoint of the instantiation process, they can be thought of as generic components without generic parameters. Furthermore, in order to make sense, a generic parameter of the subsystem must be generic parameter for at least one generic component within the subsystem. Vice versa, a generic parameter of a contained component must be a generic parameter of the subsystem unless the component is already instantiated within the subsystem.

4.7 Specifying Semantics

Up to now, the MoDeL language contains constructs to define components, their interfaces, and their relationships. Such descriptions, however, are worthless if the architect has no way to attach some “meaning” to the respective specification elements. In general, there are two levels on which the architect will have to consider such semantic issues:

- On the level of single interface definitions, the architect will have certain semantics in mind when specifying operations. Depending on the context, he will add e.g. a corresponding informal comment to the operation describing the desired behavior.
- On the level of components and their interaction, the architect influences the possible implementations by the static structure he uses. Considering the example of implementing some table module on top of a module for dynamic arrays, a suggestion for the implementation of the table has already been provided. To some extent, this can be considered as a violation of the abstraction boundary between Programming in the Large and Programming in the Small, but it is inherent to the notion of software architectures as a building plan [18].

Consequently, the architect should have means to specify what he had in mind when defining an architecture in a certain way. We can distinguish two dimensions for such additional information:

- Black-Box vs. White-Box View: The black-box view defines the semantics of interface operations in terms of interfaces only. It makes no assumptions concerning the implementation of an operation and if or how an operation could be realized (in general using interfaces of other components). A white-box view, sometimes called glass-box view, defines the behavior of operations in the context of the overall system, i.e. it does include interactions between components. In short, the black-box view is concerned with what an operation/component does and the white-box view with how it is done.
- Formal vs. Informal/Semiformal Notations: In practice, it is common to use informal plain text comments in natural language to describe the semantics of interface operations. This is probably sufficient in many situations, but certainly lacks the preciseness necessary in others. The most important reason for taking a more formal approach is to avoid misunderstandings between architect and programmer, but it may serve other purposes as well: A formal specification can be subjected to formal consistency and completeness checks, it can be used to simulate/prototype the specified system (parts), to formally prove the correctness of hand-coded implementations against the specification, or to generate test cases for such an implementation.

It should be noted that, according to whether a black-box or white-box view is taken, the term "semantics" as used above takes on two different meanings: In a black-box view, it generally denotes a mapping of input to output values of each operation. Accordingly, we call this the declarative semantics of a component. In a white-box view, on the other hand, the architect might specify a specific sequence of steps that needs to be taken inside the implementation. This kind of definition is generally called the operational semantics of a component/operation. Both declarative and operational semantics can be defined formally or informally.

Another property which can be assigned to the definition of a component's semantics is whether it is complete or not. By complete we mean that the semantics of the component (as the sum of all of its operations) is specified for all possible input values and states. Noticeably, the definition of the component's semantics may cover only "some" (important, frequent, critical etc.) cases. In practice, interface comments mention only the intended "usual" behavior. The behavior in other cases is undefined. Typically, informal specifications tend to leave missing details to the intuition of the reader.

In the following, we will mainly use the term semantics as referring to declarative semantics. Since the declarative semantics of a component and its operations are static properties, they will be discussed next. Operational semantics are concerned with the runtime behavior of the system; they will therefore be a topic in section 5. It should be noted that this distinction is not always clear; the terms semantics and behavior are indeed closely related. Consequently, other authors use different definitions and classifications in this context. For our purpose, it is sufficient to bundle the terms static specification (or, to be more precise, specification of static properties) and declarative semantics on one hand and behavior specification (specification of behavioral properties) and operational semantics on the other. Typically, operational specifications in our sense include information about the interplay of components whereas declarative specifications are "context-free", i.e. do not refer to the specification of (operations of) other components. Another characteristic property of operational specifications is that, if the specification is sufficiently complete and formalized, it can be used to prototype or simulate the system.

As far as corresponding specification languages are concerned, it is frequently not a question of language constructs but of how the language is used in order to determine whether they specify what is done or how it is done. However, formal or semi-formal languages for operational specifications often have some execution semantics which, itself, may be defined formally or semi-formally. Taking the term "specification" in a broad sense, a high-level imperative programming language can be considered an operational specification language where the language and its execution semantics are mostly defined semi-formally.

Informal semantics specifications are widely used, e.g. in the form of comments. These specifications take, in general, a black-box view on the component, are more declarative than operational, and are almost always incomplete in the above sense. For obvious reasons, they do not need further discussion here.

More interesting is the question of what approach should be taken if a formal specification is required. Since we focus on architecture modeling, it is neither possible nor sensible to discuss this question in the general context of formal system specifications. Analyzing comparative case studies (cf. e.g. [19]) shows that it is hardly possible to judge the expressiveness of formalisms without a specific intention in mind. To clarify the key requirements for a formalism suitable for our purpose, we can state the following:

1. The formalism should be able to express the semantics of (elements of) MoDeL component interfaces. Noticeably, there should be a clear mapping between interface resources and specification elements.
2. Specifications should be static by nature, we are not (yet, cf. section 5) considering control flow issues, component interaction, or algorithms.
3. We are not (yet, cf. section 6.1) considering a possibly concurrent and/or distributed realization of the system. Terms like processes/tasks, synchronization, timeout etc. play no role in our logical architecture view and should therefore be avoided in a corresponding specification.
4. It should be possible, but not a requirement to formally specify the complete system.

We can inductively conclude:

- As given by 1, we mainly want to describe the semantics of operations in a component's interface. From 2 follows that we can describe an operation only in terms of states. The state space should provide means to define the applicability of an operation (preconditions), an operation's impact (if any) on the current state (postconditions), and an operation's result. These states must be abstract in the sense that they are meaningful with respect to the interface. Since the interface, in turn, consists of operations, nothing should be in the state space which is not necessary for the specification of operations in the above sense.

- Considering MoDeL's module types as introduced above, it is obvious for data object and data type modules that the abstract states which can be used in a formal specification are equivalence classes of physical states of corresponding data objects and data type instances. For specification purposes, a data abstraction module (being specified as abstract states and operations on these states) is frequently viewed as an abstract machine. For functional components, a more difficult situation arises since there is no associated state. There is no general solution to this problem: If, for example, a functional component's operation computes the square root of a positive real number, it is probably not necessary at all to provide a formal specification: An informal appeal to mathematical intuition will sufficiently explain the input/output relation of this operation. If the component is used to make complex transformations on a data object/data type instance or to transform one data object/data type instance into another, it could be desirable to specify its semantics in terms of the related input/output data structures. Finally, if the component's purpose is to set up a certain control flow among subtasks, static specifications are by definition not appropriate.

Especially points 2 and 3 from the above list show that the intention behind many existing formal approaches like CSP, Esterel, FOCUS, Estelle, LOTOS, or SDL is not quite in line with our intended purpose. For the remaining formalisms, we have to answer the question of how states are represented. We consider two options:

- In algebraic approaches ([20]), states are expressed implicitly, i.e. the state change induced by some operation is defined by how it affects subsequent operations.
- Model-based approaches like Z, VDM-SL, or B/AMN explicitly define the state space, usually in terms of typed set theory.

Basically, both approaches fulfill the requirements from the above list. However, in practice, model-based approaches tend to be easier to understand: Defining states explicitly allows the reader to focus on one operation at a time instead of having to understand an operation in the context of the other operations in an interface.

Eventually, among the model-based formalisms, we have chosen a systematic use of Z [21] to accompany MoDeL interface definitions. The main reason for this decision was that all other formalisms with a broader acceptance include imperative constructs to cover operational aspects of semantics (so-called wide-spectrum formalisms). For our purposes, this is not required and might even lead to "overspecified" interfaces.

5 The Dynamic View

As was pointed out in the previous section, it is necessary to document the semantics of a software system on two levels: Declarative semantics to understand the purpose of a component and its interface resources and operational semantics to understand the relationships between components and their interplay. Formal and informal declarative semantics definitions can be used in MoDeL in the form of Z specifications and comments, respectively, augmenting interface specifications with the corresponding information. Here, we take up the discussion of specifying operational semantics.

A variety of specification languages for the semantics of software systems and their components have been developed over the past years. One major direction originates in the structured approach to requirements analysis and design with its data, functional, and control model of a system ([22]). From the respective languages ER (and its many extensions), SA, and SA/RT, the latter two can be allocated to behavioral descriptions. The second direction emerged under the collective term object-oriented modeling and produced a wide variety of approaches [23] adopting and redefining structured concepts as well as adding new notations. In particular, we have considered the Object Modeling Technique (OMT) [24], the Object-Oriented Analysis/Design (OOA/D) method proposed by [1], and the Unified Modeling Language UML [2], [3].

Based on this work, we have defined an approach similar to Collaboration Diagrams as in OOA/D for the MoDeL sublanguage InterMoDeL for describing operational semantics of architectures. InterMoDeL specifications are called interaction diagrams. Like CDs, interaction diagrams allow the description of exemplary behavior. However, InterMoDeL is based on interacting components, not on interacting objects. This noticeably means that instances of data type modules are not shown as nodes in interaction diagrams (although they may appear as parameters in operation calls).

Interaction diagrams are composed of the following elements:

- Nodes in the diagram are components (modules or subsystems) as defined for architecture diagrams.
- Components may be connected with uses relationships. Uses relationships indicate the call of an operation offered by the resource provider. Obviously, for a direct uses relationship to be legal, a corresponding usability relationship in the architecture diagram must exist. Every uses relationship features one or more labels which, in turn, consist of a sequence number and an operation name with a signature.

Every label reflects a call of the operation as indicated by the label. The "actual parameters" shown with the call must match the formal parameter signature as given in the textual interface specification of the called component. Sequence numbers in InterMoDeL interaction diagrams are hierarchical as in UML.

It should be noted that, by assuming reference semantics for data type modules, there can be no "hidden" calls to a module. As an example, complex data types frequently require a used-defined implementation to copy an instance of this type. Using value semantics, this operation may be called implicitly by an assignment operator or by passing a variable of that type by value in a procedure call. In contrast, with reference semantics, every call inducing the execution of code within the data type module refers to an operation with a name and signature explicitly defined in the features part of the interface. This enhances the readability of corresponding interaction diagrams and makes consistency checks between architecture diagrams, interaction diagrams, and module implementations easier.

Interaction diagrams may also contain indirect uses relationships. They feature the same kind of label as direct uses relationships have. An indirect uses relationship indicates an operation call where the resource employer is not aware of the component in which the operation is implemented. This occurs e.g. when some component calls an operation in another component via a callback (procedure pointer) which itself has been passed to the calling component previously. For an indirect uses relationship, there need not be a parallel usability relationship. In fact, it frequently happens that indirect uses appear in the reverse direction to a "corresponding" usability relationship.

Another (technically closely related) example for indirect uses relationships appears in the context of specialization/polymorphism. When a component calls an operation of a data type, the implementation actually executed by this operation call might be located in a specialized data type module. According to the intended semantics of the diagram, the architect may use either a direct uses relationship to the operation of the supertype the calling component statically imports or an indirect uses relationship to the subtype operation which is actually called.

6 Concrete Architectures

As was already mentioned, there is a number of reasons which make it reasonable to modify a logical architecture described with LogMoDeL and InterMoDeL as a first step towards an implementation. We have introduced the term concrete architecture to reflect the results of such steps. The specific transformations from logical to concrete architecture are driven by specific needs, and they may or may not be applicable independently. As a consequence, the concrete architecture will evolve in a sequence of incremental transformation steps from the logical architecture. All of these steps select a realization variant for some abstract situation given by the logical architecture. In this sense, a logical architecture contains the logical essence of a host of concrete architectures describing special realization choices.

Introducing logical and concrete architectures allows us to model a software system on a pure structural level without having to implement it exactly that way. On the other hand, adaptations or extensions of the architecture necessary to fulfill certain realization constraints are still planned and prepared during the design process (and not, as frequently found in practice, as an "on-the-fly" activity during the implementation phase). Furthermore, logical and concrete architectures as well as the dependencies and differences between them are described and documented separately: Both architectures as well as their correspondences represent individual design decisions made by the architect.

Still open, though, is the question of which architectural concepts are on logical level and which are on concrete level. There is no general answer to this question; what is considered "logical" or "concrete" strongly depends on the respective software system and its properties. A rough guideline results from distinguishing between "inherent" and "derived" properties of the system:

- We can characterize a property of a software system as "inherent" if it is tightly connected to the problem to be solved. For example, a business administration system will always store its data in some sort of database, a CAD tool will always have a graphical user interface, a screenshot capturing tool will refer to some particular window system, and a telecommunications switching system will always be a concurrent system. In general, it is not very useful to leave corresponding design decisions to the concrete level as this overloads the logical level with abstractions of no practical use.

- “Derived” properties are not introduced by the problem itself but by a particular solution. Taking up the examples from above, we can state that the choice of a concrete database system and an appropriate database schema are derived design decisions for the architecture of a business administration system. The same holds for the selection of a particular GUI for a CAD tool or a certain window system version for the screenshot capturing tool. Corresponding architecture modifications should be made on the concrete level.

So, for example, being a concurrent system is probably an inherent property of e.g. a telecommunications switching system while it would be a derived property of e.g. a compiler. Accordingly, respective specifications will occur on logical level for the former and on concrete level for the latter type of software.

6.1 Concurrency

Although, in this sense, architecture language concepts for concurrency and distribution are not generally allocated to logical or concrete level, we discuss them in the context of concrete architectures. As our approach origins more from the area of main-stream application systems than from embedded or real-time systems, concurrency and distribution are typical derived properties: They are introduced to increase performance or to allow different users access to shared information, but they are not part of the problem domain. The respective modeling concepts to be introduced in the following, however, can be used on logical level as well.

If we now want to describe concurrency on architecture level, we first have to decide what a concurrent component actually is. The notion of active objects is frequently used in current literature to introduce concurrency on design level. Since the term active generally appears in different semantics, we sketch some definitions here.

1. [1] defines active objects as objects which encompass an own thread of control. This definition and others closely relate threads or processes with single components, noticeably data type or data object modules (classes/objects). A similar approach, although not restricted to data abstraction components, can be found in the task concept of the Ada programming.
2. Functional modules are abstractions of operations; they have no state and serve transformation or controlling purposes. Data abstraction modules encapsulate state or a template for state; they hide the internal representation of this state and allow access to it only via a set of corresponding operations preserving the abstraction’s semantics. In this sense, functional components act and data abstraction components are acted upon, and it seems natural to apply the term active to functional and passive to data abstraction components.
3. When two components interact, e.g. by a procedure call from one component into the other, the called component performs some action on behalf of the caller. Like under point 2, the calling component acts and the called component is acted upon, so we may apply the terms active and passive accordingly. We can readily generalize this scenario to all situations where one component induces some activity in another, be it directly (e.g. procedure call) or indirectly (e.g. event triggering).

4. A special variant of 3 concerns the situation when two components interact for execution control purposes, i.e. one component manipulating the thread of control of another component (start, terminate, suspend, resume etc.). Semantically, this is on a different level as 3, although it may be realized in just the same way (e.g. by procedure calls). Therefore, we distinguish the corresponding notions of active and passive in the thread controlling sense from those concerning the actual abstraction of a component.

Obviously, 1 directly introduces concurrency on design level. 2 and 3 describe definitions which can be applied to sequential systems as well. Point 4 is a rather complicated case because the passive component in this sense has to have a thread for the definition to make sense; therefore it is an active component in the sense of 1. The active component in 4 might or might not be active in the sense of 1. Be this as it may, the definition in 4 already requires some notion of concurrency. To avoid confusion with other semantics of active and passive, we use the term acting component for a component which has an own thread of control and reacting component for a component which has not.

Choosing the semantics described in 1 as the foundation for our terminology does not mean that the other semantics for active and passive play no role in the design of an architecture description language for concurrent systems. In fact, relating the notions of acting and reacting components to active and passive components in the sense of 2 to 4 yields some valuable ideas presented in the following.

Comparing the definitions of acting and reacting to those of functional and data abstraction components shows that there is an intuitive relationship between the property of owning a thread of execution and the abstraction decision represented by a module's type. If a data abstraction component's purpose is to hide the details of some state's internal representation, to define what operations are semantically sensible on the state, and to map these operations onto some manipulation of the internal representation, we observe that there is no need for such a component to own a thread of control. The execution of access operations of the component is always (possibly indirectly) triggered by some functional component and can therefore be performed in the corresponding thread of control. On the other hand, the execution of transformation or controlling activities may or may not happen concurrently. Therefore, we restrict acting components to functional abstractions, i.e. functional components may or may not be acting, while data abstraction components are always reacting.

Relating our definitions to the active/passive semantics described by 3 leads to the question whether an acting component can be acted upon in the sense of (directly or indirectly) calling one of its interface operations. A negative answer would result in the restriction that an acting component may have no interface. In the reverse direction, we see that a component with an empty interface surely has to be acting: If it has no own thread of control and no other control flow can enter through the interface, the component is obviously useless. But may an acting component have an interface and be passive in the sense of 3? The intuitive notion that a component either does something of its own or it acts on behalf of other components leads to the conclusion that this is not the case.

Considering some typical examples of functional abstractions, we observe that there are indeed some sensible interface operations for acting components. But we also note that these are operations which directly or indirectly manipulate the component's thread of control. We therefore allow exactly such thread controlling operations in the interface of acting components, even though the control semantics of an interface operation might be implicit.

6.2 Function Type Modules

We can now come back to the question of function type components. It may seem trivial at first sight that it is possible for an operation in an acting component's interface to create a new thread of control instead of manipulating a thread encapsulated by a function object. Up to now, we have considered acting components on instance level, i.e. equipped with exactly one thread of control. But just like we do for data abstraction components, we can shift the concepts for functional components to type level. This allows us to distinguish between function object and function type components: The former represent a single (set of) computation(s), and the latter are templates for instantiating such computations at runtime. In sequential systems, we do not need to bother with function types because a single instance of every function is sufficient. However, in a concurrent system, function types are necessary since their instantiation is the logical counterpart of the creation of a new thread of control.

Considering the similarities between data and function type modules, the question arises how far the analogy between these component types goes. More precisely, on architectural level, we have to define whether function type instances are allowed as parameters in operation signatures and whether there is a useful meaning for specialization relationships between function type modules.

We can readily resolve this issue with a simple construction: Data type module instances carry a state and operations manipulating this state. Function objects may have no state, so function type instances have no state as well. However, the thread of control attached to a function type instance represents a state with respect to the underlying execution engine, e.g. an execution point, a call stack etc. This thread state can be interpreted as the counterpart of a data type instance state.

6.3 Synchronization

To consider the implications of having several acting components in a software system, we first have to define more clearly what is meant by a thread of control and a process. Without being precise about this term so far, we have implied some independence concerning their execution. However, taking a closer look, different degrees of independence can be identified.

According to the extent of context information carried along with a thread of control, we distinguish three levels of processes:

- A thread of control consists of an execution point ("program counter"), a call stack, and an address space. In this case, the terms task, heavy-weight process, or simply process are frequently used. For a heavy-weight process, all state information, be it local (in a procedure) or global, can be accessed only using the control flow of that process.
- A thread of control consists of an execution point and a call stack. This is generally denoted as a light-weight process or thread. Local state information, as it is stored in the call stack, is uniquely instantiated and associated with every thread. Global state, however, is accessible by all threads, possibly at once.

- A thread of control exists of an execution point only. All state information is shared amongst all "processes". This case is rather rare in mainstream application systems, though it may occur in embedded real-time systems.

In the following, we use the term process as referring to any of these forms.

Whatever approach is taken, at certain points in time, some processes will generally have to cooperate in some way. Cooperation involves one or more of the following aspects:

1. Passing of information between processes may occur with a call of an interface operation. The details of what information is exchanged (input/output/exception parameters with their respective types) are covered by the signature of the operation and therefore already part of the architecture specification of an interface as introduced so far. For light-weight processes, arbitrary data may be communicated. Heavy-weight processes may not exchange references to data inside their address spaces, i.e. only values of atomic types or references to function type instances can be passed. Several techniques exist to handle the communication of more complex data structures by "flattening" the information into a stream of atomic values (marshalling) on employer side and recomposing the complex value on provider side (unmarshalling).
2. Synchronization of access to shared data, i.e. which operations of a component may be called concurrently and which may not, is not apparent from the interface so far. Furthermore, it has to be denoted whether the employer or the provider is responsible for ensuring that the synchronization rules are followed. We will extend interface specifications to express this information below. Note that this is not relevant for heavy-weight processes. For light-weight processes, the corresponding information must be given for data abstraction components. In the context of fly-weight processes, functional component specifications might be subject to synchronization extensions as well.
3. For control flow organization, we make the assumption that some process interaction in which the provider process is not able to accept a request will suspend the employer process until the request can be handled. For heavy-weight processes, this may occur if the thread of control in the provider process is currently executing some other code. For light-weight processes, the same may happen if the provider component has the responsibility of synchronizing access to its state and another control flow currently holds access to resources which may not be used concurrently with the requested resources. In both cases, we assume that the request is queued, the requesting process suspended, and the queued requests are handled in the respective order by the provider process, thereby resuming the control flows of the requesting processes.

To specify synchronization constraints as required by point 2 above, we have introduced a virtual resource model akin to the concept of monitors [25]. This model provides means to abstract from the state of a data object or data type instance in the form of so-called mutexes (this terminology is derived from the threading facility of the Modula-3 programming language). Synchronization requirements can be expressed as dependencies of operations from these mutexes.

A mutex can be declared in an interface. It represents a resource or set of resources which can be accessed by at most one process at a time. An operation may depend on a resource on two levels: If the execution of an operation demands that the resources represented by a mutex should be made available to one process only, we interpret this as the operation "locking" the mutex. Every mutex can be locked by at most one process. To be more precise, only one process at a time may execute an operation which locks a certain mutex. Other processes needing access to the same mutex have to wait for the locking process to end the execution of the respective operation. Frequently, however, a process does not have to lock a mutex, it is sufficient for the operation's execution that it is not locked by any other process. In this case, we say that the operation "requires" the lock. Locking a mutex is a stronger condition than requiring it, i.e. an operation locking a mutex requires it as well. Finally, a process cannot lock a mutex as long as an operation is executed which requires it. MoDeL furthermore allows the architect to specify whether the provider module or the employer component(s) is responsible for protecting the mutex.

A noticeable advantage of the explicit modeling of virtual resources with mutexes, apart from the possibility to specify fine-grained access control, is the ability to express inter-module synchronization schemes. Mutexes, as anything else in a component's interface, can be imported by other interfaces and may be used to formulate synchronization requirements there.

7 Summary

This paper summarizes some of the key concepts of the architecture description language MoDeL as introduced in [10]. Its basic properties are:

- The understanding of the contents and purpose of a software architecture is that of a blueprint for the system under consideration. Its core abstraction results from the distinction between an interface and its realization.
- The basic unit of modeling is a component that may range in size and complexity between a module and a complete software system. The abstraction provided by a component depends on its logical characteristics and purpose in the context of the system.
- The language itself is multiparadigmatic. It supports different kinds of abstractions for components and relationships, modeling on different levels of the logical hierarchy, and multiple levels of detail between pure logical structure and actual realization structure.
- Different views concerning static and dynamic as well as logical and physical properties are supported. The relationships between (elements of) these views are precisely defined.

References

1. Booch, G.: Object Oriented Analysis and Design with Applications. Benjamin/Cummings, Redwood City (1994)
2. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, Reading (1999)

3. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, Reading (1999)
4. Altmann, W.: A New Module Concept for the Design of Reliable Software. In: Raulefs, P. (ed.) Workshop on Reliable Software, pp. 155–166. Hanser-Verlag, Munich (1979)
5. Gall, R.: Structured Development of Modular Software Systems – The Module Graph as Central Data Structure. In: Proceedings of the Workshop on Graphtheoretic Concepts in Computer Science 1981, pp. 327–338. Hanser-Verlag, Munich (1982)
6. Lewerentz, C., Nagl, M.: Incremental Programming in the Large: Syntax-Aided Specification Editing, Integration, and Maintenance. In: Shriver, B. (ed.) Proceedings of the 18th Hawaii International Conference on System Sciences, Honolulu, vol. II, pp. 638–649 (1985)
7. Lewerentz, C.: Extended Programming in the Large within a Software Development Environment. ACM SIGSOFT Software Engineering Notes 13(5), 173–182 (1988)
8. Nagl, M.: Softwaretechnik: Methodisches Programmieren im Großen. Springer, Berlin (1990)
9. Börstler, J.: Programmieren-im-Großen: Sprachen, Werkzeuge, Wiederverwendung. Umeå University Report UMINF 94.10, Doctoral Dissertation, Aachen University of Technology, Umeå University (1994)
10. Klein, P.: Architecture Modeling of Distributed and Concurrent Software Systems. Doctoral Dissertation, Aachener Beiträge zur Informatik, Band 31, Wissenschaftsverlag Mainz in Aachen, Aachen (2001)
11. Kleppe, A.: MDA Explained, The Model Driven Architecture: Practice and Promise. Addison-Wesley, Reading (2003)
12. Kruchten, P.: Architectural Blueprints—The "4+1" View Model of Software Architecture. IEEE Software 12(6), 42–50 (1995)
13. Stevens, W., Myers, G., Constantine, L.: Structured Design. IBM Systems Journal 13(2), 115–139 (1974)
14. Perry, D., Wolf, A.: Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes 17(4), 40–52 (1992)
15. Parnas, D.: On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM 15(12), 1053–1058 (1972)
16. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, Reading (1995)
17. Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism. Computing Surveys 17(3), 471–522 (1985)
18. Kiczales, G.: Towards a New Model of Abstraction in Software Engineering. In: Yonezawa, A., Smith, B. (eds.) Proceedings of the International Workshop on New Models for Software Architecture 1992; Reflection and Meta-Level Architecture, Tokyo, pp. 1–11 (1992)
19. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
20. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 2 – Module Specifications and Constraints. Springer, Berlin (1990)
21. Spivey, J.: The Z Notation – A Reference Manual, 2nd edn. Prentice Hall, New York (1992)
22. Kohring, C.: Ausführung von Anforderungsdefinitionen zum Rapid Prototyping – Requirements Engineering und Simulation (RESI). Doctoral Dissertation, Aachen University of Technology. Shaker-Verlag, Aachen (1996)

23. Hutt, A. (ed.): Object Analysis and Design – Description of Methods. Wiley, New York (1994)
24. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs (1991)
25. Hoare, C.: Monitors: An Operating System Structuring Concept. Communications of the ACM 17(10), 549–557 (1974)