Achieving adaptability through separation and composition of concerns

Mehmet Aksit, Bedir Tekinerdogan, Lodewijk Bergmans
TRESE Project, Department of Computer Science, University of Twente,
P.O. Box 217, 7500 AE Enschede, The Netherlands.
email: { aksit | bedir | bergmans }@cs.utwente.nl
www server: http://wwwtrese.cs.utwente.nl

Abstract

This paper discusses separation and composition of concerns as a means for improving adaptability of object-oriented programs. Separation of concerns results in a weak coupling of the concerns and as such satisfies the need for increased flexibility and reusability. We will illustrate the separation of concerns mechanism for the conventional object model and set out the requirements for an enhanced and adaptable object model. We propose the composition filters model as a framework for language extensions. The composition filters model separates the basic application code from the more special purpose concerns such as synchronization, real-time constraints and multiple views. Its applicability to solving various modeling problems is briefly illustrated.

1 Separation and composition of concerns

Large scale and complex software systems have to incorporate and deal with a variety of special computing concerns such as synchronization, real-time behavior and coordinated behavior. Separating these concerns both at the conceptual and the implementation level is generally considered important to manage complexity and increase the adaptability [31][12]. In the (conventional) object-oriented (OO) model [36], the separation of concerns principle is supported basically in three ways:

1. By defining objects as the models of the real-world concepts, which are "naturally" separated from each other;
2. By separating the concerns of providing an abstract object interface and the implementation of it; and
3. By grouping functions together around objects so that functions which are less related are structurally separated from each other.

To be able to construct complex software systems, the separated concerns must be put together using minimum effort. The OO model provide various ways in composing concerns together:

1. In the implementation part of an object, the structure and the behavior of the nested implementation objects can be composed under the definition of the encapsulating object;
2. Both inheritance and delegation mechanisms define composition of behavior. For example, in inheritance, the operations defined within a sub-class is composed with the operations of its super-class(es);
3. By defining a set-of protocols among cooperating objects, the software engineer may create more complex system structures provided that each component (or object) fulfil the protocol specification.

2 Application Domain Concerns

Application domains may define additional concerns. Consider for example, an electronic mail object. This object provides operations for defining the sender, receiver and the content of the mail. In addition, various operations are defined to approve and deliver the mail to its destination. An important requirement here is that only the "system objects" are allowed to invoke the approve and deliver operations. In addition to the previously mentioned concerns, a mail object has therefore an additional concern: "multiple-views". Each mail object has a user-view and a system-view that restrict the operations of the mail object with respect to the clients of the mail object.

To implement the mail object, each concern must be mapped to an OO concept. For example, views can be implemented as operations. The operation approve, for instance, can be executed if the operation checkSystemView returns True. This implementation, however, blurs the separation of the multiple-views concern because the views are then realized within the operations of objects.

The problem appears if we want to extend/modify the mail object. For example, we may further partition the user view as sender and receiver views. We may extend the sender and receiver views to group-sender and group-receiver views. We may like to give a warning message if the same operation is invoked for the same mail object twice. In almost all these cases, the OO model cannot express these extensions without redefining the previously defined mail object. This is because the mail object cannot implement multiple-views on objects as a separate and composable concern.

Obviously, there can be many other concerns. For example, various synchronization constraints can be defined for the mail object. A request for send, for instance, can be delayed if the sender, receiver and the mail content have not been defined yet. Other possible concerns are for example, addressing the history information, evolution of behavior, coordinated behavior, security and reliability measures, real-time behavior, distribution aspects, etc. Since, the OO model may not separate these concerns adequately, the resulting programs are likely to be less adaptable and reusable. Several publications identified the composability problems for certain concerns such as atomic transactions [17], synchronization [27] and real-time specifications [6].

The proposed design patterns in [14] cannot solve these problems adequately because the composability features of the patterns are defined by the capabilities of the conventional OO model.

3 Extensions to the Object Model

One may define a composable model for a particular concern by identifying the orthogonal components and the composition operators for that concern. For example, the multiple view problem can be modeled by representing views as explicit concepts and by defining accept functions between views and the operations of the mail object. These additional concepts and operators have to be integrated with the OO model. This can be realized basically in two ways:

1. Using special language constructs: Many research proposals introduce new language constructs and/or computation models to tackle a given problem. For example, [21], [34], [18] and [23] introduce language constructs to model synchronization, real-time, coordinated bahavior and multiple-views concerns, respectively. The introduced language constructs must be uniformly integrated with the composability features of the underlying object model. Otherwise, the resulting programs cannot be fully composable. In most publications, expressiveness is the major concern and the proposed language constructs are hardly composable.
2. Meta-level programming: Meta-level programming can be used to solve specific concerns at a meta-level. Reflection techniques can be used to keep different levels consistent with each other. For example, views and accept functions can be defined at a meta-level without modifying the basic structure of the mail object. A number of papers presented meta-level solutions for various problems, such as distributed architectures [28], atomic transactions [32], concurrent programming [20], operating system structuring [37] and compiler design [25].The challenge here is to define reflection techniques which supports, in addition to the basic OO concerns, a large number of possible application defined concerns in a composable manner. Most research activities in this area do not address the composability issues of the proposed meta-level concerns.

Apart from our work, recently, a number of publications have appeared to address the composability problems in object-oriented modeling [30][29][13][19]. Recently, the concept of aspect-oriented programming (AOP) has been introduced which focuses on a basic categorization of concerns [22]. Aspect-Oriented Programming enables programmers to first express each of a system's aspects of concern in a separate and natural form, and then automatically combine those separate descriptions into a final executable form using a tool called an Aspect Weaver™.

We have extended the conventional OO model with the concept of composition filters (Cfs). We formalize four objectives that should be fulfilled when making extensions to a language model:

1. The language model must be rich enough to express major concerns such as synchronization, real-time specifications, control flow specifications, etcetera.
2. The adopted language mechanisms must be uniformly integrated with the conventional object model.
3. It must be possible to freely combine several independent concerns into a single object, whenever this combination is semantically meaningful.
4. Objects that are extended with the new concerns must be adaptable, extensible and reusable without causing inheritance anomalies.

The composition filters model is based on the following assumptions:

- The object-oriented model as defined by current methods and languages has many useful features and therefore it must be kept as an abstraction mechanism.
- To solve the modeling problems for different concerns, the object model must be enhanced.
- Since more than one problem can be experienced for the same object, enhancements must be specified independent from each other;
- Extensions have to be specified at the interface of objects, preferably in a consistent and declarative manner.

The composition-filters object model provides a mechanism for adding an open-ended range of concerns to object models without violating their basic mechanisms. Furthermore, it allows for independent specification of these concerns and the composability of objects. The composition filters approach can be seen as a specialization of the before mentioned AOP.

In the following sections we will give an overview of the composition filters model and the integration of composition filters with the conventional object model.

4 An overview of the composition-filters model

The composition filters model is a modular extension to the conventional object model as adopted e.g. by Smalltalk and C++. The behavior of a conventional object can be modified and enhanced through the manipulation of incoming and outgoing messages only. To achieve this, the kernel object is surrounded by a layer called the interface part. The resulting model and its components are shown in Figure 1.

The most significant components in the CF model are the *input filters* and *output filters*. A single filter specifies a particular manipulation of messages. Various filter types are available. The filters together compose the behavior of the object, possibly in terms of other objects. These other objects can be either *internal objects* or *external objects*. Internal objects are encapsulated within the composition filter object whereas external objects remain outside the composition filters object, such as globals or shared objects. The behavior of the object is a composition of the behavior of its internal and external objects.

In addition, –part of– the behavior of the object will be implemented by the kernel object, which is therefore also referred to as the implementation part. On the interface of the kernel object appear two types of methods: normal methods and *condition methods*. The normal methods may be invoked through messages, if the filters of the object allow this. Condition methods are essentially Boolean expressions that provide information about the state of the object. The condition methods are used by the filters to decide how to manipulate messages. As an example, a specific filter may reject messages, based on their properties or based on the state of the object.

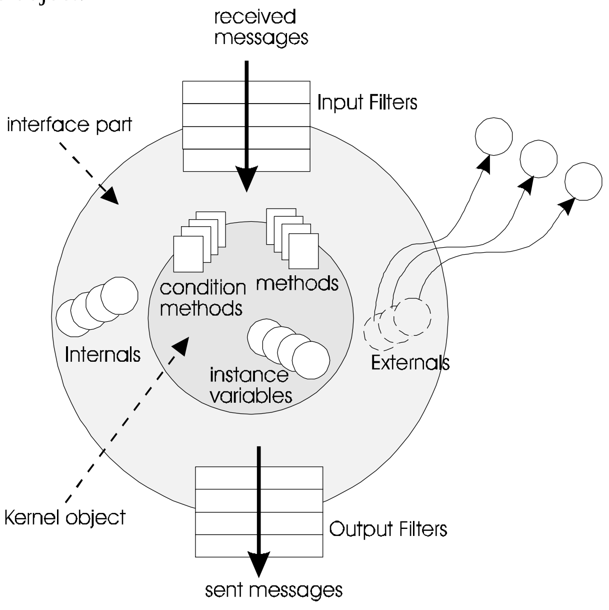

Figure 1. The components of the composition-filters model.

4.1 The principle of message filtering

We will explain the basic mechanism of message filtering by composition filters with the aid of Figure 2. The discussion focuses on input filters, but output filters work in exactly the same manner. The main difference is that output filters deal with sent messages instead of received messages.

To understand the schema the following should be kept in mind: filters are defined in an ordered set. A message that is received by an object is first reified, i.e.

a first-class representation of the message is created[1]. The reified message has to pass the filters in the set, until it is discarded or can be dispatched. Dispatching means that the message is activated again, for example to start the execution of a method body, or to be delegated to another object. Each filter can either accept or reject a message. The semantics associated with acceptance or rejection depend on the type of the filter.

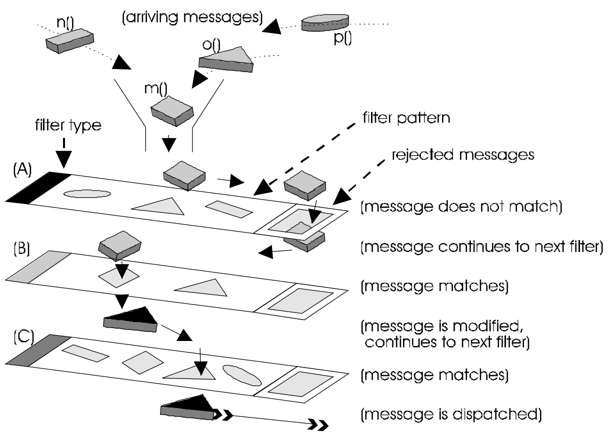

Figure 2. An intuitive schema of message filtering.

Figure 2 visualizes the processing of messages by three filters, A, B and C. An object can receive a variety of messages, in the figure exemplified by m(), n(), o() and p(). All received messages are subject to manipulation by the successive filters. Different types of filter exist for different manipulations on messages. Each filter tries to match messages based on a specific pattern. A common syntax is used by all filters for defining these patterns. The matching process can be defined in terms of message properties, but may also depend on the current state of the object.

We follow the message m() as it passes through the filters. In Figure 2, message m() does not match with the pattern defined by filter (A). Thus, the message is *rejected* by this filter. In the example, the rejected message is simply passed on to the next filter.

The message will then be evaluated by filter (B). The pattern that is defined by this filter matches with the message. This is referred to as *acceptance* of the message by the filter. This initiates a particular action, that depends on the filter type: the message may be manipulated and modified. In the example of filter (B), the message is modified (designated in the figure by its changed shape and color), and then passed on to the next filter.

For the last filter in the example, filter (C), the pattern also matches the message. The acceptance of the message in this case causes the message to be dispatched, for

[1] Composition filters thus apply a form of message reflection.

example to a local method of the object. The message itself contains information that determines how it should be dispatched (i.e. the target object and the message selector).

In general, every filter set should contain a filter that causes messages to be dispatched, as this is the only means to trigger the execution of a method. For output filters, dispatching means that the message is submitted to the target object. Note that upon its reception by the target object, the message must first pass the input filters of the target object.

In summary, each filter specification consists of a pattern definition and a filter type. Messages are matched against the pattern, then the filter type determines the action to be performed upon acceptance, respectively rejection. For a more detailed description of the composition filters model, we refer to [9], [24], or various other papers that each discuss a specific application of composition filters[1].

5 Solving modeling problems with filters

In order to provide a clear motivation for adopting composition filters, in this section we briefly describe a number of modeling problems which have been experienced in several practical pilot projects [4]. For each of these modeling problems, we outline the solution that can be provided by adding one or more filters to a kernel object.

5.1 Multiple views

Not all operations provided by an object should be accessible to each object that uses its services. Therefore it is desirable to define interfaces for an object that differentiate between clients, that is, between the senders of a message. For example, a public mailbox should make a distinction between a postman and others, since everybody is allowed to put a letter in it, but only a postman is allowed to empty the mailbox. Interfaces may also change depending on the internal state of the object; for example, the mailbox cannot be opened –not even by the postman– while it is locked.

We have coined the term *multiple views* in [3] to designate this problem. In a conventional object-oriented language such as Smalltalk, multiple views can only be realized by inserting explicit checks in all the methods of an object. The resulting mixing of concerns causes problems when trying to reuse and extend objects with multiple views.

An *Error* filter allows for ‘preconditions’ on messages, based on both the properties of the message (such as the identity of the sender) and the state of the object. Views are defined by condition methods, and the *Error* filter defines the

[1] At http://wwwtrese.cs.utwente.nl/~sina/ a tutorial on the composition filters model as adopted by the programming language Sina can be found.

mapping from the views to sets of messages. Several views can be combined or added later in subclasses.

5.2 Dynamic inheritance and delegation

Dynamic inheritance or delegation means that the inheritance hierarchy (delegation structure) is not fixed, but that an object can specify a set of superclasses (delegated objects) from which it may possibly inherit (delegate to) [3]. Dynamic inheritance or delegation can be needed if an application must be able to adapt its behavior due to: performance reasons or space requirements, different clients or contexts, or even to different hardware, as for example in distributed systems.

By attaching a *Dispatch* filter to an object we can provide dynamic inheritance and/or delegation. The *Dispatch* filter will forward received messages to different target objects, depending on results of condition methods. If the target object is encapsulated within the object itself, inheritance is simulated [1]. If the target object resides outside the encapsulation boundary of the receiver object, this becomes a delegation mechanism. Since the *Dispatch* filter can forward messages to different objects, *multiple* inheritance and delegation is supported as well. The *self* pseudovariable is retained as it is always the original receiver of the message.

Condition methods capture the state of an object, which may change at run-time. The results of condition methods also affect the dispatching process, and thus inheritance and delegation may change dynamically.

5.3 Coordinated behavior

Coordinated behavior can be encountered for example in distributed control systems, in which several distinct units, such as controlling algorithms, sensors and actuators, must work together in order to keep the controlled system in a consistent state.

The conventional object-oriented models do not provide high-level mechanisms to abstract coordinated behavior among several objects since the message passing semantics only involve two partner objects and the message communication is invisible at the application level. To implement coordinated behavior, the coordination-related application code must be spread over several objects which means that the application becomes more complex, less reusable, while its interaction semantics are more difficult to understand, verify and enforce.

The coordinated behavior problem [5] can be solved by attaching a *Meta* filter to the kernel object. A *Meta* filter captures incoming messages and forwards them as first class objects to a so-called *abstract communication object* which implements the coordinated behavior. The abstract communication object can be shared by any number of objects to coordinate and manipulate their communication. An important characteristic of this approach is that it allows for defining and reusing hierarchies of abstract communication objects.

5.4 Reuse versus synchronization constraints

In the previous sections we already introduced the problem of inheritance anomalies, which frequently occurs when adding synchronization constraints to objects. This problem may occur for instance when synchronization code is mixed with application code (i.e. embedded inside the method code). Then changing the synchronization is impossible without affecting the application code. Another source for inheritance anomalies comes from the lack of decomposability of synchronization specifications in some languages. It is then hard to add a new synchronization constraint, or to partly reuse or redefine the synchronization specification in a subclass.

We can use a *Wait* filter to specify synchronization constraints while avoiding inheritance anomalies [10]. The synchronization constraint condition is then specified by condition methods as an abstraction of the state of the object. A *Wait* filter defines a mapping from these conditions to the messages to which the synchronization constraint applies. Reuse and extension of (objects with) synchronization is possible, without unnecessary redefinitions.

5.5 Reuse versus real-time constraints

In real-time environments, at least some of the classes in the system impose real-time constraints upon the execution of methods. When such classes are reused in other applications, changes to either the application requirements or to the real-time constraint specifications in subclasses may result in excessive redefinitions of superclasses whereas this would be intuitively unnecessary. This so called real-time specification anomaly can arise when real-time specifications are mixed with the application code, or when specifications are not polymorphic, i.e. they cannot be used for more than one method, or when independently defined but related specifications are composed together.

A *RealTime* filter [6] can define or modify deadlines as well as other scheduling attributes of an execution. This is achieved by letting messages carry scheduling attributes[1]. At the interface of objects, *RealTime* filters can modify these attributes for selected messages and under selected circumstances (as controlled by condition methods). The filters specify a mapping from the real-time constraints to sets of messages. The resulting decoupling ensures that the real-time specification anomaly will not occur.

6 Discussion and Conclusions

[1] At least conceptually; another view would be that the scheduling attributes are associated with threads.

Although the conventional object model as adopted by object-oriented languages like Smalltalk and C++ have widely shown their benefits in building reusable and extensible software it still suffers from a variety of modeling problems. The separation of concerns principle has been identified as a useful mechanism to provide solutions for these problems. The benefits of the separation of concerns principle are better management of complexity, better understandability and increased adaptability. Since the object model may not separate the different application concerns adequately, the resulting programs are likely to be less adaptable and reusable.

The composition filters (CFs) has been proposed as a modular and orthogonal extension to the conventional object model to cope with the modeling problems in an elegant way and to provide the necessary adaptability of the programs. CFs can be attached to objects expressed in different languages [16][11]; the conventional OO model can be used to implement the basic concepts and each additional concern can be expressed as a CF.

Since different types of CFs show a similar structure, we have been investigating more primitive compositional structures than the ones provided by the CF model. We modeled the currently defined CFs using so-called message manipulators [33]. Message manipulators define logical operators for the received (and sent) messages to (or from) an object. These operators are for example, AND, Conditional-AND, OR, Conditional-OR, and Sequential manipulators. Each manipulator can be further decomposed by using sub-manipulators, etc. A "terminal manipulator" is composed of a constraint checker and accept and reject handlers. Constraint checkers are applied to messages. The accept and reject handlers are first class objects and represent the semantics of different concerns. Our conclusion here is that defining logical message composition operators with extensible semantics is a promising way for composing separated concerns together.

We believe that the concept of composition of different concerns must be also applied during the software development process. Propagation Patterns [26], for example, separate the concern of defining algorithms and class structures from each other. During software development, the so-called software artifacts are generated in various formats, from informal textual information to executable object-oriented programming concepts. Composability of design models require explicit representation of software artifacts in a composable way. In our recent work [8], we have applied fuzzy-logic based techniques to represent and compose various software artifacts. In contrast to deterministic object-level compositions, we found the fuzzy-logic based reasoning techniques more appropriate for representing design level concerns because fuzzy-logic can deal with design uncertainties.

The so-called software architecture definition languages (ADLs) [15] are used to model and structure higher-level design concepts. Most architecture definition languages, however, do not adequately address the issue of evolution and composition of different architectural concepts [12]. In this direction, we are currently defining an ADL based on the concept of composition of specializations of knowledge domains [7].

References

1. Aksit, M & Tripathi, A: Data Abstraction Mechanisms in Sina/ST, Proc. of the OOPSLA '88 Conference, ACM SIGPLAN Notices, Vol. 23, No. 11, November 1988, pp. 265-275.
2. Aksit, M., Dijkstra, J.W., & Tripathi, A: Atomic Delegation: Object-oriented Transactions, IEEE Software, Vol. 8, No. 2, March 1991, pp 84-92.
3. Aksit, M. & Bergmans, L: Obstacles in Object-Oriented Software Development, Proceedings OOPSLA '92, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, pp. 341-358
4. Aksit, M., Bergmans, L., & Vural, S: An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach, Proc. of the ECOOP '92 Conference, LNCS 615, Springer-Verlag, 1992, pp. 372-395.
5. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., & Yonezawa, A: Abstracting Object-Interactions Using Composition-Filters, In Object-based Distributed Processing, R. Guerraoui, O. Nierstrasz and M. Riveill (eds), LNCS 791, Springer-Verlag, 1993, pp 152-184.
6. Aksit, M., Bosch, J., Sterren, W. v.d., & Bergmans, L: Real-Time Specification Inheritance Anomalies and Real-Time Filters, Proc of the ECOOP '94 Conference, LNCS 821, Springer Verlag, July 1994, pp. 386-407.
7. Aksit, M., Marcelloni, F., Tekinerdogan, B., Vuijst, C., & Bergmans, L.: Designing Software Architectures as a Specializations of Knowledge Domains, University of Twente, Memoranda Informatica 95-44, December 1995.
8. Aksit, M., & Marcelloni, F: Minimizing Quantization Error and Contextual Bias Problems of Object-Oriented Methods by Applying Fuzzy-Logic Techniques, University of Twente, 1996.
9. Bergmans, L: Composing Concurrent Objects, Ph.D. thesis, University of Twente, The Netherlands, 1994.
10. Bergmans, L. & Aksit, M:Composing Synchronization and Real-Time Constraints, Journal of Parallel and Distributed Computing , No. 36, June 1996, pp. 32-52.
11. Dijk, W van., & Mordhorst, J: CFIST, Composition Filters in Smalltalk, Graduation Report, HIO Enschede, The Netherlands, May 1995.
12. Dijksta, E. W: The Structure of the T.H.E. Multiprogramming System, Communications of the ACM, No. 11, pp. 341-346, 1968
13. Forman, I., Danforth, S., & Madduri, H: Composition of Before/After Metaclasses in SOM, Proc. of the OOPSLA '94 Conference, ACM SIGPLAN Notices, Vol. 29, No. 10, October 1994, pp. 427-439.
14. Gamma, E., Helm, R., Johnson, R., & Vlissides, J: Design patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
15. Garlan, D., Allen, R., & Ockerbloom, R: Architectural Mismatch or, Why it's Hard to Build Systems out Existing Parts, Proc. of the 17th. Int. Conf. on Software Engineering, April 1995.
16. Glandrup, M: Extending C++ Using the Concepts of Composition Filters, M.Sc. Thesis, University of Twente, November 1995.
17. Guerraoui, R: Atomic Object Composition. Proc of the ECOOP '94 Conference, Springer-Verlag, 1994, pp. 118-138.
18. Holland, I: Specifying Reusable Components Using Contracts, Proc. of the ECOOP '92 Conference, LNCS 615, Springer-Verlag, 1992, pp. 287-308.
19. Hürsch, W., & Lopes, C: Separation of Concerns, Northeastern University, February 1995.

20. Ichisugi, Y., Matsuoka, S., & Yonezawa, A: A Reflective Object-Oriented Concurrent Language Without a Run-Time Kernel, Int. Workshop on New Models for Software Architecture'92, Reflection and meta-Level Architecture, Yonezawa & Smith (eds), November 1992, pp. 24-35.
21. Kafura, D.G., & Lee, K.H: ACT++: Building a Concurrent C++ with Actors, J. of Object-Oriented Programming May/June 1990, Vol. 3, No. 1, pp. 25-37.
22. Kiczales, G., Irwin, J., Lamping, J., Loingtier, J., Lopes, C,V., Maeda, C., Mendhekar, A : Aspect Oriented Programming, in: Max Muehlhaeuser (general editor) et al.: Special Issues in Object-Oriented Programming dpunkt Heidelberg, 1996.
23. Kiessling, H., & Kruger, U: Sharing Properties in a Uniform Object Space. Proc. of the ECOOP'95 Conference, LNCS 952, Springer Verlag, 1995, pp. 424-448.
24. Koopmans, P: On the Definition and Implementation of the Sina/st Language, M.Sc. Thesis, University of Twente, The Netherlands, July 1995
25. Lamping, J., Kiczales, G., Rodriguez, L., & Ruf, E: An Architecture for an Open Compiler, Int. Workshop on New Models for Software Architecture'92, Reflection and meta-Level Architecture, Yonezawa & Smith (eds), November 1992, pp. 95-106.
26. Lieberherr, K: Adaptive Object-Oriented Software the Demeter Method with Propagation Patterns., PWS Publishing Company, 1995.
27. Matsuoka, S., & Yonezawa, A: Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, in Research Directions in Concurrent Object-Oriented Programming, (eds.) G. Agha, P. Wegner and A. Yonezawa, MIT Press, April 1993, pp. 107-150.
28. McAffer, J: Meta-level Programming in CodA, Proc. of the ECOOP'95 Conference, LNCS 952, Springer Verlag, 1995, pp. 190-214.
29. Mullet, P., Malenfant, J., & Cointe, P:Towards a Methology for Explicit Composition of MetaObjects, OOPSLA'95 Conference Proceedings, ACM Sigplan Notices, Vol. 30, No. 10, October 1995, pp. 316-330.
30. Nierstrasz, O., & Tsichritzis, D (eds): Object-Oriented Software Composition, Prentice Hall, 1995.
31. Parnas, D.L: On the criteria to be used in decomposing systems into modules, Communications of the ACM 15, 12, 1972, pp. 1053-1058.
32. Stroud, R., & Wu, Z: Using Metaobject Protocols to Implement Atomic Data Types, Proc. of the ECOOP'95 Conference, LNCS 952, Springer Verlag, 1995, pp. 168-189.
33. Stuurman, C: Techniques for Defining Composition-Filters Using Message Manipulators, M.Sc. Thesis, University of Twente, August 1995.
34. Takashio, K., & Tokoro, M: DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems, Proc of the OOPSLA '92 Conference, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, pp. 276-294.
35. Vestal, S: A Cursory Overview and Comparision of Four Architecture Description Languages, Honeywell Technology Center, Minneapolis, February 1993.
36. Wegner, P: Dimensions of Object-Based Language Design, Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 168-182
37. Yokote, Y: The Apertos Reflective Operating System: The concept and its Implementation, Proc of the OOPSLA'92 Conference, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, pp. 414-434.