Modular and Composable Extensions to Smalltalk using Composition Filters

Lodewijk Bergmans, Bedir Tekinerdoğan & Mehmet Akşit

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 – current and future trends in computer science require extensions to Smalltalk. Rather than arguing for particular language mechanisms to deal with specific requirements, in this position paper we want to make a case for two requirements that Smalltalk extensions should fulfill. The first is that the extensions must be integrated with Smalltalk without violating its basic object model. The second requirement is that extensions should allow for defining objects that are still adaptable, extensible and reusable, and in particular do not cause inheritance anomalies. We propose the composition filters model as a framework for language extensions that fulfills these criteria. Its applicability to solving various modeling problems is briefly illustrated.

1. Motivation and approach

Despite the severe competition in the language market, Smalltalk remains as one of the major players among commercial object-oriented languages. This may be attributed to Smalltalk's pureness as an object-oriented language, its rich programming environment and its portability. To be able to continue with this success, however, Smalltalk has to evolve to cope with the trends in modern computing and software development.

1.1 Context

To give an indication of the kind of issues that modern programming languages must deal with, we mention a few trends in computing and application development:

- an increasing interest in embedded systems, which requires dealing with real-time issues and synchronization.
- distribution of applications across a network (e.g. based on CORBA); as a result mechanisms for concurrency control, various communication protocols, distributed transactions etc. are required.
- separate specification of workflow and work procedures, independent from the application code.
- systems have to adapt dynamically to changing circumstances.

It is important to note that more such issues are likely to appear, either due to advancements in computing, or to the requirements of specific application domains. In other words, this list is open-ended.

1.2 Extending Smalltalk

To give an impression of the kind of extensions that we have in mind, we list a number of them:

- synchronization of concurrent activities
- real-time constraint specifications
- atomic transactions
- tailorable message passing semantics
- objects representing communication protocols, so that these can be abstracted and reused
- separate and explicit control flow specifications
- dynamically adaptable (multiple) inheritance and delegation.

Note that each of these extensions deals with one specific aspect of computation.

A big question to be answered first is: do we really *need* to extend the language? Can we not deal with these issues through a library approach? Extending a language for every new feature that is desired has proven quite problematic, as e.g. C++ demonstrates. We will briefly discuss when extensions are required.

The basic object-oriented model [Wegner 87] suffers from several modeling deficiencies (see e.g. [Aksit 92b]). Most of these problems are due to the fact that object-oriented models are not capable of expressing certain aspects of applications *in a reusable way*. The question is thus not only whether the language can express an aspect at all. But also to what extent it is possible to add new aspects to objects, and to extend and/or reuse such objects.

As an example, consider the synchronization of concurrent threads. Smalltalk offers a semaphore class that allows for embedding semaphore operations within the bodies of methods. However, it has been explained in the literature [Matsuoka 93][Bergmans 94b] that this approach towards synchronization does not behave properly when trying to reuse and extend classes with synchronization. The term *inheritance anomaly* has been coined to designate the problem that programmers are forced to do superfluous overriding of inherited methods. An interesting property of the inheritance anomaly is that it is caused by the specifics of the object model (as adopted by a programming language).

Our experience has shown that many improvements to the object-oriented model are required in order to be able to deal with modeling problems [Aksit 92b]. We have developed the language-independent *composition filters model* as a generic approach to overcome these.

1.3 Requirements for extensions

We formalize four objectives that should be fulfilled when making extensions to Smalltalk:

1. Smalltalk must be rich enough to express aspects such as synchronization, real-time specifications, control flow specifications, etcetera.

2. The adopted language mechanisms must be uniformly integrated with Smalltalk's object model.
3. It must be possible to freely combine several independent aspects into a single object, whenever this combination is semantically meaningful.
4. Objects that are extended with the new aspects must be adaptable, extensible and reusable without causing inheritance anomalies.

The composition-filters object model provides a mechanism for adding an open-ended range of aspects to object models without violating their basic mechanisms. Furthermore, it allows for independent specification of these aspects and the composability of objects.

2. The composition filters extension

In order to meet the requirements for the extensions we have extended the conventional OO model with the concept of composition filters (CFs). 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[1].
- ♦ To solve the modeling problems for different aspects, 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.

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

2.1 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. The behavior of a Smalltalk object can be modified and enhanced through the manipulation of incoming and outgoing messages only. To achieve this, the Smalltalk 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 Smalltalk object, which is therefore also referred to as the implementation part. On the interface of the Smalltalk 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.

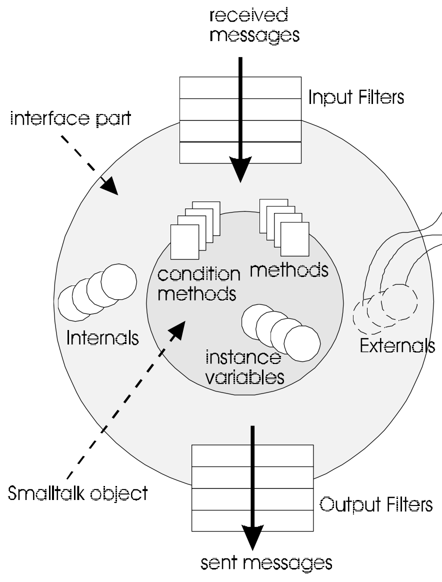

Figure 1. The components of the composition-filters model.

2.2 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.

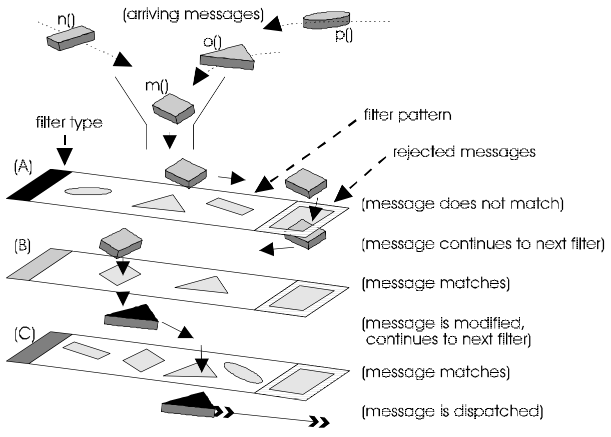

Figure 2. An intuitive schema of message filtering.

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[2]. The reified message has to

1 Grady Booch has even argued in [Booch 96] that object-orientation as an abstraction paradigm is here to stay indefinitely..

2 Composition filters thus apply a form of message reflection [Ferber 89].

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 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 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 [Bergmans 94a], [Koopmans 95], or various other papers that each discuss a specific application of composition filters[3].

3. 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 [Aksit 92b]. For each of these modeling problems, we outline the solution that can be provided by adding one or more filters to a Smalltalk object.

3.1 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 [Aksit 92a] 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 mapping from the views to sets of messages. Several views can be combined or added later in subclasses.

3.2 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) [Aksit 92a]. 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 [Aksit 88]. 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.

3.3 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.

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

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 [Aksit 93] can be solved by attaching a *Meta* filter to the Smalltalk 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.

3.4 Reuse versus synchronization constraints

In section 1.2 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 [Bergmans 94b]. 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.

3.5 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 [Aksit 94] can define or modify deadlines as well as other scheduling attributes of an execution. This is achieved by letting messages carry scheduling attributes[4]. 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.

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

4. Composable extensions

In the previous section we have outlined a range of common modeling problems and given an indication how composition filters can address these problems. However, as indicated in section 1.3, an important requirement for extensions to Smalltalk is that they are *composable*. This involves two properties:

I. It should be possible to combine solutions to problems such as synchronization, dynamic inheritance and delegation, real-time specifications, multiple views, etcetera within a single object.
II. It should be possible to compose new objects from existing ones (e.g. through multiple inheritance) while retaining the semantics of the composed objects.

The constraint that must be fulfilled in both cases is that the combined aspects do not interfere semantically.

Figure 3 illustrates an example of the composition of synchronization and real-time constraints along an inheritance hierarchy. The hierarchy consists of 5 classes, where each class introduces one or more synchronization constraints (symbolized by $\boxed{1}$, $\boxed{2}$, $\boxed{3}$ & $\boxed{4}$) or real-time constraints (symbolized by ❶, ❷ & ❸).

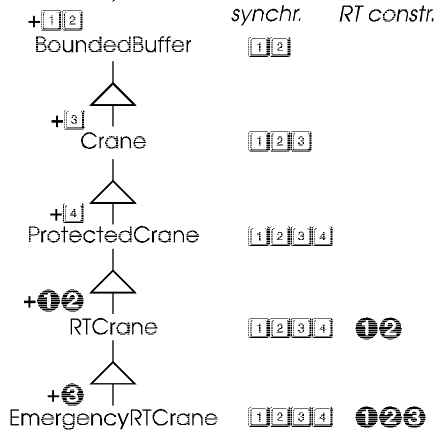

Figure 3. An inheritance hierarchy illustrating the composition of synchronization and real-time constraints

Classes RTCrane and EmergencyRTCrane show that different aspects can be composed within a single object. The hierarchy also illustrates that the aspects can be reused and extended by composing an existing class definition with a new subclass specification. The topic of composing synchronization constraints with real-time constraints and the possibly resulting semantic conflicts are discussed in detail in [Bergmans 95]. This article also describes the solutions offered through the application of the *Wait* and *RealTime* composition filters.

In general, we can combine arbitrary filter specifications in a single object specification. However, the resulting composition should also be semantically meaningful; this depends on the application characteristics. The composition filters model also supports the composition of new objects from existing ones, thereby combining their respective characteristics.

5. Conclusion

5.1 Implementation of composition filters

So far, we have developed and tested the following implementations of the composition filters model:

- ♦ An implementation of our research language *Sina*, which adopts composition filters, on top of VisualWorks. This includes the implementations of the *Dispatch*, *Error*, *Wait* and *Meta* filters [Koopmans 95].
- ♦ A prototype implementation called CFIST that adds composition filters specifications to Smalltalk classes [Dijk 95].
- ♦ A prototype implementation that adds composition filters specifications to C++ [Glandrup 95]. Currently, this features no C++ preprocessor, and therefore C++ message invocations must be replaced explicitly by macro calls.

The most important implementation issue is the reification of messages; each message invocation from and to an object with filters must be captured and reified. This is difficult to achieve in a way that is both transparent and efficient.

The efficient implementation of message filtering still remains a challenge; objects with composition filters can in principle exhibit a very dynamic behavior, for instance based on the state of the object. In general, this would make it very difficult to create efficient code. However, an interesting feature of composition filters is the declarative approach towards their specification. This allows for reasoning about filter specifications and makes it easy, for instance, to find out which behavior is fixed and which behavior is variable. Obviously, the fixed behavior can then be optimized easily, e.g. through inlining. For the variable behavior, dynamic optimization techniques e.g. as adopted by SELF [Hölzle 94] [Chambers 89] are required.

5.2 Related work

We mention the e*ncapsulators* framework [Pascoe 86] as a Smalltalk-based representative of related work. This framework offers an approach that is similar to the ideas motivating the composition-filters model: an application object can be surrounded with a layer that intercepts messages that are sent to the object and the replies of those messages. The encapsulators are special objects that implement this layer. An encapsulator defines a pre-action, that is executed upon message reception, and a post-action, that is executed when the result of the message is returned.

The main differences with the composition-filters model are that the actions in encapsulators are monolithic Smalltalk method implementations with no further fixed structure. The composition-filters model aims at providing abstractions to manage complex object behavior in such a way that the composability of aspects and objects is achieved.

Some design patterns, such as *Adapter*, *Decorator*, *Mediator* and *State* from [Gamma 95] address issues that are also covered by some applications of composition filters. However, they cannot completely overcome the inherent modeling difficulties in the conventional object models. For example, design patterns often imply cascading of interface updates, due to the aggregation-based nature of most patterns.

5.3 Future work

We are currently involved in adding composition filters to the Orbix™ implementation of CORBA IDL, and are considering to extend the Java programming language with filters. On top of our agenda is the modification of a Smalltalk virtual machine in order to provide an efficient implementation of message reification.

Furthermore, we intend to build a visual manipulation tool for specifying composition filters. Due to the simple syntax and the declarative nature of composition filters specifications, a very user-friendly filter specification through visual manipulation only is feasible. As an extension to e.g. Smalltalk, the full power of composition filters would then become available without the need for any coding and additional language syntax. The visual programming approach also fits nicely with current state-of-the-art Smalltalk environments such as Visual Age and the PARTS Workbench.

5.4 Summary

We will summarize the main points of this paper:

- ♦ When designing extensions for expressing certain aspects in Smalltalk, we must assure that the extensions are uniformly integrated with Smalltalk's object model, that all aspects can be freely combined, and that the extended objects remain adaptable, extensible and reusable.
- ♦ We propose the composition filters model as a modular extension to Smalltalk, leaving the Smalltalk object model unmodified.
- ♦ The composition-filters model offers reflection on messages in a declarative way and with a consistent notation, while providing open-endedness so that it can be applied to a range of application-domains. This model replaces numerous language constructs that would be required to offer similar –but limited– functionality when extending the language itself.
- ♦ The composition filters model provides solutions to a range of modeling problems. In section 3 a number of the problems and the solution approach are described.
- ♦ The solutions to these problems are composable, both within a single object, and through the composition of multiple objects into a single one[5]. This was discussed in section 4.
- ♦ Composition filters can be attached to objects expressed in different languages, including C++ [Glandrup 95] and Smalltalk [Dijk 95]. An important property of this approach is that an application can be developed completely in pure Smalltalk. Filters need only be added for classes that need additional expression power such as dynamic inheritance, synchronization, etcetera.

We would like to conclude with the remark that the applicability of composition filters has been established through pilot studies for a range of application domains, in particular in the context of designing application frameworks (e.g. [Tekinerdogan 94], [Vuijst 95]).

[5] A demonstration of the composition filters model that illustrates the composition of multiple aspects is scheduled during OOPSLA '96.

References

[Aksit 88] M. Aksit and A. Tripathi, Data Abstraction Mechanisms in Sina/ST, Proc. of the OOPSLA ‘88 Conference, ACM SIGPLAN Notices, Vol. 23, No. 11, November 1988, pp. 265-275.

[Aksit 91] M. Aksit, J.W. Dijkstra and A. Tripathi, *Atomic Delegation: Object-oriented Transactions*, IEEE Software, Vol. 8, No. 2, March 1991, pp 84-92.

[Aksit 92a] M. Aksit, L. Bergmans and S. Vural, *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.

[Aksit 92b] M. Aksit and L. Bergmans, *Obstacles in Object-Oriented Software Development*, Proceedings OOPSLA '92, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, pp. 341-358

[Aksit 93] M. Aksit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa, *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.

[Aksit 94] M. Aksit, J. Bosch, W. v.d. Sterren and L. Bergmans, *Real-Time Specification Inheritance Anomalies and Real-Time Filters*, Proc of the ECOOP ‘94 Conference, LNCS 821, Springer Verlag, July 1994, pp. 386-407.

[Bergmans 94a] L. Bergmans, *The Composition Filters Object Model*, proceedings of the RICOT symposium ‘*Enabling Objects for Industry*’, June 30, 1994

[Bergmans 94b] L. Bergmans. *Composing Concurrent Objects*, Ph.D. thesis, University of Twente, The Netherlands, 1994.

[Bergmans 95] L. Bergmans and M. Aksit, *Composing Synchronization and Real-Time Constraints*, University of Twente, Memoranda Informatica 95-41, (to be published in Journal of Parallel and Distributed Computing September 1996), December 1995.

[Booch 96] G. Booch, *The End of Objects and the Last Programmer*, http://www.rational.com/pat/tech_papers/tp47.html

[Chambers 89] C. Chambers, D. Ungar & E. Lee, *An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes*. Proc. of the OOPSLA ‘89 Conference, ACM SIGPLAN Notices, 24(10), October 1989, pp. 49-70.

[Dijk 95] W. van Dijk and J. Mordhorst, *CFIST, Composition Filters in Smalltalk*, Graduation Report, HIO Enschede, The Netherlands, May 1995.

[Ferber 89] J. Ferber, *Computational Reflection in Class-Based Object-Oriented Languages*, Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 317-326

[Gamma 95] E. Gamma, R. Helm, R. Johnson & J. Vlissides, *Design patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[Glandrup 95] M. Glandrup, *Extending C++ Using the Concepts of Composition Filters*, M.Sc. Thesis, University of Twente, November 1995.

[Hölzle 94] U. Hölzle, *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*, Ph.D. thesis, Dept. of Computer science, stanford University, August 1994.

[Koopmans 95] P. Koopmans. *On the Definition and Implementation of the Sina/st Language*, M.Sc. Thesis, University of Twente, The Netherlands, July 1995

[Matsuoka 93] S. Matsuoka and A. Yonezawa, *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

[Pascoe 86] G.A. Pascoe, *Encapsulators: A New Software Paradigm in Smalltalk-80*, Proc. of the OOPSLA '86 Conference, ACM SIGPLAN Notices, Vol. 21, No. 11, November 1986, pp. 341-346

[Tekinerdogan 94] B. Tekinerdogan, *The Design of an Object-Oriented Framework for Atomic Transactions*, Msc thesis, University of Twente, Dept. of Computer Science, The Netherlands, 1994

[Vuijst 94] C. Vuijst, *Design of an Object-Oriented Framework for Image Algebra*, Msc thesis, University of Twente, Dept. of Computer Science, The Netherlands, 1994

[Wegner 87] P. Wegner, *Dimensions of Object-Based Language Design*, Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 168-182