COMPOSING SOFTWARE FROM MULTIPLE CONCERNS: A MODEL AND COMPOSITION ANOMALIES

Lodewijk M.J. Bergmans (`lbergmans@acm.org`)
Mehmet Aksit (`aksit@cs.utwente.nl`)
`http://trese.cs.utwente.nl`
TRESE GROUP – FACULTY OF COMPUTER SCIENCE, CENTRE FOR TELEMATICS AND INFORMATION TECHNOLOGY (CTIT)
UNIVERSITY OF TWENTE, P.O. BOX 217, 7500 AE, ENSCHEDE, THE NETHERLANDS

Abstract—Constructing software from components is considered to be a key requirement for managing the complexity of software. Separation of concerns makes only sense if the realizations of these concerns can be composed together effectively into a working program. Various publications have shown that composability of software is far from trivial and fails when components express complex behavior such as constraints, synchronization and history-sensitiveness. We believe that to address the composability problems, we need to understand and define the situations where composition fails. To this aim, in this paper we (a) introduce a general model of multi-dimensional concern composition, and (b) define so-called *composition anomalies*.

1. INTRODUCTION

In this paper we discuss software *composition schemes*: a composition scheme is the conceptual model or mechanism for composition in a computation model or programming language. Composability is a key requirement for any language or computation model that intends to address multi-dimensional separation of concerns or aspect-oriented programming[1]: separation of concerns/aspects makes only sense if these can be composed together effectively into a working program. In this paper we propose a general model of multi-dimensional software, consisting of what we view as key ingredients for modeling such software.

However, the composability of software is far from trivial (see e.g. [Bergmans 97, 99]), and fails in many cases. We coin the term *composition anomaly* to describe a specific problem that causes a composition scheme to fail. Section 2 introduces our software composition model and illustrates it with a simple example. In section 3 the notion of composition anomaly is explained and defined. We illustrate the problem by mapping our example problem to an object-oriented implementation. We conclude this paper in section 5.

[1] The distinction between the two is mostly a matter of terminology, but until now, aspect-oriented programming (AOP) as illustrated by AspectJ^{TM} has focused on mechanisms that support a clear conceptual distinction between aspect- and base-level, whereas multi-dimensional separation of concerns is generally used in a more abstract and general way, e.g. including cross-lifecycle concerns.

2. A MODEL OF SOFTWARE COMPOSITION

In this section we present a general model for studying and reasoning about software composition. It relies on three fundamental observations about large and/or complex (software) systems:

- Complex (software) systems are modeled and constructed by merging a number of different dimensions, or views, of the system: multiple dimensions of concerns [Tarr 99, Bergmans 96a].
- Complex systems usually take the shape of *hierarchic systems*; systems that are composed of interrelated subsystems, each of the latter being in turn hierarchic in structure, until the lowest, atomic, level is reached [Simon 96].
- A useful model of software enables one to reason about the model instantiations. This requires the specification of the semantics and properties of the basic building blocks. Preferably, there exists well-defined models for the relevant concerns.

The following subsections will deal with each one of these observations, respectively:

2.1 MULTIPLE DIMENSIONS OF CONCERNS

For example in building architecture, one can distinguish many concerns or views, such as the floor plan, construction stability, heat characteristics, electrical wiring, plumbing, andsoforth separately. These can be modeled and specified largely independently, with a number of dependencies between them. The resulting building merges all these views. We term this as multiple dimensions of concerns, which can be visualized as follows:

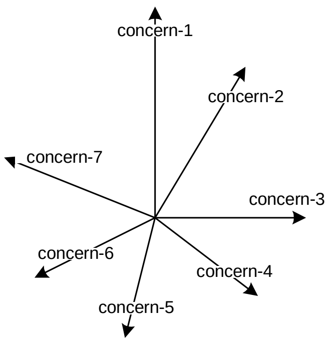

Figure 1. Multiple dimensions of concerns span a space where each point denotes a specific combination of concerns.

Each concern dimension consists of a number of concern instantiations. For example, a bounded buffer object may consist of concerns from the dimensions *data*, *behavior*, *synchronization* and *memory management*. For instance the data dimension might consist of the concerns {*storage*, *head*, *tail*}, and the synchronization dimension of the concerns {*EmptySync*, *FullSync*}.

The crucial issue in multiple dimensions of concerns is that in principle, all concerns in all concern dimensions may be coupled to one or more other concerns from any concern dimensions[1]. In the buffer example, methods (instances of the behavior concern) may refer to *data* concerns and *memory management* concerns, *synchronization* concerns may refer to methods.

Another way of visualizing the multiple dimensions of concerns, which is easier for illustrating the dependencies is the following layer-based model:

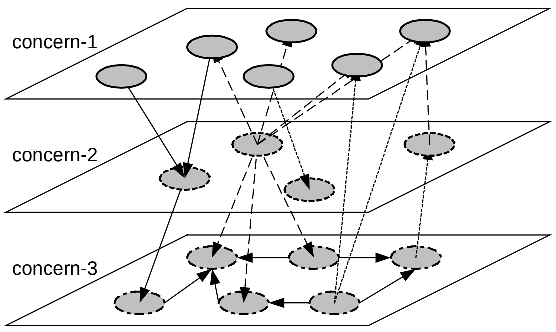

Figure 2. A representation of multiple dimensions of concerns as a set of typed graphs.

In the above figure, each layer represents one dimension of concerns, each gray circle represents a concern instance. The arrows show directed relations between the concerns. Various kind of relations (i.e. types) are possible; in an object-oriented setting, the relations may be for example inheritance, aggregation or message connections. The concern instances within a single dimension may, but need not have relations between them.

The direction of the relations defines how a relation can be specified: the relation can be specified completely on the tail side, without affecting the specification of the concern where the head of the relation points to.

We can simplify this layered representation by drawing everything in one space as a graph of typed nodes and typed relations, e.g. as in the following figure (note that this is not an equivalent of the above figure; for space reasons we have reduced the number of nodes and relations):

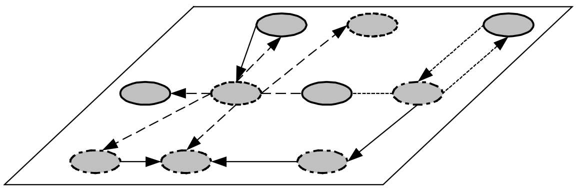

Figure 3. A representation of multiple dimensions of concerns as a graph with typed nodes and relations.

2.2 HIERARCHICAL NESTING

It has been observed by many (e.g. [Simon 96]/Rechtin/..) that complex systems are best constructed and modeled as hierarchical systems; systems that are composed of *interrelated* subsystems. Each subsystem is again a composition of hierarchically structured (sub-) subsystems, until the lowest, *atomic* level. The fact that at each level the subsystems of a system are interrelated and mutually dependent is fairly important; no interesting complex systems can be built from fully independent, isolated subsystems.

We adopt the concept of hierarchically structured systems as a means to create complex concerns composed from subconcerns. This is visualized in Figure 4 as respectively nesting and a hierarchical tree:

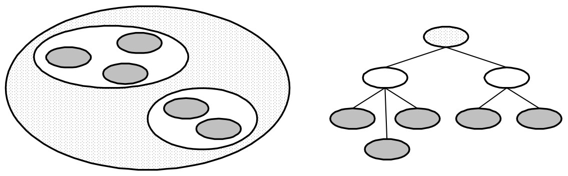

Figure 4. Two visualizations of nested c.q. hierarchically composed concerns.

The above visualizations do not show our interest in using identical concerns in different nestings/compositions. This could be modeled explicitly by allowing a lattice structure[2] instead of a tree-structure: this allows several complex concerns to refer to the same concern. This is less appealing as it does not match our intuition of hierarchic systems well. The alternative that we adopt for now is to allow duplicates of concerns, a special relation between identical concerns could make this explicit.

A wide range of decomposition strategies is possible. In [Koopman 95], the following three strategies are identified:

- Based on structures: the form, or ‘what’.
- Based on behaviors: the function, or ‘how’.
- Based on goals: the desired/emergent design properties, or ‘why’.

These categories roughly coincide with *matter*, *processes*, and *mental state* as defined in [Chi 94] as the three basic ontologies for reasoning and learning, and with *object*, *aspect*, and *sequential* decomposition as mentioned in the context of general (non-CS) design decomposition in [Michelena 97].

[1] For example in Aspect-Oriented Programming, one of the basic assumptions is that all aspects (i.e. concern dimensions) apply to the *base level*, never to each other [Bergmans 99 OOPSLAws].

[2] Or: a directed, a-cyclic, rooted graph?

The main theme of [Koopman 95] is that each of the different mixtures[1] of the above three decomposition strategies has its own advantages and disadvantages. In our case, the adopted decomposition strategies would determine the concern dimensions of the nested concerns. We do not impose any constraints on the decomposition strategies; neither on the number of strategies nor on the order in which they are to be applied.

The dependencies between the concerns are modeled by the same kind of relations that we discussed in section 2.2.

The aim of the hierarchical nesting (decomposition) is that it gives us first-class abstractions to represent complex (i.e. non-atomic) concerns. This is essential for constructing and reusing complex subsystems.

2.3 MODELING CONCERNS

Now that we have presented the structural issues of our model, it is important to consider the semantics of our basic building blocks: the concerns. Until now we have used this term in a very broad sense, but this prohibits the ability to do any reasoning at all about our composition model. Therefore it is essential to have well-defined models for representing concerns.

During the last decade, we have developed various models for expressing certain aspects such as synchronization and real-time [Bergmans 96b]. For example, the synchronization concern in [Bergmans 96b] was defined using the following abstractions:

- *Implicit state*: references to and hooks within the context of the synchronized component/software.
- *Synchronization conditions*: expressions that abstract the context (state) in a way that is relevant for synchronization.
- *Synchronization condition mapping*: determines the syntax and semantics of synchronization condition specifications.
- *Message accept sets*: defines a set of operations that are synchronized, i.e. blocked or free.

The important things about models for representing concerns are: (a) the more structure, finer granularity and more restricted semantics for the parts that specify a concern, the easier it will be to make useful statements about the behavior and composability of a set of concerns, and (b) that the crucial properties that determine the composition characteristics of the model lie in the flexibility of the mappings and the ability to add or remove sub-components incrementally.

We have developed models for various concerns, such as real-time, multiple views, coordinated behavior and error handling. Ideally, we can define a canonical model, which is suitable for appropriately modeling many different concerns. In our research, the *Composition Filters* model has served this purpose . See e.g. [Aksit 92, 93, 94][Bergmans 94, 96b] for examples of the various concerns and the composition filters model.

[1] Although any arbitrary mixture is possible, the typical goal of a design methodology is to structure the decomposition by systematically choosing the decomposition strategies.

2.4 THE COMBINED MODEL

Given the ingredients that have been described in sections 2.1, 2.2 and 2.3, we now show how these work together to model the composition of software. We illustrate this with a simple object-oriented example that composes multiple concerns. For reasons of space and comprehensibility, we omit any details on the internals of the concerns, as defined by the canonical model.

Our example consists of two independent concerns: the first is a console object, defining behavior to implement writing text to the console, and reading text (key presses) from the console. When modeling the writing and reading as separate concerns, these belong clearly to a concern dimension for behavior. The composition of these two concerns yields the *Console* concern. One may argue that this concern belongs to the behavior concern dimension as well, but in a more general case, where it also composes other concerns such as the state of the console, this can be modeled as belonging to a separate concern dimension that we refer to as *abstractions*[2]. The result is shown in Figure 5 below.

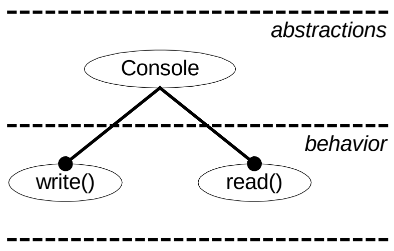

Figure 5. A simple model of the Console.

To visualize these models, we draw graphs consisting of ellipses for nodes that represent concerns, and two types of edges: the thick lines represent composition relations, with a bullet at the end of the composed (nested) concern. Thin arrows are used to represent other dependencies between concerns. The direction of the arrow shows which concern depends on the other concern that is pointed to. The relation between concern dimensions and concerns can be shown by the layering as adopted here, or by annotations (stereotypes) for each concern.

[2] Note that we do not present a methodology for identifying and modeling multiple concerns, but only a model for describing a given decomposition into concern dimensions and concerns. Other ways of decomposing are equally valid and applicable.

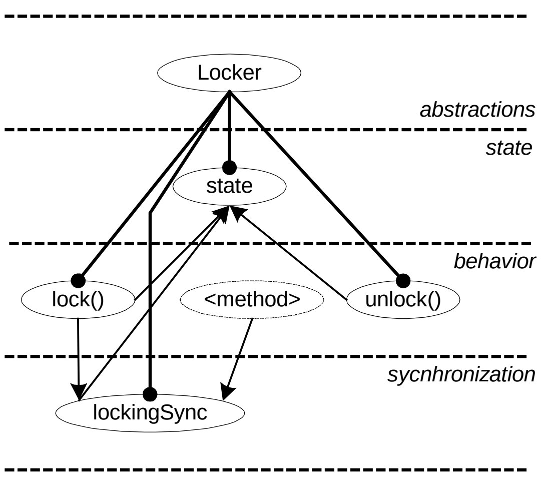

Figure 6. A visualization of the Locker concern

In Figure 6 above, we show a slightly more complex example of a *Locker* class: the purpose of this class is to be able to 'lock' an object, i.e. blocking all incoming message invocations, and later 'unlock' the object again. The blocking of messages is a separate synchronization concern dimension. It contains the concern *lockingSync*, which depends on the state. To illustrate that various methods may depend on (adopt) the *lockingSync* concern, we introduce a dummy concern <*method*> which has a dependency upon the *lockingSync*. For maintaining the state (i.e. locked or unlocked) of the *Locker*, we introduce a separate *state* concern, which belongs to the state concern dimension.

The main theme of this paper is composition of multiple/complex concerns. We will now illustrate this through the composition of *Locker* and *Console*. We simply show the intuitive/conceptual idea of composition, regardless of available composition technology. The goal of the composition is to yield a composed concern *LockingConsole*.

Plain, orthogonal composition of concerns without any interaction or dependencies between the two parts, albeit applicable in a small number of cases, is a nobrainer. In this case we are interested to apply the notion of locking upon (the methods of) *Console*. This naturally creates dependencies between the two concerns (or rather, their subconcerns): we would like the *write()* and *read()* behavior concerns within the *Console* concern to adopt the *lockingSync* concern within the *Locker*. Figure 7 shows this example in a diagram.

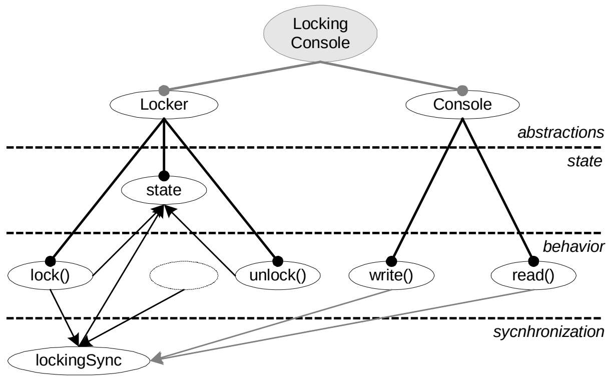

Figure 7. Visual representation of the composition of the complex concerns Locker and Console.

One of the key issues of this example composition is that it creates a composition of (concerns consisting of) multiple concern dimensions: abstractions, state, behavior and synchronization. We can view this as two levels of composition: the first is to compose multiple concerns from multiple concern dimensions into a single working abstraction, the second is to compose complex abstractions, each covering multiple concern dimensions (cf. the discussion of composability in [Bergmans 97]).

3. COMPOSITION ANOMALIES

3.1 DEFINITION

An *anomaly* means: “deviation from the common rule; irregularity” or “something different, abnormal, peculiar, or not easily classified“ (Merriam-Webster). With *composition anomaly* we refer to the cases where a *conceptually sound* composition does not work out with a specific composition scheme. In other words, a composition of two concerns that is natural and logical at the conceptual level cannot be implemented straight-forward with a particular composition scheme.

Depending on the composition scheme, it may be possible, though, to fix a composition deficiency with some patching code. Although there is no fundamental problem with the need for additional code, it turns out that this reduces maintainability in virtually all cases. This may manifest itself e.g. by requiring code replication, or additional dependencies on either implementations or interfaces.

The term composition anomaly has been derived from the term *inheritance anomaly*, as was coined by Matsuoka et.al. in [Matsuoka 90, 93] to denote the more specific case where the embedding of synchronization code in classes caused serious problems when trying to reuse and extend such code, especially through inheritance mechanisms. In those cases, it typically appeared that the problems could be patched by overriding in a subclass substantial parts of the methods defined by a superclass. We refer to [Matsuoka 90] and [Bergmans 94] for extensive analysis of these problems. One of the important conclusions from this work was that these problems are language-dependent; i.e. they are related to the chosen synchronization scheme, and its composition semantics.

3.2 AN EXAMPLE: LOCKINGCONSOLE

In Figure 7 the visual composition of concern *LockingConsole* from two concerns, *Locker* and *Console*, was shown. The figure illustrates that it only requires the definition of some new relations to connect the synchronization concerns to the methods of the console class. This is a perfectly sound composition of two different behaviors –each consisting of multiple concerns themselves– into a new, complex one. However, if we try to do such a composition in, say, the object-oriented model, it appears that this is not straightforward at all, and actually exhibits a composition anomaly.

We show the classes *Locker* and *Console* by object-oriented pseudo-code:

```
Class Locker // blocks method calls if locked
  isLocked:Boolean;
  lock() // lock all methods except unlock
    {self lockingSync(); isLocked:=true; …}
  unlock() // unlock all methods
    {isLocked:=false; …)
  lockingSync(); // implements method synchr.
  //for both locked & unlocked states
  //must be embedded at the start of methods
end;
Class Console // plain functionality
  write(s:String);  // displays the string
  read():String;    // reads from the input
  …
end;
```

Now assume that the synchronization is implemented with semaphores or a monitor, or a similar mechanism; this means that all methods that can be locked, must be enhanced with synchronization code. This code is implemented separately in method `Locker::lockingSync()`. When composing the two classes, all methods that are to be locked, must insert this code, i.c. a call to this code within the methods. The most effective way to do this is by overriding the methods with two 'super calls'; this reuses the code from the respective superclasses:

```
Class LockingConsole
  // compose through multiple inheritance:
  inherits Locker, Console;
  write(s:String)
   {Locker::lockingSync(); Console::write(s)}
  read()
   {Locker::lockingSync(); Console::read()}
end;
```

Although this example even does not replicate code, it infringes the maintainability of the application since (a) the composition requires overriding all methods of class *Console* that are lockable, (b) the overridden methods depend on the interface of the methods of *Console*, (c) all –future– methods of class *LockingConsole* and both its subclasses must add the call to *lockingSync()*, and (d) for all lockable methods that are added to classes *Console*, *Locker*, or their superclasses, a new redefinition must be added to class *LockingConsole*.

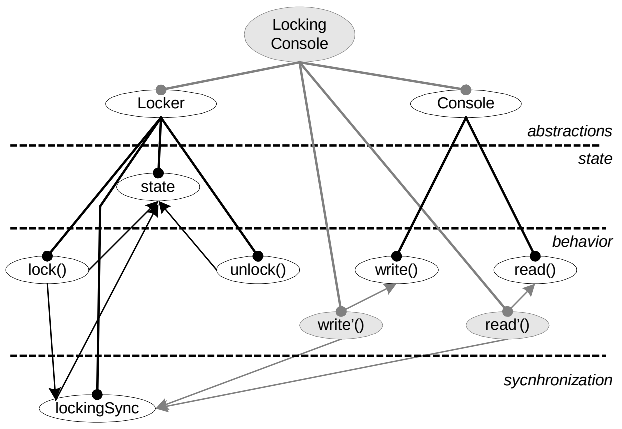

Figure 8. A visualization of the OO implementation of LockingConsole*; the gray parts are newly added.*

Figure 8 shows this implementation of the composition, it shows in particular that this composition scheme requires the addition of the two new concerns *write'()* and *read'()*, whereas the intended composition as shown in Figure 7 does not require this. Note that an alternative of adding this concern would be to add dependency relations from *write()* and *read()* to *lockingSync*. Although this would discard the need for adding the two new concerns *write'()* and *read'()*, it actually requires modifying the original definitions of *write()* and *read()*, which is not acceptable either.

Summarizing, we can give the following definition of inheritance anomaly in terms of our composition model: an anomaly occurs if a composition C_c of two or more concerns requires (a) adding new concerns except for C_c, (b) modifying concerns, or (c) adding new relations from any other concern except C_c.

4. GOALS FOR COMPOSING CONCERNS

Our goal is to achieve the ability to construct and maintain software incrementally. This means that for any give complex piece of software, it should be possible to perform a set of incremental transformations. However, we only need to consider transformations that are logically sound: for example, consider a data structure concern and a memory management concern that are tightly coupled. Assuming that it conceptually makes no sense to remove the data structure concern while demanding that the memory management concern is unaffected, then there is no need to demand such a transformation to be supported by the composition scheme.

We define the following list of simple transformation upon a given model:

- Ability to *add* new *concerns* to a system with minimum (zero) impact on existing concern specifications. Note that this implies that the dependencies between the new and existing concerns are to be specified without affecting existing concerns.
- Ability to *modify* one or more *concerns* with minimum (zero) impact on the rest of the system. Compare this to the principle of encapsulation in object oriented programming.
- Ability to *remove concerns* from a system with minimum (zero) impact on the remaining concern specifications. This assumes that there are no inherent dependencies upon the removed concern. Naturally, this may affect the dependency relations between the concerns.
- Ability to *add* a *dependency* between two or more concerns without impact upon the concern(s) that is depended on, and with minimum impact upon the dependent concerns.
- Ability to *modify* a *dependency* between two or more concerns without affecting the concerns that is depended upon, and with minimum (or zero) impact upon the dependents.
- Ability to *remove* a *dependency* between two or more concerns without impact upon the concern(s) that is depended on, and with minimum impact upon the dependent concerns.

Note that for these transformations, we have not made any assumptions about the internal model of the concerns; adopting an internal model will lead to additional transformations and refinement of the descriptions above.

5. CONCLUSION & FUTURE WORK

In this paper we have presented the groundwork for further analysis and better understanding of the issues in composing software from multiple concerns. The model that we presented in section 2 is based on some basic assumptions (axioms), which we have tried to motivate from experience and literature on designing complex systems. In combination, these lead to a model that in our opinion is suitable for reasoning about the composition of complex (software) systems.

The definition of composition anomalies in section 3 is a generalization from the notion of inheritance anomaly in concurrent object-oriented programming. Its value lies in the availability of a tool for reasoning about composability and composability problems, especially if a formalization of the model and anomaly definition is available.

The point of the goals that have been presented in section 4 is to provide a foundation for reasoning about requirements for composition schemes and comparing different composition schemes.

As may be clear, the work presented in this paper is just the beginning of much future (and some recent) work, we describe some of the issues:

- Formalization of the software composition model: this is important for solid reasoning and definitions. We have made first steps in this direction, adopting a model that is similar to Harel's work on *higraphs* [Harel 88].
- Formalization of anomalies: we are currently working on the formalization of the composition anomalies, which will at least give a precise answer to questions when an anomaly is occurring or not.
- Modeling concerns & canonical model: we have some ideas about this, which we are prototyping. It is our intention to create specializations like composition filters, a plain object-oriented model, and other recent composition schemes.
- The previous item can serve for the definition of a common framework for analyzing and comparing various composition schemes.
- We consider the list of transformations as a goal for composability of multi-dimensional concerns; more specific requirements for composition schemes have to be stated. We have outlined these e.g. in [Bergmans 00].
- Finally we want to be able to define better, more powerful and more composable composition schemes, based on what we learned from the previous activities.

Acknowledgement
This research has been carried our partly in the context of the AMIDST project, which is funded by the Telematics Institute.

REFERENCES

[Aksit 88] M. Aksit & 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 92] M. Aksit, L. Bergmans and S. Vural. *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*, Proc. of ECOOP '92, LNCS 615, Springer-Verlag, 1992, pp. 372-395

[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 ECOOP '94, LNCS 821, Springer Verlag, July 1994, pp. 386-407

[Aksit 99] M. Aksit, *Object-Oriented Software Architectures*, post-graduate course for PAO-Informatica, course notes, 1999

[AspectJ 99] *AspectJ Language Specification*, XEROX Corporation, URL: http://www.aspectj.org, 1999

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

[Bergmans 96a] L. Bergmans, *Aspects of AOP: Scalability and application to domain modelling*, position paper for the first 'Friends of AOP' workshop, XEROX PARC, Palo Alto, 1996

[Bergmans 96b] L. Bergmans & M. Aksit, *Composing Synchronization and Real-Time Constraints*, Journal of Parallel and Distributed Programming, September 1996

[Bergmans 97] L. Bergmans, *An Introduction to Composability*, in the workshop report of the ECOOP'96 Workshop on Composability in Object-Oriented Programming, in [Mühlhaüser 97], 1997

[Bergmans 99] L. Bergmans & M. Aksit, *Analyzing Multi-Dimensional Programming in AOP and Composition Filters*, position paper for the OOPSLA'99 Workshop on Multi-Dimensional Separation of Concerns, 1999

[Bergmans 00] L. Bergmans & M. Aksit, *Aspects & Crosscutting in Layered Middleware Systems*, position paper for the workshop on Reflective Middleware, in conjunction with Middleware 2000, 2000

[Booch 94] G. Booch, *Object-Oriented Analysis & Design-with Applications*, 2nd edition, Benjamin/Cummings Publishing Company, 1994

[Chi 94] M. Chi, J. Slotta, N. de Leeuw, *From things to processes: A theory of conceptual change for learning science concepts. In Learning and Instructions*, Vol. 4., pp. 27-43, Elsevier Science

[D'Hondt 99] M. D'Hondt & Th. D'Hondt, *Is Domain-Knowledge an Aspect?*, position paper for the ECOOP'99 Workshop on Aspect-Oriented Programming, to be published in Springer-Verlag ECOOP workshop proceedings, 1999

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

[Harel 88] D. Harel, *On Visual Formalisms*, Communications of the ACM, Vol. 31, No. 5, pp. 514-530, May 1988

[Harrison 93] W. Harrison & H. Ossher. *Subject-oriented programming (a critique of pure objects)*. In proceedings of OOPSLA '93, September 1993.

[Kiczales 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, *Aspect-Oriented Programming*. In proceedings of ECOOP '97, Springer-Verlag LNCS 1241. June 1997.

[Koopman 95] P. Koopman, *A Taxonomy of decomposition Strategies based on Structures, Behaviors, and Goals*, In *Design Theory & Methodology '95*, 1995

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

[Lopes 98] C.V. Lopes & G. Kiczales, *Recent Developments in AspectJ*, in *Object-Oriented Technology-ECOOP'98 Workshop Reader*, position paper for the Workshop on Aspect-Oriented Programming, pp. 398-401, LNCS 1543, 1998

[Matsuoka 90] S. Matsuoka, K. Wakita & A. Yonezawa, *Synchronization Constraints with Inheritance: What is Not Possible- So What is?*, Tokyo University, Internal Report, 1990

[Matsuoka 93] S. Matsuoka & A. Yonezawa, *Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, in Research Directions in Concurrent Object-Oriented Programming, (eds.) G. Agha, P. Wegner & A. Yonezawa, MIT Press, April 1993, pp. 107-150

[Michelena 97] N. Michelena & P. Papalambros, *A Hypergraph Framework for Optimal Model-Based Decomposition of Design Problems*, Computational Optimization and Applications, *Vol. 8, No.2, September 1997, pp. 173-196*

[Simon 96] H.A. Simon, *The Sciences of the Artificial*, 3rd edition, The MIT Press, Cambridge (MA), 1996.

[Tarr 99] P. Tarr, H. Ossher, W. Harrison & S.M. Sutton, Jr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. In proceedings of ICSE 21, May, 1999.