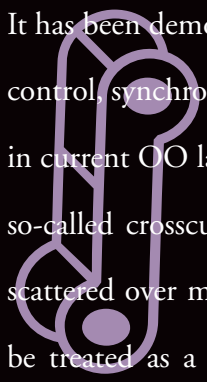
COMPOSING CROSSCUTTING CONCERNS USING COMPOSITION FILTERS

Supporting both intraclass and interclass crosscutting through model extension.



Lodewijk Bergmans
and Mehmet Aksit

It has been demonstrated that certain design concerns, such as access control, synchronization, and object interactions cannot be expressed in current OO languages as a separate software module [4, 7]. These so-called crosscutting concerns generally result in implementations scattered over multiple operations. If a crosscutting concern cannot be treated as a single module, its adaptability and reusability are likely to be reduced. A number of programming techniques have been proposed to express crosscutting concerns, for example, adaptive programming [9], AspectJ [8], Hyperspaces [10], and Composition Filters [1]. Here, we present the Composition Filters (CF) model and illustrate how it addresses evolving crosscutting concerns.

The filters in the CF model can express crosscutting concerns by modular and orthogonal enhancements to objects. Here, modularity

means that filters have well-defined interfaces and are conceptually independent of the implementation (language) of the object. Orthogonality means that filter specifications do not refer to (the specification of) other filters. These two properties increase adaptability and reusability of concerns because filters can be applied to different languages and can be combined easily. Previous publications on the CF model focused on concerns that crosscut a single object [4]; a generalization of the CF model for expressing concerns that crosscut multiple objects is presented here.

An Example: Office Automation for Social Security Services

The objective of the example application is to support the activities of a government-funded agency responsible for the implementation of disability insurance laws. The core activity of the agency can be represented by five tasks, which are shown in Figure 1. Task *RequestHandler* processes incoming requests and initializes a new document. Task *RequestDispatcher* implements the evaluation and distribution of the requests to the other tasks: *MedicalCheck, Payment* or *OutputHandler*. *MedicalCheck* is responsible for evaluating the client's disability; *Payment* is responsible for issuing bank orders; and *OutputHandler* is responsible for the interaction with clients.

Software Design of the Example Application. In this article, we ignore user-interface issues and concentrate on the core application classes shown in Figure 2. One major concern in the application is to model the documents that represent requests and another is to model tasks that process the requests. The operation of the system mainly consists of creating and editing documents, and forwarding the documents to other tasks as defined by office procedures. For presentation purposes we display only a subset of the attributes and methods in Figure 2. Class *Document* is the common superclass of all document classes and defines the attributes *id*, *clientName*, and *clientAddress*, used for storing information about the client. The attribute *category* specifies the classification of the client with respect to the disability laws. The attribute *responsible* represents the clerk responsible for processing the request. Class *Document* has 10 methods, which are used to read and write the attributes. Class *ClaimDocument* is a specialization of *Document* and is used to represent the claims of clients. This class declares a number of attributes and methods regarding the motivation and size of a claim, the medical status and the approved claim amount. All the task classes in the system implement the interface *TaskProcessor*. Here only the method *processDocument* is shown, which accepts a document as an argument and opens a dedicated editor for it.

Implementing the System Using Composition Filters. All current OO languages and systems share

Figure 1. Tasks in the example system.

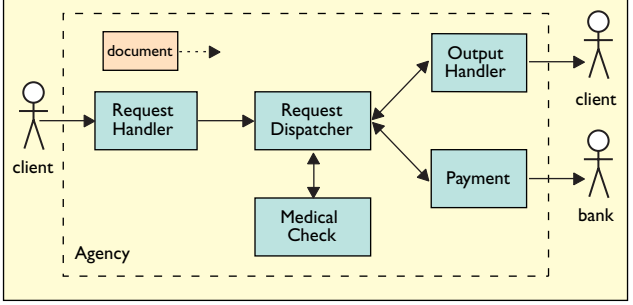
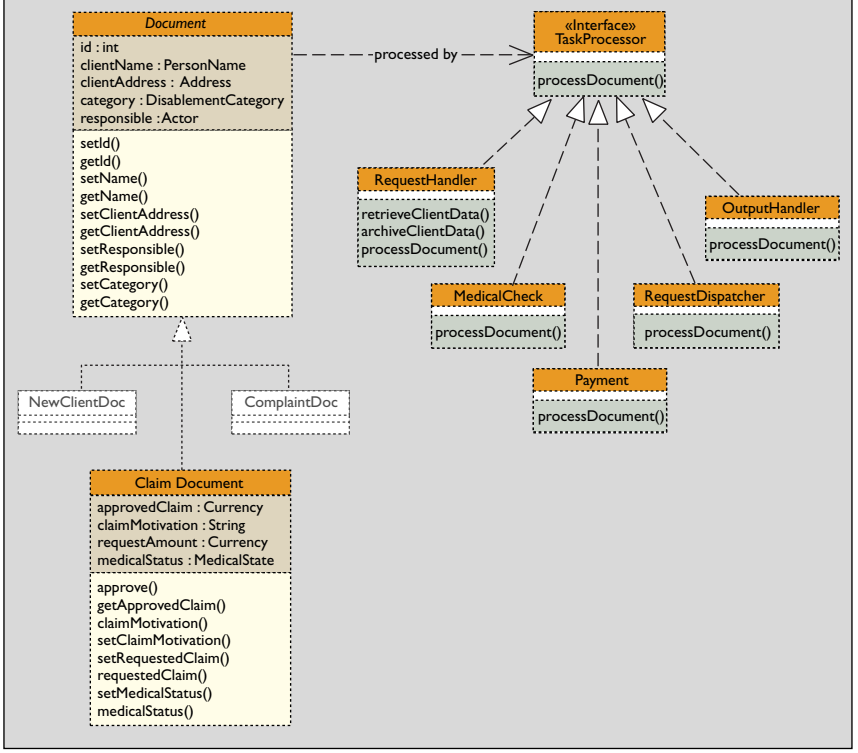
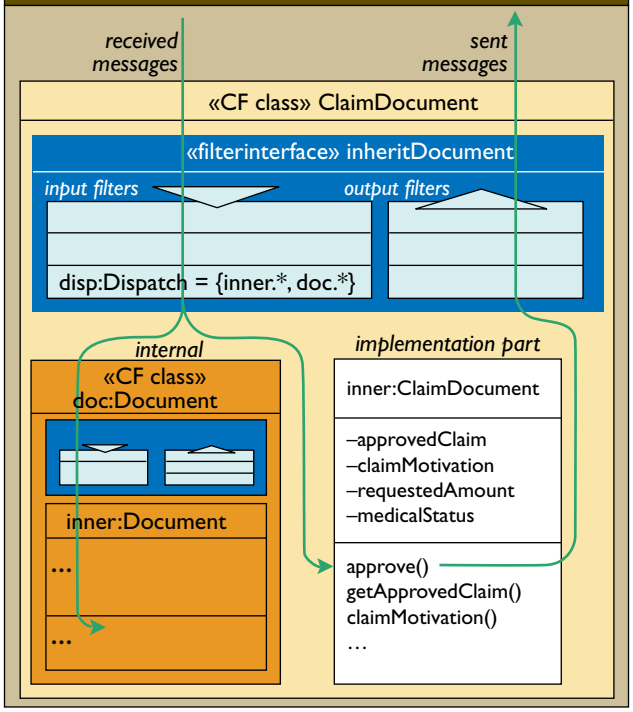


Figure 2. An overview of the core application classes in the example application.



a similar message-passing model. To specify filters independent of a specific programming language, filters are defined as functions, which manipulate messages received and sent by objects. Since an object's observable behavior is determined by the messages it receives and sends, filters are capable of expressing a large category of concerns such as inheritance and delegation, synchronization, real-time constraints, and interobject protocols. Because of the modularity of filter specifications, these concerns can be attached to objects as enhancements.

Figure 3. A UML-based representation of CF class ClaimDocument.



We will now explain how the CF model can be used to define class *ClaimDocument*, which was introduced previously. A CF class aggregates zero or more internal classes and composes their behavior using one or more filters. An internal class may be again a CF class or it may be a class expressed in any OO language, for example Java. In Figure 3, the yellow rectangle with the UML stereotype «`CF class`», represents a CF implementation of class *ClaimDocument*. We will refer to this class as *CF-ClaimDocument*, which aggregates two other classes. Here, *CF-Document* (shown as an orange rectangle) is a CF representation of class *Document*. The implementation part, represented by class *ClaimDocument* (shown as a white rectangle), defines an implementation of the (noncrosscutting) concerns of class *ClaimDocument*, for instance in a language such as Java or C++. The blue boxes with arrows pointing in and out correspond to so-called input filters and output filters, respectively. Each message that is received must first be individually processed by all filters in the input filter set. Eventually, the received message may be dispatched to one of the methods of an aggregated object. During the execution of a method, all messages sent to other objects must first pass the output filters. Here, class *CF-ClaimDocument* has only one filter named *disp*, which is an instance of the filter type named *Dispatch*. This example of a dispatch filter effectively implements inheritance from class *Document*. The dispatch filter is initialized with the filter expression between curly brackets. In this example, the filter expression "`{inner.*, doc.*}`" consists of two filter elements, separated by the comma. Whenever a message arrives at the filter, an attempt is made to match the message with each of the elements, from left to right. The matching process primarily involves the target and the selector of the message. If the matching process succeeds, the message is said to be accepted by the filter, if not, the message is said to be rejected. The type of the filter determines the semantics of acceptance or rejection.

The first filter element of the dispatch filter in the example consists of a target defined as "`inner.`" and the wildcard "`*`" as the selector. The pseudovariable *inner* refers to the implementation part of the object. The wildcard "`*`" will match if the selector of the message is in the signature[1] of the target (*inner*). The second filter element is similar, but with *doc* as the target. This filter element will cause the selector of a message to match all the method names in the signature of class *CF-Document*. In case of rejection by a dispatch filter, the message will proceed to the next filter. If there is no subsequent filter available in the filterset an exception will be raised. In case of acceptance, the dispatch filter causes the message to be dispatched to the object that corresponds to the target of the (first) matching element.

In Figure 3, two possible message flows are shown. One message is dispatched to the internal object *doc* of class *CF-Document*, and the other is dispatched to the implementation object. Since *CF-Document* is a CF class, the messages dispatched to *doc* have to pass the input filters defined by *CF-Document* as well. The Dispatch filter type can also be used to express multiple inheritance, dynamic inheritance, delegation, or a combination thereof, through appropriate filter expressions.

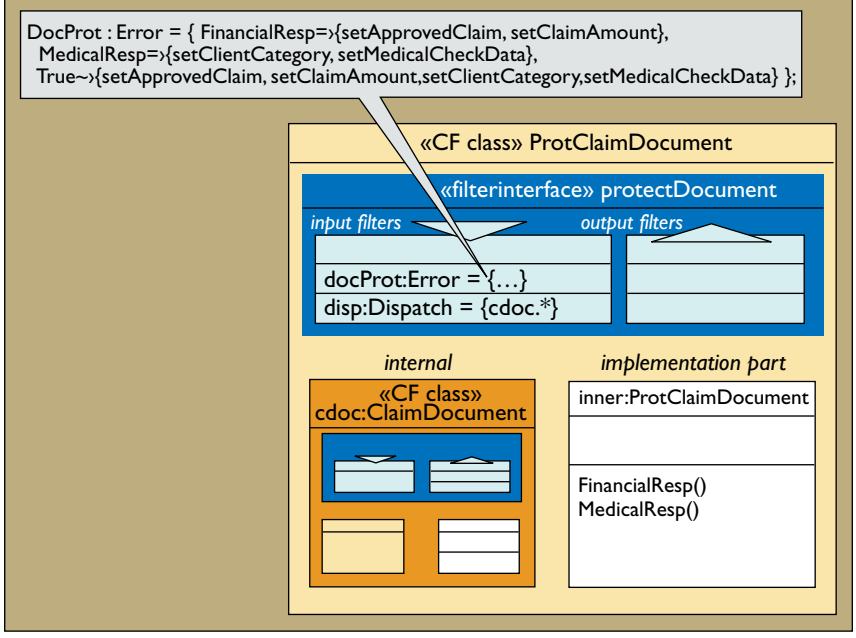
Every filter is declared and initialized using the same syntax. A number of filter types have been intro-

[1]That is, the set of messages that an object supports on its interface.

Table 1. Overview of filter types and their semantics.

Filter Type	Accept Action	Reject Action	Publications
Dispatch	Dispatch message to the target of the message	Continue to the next filter in the set	[1, 6]
Error	Continue to the next filter in the set	An exception is raised	[1, 6]
Wait	Continue to the next filter in the set	The message is queued as long as the evaluation of the filter expression results in a rejection	[5]
Meta	The reified message is sent as a parameter of another (meta) message to a named object. The object that receives the meta message can observe and manipulate the message, then reactivate its execution	Continue to the next filter in the set	[2]
RealTime	Adjust RT properties of the message (thread)	Continue to the next filter in the set	[3, 4]

Figure 4. A UML-based visualization of CF class ProtClaimDocument with two input filters.



duced to express certain crosscutting behavior. Some commonly used filter types are shown in Table 1.

Evolution Step 1: Protecting Documents. In the initial system, the methods of the document classes are available for all task objects. This means that, for example, a clerk who is responsible for registering requests would be able to modify medical or financial information. To avoid this, the implementation of methods should be extended with some source code that verifies the identity of the caller. This type of selective behavior is referred to as multiple views [1]. To implement the views, we introduce a subclass of *ClaimDocument* called *ProtClaimDocument*. Figure 4 represents class *CF-ProtClaimDocument* and its components. The class consists of one filter interface named *protectDocument*, one internal object named *cdoc*, which is an instance of class *ClaimDocument* that has been described before, and an implementation, or inner part represented as class *ProtClaimDocument*. The filter interface defines two input filters, called *docProt* and *disp*.

In the CF model, the message manipulation process can be affected at run time through side effect-free Boolean expressions called conditions. In this example, class *ProtClaimDocument* declares two conditions called *FinancialResp* and *MedicalResp*, which evaluate to true if the responsibility of the sender of a message is financial or medical, respectively. These conditions are used within the filters, but defined in the implementation part.

In Figure 4, class *CF-ProtClaimDocument* illustrates how an Error filter can be used to enforce the required multiple views on the documents. As shown in the blue rectangle, this class declares two filters: *docProt* of type Error and *disp* of type Dispatch. The Error filter specification is shown in the gray callout box. The first line of this specification states that the messages *setApprovedClaim()* and *setClaimAmount()* match the filter expression if the condition *FinancialResp* evaluates to true. If *FinancialResp* is false or the received message is not *setApprovedClaim()* or *setClaimAmount*, the second line of the filter expression is evaluated. If the condition *MedicalResp* evaluates to true, and the received message is *setClientCategory()* or *setMedicalCheckData()*, the message is accepted, otherwise the last element of the filter expression is evaluated. Here, the condition is True, followed by the exclusion operator, which is denoted by the characters "~>". This operator specifies that if the condition is true, all mes-

Table 2. An overview of the crosscutting of the concerns.

concern: error check → / ↓ methods	FinancialResp	MedicalResp	any (true)
setApprovedClaim	✔		
setClaimAmount	✔		
setClientCategory		✔	
setMedicalCheckData		✔	
other (*)	✔	✔	✔

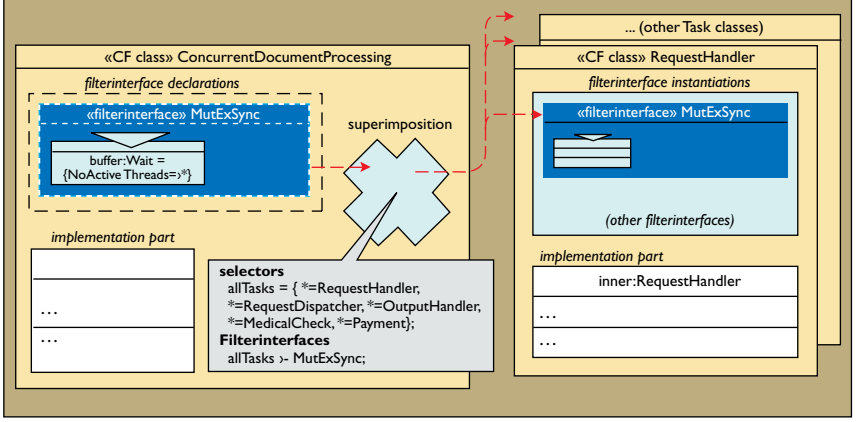
sages match, except the ones that are specified on the right-hand side of the operator.

Table 2 depicts the relation between the crosscutting concerns and methods. Here the rows show a selected set of methods of class *ProtClaimDocument* and the columns represent the crosscutting concerns. In a conventional OO language such as Java, these concerns have to be repeatedly implemented in the corresponding methods. In the CF implementation, these concerns are expressed in filters separate from the methods of the document class.

Evolution Step 2: Document Queues. In the example office system, tasks are executed in parallel. When documents are forwarded, it may not be possible to start processing the next task immediately. To avoid blocking the execution of the preceding task, a document queue is to be defined for each task processor. In addition, the task processors must provide mutual exclusion; while an object is busy executing a task, every incoming message is queued until the object has finished the execution of the previous task. We will assume the synchronization code that implements mutual exclusion can use an operation, say *noActiveThreads()*, which returns true if there is no active thread in the context of operation. The synchronization code must apply to every task processor class, and is repeated within the implementation of several methods of the same task processor class: a typical example of a crosscutting behavior.

The CF model can express various synchronization mechanisms in a modular manner. To avoid replicating the synchronization code in all task classes, the CF model provides the superimposition mechanism to impose filter interfaces onto one or more objects. Figure 5 illustrates the superimposition mechanism for concurrent processing of tasks.

Figure 5. An example of a CF class with superimposition.



In the figure, the yellow rectangle on the left represents a CF implementation of class *ConcurrentDocumentProcessing*, which is used to model the crosscutting queuing concern. The yellow boxes on the right symbolize all the task processor classes to be extended with a document queue.

The blue rectangle in *ConcurrentDocumentProcessing* represents the filter interface *MutExSync* of this class. The call-out in the upper-left corner shows the filter expression in more detail. *MutExSync* declares a single input filter named buffer of type Wait. For this filter type the expression "`{NoActiveThreads=>*}`" means that if the condition *NoActiveThreads* is true, all messages can pass this filter. Otherwise, incoming messages will be blocked, until the condition becomes true, that is, no more threads are active in the corresponding object.

The white rectangle in *ConcurrentDocumentProcessing* represents the implementation of the condition *NoActiveThreads*. The blue cross in *ConcurrentDocumentProcessing* symbolizes the superimposition specification, which denotes a set of objects where the filter *MutExSync* is inserted. Incoming and outgoing messages of the designated objects have to pass through this filter interface.[2] The dashed red lines indicate the process of superimposition. Note that the inserted filter *MutExSync* is shown on the right as the input filter of the task classes.

The definition of the superimposition that maps this filter interface to the task classes is shown in the bottom center call-out of Figure 5. In this definition,

[2]The filter interfaces are arranged sequentially in the order of instantiation.

first a selector named *allTasks* is declared, which is an expression (query) for selecting a set of concern instances. In this particular case, the expression specifies that all the instances of classes *RequestHandler*, *RequestDispatcher*, and so on are selected. The superimposition specification continues by specifying how filter interfaces are superimposed: for example, "`allTasks<-MutExSync`" in the 'interface' clause means that the filter interface *MutExSync* is to be superimposed upon all instances that are defined by selector *allTasks* of the same class. The result of this specification is that all the task instances are enhanced by superimposing the implementation of condition *NoActiveThreads*, and the filter interface *MutExSync*.

Conclusion

Here, we discuss the important characteristics of the CF model and offer a brief evaluation of these characteristics with respect to the examples that have ben presented in this article.

Declarative: Concerns are specified declaratively using a simple message manipulation language. This specification allows for various implementation strategies. For example, ComposeJ is a compiler, which inlines filter code within the appropriate methods. The Sina language compiler implements filters as metaobjects. Approaches such as adaptive programming [9] and AspectJ [8] typically adopt a dedicated declarative specification to express the crosscut and a general-purpose language to express concerns.

High-Level Semantics: Filter types encapsulate the accept and reject semantics of filters. The semantics of filter types are well defined and highly expressive within the domain of the concern [4]. As shown in the example application, Error, Dispatch, and Wait filters could effectively express multiple views, inheritance, and synchronization, respectively. Most related work adopts general-purpose programming languages for specifying concerns, in which case little or no reasoning about the semantics of the composed concerns is possible.

Open-Ended: The CF model is open-ended in at least three ways:

- New filter types can be introduced; for example, in [3] we introduced the filter type RealTime, which can be used to express the timing constraints on message executions.
- Filter expressions may contain wild cards to designate an open-ended set of messages. For example, to implement mutual exclusion for all messages, the filter expression "`{NoActiveThreads=>*}`" was used. Wild cards are also used by AspectJ, for instance.
- Various kinds of composition operators have been defined and new operators can be added. For example, in Figure 4, to compose different views at the interface of the protected claim documents, the Conditional-OR operator ',' and the exclusion operator "~>" are used. To compose the error and dispatch filters, the Conditional-AND operator ';' is used. Similar to the CF model, HyperJ [10] also supports an open-ended set of composition operators.

Strong Encapsulation: in the CF model, superimposition of filter interfaces, objects, methods, and conditions is restricted to the interface level. Therefore superimposed concerns do not rely on the details of the implementation, as such even the implementation language is encapsulated.

Modular: The CF model unifies OOP with AOP. Adding document queues to task objects, is an example of this modular extension. In Figure 2, task classes are specified independent of the queuing concern using a conventional OO notation. The queuing concern is specified using a filter. The superimposition specification is used to integrate the queuing concern with the tasks.

Composable: All filters are specified using the same message manipulation language. Filters can be composed as desired through the application of several composition operators. For example, filters can be declared sequentially so that concerns can be enforced on messages in a particular order. In Figure 5, for instance, first mutual exclusion and then the message dispatching concerns are enforced; the messages are only accepted for execution if the corresponding object is idle. In Figure 4, various alternative views are enforced on the claim documents. Each filter has specific, well-defined semantics, determined by the filter type. This supports reasoning about compositions of filters and the feasibility of these compositions. The dispatch filter also illustrates composition of the signatures of objects.

Here, we have presented the Composition Filters model, a modular and orthogonal extension to the OO model that supports both intraclass and inter-

class crosscutting. Important characteristics of this model are: declarative, high-level semantics, open-ended, strong encapsulation, modular, and composable. Much detail of the model has been omitted here; for more information, including versions of the examples in this article with the full code, see the TRESE Web site: trese.cs.utwente.nl/composition_filters. C

References

1. Aksit, M., Bergmans, L., and Vural, S. An object-oriented language-database integration model: The composition-filters approach. In *Proceedings of ECOOP '92*, Springer-Verlag, 1992.
2. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., and Yonezawa, A. Abstracting object-interactions using composition-filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, Eds., *Object-based Distributed Processing*, Springer-Verlag, 1993.
3. Aksit, M., Bosch, J., Sterren, W., and Bergmans, L. Real-time specification inheritance anomalies and real-time filters. In *Proceedings of ECOOP '94*, Springer Verlag, 1994.
4. Aksit, M. and Bergmans, L. Guidelines for identifying obstacles when composing distributed systems from components. In M. Aksit, Ed., *Software Architectures and Component Technology: The State of the Art in Research and Practice.* Kluwer Academic Publishers, 2001.
5. Bergmans, L. and Aksit, M. Composing synchronization and real-time constraints. *Journal of Parallel and Distributed Programming* (Sept. 1996).
6. Bergmans, L., Aksit, M., and Tekinerdogan, B. Constructing reusable components with multiple concerns using composition filters. In M. Aksit, Ed., *Software Architectures and Component Technology: The State of the Art in Research and Practice.* Kluwer Academic Publishers, 2001.
7. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *Proceedings of ECOOP '97*, Springer-Verlag, 1997.
8. Kiczales, G., Hilsdale, E., Hugunin, L., Kersten, M., Palm, J., and Griswold, W. *An overview of AspectJ*; aspectj.org.
9. Mezini, M. and Lieberherr, K. Adaptive plug-and-play components for evolutionary software development. In *Proceedings of OOPSLA '98*, 1998.
10. Ossher, H. and Tarr, P. Multi-dimensional separation of concerns and the hyperspace approach. In M. Aksit, Ed., *Software Architectures and Component Technology: The State of the Art in Research and Practice.* Kluwer Academic Publishers, 2001.

Lodewijk Bergmans (bergmans@cs.utwente.nl) is Assistant Professor in the Department of Computer Science at the University of Twente, The Netherlands.

Mehmet Aksits (aksit@cs.utwente.nl) is Professor of Software Engineering and Chair of the Department of Computer Science at the University of Twente, The Netherlands.

This work was supported by the Telematica Instituut through the AMIDST project, and by IST Project 1999-14191 EASYCOMP.