

Abstracting Object Interactions Using Composition Filters

Mehmet Aksit¹, Ken Wakita², Jan Bosch¹, Lodewijk Bergmans¹ and Akinori Yonezawa³

¹TRESE project, Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.

email: {aksit, bosch, bergmans}@cs.utwente.nl

²Department of Information Science, Tokyo Institute of Technology, 2-12-1 Oh-okayama, Meguro-ku, Tokyo, 152, Japan. email: wakita@is.titech.ac.jp

³Dept. of Information Science - Faculty of Science- University of Tokyo, Hongo, Bunkyo-ku, Tokyo 113 Japan. email: yonezawa@is.s.u-tokyo.ac.jp

Abstract

It is generally claimed that object-based models are very suitable for building distributed system architectures since object interactions follow the *client-server* model. To cope with the complexity of today's distributed systems, however, we think that high-level linguistic mechanisms are needed to effectively structure, abstract and reuse object interactions. For example, the conventional object-oriented model does not provide high-level language mechanisms to model layered system architectures. Moreover, we consider the message passing model of the conventional object-oriented model as being too low-level because it can only specify object interactions that involve two partner objects at a time and its semantics cannot be extended easily. This paper introduces *Abstract Communication Types* (ACTs), which are objects that abstract interactions among objects. ACTs make it easier to model layered communication architectures, to enforce the invariant behavior among objects, to reduce the complexity of programs by hiding the interaction details in separate modules and to improve reusability through the application of object-oriented principles to ACT classes. We illustrate the concept of ACTs using the composition filters model.

1. Introduction

The dynamic semantics of object-oriented languages are based on the *message passing* mechanism. A message is a request for an object to carry out one of the object's operations. Since objects can only communicate by sending messages, message passing is the basic means for creating executions in the system.

To cope with the complexity of today's distributed systems, we think that high-level linguistic mechanisms are needed to effectively structure, abstract and reuse object interactions.

Originating from the construction of operating systems, large distributed systems are structured in terms of vertical layers. Functionally, each layer communicates with its peer-level layer, although physical data exchange occurs with the adjacent layers. The conventional object-oriented model does not provide high-level language mechanisms to model layered system architectures. Moreover, we consider the message passing model of conventional object-oriented languages as being too low-level because it can only specify communications that involve two partner objects at a time and its semantics cannot be extended easily. Mechanisms like inheritance and delegation only support the construction and behavior of objects but not the abstraction of communication among objects. These mechanisms therefore fail in abstracting patterns of messages and larger scale synchronization among objects.

We have applied the *composition filters* model to abstract communications among objects. In this approach, the basic object model is extended *modularly* by introducing *input* and *output composition filters* that affect the received and sent messages respectively. This mechanism enables software engineers to abstract communications among objects into a first-class object called *abstract communication type*¹ (ACT). ACTs make it easier to model layered architectures, to enforce the invariant behavior among objects, to reduce the complexity of programs by hiding the interaction details and to improve reusability through the application of object-oriented principles to ACT classes.

This paper is organized as follows. The next section describes the problems in object-oriented modeling which form the motivation for abstracting inter-object communications. Section 3 studies the background and related work, including the composition filters model. Section 4 first gives a list of requirements to effectively integrate communication abstractions with the object-oriented model. It then introduces ACTs and explains how ACTs can be expressed using composition-filters. Section 5 presents examples in 3 categories: examples of inter-object invariant behavior, inter-object synchronization, and coordinated behavior. Section 6 evaluates the ACT concept as presented and gives conclusions.

2. The Problem Statement

The conventional object models lack support for abstracting object interactions. This reveals itself through a number of problems that are encountered in object oriented software development:

1. Lack of Support for Meta-levels and Reflection:

Assume for example that object *A* sends a message to a remote object *B* by executing the message statement

¹ The term abstract communication type is derived from abstract data type and may refer to both objects and classes. Terms ACT object and ACT class will be used to refer to an object or class respectively.

B.moveTo(X, Y);

For A, the details of this execution are abstracted. However, in reality, this message must be intercepted by the underlying layer to determine, for example, the physical location of the *receiver* of the message.

From the object-oriented modeling perspective, this requires reflection² of messages. In message reflection, the so-called *message reification* operation allows the meta-layer to process the explicit representation of the reified message [Barber 89].

Conventional object-oriented methods [Booch 90, Coad&Yourdon 91a, Coad&Yourdon 91b, Champeaux 91, Rumbaugh 91] do not provide support for reflective system development. Conventional object-oriented languages (such as C++) provide only a limited or ad-hoc reflection [Madany et al. 92].

2. *Complexity and Lack of Reusability*: The manageability of programs is affected by the complexity of interactions among modules. In object-oriented programs, the code for describing the interactions is distributed over the participating objects. This causes a mixture of functional and interaction related code, which affects both maintainability and extensibility.

Different classes may adopt identical patterns of communication and synchronization. Similarly, a single class might participate in various patterns of communication. Thus, hardcoding the interaction patterns in a class severely reduces the reusability (of the class itself, and of the interaction code). Especially reuse through extension (subclassing) is an important issue.

3. *Enforcing invariant behavior*: If the code that implements the invariant behavior is distributed over a number of objects, verifying the invariants is far from trivial. A single module that explicitly represents the interaction between objects is an attractive approach for ensuring the invariant behavior of this interaction.

² A reflective system is a system which incorporates models representing (aspects of) itself. This self representation is *casually connected* to the reflected entity, and therefore, makes it possible for the system to answer questions about itself and support actions on itself. Reflective computation is the behavior exhibited by a reflective system. The term reflection was introduced by [Smith 82] as a technique to structure and organize self-modifying procedures and functions. In [Maes 87] reflection was applied within the object-oriented framework. Recently a considerable amount of work has been done in object-oriented reflection, for example, in concurrent programming [Ichisugi et al. 92], operating system structuring [Yokote 92], compiler design [Lamping et al. 92] and real-time programming [Honda&Tokoro 92].

3. Background and Related Work

This section describes the background and related work for ACTs. It consists of two main sections: in the first section the related work in analysis and design, and programming models is described. In the second section the composition-filters model is explained. We will apply the composition-filters model for expressing and illustrating ACTs.

3.1. Related Work in Object Interactions

This section describes the work that has been done with respect to object interactions. We first describe the attention that object-oriented analysis and design methods pay to modeling object interactions, and then one specific modeling approach, *Contracts*. Then we discuss two programming models, respectively *Scripts* and reflective computation, how they can be applied for abstracting object interactions.

Object-Oriented Analysis and Design Methods

Most object-oriented analysis and design methods model interactions among objects, usually after identifying *inheritance* and *part-of relations*. Different terms are used to express object interactions such as *object diagrams* [Booch 90], *process model* [Champeaux 91], *message connections* [Coad&Yourdon 91a], *data-flow diagrams* [Rumbaugh 91] and *collaboration graphs* [Wirfs-Brock et al. 90]. The Demeter system [Lieberherr et al. 91] is a Computer-Aided Software Engineering (CASE) environment which provides a tool to generate repeated operations called *propagation patterns*. In addition, the Demeter system incorporates a design rule for minimizing interactions between objects [Lieberherr&Holland 89]³. Object-Oriented Design by Coad and Yourdon [Coad&Yourdon 91b] introduces a *task management* component which aims at defining object interactions.

Object-oriented analysis and design methods model interactions among objects in a way similar to object-oriented languages. Basically, they define graph structures that represent execution threads and therefore these methods have the same limitations as programming languages. The *task management* component [Coad&Yourdon 91b] can be considered as a module to model object interactions. In this method, however, there is no emphasis on using these constructs for this purpose. Moreover, it does not provide solutions to the problems as presented in section 2.1.

³ *Contracts* were developed as a part of the research activities related to the Demeter system.

Contracts

In the area of object-oriented modeling, the idea of specifying object interactions as an explicit module is applied by *contracts*⁴ [Helm et al. 90, Holland 92]. Contracts are used to specify the *contractual obligations* that a set of *participants* must satisfy. It is possible to *refine* a contract in order to make it more specific and it is possible to *include* existing contracts in a new contract. In its first version [Helm et al. 90] a declarative language was introduced to define contractual obligations. In the second version [Holland 92], however, a procedural language was adopted instead of a declarative one. In the following we refer only to the second version of contracts.

A contract specification includes the specification of the participating objects, the contractual obligations of all participants, the invariants to be maintained by the participants and the method which instantiates a contract.

A contract can be seen as an *abstract class*, defining both abstract and concrete methods for its participants. The abstract methods must be provided by the participants themselves. The concrete methods of the contract (or its refinement) override the concrete implementations of the participants. A contract may also define variables that are shared by all the participants. In order to put a contract to use, a conformance declaration must be made which initializes the contract with actual participants. Obviously, these participants have to satisfy the contractual obligations of the contract. An object may participate in several contracts. Contracts offer two alternatives: either the methods are implemented at the contract specification, or they are distributed over the participating classes.

Contracts are primarily targeted as a design tool. Contracts are quite useful for the implementation of coordinated behavior and the abstraction of object interactions but are unable to reflect upon the actual message interactions between objects for purposes such as monitoring and manipulating messages. Contracts are treated differently from normal classes. Contracts also do not address concurrency and synchronization issues.

Scripts

A language construct called *scripts* [Francez 86] was introduced to abstract patterns of messages into a module. A script is a parameterized program section in which processes *enrol* in order to participate. The concept of *enrolment* is similar to the subroutine call mechanism whereby the execution of the role in a given script instance is a logical continuation of the enrolling process. A script consists of formal process parameters called *roles*, data parameters and a concurrent program section called the *body*. Processes can enrol in scripts by means of *enrol in* statements.

Scripts are program modules and do not provide mechanisms for object-oriented computing. Scripts, for example, do not allow users of the system to create several

⁴ Apart from the object-oriented language Sina.

instances belonging to the same communication module. Inheritance or delegation mechanisms are also not defined for scripts thereby resulting in a less systematic reuse of communication abstractions.

Reflective Computation

In principle, languages that provide full reflection are able to represent object interactions. However, full reflective languages have complicated semantics and may bring unnecessary additional complexity. One particular example of a restricted reflective language is MAUD [Agha et al. 92]. Each object in MAUD owns three meta-objects called a *dispatcher*, a *mail queue* and *acquaintances*. The sent and received messages are handled by the dispatcher and mail queue objects respectively. The acquaintances object contains a list of objects that may be addressed by its owner object. In the MAUD language, one can implement coordinated behavior by replacing the meta-objects with the objects implementing the required protocol. To install a protocol for an object the original mail queue and dispatcher must be replaced by a pair implementing the required protocol.

In MAUD, a shared protocol among objects is implemented by mail queues and dispatchers. Coordinated behavior is distributed among mail queue and dispatcher objects which are added to all participating objects. Therefore designers cannot define and reuse coordinated behavior as a single entity.

Apertos is an object-oriented reflective operating system [Yokote 92] designed for open and mobile computing environments. Apertos introduces object/metaobject separation in the operating system design. An object is associated with a group of metaobjects and a metaobject defines the semantics of its object. An object can change its metaobject (or group of metaobjects) by migration. Although Apertos provides a general reflective system framework, it does not emphasize abstraction and reuse of interactions among objects.

3.2. The Composition Filters Model

We will first briefly introduce the components of the composition-filters object model and then present them in greater detail later. This computation model is adopted by the Sina language⁵. In Sina, operations and local variables are called *methods* and *instance variables*, respectively. As illustrated by Figure 1, a composition-filter object consists of two parts: an *interface* and an *implementation* part. The interface part deals with incoming and outgoing messages. It consists of one or more *input*

⁵ The early version of the Sina language was published in [Aksit&Tripathi 88, Tripathi&Aksit 88, Aksit et al. 91]. This version introduced only a simple filter mechanism which was then called *predicates*. The recent version of the language was published [Aksit et al. 92, Bergmans et al. 92]. These publications did not address the issues related to abstract communication types. The preliminary version of ACTs was first published in [Aksit 89a].

and *output filters*, optional *internal* and *external* objects and *method header* declarations.

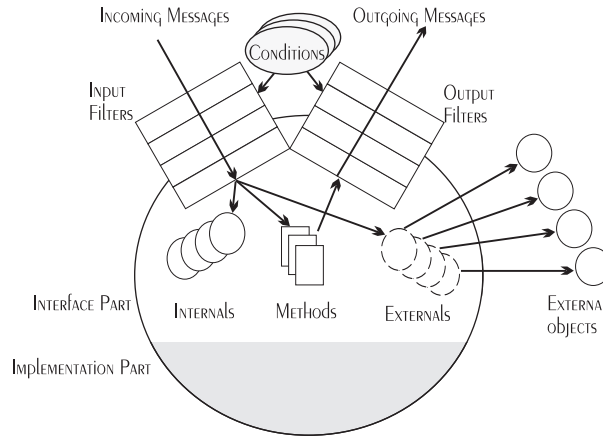


Figure 1. The interface components of the composition-filters object model.

Filters are controlled by *conditions*⁶. Filter names, method headers and condition names can be made visible to the *clients* of the object, however, their implementations are defined in the implementation part and invisible. In Figure 1, a possible effect of the input filters is shown. If a message passes through the input filters it can be further delegated to internal objects, methods or external objects. In addition, Figure 1 depicts the effect of output filters on the outgoing messages. All the messages that originate from method executions within the object and are sent to objects that are outside the boundaries of the current object pass through the output filters. Without filters, our model is very similar to the conventional object model.

The implementation part contains method definitions, instance variable declarations, definitions of conditions and an optional initialization operation. The implementation part is fully encapsulated within the object.

The Interface Part

As an example of a simple class consider the interface part of class *Point*. We present our examples following the Sina language notation.

⁶ In [Aksit et al. 92] conditions were called *states*.

```

class Point interface
  comment This class implements a graphical point;
  conditions
    Initialized; // this condition is only valid after the object has been initialized
  methods
    moveTo(Integer, Integer) returns Nil;
    // changes the coordinates of the point
    getX returns Integer;
    // reads the current x location of the point
    getY returns Integer;
    // reads the current y location of the point
  inputfilters
    disp : Dispatch = { True=>inner.moveTo, Initialized => inner.* };
end;

```

Figure 2. Definition of the interface part of class *Point*.

The methods that are to be visible at the interface of the object are declared in the interface part by *method headers* following the keyword `methods`. Class *Point*, for instance, declares the methods *moveTo*, *getX* and *getY* for changing and reading the coordinates of the point respectively. The actual implementations of these methods are encapsulated within the implementation part. An appropriate message must be sent to an instance of class *Point* to invoke one of these methods.

An input filter specifies conditions for message acceptance or rejection and determines the appropriate subsequent action. The output filters handle outgoing messages and are studied in section 4. After the keyword `inputfilters`, class *Point* defines a single input filter called *disp* of class *Dispatch*⁷ using the expression

```
disp: Dispatch = { ... };
```

An input filter of class *Dispatch* is used to initiate execution of a method when the corresponding message passes successfully. The filtering condition, between the brackets "{" and "}", is specified as

```
{ True=>inner.moveTo, Initialized => inner.* }
```

On the left hand side of the characters "=>", a necessary condition is specified, denoted by the condition identifiers, *True* and *Initialized* in this case.

⁷ The current version of the Sina language provides a number of primitive filters such as *Dispatch*, *Meta*, *Error*, *Wait* and *RealTime*. The *Dispatch* filter is explained in this section. The *Meta* filter will be studied in section 4. The *Error* filter is similar to the *Dispatch* filter but it does not provide a method dispatch; it raises an error condition if a message does not pass through the filter [Aksit et al. 92]. The *Wait* filter is used for synchronization [Bergmans et al. 92]. The *RealTime* filter is used for realtime computations [Aksit&Bosch 92]. These filters can be used as both input and/or output filters. An input filter composes the signature of its object whereas an output filter specifies how its object sends messages to other objects. An important feature of all these filters is that they are orthogonal to each other and, therefore, they can be combined freely.

Conditions are similar to logical propositions. The names of the conditions are declared in the interface part following the keyword `conditions` and their definition is provided in the implementation part. Conditions may reflect the values of instance variables, but may reflect external variables as well. In this example, the condition *Initialized* is set to *true* if the instance variables of class *Point* have been initialized.

The received message is matched with the method names specified on the right hand side of the characters "`=>`". The character "*" indicates a wild-card or don't care condition; if the message matches with any of the method names provided by class *Point* it will be accepted for execution. An alternative could be to list all the method names explicitly. The pseudo-variable *inner* denotes the methods defined by *Point*.

An optional *internal* clause may be used to declare encapsulated objects whose behavior can be made (partially) visible on the interface of the encapsulating object by filter specifications. Internal objects differ from instance variables, because internals are used to compose the behavior of the object, whereas instance variables represent the local data of the object. An *external* clause may be used similarly to declare exterior objects that are to be accessible to this object. The use of internals and externals will be explained when inheritance mechanisms are introduced.

The Implementation Part

The components of the implementation part are exemplified by class *Point* as shown in Figure 3.

Instance variables are declared in the `instvars` clause. Instance variables are fully encapsulated and can be objects of arbitrary complexity. Class *Point* declares 3 instance variables named *x*, *y* and *initializeDone*. Only the methods defined within the object's class may access the instance variables directly, external clients of an object or even its subclasses cannot do this.

The implementations of the conditions are defined by message expressions. The structure of a condition implementation is similar to the structure of a method. However, a condition implementation always results in a *Boolean* value and is free of *side effects*.

The initialization method of an object is defined in the `initial` clause. This method is executed immediately after object creation.

The last component of the implementation part is the definition of the methods. A method consists of a series of message expressions. The control flow may be controlled by a set of standard *control statements*.

```

class Point implementation
  comment This class implements a graphical point;
  instvars
    x, y: Integer
    initializeDone: Boolean;
  conditions
    // the conditions that were declared in the interface part are implemented here
    Initialized:
      begin return initializeDone end;
  initial
    begin initializeDone:= false; end;
    // here the initial method is defined, which is executed immediately after object creation.
  methods
    moveTo(x, y: Integer) begin ....; initializeDone:=true end;
    getX begin .... end;
    getY begin .... end;
end;

```

Figure 3. The implementation part of class *Point*.

Message Evaluation by Filters

A filter is a *first-class* object that determines whether a particular message is either *accepted* or *rejected* and what action is to be performed in either case. Each filter is declared as an instance of a filter class. A programmer may define an arbitrary number of filters for an object. Each filter can be an instance of an arbitrary filter class. The complete set of input filters of an object determines the conditions for message acceptance and determines which method will be executed upon acceptance. Figure 4 illustrates how a message is evaluated by a set of filters.

This example consists of three filters *A*, *B* and *C*. A received message *m* has to pass through all the filters to result in a successful dispatch. Every filter consists of a number of filter elements (two or three in this example). When a message is to be evaluated by a filter it will be checked against the elements of the filter in left-to-right order. A filter element consists of three parts:

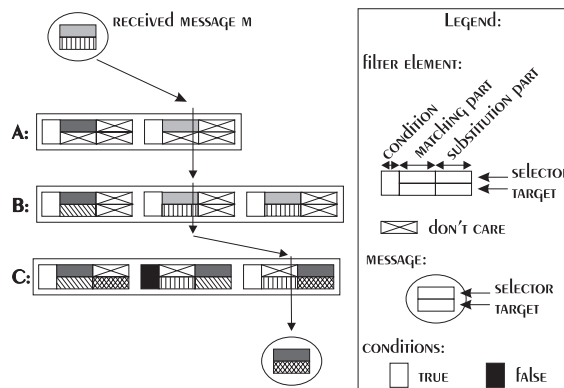


Figure 4. Message acceptance by filters.

- A *condition*, which specifies a necessary condition to be fulfilled in order to continue evaluating a filter element;
- A *matching part*, in which the evaluated message is matched against a defined pattern;
- A *substituting part*, where (parts of) the message can be replaced.

In filter *A*, the selector of the received message is matched against the selector of the matching part of each filter element; when the filter element does not match the subsequent filter element is tried. In filter *A*, although both of the conditions are *true*, only the second element matches the message since the selector of the first filter element does not match. The message is accepted by filter *A* and can then proceed to the next filter.

In filter *B*, matching is not restricted to the selector of the message, but involves the target of the message as well. The first element of *B* does not match but the second and third elements do. Due to the left-to-right ordering, the message matches on the second filter element and proceeds to the next filter.

Filter *C* demonstrates the full expressiveness of filter evaluation. It introduces substitution of selectors and targets. In the filter, the first filter element does not match and the second filter element has a condition that is *false*. The message is accepted at the third element and new values for the target and selector are substituted.

Since there is no subsequent filter, the type of the filter determines what will happen with the message. Commonly the last filter is of class *Dispatch*, which results in delegation of the request message to its target object.

The conditions, the matching and the substitution as provided by filters, provide a generic mechanism for selecting messages based either on their properties (selector or target), or on some condition specified by the receiving object. They also support the renaming of message selectors and redirection of messages (by substituting new targets). Based on the acceptance or rejection of a message, the filter can perform appropriate actions such as bouncing or blocking a rejected message or delegating an accepted message.

Inheritance and Delegation Through Input Filters

This section demonstrates how input filters can be applied to realize basic object-oriented data modeling techniques, such as inheritance and delegation. In section 4 we will explain how filters can be used to define ACTs.

In the composition-filters model, inheritance is not directly expressed by a language construct but is simulated by input filters. In order to inherit from a class an *internal object* must be declared as an instance of that class. Inheritance is simulated by delegating messages to the methods provided by this instance object. This is exemplified by class *ReferencePoint*, shown in Figure 5.

```

class ReferencePoint interface
  comment This class is a subclass of class Point and is used
           as a reference point for a set of other points ;
  internals
    myPoint : Point; // instance of the 'superclass'
  methods
    display returns Nil; // displays itself on the current point
  inputfilters
    disp: Dispatch= { True=>myPoint.*, True=>inner.* };
end;

```

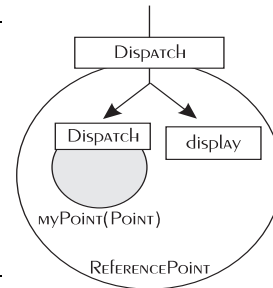


Figure 5. The interface part of class *ReferencePoint*.

Class *ReferencePoint* declares an internal object *myPoint* of class *Point* and introduces one method *display*. The method *display* makes the graphical object visible at the current location.

The filter *disp* of class *Dispatch* contains two filter elements. The condition *True* preceding each filter element means that the target-selector pair(s) on the right-hand side will always be checked. These two filter elements have the following meaning:

First filter element:

The first element of the filter, *myPoint.** specifies that all the incoming messages are delegated to the internal object *myPoint*, provided that these messages are supported by class *Point*. Since the methods of *Point* are now available to *ReferencePoint* through an instance of *Point*, class *ReferencePoint* inherits the operations of class *Point*. This technique for simulating inheritance is also referred to as *delegation-based inheritance*.

When an instance of class *ReferencePoint* is created, its internal object *myPoint* is also created. An important feature here is that instance variables of the *superclass* are only accessible through operations provided by the superclass.

The second filter element:

If the first filter element does not match with the message, the second filter element is evaluated. Instead of delegating to an internal object such as *myPoint*, this filter element delegates the message to the pseudo-variable *inner*⁸. By

⁸ Apart from the pseudo-variable *inner*, two other pseudo-variables, *self* and *server*, are also available as a means of self-reference. The variable *inner* allows direct internal access on the objects' own methods. *self* refers to the instance of the class which defines the method. If, for example, *myPoint* refers to *self*, it will refer to *myPoint* but not *aReferencePoint*. We introduced *inner* to avoid infinitely nested compositions. Such nested compositions can be created if only *self* is used. In order to refer to the object that originally received the message, *server* is used as a target. For example, if *myPoint* refers to *server*, it will refer to *aReferencePoint*. Note that *server* is dynamically bound and is equivalent to Smalltalk *self*.

declaring *inner* as a target object, class *ReferencePoint* makes the methods defined and implemented by itself available to its clients.

Note that since the filter elements are evaluated from left to right, the first element prevails over the second one. The order of the filter elements can be manipulated to bind messages to the desired targets⁹.

Instead of using an internal object as a target, the programmer may also delegate the incoming messages to an *external object* by declaring the target name in the *externals* clause. Because external objects are not encapsulated within the object, they can be shared by other objects. In addition, contrary to the *internals* clause, an external declaration does not result in automatic object creation.

4. Abstract Communication Types

4.1. Requirements for Abstract Communication Types

We have identified the following requirements for defining effective communication abstractions:

1. *First-class property*¹⁰: If the communications among objects show a well-defined, meaningful, complex and/or reusable behavior, then they must be explicitly represented by one or more ACTs. The rationale for this requirement is that if communications among objects are well-defined and meaningful, they are likely to be problem domain entities; if they are complex, then they can be managed by the object-oriented techniques such as encapsulation and inheritance; if they are reusable, they must be defined as classes (objects) since classes (objects) are the unit of reuse.

An ACT class must be able to reuse other classes in the system so that ACT frameworks may be constructed.

2. *Large scale synchronization*: ACTs must be able to express various concurrency and synchronization schemes. We believe that distributed applications can be conveniently constructed using ACTs. Therefore, ACTs must have rich semantics to express various concurrency and synchronization mechanisms, such as asynchronous communications, broadcasts, coordinated terminations, distributed concurrency control algorithms, etc.
3. *Reflection upon messages*: An ACT must be capable of reflecting upon messages, such as for monitoring, logging, affecting synchronization semantics and message contents, or redirecting messages.

⁹ This is especially useful for solving name conflicts that are due to multiple inheritance.

¹⁰ First-class property means an ACT object is treated as an ordinary language object.

4. *Uniform integration of communication semantics*: Considering ACTs as objects only is not sufficient. Communication mechanisms defined by an ACT must be uniformly integrated with the operations implemented by the participating objects. An ACT must be considered as the *extended identity* of the participating objects¹¹.

4.2. Basic Concepts

An ACT class is an ordinary Sina class with the same syntax and semantics. What makes a class an ACT class is the way its behavior is composed with its participating objects. An ACT class operates on first-class representations of messages. For converting a message into its first-class representation, we introduce a new filter class called *Meta* filter. An instance of *Meta* filter has a structure similar to the *Dispatch* filter. The difference here is that if the received message is accepted by a *Meta* filter it is first converted to an instance of class *Message* and then passed as an argument of a new message to the ACT object. The conversion operation is also known as *reification*. The ACT object can retrieve the necessary information from the message argument. An ACT can also modify the contents of the message by invoking the operations of class *Message*. Finally, an ACT can convert an instance of *Message* back to a message execution. The detailed explanations of class *Meta* filter and *Message* are presented in sections 4.3 and 4.4, respectively.

ACTs can be further classified as *abstract sender types* (ASTs) and *abstract receiver types* (ARTs)¹². ASTs and ARTs are responsible for abstracting one-way communication among objects. Various ways of composing ACTs are illustrated in Figure 6.

In Figure 6(a), each object has an output *Meta* filter which intercepts and delegates the outgoing messages to the internal AST object. The internal ASTs that are encapsulated by different objects may all belong to the same class to enforce common protocols among objects. The AST object is responsible for abstracting the communication that originates from the sender object. The sender object inherits the behavior of the AST object in object communication. This mechanism uniformly integrates the communication semantics of the AST object with the sender object. Typical applications of this architecture are asynchronous communications, encoding messages etc.

¹¹ The semantics of an ACT object can not be integrated uniformly with the behavior of interacting objects just by executing message calls. After each message call, the context of the original call (such as the pseudo-variable *self*) is changed. As a consequence, this may result in a less reusable coordinated behavior since the ACT object can not polymorphically refer to the participating objects. This is equivalent to the *self-problem* as defined in [Lieberman 86].

¹² This is an intuitive classification. We found out that in practice designers of ACTs tend to talk about ACTs that send or receive messages.

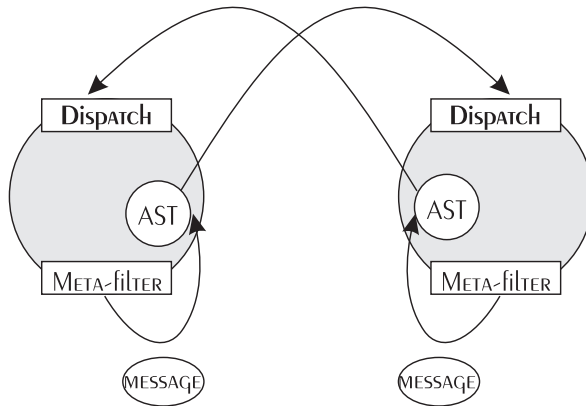


Figure 6(a). Outgoing messages are delegated to an internal AST object.

The architecture in Figure 6(b) is similar to 6(a), except that a shared external AST object is used instead of an internal one. This allows communicating objects to share the behavior with a common state. For example, this AST object can store the names of the receiver objects in a multicast implementation.

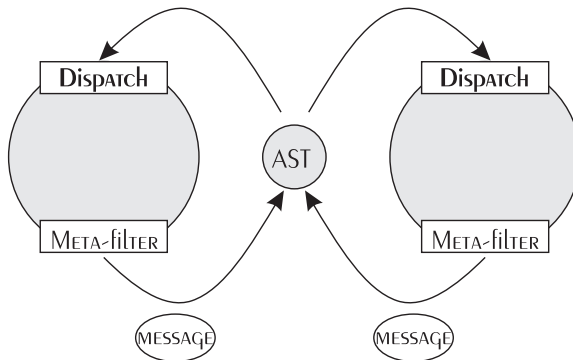


Figure 6(b). Outgoing messages are delegated to an external shared AST object.

In Figure 6(c), each object has an input *Meta* filter which intercepts and delegates the received messages to the external ART object.

The ART object is responsible for handling incoming messages. Examples are one-way constraint solvers, security protocols, data handlers in atomic transactions, decoding messages, etc.

Figure 6(d) combines the functionalities of AST and ART types into a single external ACT object. This object handles both incoming and outgoing messages. Typical examples are *coordinated behavior*, *multi-way constraint solvers*, *distributed algorithms* etc.

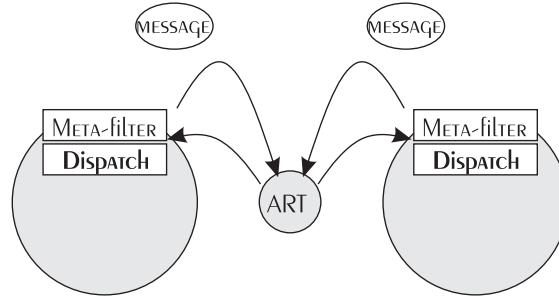


Figure 6(c). Composition of an external ART object with the participating objects.

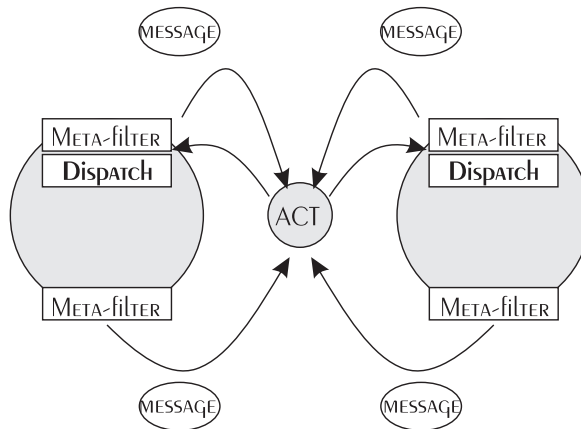


Figure 6(d). Delegating all communication to an ACT object.

4.3. Modeling Software Using ACTs.

In our analysis and design method, we apply the ACT concept as an object-oriented modeling technique. As illustrated by Figure 7(a), during the class (object) identification phase we explicitly search for classes that represent interactions among objects. Typically, these classes manifest themselves as action abstractions, distributed algorithms, coordinated behavior, inter-object constraints, etc. ACT classes are not procedural abstractions but they are problem domain entities and have a well-defined behavior.

In some cases, the analyst may fail in identifying ACT classes. After the identification of inheritance and part-of relations among classes, we specify object interaction patterns. If there is a well-defined pattern among objects and if this pattern is meaningful in the problem domain, then we represent them as ACT objects. As shown in Figure 7(b), in such a case we move the object-interaction behavior (code) to an ACT object.

Many object-oriented methods define *associations* [Rumbaugh 91] between objects. Most associations represent message exchange between these objects and can be conveniently represented by ACTs.

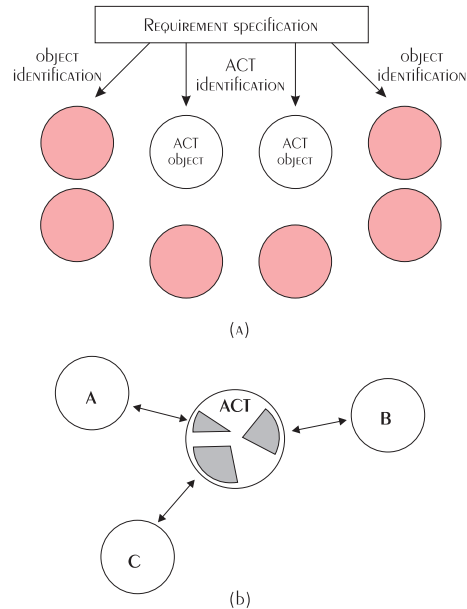


Figure 7. Identifying ACTs using (a) requirement specification and (b) object interaction patterns.

We have applied the object-oriented analysis and design techniques to a large number of applications [Aksit&Bergmans 92]. In various applications we could benefit from mechanisms that could abstract object interactions. One example was the administration system for social security services [Greef 91]. In this system, different objects were coordinating together to calculate payments. These calculations were the implementations of laws and could be abstracted by ACTs. Another example was the chemical process control system for a distillation process which was developed at the department of Chemical Engineering, University of Twente [Jonge 92]. In this system various optimization algorithms were distributed to different components. The algorithms were solving some well-defined differential equations and could be modeled by ACTs that implemented these algorithms. Distributed system design clearly demonstrated the need of abstracting interaction patterns [Aksit 89b, Bempt 91, Bergmans 90, Dolfing 90, Zondag 90]. In the distributed system design we could benefit from ACTs, for example, in building *layered architectures*, dedicated distributed *concurrency control* mechanisms and implementing *security protocols*.

4.4. Class Meta-Filter

Instances of class *Meta*-filter are used to reify messages that pass through them. The reified message is passed as an argument of the new message to an ACT object. Reification is needed to allow the ACT object to invoke operations on the instance of *Message*. Consider the following example:

```
aMetaFilter : Meta = { aCondition => [self.aMethod] anACT.aMethodOfAnACT };
```

There is no difference between a *Meta* filter and other filters in the manner a filter expression is evaluated. However, when the message is accepted by a filter element, which means both *aCondition* is *true* and the message is *self.aMethod*, a new message is created and the original message becomes the argument of the new message. The new message is composed of *anACT.aMethodOfAnACT(aMessage)*, where *aMessage* is the reified original message. If the received message does not match with a *Meta* filter it is passed to the next filter. The semantics of class *Meta* filter are presented in Appendix B.

4.5. Class Message

A message in the system becomes accessible when it is reified by a *Meta* filter and passed to an ACT as an argument of class *Message*. Class *Message* defines a number of methods for accessing and changing the *receiver*, *sender*, *server*, *selector* and *arguments* of the message. In addition, it provides methods for copying, reactivating and replying to the message. The accessing and changing operations are self explanatory. We will now describe the other methods.

The method *copy* returns a copy of the message. The *sender* of the copied message is undefined unless it is explicitly initialized. The reactivating method *fire* causes the message to continue with its execution. The method *reply* accepts an argument and sends this argument as a reply message to the sender, stored internally in the message. The interface methods of class *Message* are described in Appendix A.

4.6. Implementation Issues

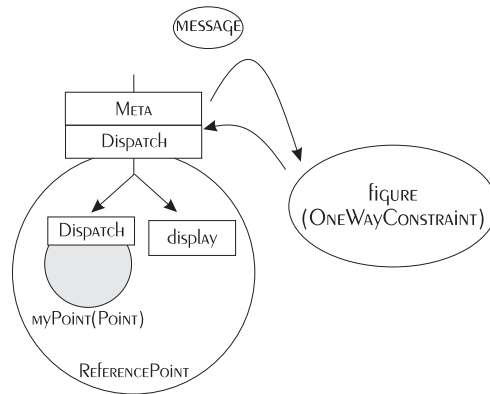
Currently, we are carrying out a research activity for the efficient implementation of composition-filters. We are experimenting with a Sina compiler that generates C++ and Smalltalk code. In most cases, ACTs do not impose significant execution overhead, since the code that is executed by an ACT can be *inlined* into the object that owns the meta filter. This is because the name of the ACT object is explicitly named in the filter initialization part.

5. Examples of Abstract Communication Types

5.1. Example for an Inter-Object Invariant Behavior: One-Way Constraints

An instance of class *ReferencePoint* is supposed to store the reference coordinates of a figure. When the coordinates of the reference point are changed, then all the dependant graphical objects must be updated accordingly. Thus a figure can be considered as a constraint among the graphical elements that form the figure. We consider such a constrained behavior as a typical example of an ACT.

To compose this constraint behavior with *ReferencePoint*, Figure 8 extends the interface part by declaring object *figure* of class *OneWayConstraint* in the externals clause and by adding a new input filter called *constraint* of class *Meta*.



```

class ReferencePoint interface
  comment this class is a subclass of class Point and is used as a reference point for a set of other
    points ;
  externals
    figure : OneWayConstraint; // instance of the 'ART class'
  internals
    myPoint : Point; // instance of the 'superclass'
  methods
    display returns Nil; // display itself on the current point
  inputfilters
  {
    constraint : Meta= { True => [*..moveTo]figure.applyConstraint };
    disp: Dispatch= { True=>myPoint.*, True=>inner.* };
  }
end;

```

Figure 8. Redefinition of the interface part of class *ReferencePoint* .

Class *ReferencePoint* now has two filters enclosed by the characters "{" and "}". The filter *constraint* of class *Meta* contains a single filter element. The condition *True* preceding the filter element means that the target-selector pair(s) on the right-hand side will always be checked. The filter element consists of matching and substitution parts:

Matching part:

The matching part of the filter "[*:moveTo]" means that all the incoming messages with the selector *moveTo* will match. The received message will be converted to an instance of class *Message* if there is a match. If the received message does not match with the *Meta* filter it is passed to the next filter.

The substitution part:

After the message conversion the message is sent as an argument of the message "*figure.applyConstraint(aMessage)*". Object *figure* is declared in the externals clause and is responsible for enforcing the constrained behavior among the elements of *figure*. After updating the dependant graphical elements, *figure* converts the message back to the execution form which then passes though the second filter called *disp* of class *Dispatch*. The second filter dispatches the message to its target.

```
class OneWayConstraint interface
  comment this class implements a one way constraint enforcing mechanism ;
  methods
    applyConstraint( Message ) returns Nil; // this is the independent reference message
    putDependants( OrderedCollection(Any) ) returns Nil; // dependant objects are supplied
    size returns Integer; // number of dependant objects
    putConstraints( OrderedCollection(Block) ) returns Nil; // store constraints for dependants
    getConstraints returns OrderedCollection(Block); // retrieve constraints
  inputfilters
    disp : Dispatch = { true => inner.* };
end;
```

Figure 9. The interface part of class *OneWayConstraint*.

Class *OneWayConstraint* is an ART and is a general one-way constraint solver which provides the consistency of the dependant variables when the independent variable changes. In the following example variables *y* and *z* are dependants of *x*:

$$y = f1(x) \qquad z = f2(x)$$

OneWayConstraint introduces five methods. The method *applyConstraint* accepts a single argument of class *Message*. This argument is used as the independent value for the one-way constraint solver. The method *putDependants* accepts an ordered collection of objects of any type and stores them internally as dependant objects. The method *size* returns the number of dependant objects. The method *putConstraints* accepts an ordered collection of instances of class *Block* as an argument. Class *Block* represents a Sina method implementation. Each block is a constraint expression to be solved and corresponds to the object that is stored at the same index location of the ordered collection of dependants. For example, constraints on figure elements can be expressed as

[moveTo(message.argument(1) + ΔX , message.argument(2) + ΔY)]

Where *message* is the argument provided to the method *applyConstraint*. The method *argument(i)* returns the i_{th} argument of this message. ΔX and ΔY are the coordinates relative to the reference point.

The method *getConstraints* retrieves the ordered collection of *Blocks*.

Note that class *OneWayConstraint* is a generic class and can be reused in other applications.

In the following example class *BoundedFigure* inherits from class *OneWayConstraint* and restricts the coordinates of the figure within a certain frame. *BoundedFigure* introduces two new methods called *putFrame* and *getFrame* and overrides the method *applyConstraint*. The method *putFrame* accepts an argument of class *Rectangle* and stores it as the boundary of the figure. The method *getFrame* returns the current frame of the figure. The method *applyConstraint* of *OneWayConstraint* is now overridden because the allowed coordinates of the figure are restricted.

```

class BoundedFigure interface
  comment This class inherits from OneWayConstraint and extend it further by putting a frame;
  internals
    figure : OneWayConstraint;
  methods
    putFrame( Rectangle ) returns Nil;
    getFrame returns Rectangle;
    applyConstraint( Message ) returns Nil;
  inputfilters
    disp : Dispatch = { true => { inner.*, figure.* } };
end;

```

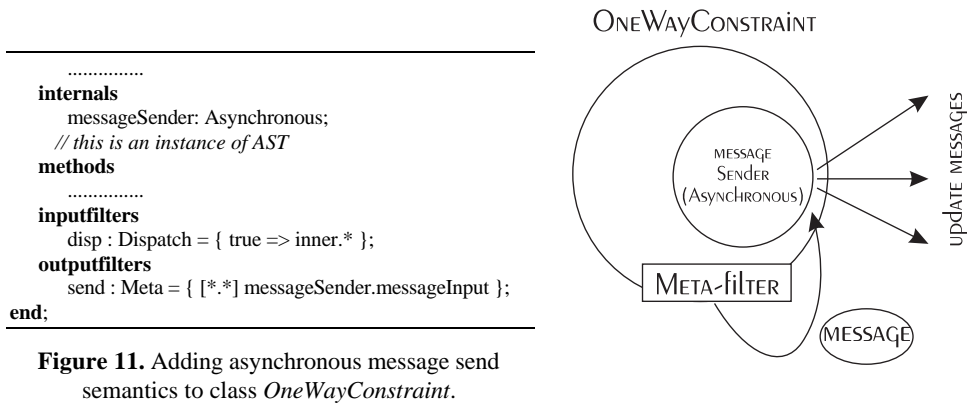
Figure 10. The interface part of class *BoundedFigure*.

5.2. Example for Inter-Object Synchronization: Asynchronous Message Send

The update messages sent by the constraint solver can be executed asynchronously. In Figure 11, class *OneWayConstraint* is extended by defining a new output filter called *send* of class *Meta*. This filter converts the outgoing messages to an instance of *Message* and passes it to the internal object *messageSender* of class *Asynchronous*. Class *Asynchronous* provides asynchronous message passing and its definition is given in Figure 12.

Class *Asynchronous* is an AST and defines a single method called *messageInput*. This method accepts an instance of class *Message* as an argument and replies to this message immediately by returning the object *nil* to the sender. It then activates the message by invoking the method *fire* on this message. Note that the matching part in the dispatch filter "[self.*]" will match with any message that is sent to an instance of class *Asynchronous*.

In Sina, unless mutual exclusion is provided by a filter [Bergmans et al. 92], methods may be executed concurrently. This class therefore may execute concurrent *messageInput* invocations.



```

class Asynchronous interface
comment this class implements an asynchronous message passing mechanism;
methods
messageInput ( Message ) returns Nil; // message to be sent asynchronously
inputfilters
disp : Dispatch = { true => [self.*] inner.messageInput};
end;

class Asynchronous implementation
methods
messageInput( originalMessage: Message);
begin
originalMessage.reply(nil);
originalMessage.fire;
end;
end;

```

Figure 12. The interface and implementation parts of class *Asynchronous*.

5.3. Example for Coordinated behavior: Atomic Transactions

For computer-aided engineering applications figures can be processed to calculate certain features such as volume, weight, etc. In the one-way constraint implementation of Figure 11, dependant objects are updated by sending them a number of asynchronous messages. During the update operation the figure is inconsistent and, if there are other processes accessing this figure, the results of their computation may be inconsistent as well.

Atomic transactions have proven to be a useful mechanism to preserve consistency [Haerder&Reuter 83]. *Serializability* and *indivisibility* are the two important properties of atomic actions. Serializability means that if several actions are executed concurrently, they manipulate the affected data as if they were executed serially in some order. Indivisibility means that either all or none of the atomic actions are performed.

The implementation of class *OneWayConstraint* is extended in Figure 13 by defining a second output filter named *atomic* of class *Meta* to enforce consistent updates. This filter converts the message that is fired by *messageSend* to an instance of *Message* and passes it to the internal object *atomicUpdate* of class *TransactionManager*.

The interface definition of class *TransactionManager* is given in Figure 14. Class *TransactionManager* inherits from class *CommitReceive* and provides two methods called *size* and *transaction*. The method *size* accepts an integer argument and stores it internally as the size of the transaction. The method *transaction* accepts an argument of class *Message* and executes this message together with other messages as an atomic transaction.

In our example class *TransactionManager* has an instance variable called *commitSend* which implements a commit protocol. This protocol is explained with the help of Figure 15(a-d). In 15(a) *commitSend* receives the transaction as a message list from *TransactionManager* and *fires* them one by one.

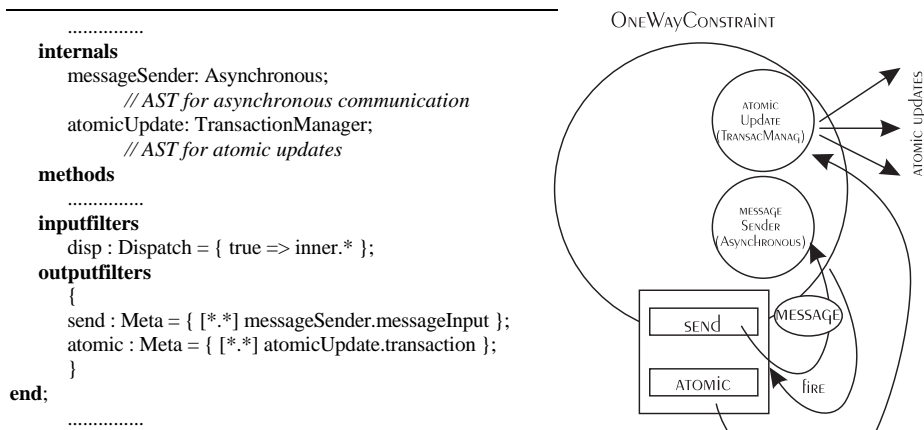


Figure 13. Adding atomic transaction semantics to class *OneWayConstraint*.

```

class TransactionManager interface
comment this class sends a set of messages as a transaction;
internals
myCommitReceive: CommitReceive; // Inherits from CommitReceive. It is used to commit or
// abort the transaction
methods
size(Integer) returns Nil; // size of the transaction block
transaction( Message ) returns Nil; // an element of a transaction block
inputfilters
disp : Dispatch = { true => inner.* };
end;

```

Figure 14. The interface part of class *TransactionManager*.

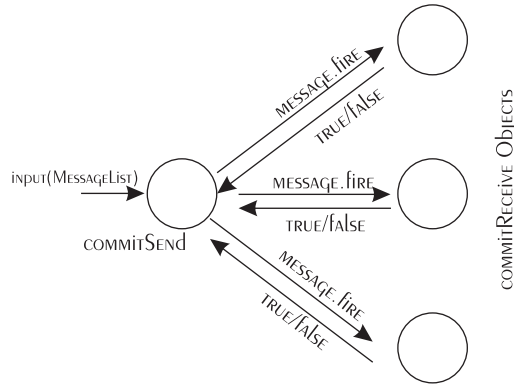


Figure 15(a). Transaction starts.

The receiver object must incorporate an ART of class *TransactionManager*. Class *TransactionManager* inherits from class *CommitReceive* which is responsible for handling *transaction commit* and *abort* messages. When a message is first received by *CommitReceive*, it goes from the *idle* to the *commit pending* state, and returns *true* as shown in Figure 15(d).

As shown by Figure 15(b), if all the responses to *commitSend* are *true*, then the transaction commits.

In Figure 15(c) is shown that when a message is returned as *false* the transaction aborts. During the *commit pending* state, if *CommitReceive* receives a new request it returns *false* and thus causes the abortion of the corresponding transaction.

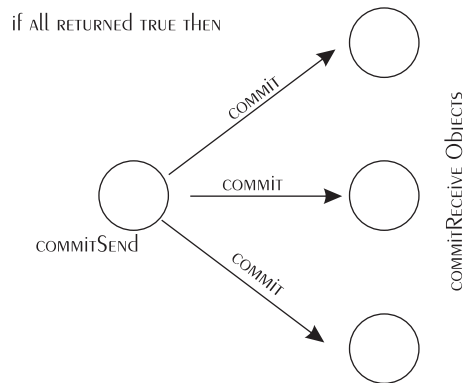


Figure 15(b). If all succeed then transaction commits.

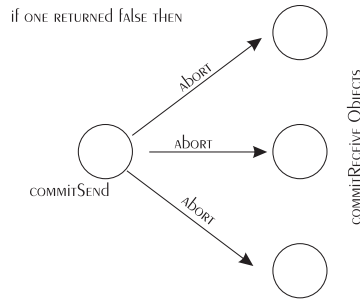


Figure 15(c). If one fails, then the transaction aborts.

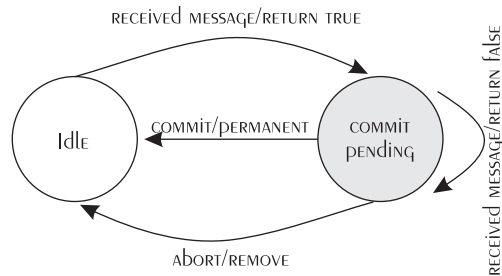


Figure 15(d). The state transition diagram of class *CommitReceive*.

Objects that require transactional behavior must incorporate an instance of class *TransactionManager* as an ART. Class *AtomicPoint*, shown in Figure 16, represents the dependant graphical points which are to be updated when their reference point is changed. This class inherits from *Point* and delegates any *moveTo* message as an instance of *Message* to its internal object *atomic* of class *TransactionManager*. The second filter dispatches to the internal objects *myPoint* and *atomic*, if the received message passes through it. Since *TransactionManager* inherits from class *CommitReceive* it responds to *commit* and *abort* messages.

```

class AtomicPoint interface
  comment This class makes point an atomic point;
  internals
    myPoint : Point;
    atomic: TransactionManager;
  inputfilters
    makeAtom: Meta = { true=> [moveTo] atomic.commitInput};
    disp : Dispatch = { true => atomic.abort, atomic.commit, myPoint.* };
end;

```

Figure 16. The Interface part of class *AtomicPoint*.

6. Evaluation and Conclusions

To illustrate the useful features of ACTs, we presented examples in 3 categories: examples of inter-object invariant behavior, inter-object synchronization, and

coordinated behavior. Figure 17(a) shows the relations among the classes as defined in this paper. Figure 17(b) organizes these classes into a layered architecture.

In this section we analyze the composition-filters approach with respect to the problems and requirements we identified in section 2 and 4.1, respectively. First we discuss how ACTs provide solutions to the problems in section 2, and how this is illustrated by the examples in the previous chapter.

1. *Lack of Support for Meta-levels and Reflection:* ACTs can be used for intercepting and manipulating messages. Interception of messages is achieved by the input and output filters of an object, whereas manipulation of messages is made possible by *Meta* filters, since these transform messages into first-class objects. This will allow the software engineer to model and implement layered architectures and extend the message passing semantics of the object-oriented model if needed. Figure 17(b) shows the layered architecture as defined in this paper.
2. *Complexity and Lack of Reusability:* ACTs can make the complexity of programs manageable by moving the interaction code to separate modules. This allows for reducing the number of inter-module relations and hiding communication details. Classes *OneWayConstraint*, *Asynchronous* and *TransactionManager*, for example, represent inter-object interactions. The details of these interactions are abstracted by the methods. Note that *OneWayConstraint*, *Asynchronous* and *TransactionManager* are generic classes and may be used in various applications.

Programmers may apply object-oriented techniques, such as inheritance and delegation, to achieve a more systematic reuse of these components. Inheritance mechanisms will allow software engineers to construct *application frameworks* for different communication protocols. For example, constraint-based systems, distributed concurrency control and recovery protocols, security protocols, distributed scheduling and optimization algorithms, etc. can be expressed using ACTs. The software engineer can tailor these frameworks for his/her particular needs. Properly designed ACTs can be highly reusable.

As illustrated by *BoundedFigure*, ACTs can be extended through the use of inheritance. Another possible extension could be to subclass *TransactionManager*, for instance, to implement weak atomicity for some actions. Thus, the implementation of ACT classes can be changed without affecting the participant objects. For example the implementation of class *TransactionManager* could be changed to *two-phase commit protocol*, without affecting the instances of class *OneWayConstraint*.

3. *Enforcing invariant behavior:* It is easier to enforce the invariant behavior among objects if there is a module explicitly representing this behavior. For example, constraints among objects are enforced by a single class *OneWayconstraint*. Otherwise, all the interacting-code among display objects had to be taken into account.

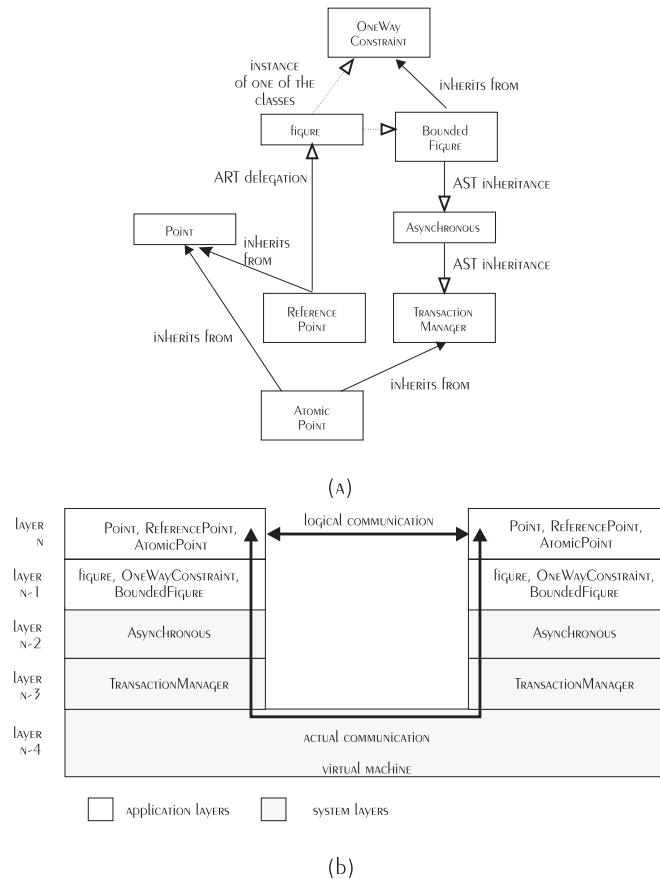


Figure 17. Example classes (a) relations among classes (b) classified into layers of abstractions.

We will now evaluate ACTs with respect to the requirements that were stated in section 4.1:

1. *First-class property:* ACT classes are first-class modules because they are just like other Sina classes. What makes a class an ACT class is that it manipulates messages as first-class objects, and the way it is composed with other classes. Inheritance and/or delegation of behavior is provided for ACT classes through the use of composition-filters.
2. *Large scale synchronization:* ACT classes can implement large-scale synchronization among participating objects. A typical example is class *TransactionManager*. Sina provides mechanisms for concurrency and synchronization since it is a concurrent language [Bergmans et al. 92].
3. *Reflection upon messages:* Through the use of classes *Meta* and *Message*, messages can be manipulated because they are abstracted by the methods of class

Message. For example, classes *OneWayConstraint* and *BoundedFigure* manipulate the arguments of messages to enforce the consistency of dependant objects.

4. *Uniform integration of communication semantics*: ACTs are incorporated with the participating objects by using composition-filters. Since composition-filters also are the basic means for expressing the basic object-oriented data abstraction mechanisms, ACTs are fully integrated with the object model.

The contribution of this paper is to introduce the concept of ACTs. Realization of ACTs is made possible by the introduction of a new type of filter: called *Meta*. Currently, we are experimenting with ACTs in building object-oriented distributed transaction frameworks [Tekinerdogan 92]. We also investigate mechanisms to improve fault-tolerance, for example, by defining ACTs that manage replicated objects transparently. The concept of ACTs as introduced in this paper can be effectively used with the other filter mechanisms presented in our earlier publications. The composition-filter mechanism is adopted by the Sina language and an ICASE environment called *ObjectComposer* [Pool&Bosch 92].

References

- [Agha et al. 92] Agha et al, *A Linguistic Framework for Dynamic Composition of Fault-Tolerance Protocols*, Working paper, Department of Computer Science, University of Illinois at Urbana-Campaign, 1992.
- [Aksit&Tripathi 88] M. Aksit & A. Tripathi, *Data Abstraction Mechanisms in Sina/ST*, OOPSLA '88, pp. 265-275, September 1988.
- [Aksit 89a] M. Aksit, *Abstract Communication Types*, in *On the Design of the Object-Oriented Language Sina*, Ph.D. Dissertation, Chapter 4, Department of Computer Science, University of Twente, The Netherlands, 1989.
- [Aksit 89b] M. Aksit, *Atomic Delegations*, in *On the Design of the Object-Oriented Language Sina*, Ph.D. Dissertation, Chapter 5, Department of Computer Science, University of Twente, The Netherlands, 1989.
- [Aksit et al. 91] M. Aksit, J.W. Dijkstra & A. Tripathi, *Atomic Delegation: Object-oriented Transactions*, IEEE Software, Vol. 8, No. 2, March 1991.
- [Aksit&Bergmans 92] M. Aksit & L.M.J. Bergmans, *Obstacles in Object-Oriented Software Development*, OOPSLA '92, pp. 341-358, Vancouver, Canada.
- [Aksit et al. 92] M. Aksit, L.Bergmans & S. Vural, *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*, ECOOP '92, LNCS 615, pp 372-395, Springer-Verlag, 1992.
- [Aksit&Bosch 92] M. Aksit & J. Bosch, *Issues in Real-Time Language Design*, NATO Advanced Study Institute on Real-Time Systems, Position paper to be published as LNCS, Springer-Verlag, Sint Maarten, October 1992.
- [Barber 89] J. Barber, *Computational Reflection in Class-Based Object-Oriented Languages*, OOPSLA '89, pp. 317-326, October 1989.

- [Bempt 91] M. v.d. Bempt, *Construction of Hierarchies in Distributed Computer Systems*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, November 1991.
- [Bergmans 90] L.M.J. Bergmans, *The Sina Distribution Model*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, March 1990.
- [Bergmans et al. 92] L. Bergmans, M. Aksit, K. Wakita & A. Yonezawa, *An Object-Oriented Model for Extensible Synchronization and Concurrency Control*, Memoranda Informatica 92-87, University of Twente, January 1992.
- [Booch 90] G. Booch, *Object Oriented Design (with applications)*, Benjamin/Cummings Publishing Company, Inc., 1990.
- [Campbell et al. 89] R. H. Campbell & et al. *Principles of Object-Oriented Operating System Design*, Report UIUCDCS-R-89-1510, University of Illinois at Urbana-Champaign, USA.
- [Coad&Yourdan 91a] P. Coad & E. Yourdon, *Object-Oriented Analysis*, 2nd Edition, Yourdon Press, 1991.
- [Coad&Yourdan 91b] P. Coad & E. Yourdon, *Object-Oriented Design*, Yourdon Press, 1991.
- [Champeaux 91] D. de Champeaux, *Object-Oriented Analysis and Top-Down Software Development*, ECOOP '91, pp. 360-375, July 1991.
- [Dolfing 90] H. Dolfing, *An Object Allocation Strategy for Sina*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, November 1990.
- [Francez 86] N. Francez et al, *Script: A Communication Abstraction Mechanism and Its Verification*, Science of Computer Programming, 6, 1, pp. 35-88, 1986.
- [Greef 91] N. de Greef, *Object-Oriented System Development*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, 1991.
- [Haerder&Reuter 83] Haerder & A. Reuter, *Principles of Transaction-Oriented Database Recovery*, ACM Computing Surveys, Vol. 15, No. 4, pp. 287-317, December 1983.
- [Helm et al. 90] R. Helm, I. Holland & D. Ganghopadhyay, *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems*, OOPSLA '90, pp. 169-180, 1990.
- [Holland 92] I.M. Holland, *Specifying Reusable Components Using Contracts*, ECOOP '92, LNCS 615, pp. 287-308, Utrecht, June 1992.
- [Honda&Tokoro 92] Y. Honda & M. Tokoro, *Soft Real-Time Programming Through Reflection*, Int. Workshop on New Models for Software Architecture'92, Reflection and meta-Level Architecture, Yonezawa & Smith (eds), pp. 12-23, November 1992.
- [Ichisugi et al. 92] Y. Ichisugi, S. Matsuoka & A. Yonezawa, *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), pp. 24-35, November 1992.

- [Jonge 92] E. Jonge, *Object-georiënteerde Analyse, Ontwerp en Implementatie van een Batchdestillatiebesturing*, M.Sc. Thesis, Department of Chemical Engineering, University of Twente, The Netherlands, January 1992.
- [Lamping et al. 92] J. Lamping, G. Kiczales, L. Rodriguez & E. Ruf, *An Architecture for an Open Compiler*, Int. Workshop on New Models for Software Architecture'92, Reflection and meta-Level Architecture, Yonezawa & Smith (eds), pp. 95-106, November 1992.
- [Lieberherr et al. 91] K. Lieberherr *et al.*, *Graph-Based Software Engineering: Concise Specifications of Cooperative Behavior*, Northeastern University, Tech. Report: NU-CCS-91-14, September 1991.
- [Lieberman 86] H. Lieberman, *Using Prototypical Objects to Implement Shared Behavior*, OOPSLA '86, pp. 214-223, 1986.
- [Lieberherr&Holland 89] K. Lieberherr & I. Holland, *Assuring Good Style for Object-Oriented Programs*, IEEE Software, pp. 38-48, September 1989.
- [Maes 87] P. Maes, *Concepts and Experiments in Computational Reflection*, OOPSLA '87, pp. 147-155, October 1987.
- [Pool&Bosch 92] S. Pool & J. Bosch, *ObjectComposer ICASE Environment*, OOPSLA '92 conference demonstration, October 1992.
- [Rumbaugh 91] J. Rumbaugh *et al.*, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
- [Smith 82] B.C. Smith, *Reflection and Semantics in a Procedural Language*, MIT-LCS-TR-272, Mass. Ins. of Tech., Cambridge, MA, January 1982.
- [Tekinerdogan 93] B. Tekinerdogan, *The Design of a Framework for Object-Oriented Atomic Transactions*, Draft M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, 1993.
- [Tripathi&Aksit 88] A. Tripathi & M. Aksit, *Communication, Scheduling and Resource Management in Sina*, JOOP, Vol. 1, No. 4, November/December 1988, pp. 24-37.
- [Yokote 92] Y. Yokote. *The Apertos Reflective Operating System: The concept and its Implementation*, OOPSLA'92, pp. 414-434, October 1992.
- [Wirfs-Brock et al. 90] R. Wirfs-Brock *et al.*, *Responsibility-Driven Design*, Prentice-Hall, 1990.
- [Zondag 90] E. G. Zondag, *Hierarchical Management of Distributed Objects*, Memoranda Informatica 90-73, 1990.

Appendix A - Specification of class Message

In this appendix the relevant methods of class *Message* are described. As described, class *Message* has fields for the *receiver* object, the *sender*, the *server*, the *method selector* and the *arguments*.

- `getReceiver` **returns** Any;
returns the receiver of the message object.
- `putReceiver(Any)` **returns** Nil;
changes the receiver of the message object into the argument object.
- `getSender` **returns** Any;
returns the object that sent the message.
- `putSender(Any)` **returns** Nil;
changes the sender of the message into the argument object.
- `getServer` **returns** Any;
returns the object that originally received the message, but that delegated it to the receiver object.
- `putServer(Any)` **returns** Nil;
changes the server of the message into the argument object.
- `getSelector` **returns** Identifier;
returns the method identifier that is stored in the message.
- `putSelector(Identifier)` **returns** Nil;
changes the method identifier into the argument identifier.
- `getArgument(Integer)` **returns** Any;
returns the argument referred to by the integer argument.
- `putArgument(Integer, Any)` **returns** Nil;
changes the argument referred to by the integer argument into the argument object.
- `copy` **returns** Message;
returns a copy of the message.
- `fire` **returns** Nil;
activates the message. If the receiver object is not changed, the message is evaluated by the subsequent filter. Otherwise is the message sent to the new receiver object, where it will be evaluated as any message.
- `reply(Any)` **returns** Nil;
sends the argument object as a reply message to the sender of the message.

Appendix B - The Semantics of the Message System

This appendix gives a formal description of the message system. A message is represented as

$$msg = (o_s, o_r, o_v, \sigma, [a_1, \dots, a_n])$$

Where, o_s is the sender id and o_r is the receiver id, o_v is the server object id, σ is the message selector, and $[a_1, \dots, a_n]$ are message arguments.

The input filter set consists of filters $F_{i,1}, \dots, F_{i,n}$ and the output filters set consists of filters $F_{o,1}, \dots, F_{o,m}$. Each filter F_i has a message queue MQ_{F_i} . A filter of class *Error* is always added as a last filter $F_{i,n+1}$ generating an error if a message is not dispatched so far. A filter of class *Dispatch* is always added as a last filter $F_{o,m+1}$ to send a message once it passed the m output filters.

In appendix A, methods of class *Message* were introduced. Now we will describe the semantics of the methods *copy*, *fire* and *reply*. In A(1), the method *copy* results in a new message with the same structure except the *sender* object is now replaced by *nil_obj*. The method *fire* as defined in A(2), puts the message in the message queue of the next filter. This filter is determined according to the declaration order. The sender of the *fire* message receives *nil_obj* as a result of this invocation. In A(3), the method *reply* sends its argument as a reply message to the sender of the original message. Similar to the previous formula, the sender of the *reply* message receives *nil_obj* as a result of this invocation.

$$\text{copy} \rightarrow (\text{nil_obj}, o_r, o_v, \sigma, [a_1, \dots, a_n]) \quad \text{A(1)}$$

$$\text{fire} \rightarrow \begin{cases} MQ_{O_{R_{F_{i+1}}}} = MQ_{O_{R_{F_i}}} \cup \{msg\} \\ \text{nil_obj} \end{cases} \quad \text{A(2)}$$

where O_R is the reifying object
and F_i is the reifying Meta filter

$$\text{reply}(rep_obj) \rightarrow \begin{cases} rep_obj \xrightarrow{\text{reply}} o_s \\ \text{nil_obj} \end{cases} \quad \text{A(3)}$$

Each message is removed from the message queue of the current filter and evaluated according to the algorithm as described in section 4. The filter can either *accept* the message or *reject* it. In each case the filter will perform some action depending on its type. The actions performed by filters *Dispatch* and *Meta* are described in the following:

The function $\text{execute}(msg)$ is used to start execution of the method as a result of filter evaluation. The *Dispatch* filter is defined in A(4). If the received message is accepted and if the *target* of the message is *self*, then the corresponding method is executed. If, however, the *target* object is not *self*, then the accepted message is put in the message queue of the first filter of the *target* object. If the message is not accepted, then it is put in the message queue of the next filter. The *Meta* filter is

defined in A(5). If the message is accepted, msg is converted into msg' and msg' is put in the message queue of the first filter of the specified ACT. If the message is not accepted, then it is put in the message queue of the next filter. The conversion operation creates a new message msg' with the current object as the *sender*, the ACT object a *receiver* and *server*, the message selector σ_{ACT} as specified in the filter expression and the original message msg as the argument of the message.

$$F_i(msg):Dispatch \rightarrow \begin{cases} \text{execute}(msg) & \text{if accepted and self} = O_r \\ MQ_{O_r} = MQ_{O_r} \cup \{msg\} & \text{if accepted and self} \neq O_r \\ MQ_{F_{i+1}} = MQ_{F_{i+1}} \cup \{msg\} & \text{otherwise} \end{cases}$$

A(4)

where $MQ_{O_r} = MQ_{F_{i,1}}$ of O_r

$$F_i(msg):Meta \rightarrow \begin{cases} MQ_{ACT} = MQ_{ACT} \cup \{msg'\} & \text{if accepted} \\ MQ_{F_{i+1}} = MQ_{F_{i+1}} \cup \{msg\} & \text{otherwise} \end{cases}$$

A(5)

where $(\text{self}, ACT, ACT, \sigma_{ACT}, [msg]) = msg'$

and $MQ_{ACT} = MQ_{F_{i,1}}$ of ACT

and $ACT = O_r$