The Filter - A Paradigm for Interfaces

Raimund K. Ege
Oregon Graduate Center

Technical Report No. CSE-86-011
September 1986

Abstract

Interfaces play a crucial role in today's computer technology. Much effort is spent to design and program user interfaces. This paper describes a new approach to this area of research that is based on the concept of separating the presentation from the data, and describing the presentation declaratively. The source and view of data are then related using constraints to control the interface between them. Source and view objects are strongly typed to allow construction of higher-level interfaces from composition of lower-level units.

This paper introduces the basic concepts of object, constraint and filter and shows how they can be used to describe an interface. The syntax and semantics of the object and filter type definition is given and related to the theory. Objects and filters are built from basic atoms using constructors, inheritance and constraints. The critical issues in a possible implementation are described. The paper concludes with ideas on how to build an interface specification tool based on the filter paradigm and illustrates a geometric theorem as sample interface to a data structure.

1. Introduction

Interfaces are a crucial part of any computer, not only between users and the computer, but also between programs running in a computer system or between different components of a computer system. The quality of an application is partly judged by the quality of the user interface. Significant effort is spent on designing and programming the interface part of any application. This research is aimed at reducing this effort. One goal is to provide the designer with a method or model to produce interfaces that are acceptable to the user in respect to style, usability and efficiency. Another goal is to reduce programming by automatically generating interfaces and re-using parts of existing interfaces. We are not proposing a particular style of interfaces, but a new abstraction for building interfaces.

Object-oriented Systems

In an object-oriented environment an application is represented by objects. These objects communicate with each other and with the user of the application. An important step towards efficient user-interface design was made when it was realized that interface and application should be decoupled. Thus the interface is also represented by objects. This research is concerned with workstations that are able to display a great range of objects using high-resolution bitmap displays. The user perceives the objects presented to him on the screen and interacts with them by using the input devices of a modern workstation. Interface objects and application objects are related. For example, the user sees a graphical representation of a tree, however the application knows of the tree as a nested collection of records. Conceptually, the tree exists only once and has many aspects. Each participant in such a system looks at an object in the universe and models it in his own world. In the tree example, the user looks at a graphical image of the tree consisting of bits on the bitmap display, the application program views the tree in terms of bytes and addresses of memory. We can picture that abstraction as looking at an object in the universe through telescopes using different filters. Figure 1.1 illustrates this conceptual model. User and application each have their own view of the universe.

To define an interface according to this metaphor we could initially specify the object in the universe and the filtering mechanisms for both participants in the interface. The specification of the object would be very elaborate because in describing the object we have to

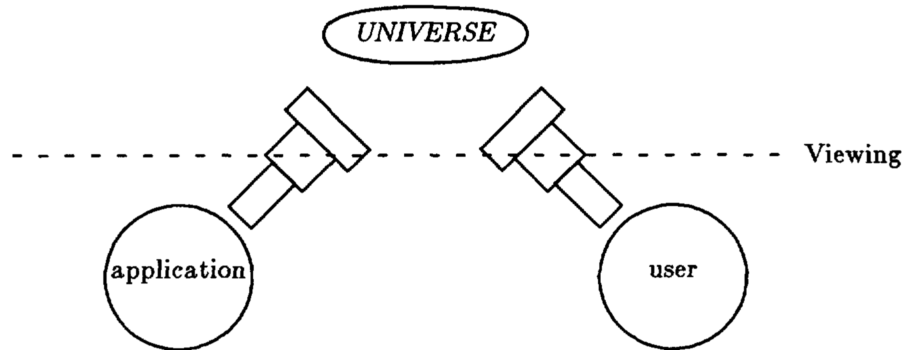

Figure 1.1: Conceptual model.

consider all the perspectives from which we ever want to look at it. The filtering mechanisms, however, would simply pass through a portion of the object information without change. Instead, the approach we are taking is to define the two objects that result from the filtering. The object in the universe disappears and the two filtering mechanisms are combined into one. So, instead of having the user and the application look at the same object using a filtering technique (Figure 1.1), we define two separate objects, one in the user's reality and one in the application's reality, which are connected via a control mechanism (Figure 1.2). We could call the control mechanism a 'channel' [Kay 83] or a 'mediator' [Goldberg 85], but because we want to reflect the original idea of filtering aspects of an object in the universe into the reality of the interface, we name it 'filter'.

Constructing Filters

Our notion of an interface has three parts as illustrated in Figure 1.2. Two objects, ***source*** and ***view***, are connected by a ***filter***. The ***source*** and ***view*** are objects that can be part of the application's or the user's object space. The ***filter*** component relates the two objects. It

Figure 1.2: Filter paradigm.

constrains the *source* and *view* objects to be representations of a conceptual object that has them as two different views. Both objects belong to their respective environment, which could be the memory of a program in an application or the user's display screen. In general, a filter can connect any two objects. Bigger filters can be constructed from smaller filters using intermediate objects or sharing subparts of larger objects. Thus we can build filters from subfilters, but the result is still a *filter* connecting a *source* and *view* object. If either object is changed the *filter* has to enforce the conceptual equality. If the application changes the data in the memory of the program, this change has to be reflected on the screen. If the user expresses a change to the representation on the screen, the memory of the program is updated.

In contrast, consider a screen editor. The data on the screen (*view*) reflects the contents of a file stored somewhere on a disk (*source*). The communication protocol between the two objects, screen view and disk file, is well defined. The disk file is displayed initially, the user updates his screen view and finally the new version is saved back to the disk file. Here the equality of the two objects is not maintained at all times. The constraint enforcement is separated into two phases, one at the beginning of the editing session and one at the end. Another example is a spreadsheet program where relations between objects (numbers) can be expressed by equations. Subsequent numbers can be defined in terms of previously defined numbers. In common spreadsheet programs changes are only forwarded in one direction through the equations. In our notion, changes on either side of the equation are reflected on the other side.

This paper represents the first step towards the goal of automatic generation of interactive displays using the filter paradigm. It defines a filter specification language. Future research will be aimed at implementing this language using constraint satisfaction techniques and at an interactive tool to specify interfaces graphically. In the next section we describe related ideas and distinguish our approach from the ongoing research in the field. Section 3 introduces the basic concepts, which are objects, constraints and filters. Section 4 gives syntax and semantics for the object and filter specification language. Section 5 discusses the issues that concern the implementation of a prototype interface with filters. We conclude this paper with ideas on how to build an interface specification tool based on the filter paradigm. In the appendix we describe an implementation of a geometric theorem using our filter paradigm.

2. Related Work

The goal of this research is to provide a high-level specification of interfaces and a good model for modular construction of displays that will allow automatic generation of interactive displays. The areas that are involved in seeking a solution are:

- interface design and specification
- displaying objects in an object-oriented environment
- constraint languages and satisfaction systems.

In designing any interface, we have to realize the two major issues of communicability, which is how the user can express his wishes, and perceptibility, which is how the user perceives the data presented to him [Bornique 85]. Thus, an interface will have two components. Viewing an interfaces as a two-way street, a user interface management system can implement a mediator to control an input and output pipe [Takala 85]. Editing can be used as an abstraction of user interaction [Scofield 85]. Also, there are semantic issues to be observed. Using the Vienna Development Method, a group at University of Stuttgart [Studer 84] defined dialog concepts not only in terms of windows, menus, etc., but also for interactive concepts such as user input, error

handling and undo operations. Several other theoretical approaches using algebraic techniques can be used to specify user interfaces and interaction axiomatically [Chi 85].

Our approach was guided by experience with the Smalltalk model-view-controller (MVC) paradigm [Goldberg 83]. This paradigm employs the idea that all data are kept by a model. The presentation is kept in a view and a controller handles the interaction. As mentioned in the first section, this arrangement makes it necessary that the model knows about any aspect from which it can be observed. Programming experience has shown that this paradigm is hard to follow. The Smalltalk Interaction Generator (SIG) tried to add a declarative interface on top of the MVC mechanism [Nordquist 85, Maier 86]. Objects need type information to guide an automatic display. The Incense system [Myers 83] uses type information supplied by a compiler to display objects. The user can influence the display format but cannot update through this system. The display function in Allegro [Ege 84] also deals with viewing database information using the scheme of a network database system.

Constraints are used to specify relations and dependencies in a so-called *active* database interface system [Morgenstern 83]. Other systems use constraints as their major construct, such as ThingLab [Borning 79], which allows constraints to be expressed in a graphical manner. An early system that employed constraints to express graphical relations was Sketchpad [Sutherland 63]. The language Ideal, used in typesetting graphical pictures, is based on constraints and demonstrates their power and usefulness[1] [Van Wyk 82]. Bertrand [Leler 86] is a language that can specify and generate constraint satisfaction systems. It has been demonstrated that it can be used to build graphics constraint languages.

3. Basic Concepts

As described in the first section of this paper, our filter paradigm employs objects, constraints and filters. In an object-oriented system objects represent entities that we want to model. These objects have structure that are defined by their types. Constraints can be defined for an object and between objects. A filter is an object that represents a constraint that is defined between two objects of specific types. Objects can be connected by filters to form an interface. Objects in such an interface are called *source* and *view* objects. The *source* object is displayed by the *view* object when the interface is used for displays, but we can use filters to construct other interfaces. Section 3.1 gives an introductory example. Sections 3.2, 3.3 and 3.4 explain the basic concepts of object, constraint and filter and give examples. Section 3.5 covers the introductory example in more detail.

3.1. QuadArrayMirror Example

As introductory example, let us consider objects that are QuadArrays. A QuadArray is a tree-like data structure. Each node in the structure has either four subnodes or is a leaf node. A text string is attached to each node as a label. As interfaces can be constructed between any types of objects, we want to construct an interface between two objects of type QuadArray. The task of the interface is to act as a mirror, reflecting one object to the other. The text labels at the nodes and the structure of the QuadArrays are to be reversed. We start out to define a subpart of the interface that reverses a text string. Each label on one side of the

[1] Figures 1.1, 3.3, 4.10 and 5.4 of this paper were produced using Ideal. The implementation of some graphical objects described in Section 5.2 is similar to Ideal.

interface needs to be related to the corresponding label on the other side of the interface in terms of text reversal. We call such a relationship a filter in general, and TextMirror filter for this specific relation. This relationship can be expressed as a constraint. The constraint is not static. If the label on one side is changed, the other label is changed accordingly, preserving the mirror interface. If the QuadArrays on either side of the interface consist of only one node, this TextMirror filter would be a correct implementation of our interface, if the nodes containing the labels are otherwise identical. If subnodes are added to the QuadArray, the filter has to allow TextMirror subfilters for the label of the subnodes. TextMirror filters then connect the label of the first subnode of one side to the label of the fourth subnode of the other side, etc. By constructing an interface for QuadArrays that have one sublevel from TextMirror filters, we have created a new filter. The new filter not only holds information on which nodes are connected to which other nodes, but also knows the structure of the connections. The concept of adding four more TextMirror subfilters for each sublevel can now be applied recursively to define the desired interface as a QuadArrayMirror filter.

Figure 3.1 illustrates this example as it was implemented in Smalltalk[2]. The QuadArray

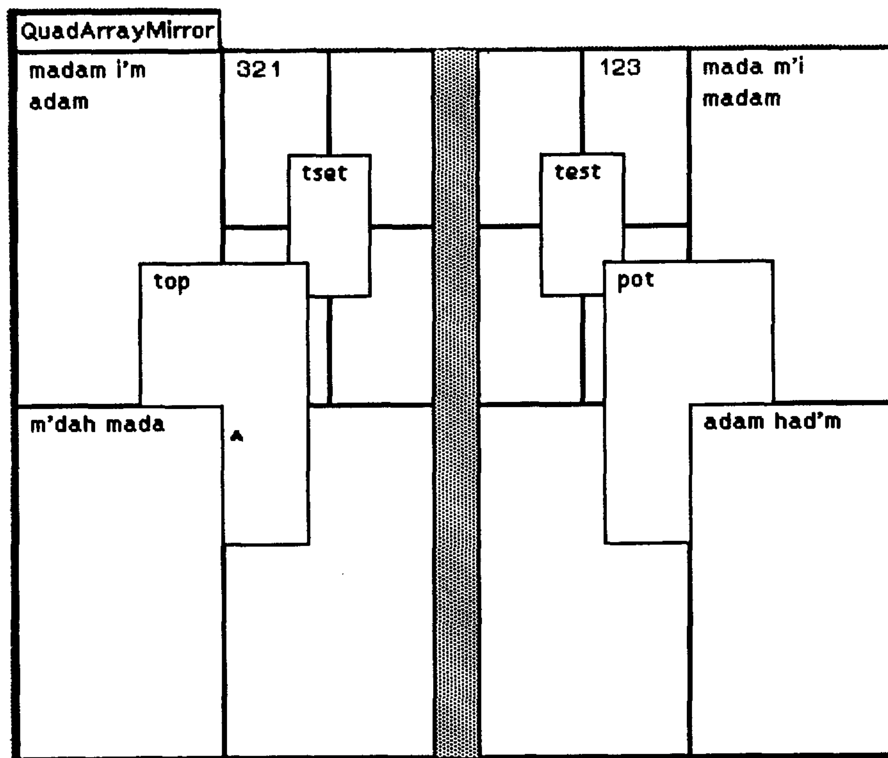

Figure 3.1: QuadArrayMirror interface.

[2] See Section 5 on implementation.

objects are displayed using the SIG system [Nordquist 85]. Two QuadArrays are visible. We look at them from the top. The node in the middle is the top node and is surrounded by its subnodes. The subnodes are again QuadArrays that can have subnodes.

This examples illustrates that, in order to create interfaces, we need information about the type of the objects that participate, that objects can be related using constraints, and that higher-level interfaces can be constructed from filters.

3.2. Objects

Objects are present at the two sides of a filter. If the interface is more complex, objects can also serve as intermediates in a chain of sub-interfaces. We want to define types for objects. Types are needed because the filters need information about the structure of the *source* and *view* objects, and because we want to check the type of an object when we use it to compose filters to create larger interfaces. The type is defined in terms of attributes an object has. The attributes can be of any known type. Following the syntax of well-formed terms [Ait-Kaci 84], we call attributes ***addresses***. We distinguish between ***atomic*** objects, which are of an atomic type such as integer, boolean, character, and ***structured*** types, which are formed from atomic types using the structural constructors set, array, condition and recursion. Addresses for a type can be defined explicitly. In addition a type inherits all addresses from all its subsuming types. Conditions can be placed on addresses to ensure well-formedness of a type.

An example object type is **QuadArray** (Figure 3.2). It contains three addresses: *label*, *elements* and *subarray*. They denote the constituent types **TextString**, **Integer** and array of **QuadArray**. We assume that the type **TextString** is already defined. All types include the value NIL, which expresses the fact that the address is undefined. In the example, this value can be used to terminate the recursion. The iteration in the *subarray* address uses the address *elements* to express the iteration factor. The *elements* address is also used in the **constraint** statement, which constrains the value held at this address to be the integer constant 4.

An important notion is subtyping. Subtypes can be defined implicitly by using the same and more addresses as an existing object type, or explicitly by inheriting addresses from another object type. This inheritance leads to a type hierarchy, which can be useful when determining which objects can be plugged together in filters to form an interface. If object types belong to a type lattice (Figure 3.3), then a filter that is defined for a specific object type can also be applied to all its subtypes. The symbols '$\top$' and '$\bot$' denote the top and bottom in the type hierarchy. Ait-Kaci describes algorithms to compute upper and lower bounds in a type hierarchy [Ait-Kaci 84]. As an example, consider an interface that is built from a filter that is defined for a *view* object of type DisplayMedium. The filter constrains a *source* object to be displayed

---

Object Type QuadArray
    label → TextString
    elements → **Integer**
    subarray [elements] → Quad_Array
    constraint IntegerIdentity (elements, 4)
end

Figure 3.2: QuadArray object type.

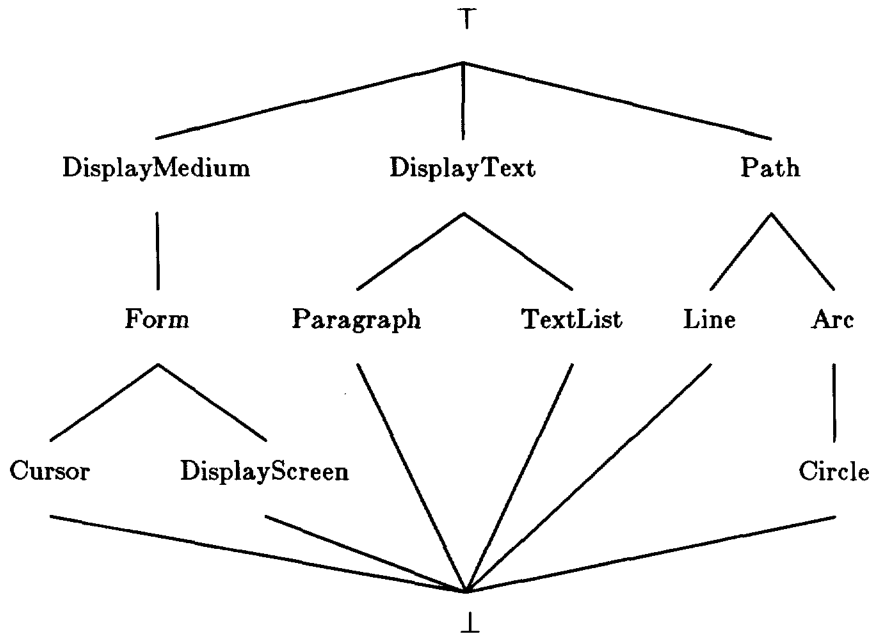

Figure 3.3: Type lattice for display classes in Smalltalk.

on the DisplayMedium. The DisplayMedium is only an abstraction for different subtypes. So, we expect this filter to display the same *source* object on a *view* object of type DisplayScreen, which is a subtype of DisplayMedium.

3.3. Constraints

Constraints are the backbone and the basic building tool in our filter paradigm. We have to distinguish three areas where we use constraints. First, filters represent constraints and are used to express interfaces. Second, constraints are used to place conditions on objects as part of the object type definition. Third, complex constraints, for e.g. arithmetic, can be solved using an external constraint satisfaction system.

Our filter paradigm is based on the idea that we constrain the participating objects relative to each other. We provide a model to structure this relation between the *source* and *view* objects. The QuadArrayMirror filter (see Section 3.1) illustrates how to construct a mirror interface by decomposing the overall mirror constraint into subfilters. Thus, a way to implement the filter paradigm would be to compile the filter description into a constraint. This constraint can be solved and maintained using existing constraint satisfaction systems [Borning 79, Leler 86]. In viewing the interface description as an overall constraint, we can allow interpretation, compilation and optimization of the interface. The constraints also help us in discussing the semantics of the interface.

Constraints are also used to express conditions that an object must fulfill. For example the object QuadArray (see Figure 3.2) has exactly four subnodes. But we could also impose more complex restrictions on an object and an object can restrict addresses that it has inherited. As an example, consider the Cursor subtype of Form (see Figure 3.3). In its type definition it restricts the display bitmap to have height and width 16. This mechanism is very helpful when defining types.

The basic building block in our paradigm is a filter atom. The QuadArrayMirror filter used the TextMirror subfilter, which is built from CharacterIdentityAtom filter atoms[3]. We do not expect that all interfaces can be decomposed to identity subfilters. Thus, we provide an interface to a constraint satisfaction system[4]. For example, if we want to describe graphical objects, we need arithmetic constraints to express basic positional relationships. To define a point as the middle of a line between two other points we need to solve the constraint:

Middle = (PointOne + PointTwo) / 2.

In the appendix we give details of how this constraint can be implemented.

3.4. Filters

The filter represents a constraint that has to be maintained between two objects. The filter is defined for specific types of objects and is identified by its type name. The QuadArrayMirror filter example in Section 3.1 illustrates how we can decompose an interface using filters. The TextMirror subfilter is itself defined in terms of atomic subfilters. We have to distinguish filter atoms, which have to be provided, and higher-level filters, which are constructed from filter atoms or other constructed filters. We will refer to constructed filters as ***filter packs***.

Our filter specification languages provides constructors to declare ***filter packs***. The set constructor declares several arbitrary subfilters. The iteration constructor declares a number of identical subfilters. The condition constructor declares a subfilter if an expression is true. Recursion allows us to declare subfilters recursively. Each constructor establishes subfilters and also keeps information on how the subfilters are related. A subfilter is established by giving its type name and associating *source* and *view* objects to it. Associating *source* and *view* objects to a subfilter is done by passing references to the subfilters.

In general, we can distinguish end-to-end and side-by-side subfilter combination. In end-to-end construction, the filter is composed from two subfilters. The *view* object of the first subfilter and the *source* object of the second subfilter agree on a common intermediate object. Figure 3.4 shows a filter pack that constructs an identity filter for integers from two filter atoms (IntegerCharacterAtom and CharacterIntegerAtom), which serve as conversion filters between an integer and a character object[5]. The intermediate object is of type character. The make statement tells what subfilters to instantiate and how to relate their sources and views in order to instantiate a PseudoIdentity filter. Note that the filter definition introduces a local variable that constitutes the connecting object. The resulting filter is not exactly an identity filter because the intermediate object is of type character. The integer to character conversion filter atom restricts the participating integer numbers to one digit. Figure 3.5 diagrams the filter pack using end-to-end construction.

In side-by-side construction the filter is composed from a set of two or more subfilters. The *view* object of the first subfilter and the *view* object of the second subfilter are part of the

---

[3] Equality filter atoms are described in Section 4.3.1.

[4] See Section 4.3.3 on constraint filter atoms.

[5] See Section 4.3.2 on conversion filter atoms.

```
Filter Type PseudoIdentity ( source : Integer , view : Integer )
        var
                c → Character
        make set of
                IntegerCharacterAtom ( source , c )
                CharacterIntegerAtom ( c , view )
end
```

Figure 3.4: PseudoIdentity using character intermediate object.

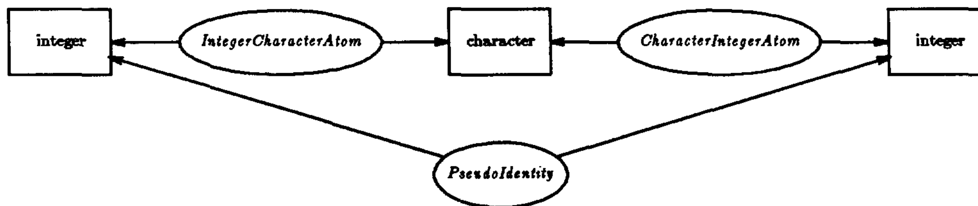

Figure 3.5: PseudoIdentity as a filter pack.

view object of the constructed filter pack. Analogously, the *source* objects are part of the *source* object of the filter pack. Figure 3.6 and 3.7 illustrate the NumberString filter. It is defined for objects of type Dual and Pair. The object type Dual defines an array of two characters. The object type Pair defines two integers digits. One integer represents the first digit of a number, the other the second, which are concatenated into a string representing the whole number. The filter is built from two instances of the IntegerCharacterAtom filter atom, which was also used in the last example. The IntegerCharacterAtoms are instantiated with references to their *source* and *view* objects that are part of the *source* and *view* objects of the NumberString filter.

3.5. QuadArrayMirror revisited

Using the object type definition of QuadArray from Figure 3.2, we now give a more detailed description of the QuadArrayMirror interface example. This filter is constructed, using the set constructor, from a TextMirror subfilter[6] and four QuadArrayMirror subfilters. Figure 3.8 illustrates the filter definition. The **set of** statement establishes one subfilter for the label subparts of the *source* and *view* objects, and a conditional subfilter constructor for the subarrays. The condition constructor establishes the iteration subfilter constructor only if one of the subarrays of *source* and *view* is not NIL. The iteration construct establishes QuadArrayMirror

[6] We assume for this discussion that the TextMirror subfilter is already defined.

```
Object Type Dual
        field [ 2 ] → Character
end

Object Type Pair
        first → Integer
        second → Integer
end

Filter Type NumberString ( source : Pair , view : Dual )
        make set of
                IntegerCharacterAtom ( source.first , view.field[1] )
                IntegerCharacterAtom ( source.second , view.field[2] )
end
```

Figure 3.6: Object and filter type definition for NumberString.

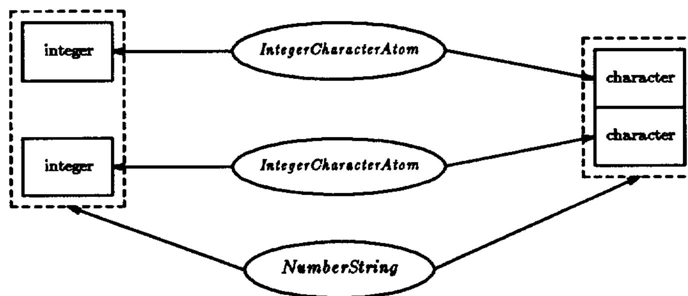

Figure 3.7: NumberString filter.

```
Filter Type QuadArrayMirror ( source : QuadArray , view : QuadArray )
    var  factor → Integer
    make set of
            TextMirror ( source.label , view.label )
            condition source.subarray ≠ NIL or view.subarray ≠ NIL
                    iteration factor times i
                            QuadArrayMirror ( source.subarray[i], view.subarray[factor-i+1] )
    merge source.elements =○= factor
            view.elements =○= factor
end
```

Figure 3.8: Filter type for QuadArrayMirror.

subfilters recursively. The iteration factor serves as a local variable that is bound to the

elements address of the *source* and *view* objects. The symbol '=○=' expresses that fact in the **merge** statement.

This example has been implemented in Smalltalk for experimentation. QuadArray objects are instances of a Smalltalk class. The QuadArrayMirror filter is implemented as an object that keeps track of two QuadArrays to be controlled. Each subfilter connects one node in the *source* object to one node in the *view* object. The QuadArrayMirror filter is capable of creating and deleting subfilters automatically as the structure changes in the QuadArrays. All constraints representing the filter were translated into Smalltalk code.

Using end-to-end composition (see last section), we can now build a higher-level filter connecting two objects of type QuadArray using a third, intermediate, object of type QuadArray and a set of two QuadArrayMirrors. The result is an identity filter. Obviously, there is a much simpler filter equivalent to this construction. An optimizing implementation could detect the redundancy and eliminate the middle object.

This example showed an interface from objects of type QuadArray to objects of the same type. But what we really want would be an interface from a terminal to objects of type QuadArray. In order to build such an interface we will have to define filters that take the QuadArray step by step, in end-to-end combination, to the desired screen representation. Then we connect these subfilters and all the necessary intermediate objects, and so define an interface. Section 5 on implementation describes some of the concepts that we need when we approach the screen, such as how to render graphical objects and sensor input according to our filter paradigm.

4. Filter Specification

4.1. Introduction

A filter defines a mediator between two objects. We described earlier how we understand that logical relationship and how we hope to build interfaces based on this notion. This section presents a complete definition of a filter specification language. We present a syntax for a filter type definition. A filter type provides the framework and structure for building a filter relation between the underlying objects. The *source* and *view* objects are typed in order to insure well constructed filters and legal filter compositions. When composing filters together to form a larger interface, the intermediate objects have to be checked as to whether they fit the type framework. Filter composition can also impose constraints on the *source* and *view* objects.

Object types are built from atomic types by grouping previously defined types together to form new types. Several grouping mechanisms are provided. New object types can be defined as subtypes of existing types to create a type hierarchy. Filter types are built from atomic filters, which establish low level relationships between object types, or by grouping according to provided mechanisms. In order to do construct filter types correctly, type information from the underlying *source* and *view* objects is needed. The structure of the filter type reflects the structure of the constituent objects.

The syntax for object and filter types is given in Section 4.2, 4.3, and 4.4. Examples are given that illustrate the definitions. The object type examples in Section 4.2 are used in the filter type examples of Sections 4.3 and 4.4. The meaning of a filter specification is discussed in Section 4.5 on semantics. This specification is done in terms of an informal description of semantic functions. The discussion distinguishes between the static and behavioral case, where the static case is for creating a filter from existing objects, and the behavioral case deals with

input or change behavior.

4.2. Object Types

In spite of the cliché that "typing is for people with poor memory and is only for the benefit of the compiler," in order to allow filter composing and to ensure correctness of a filter construct, we need the notion of type for the objects that participate in the filter definition. The idea is to separate type checking from run-time to get well-formed programs. This idea is well-founded in the literature [Milner 78].

We want to support the notions of aggregation and specialization as a type definition mechanism [Albano 83] [Borgida 84]. With aggregation we can build higher level objects from lower level components. Specialization allows us to refine an existing type by adding more type information to it.

Aggregation builds objects from components. Each component has an address and is of an object type that is already defined. The lowest level components, from which any object type is built, are predefined atomic types. For atomic types we allow:

- **Integer** for integer numbers
- **Character** for single characters
- **Boolean** for truth values 'true' and 'false'
- **Bit** for bit values '0' and '1'

All object types include the value NIL that denotes the undefined value.

In specialization we can refine an existing object type by inheriting its components and adding new ones. We are using strict inheritance, which means that all components are inherited by the specialized object type. This is in contrast to default inheritance, where not all attributes have to be inherited [Borgida 84].

These aggregation and specialization mechanisms allows us to build a type hierarchy in two ways: Explicitly, by specializing existing object types, and implicitly, by aggregating the same components as in an existing types plus more others [Albano 83]. In addition, all object types can impose intra-object constraints on their components. Thus we can create a type hierarchy or a partial order on the object types.

Object types are defined using the notion of well-formed terms [Ait-Kaci 84] extended with some syntactic sugar to express condition and iteration in an easy way. They are defined as follows:

```
Object Type <Name>
        inherit from <object types>
        <address_expr_list>
        constraint <constraint_name>
end
```

Where

<Name>
: is a unique name for this type of object. Subsequent object and filter definitions can use this name when constructing more complex objects or filters. The name has to start with a capital letter.

<object types>
: are the names of previously defined object types, from which components are inherited. If inherited components have the same address in more than one subsuming type, then the

addresses are concatenated with the names of the object types where they are defined.

<address_expr_list>
: establishes the structure of the object. It is built from address expressions. There are address expressions of basic, iteration and condition form, which are defined below.

<constraint_name>
: is the name of a constraint that has to be satisfied for this object type to be well-formed. The constraint could be a previously defined filter type or a constraint that is supplied from outside. The constraint or filter can name addresses from the current object type definition or from supertypes that were mentioned in the **inherit from** statement.

The **inherit from**, <address_expr_list> or **constraint** statements can be omitted if not needed. The <address_expr_list> is a list of one or more address expressions. The basic address format is :

<address> → <object_type>

where <address> is a label for this subpart of the definition and <object_type> refers to another **Object Type** definition or to an atomic type. We also allow literals of an atomic type. Literals of type **Integer** are denoted by numbers, of type **Character** by quoted characters, of type **Boolean** by the words 'true' and 'false', and of type **Bit** by '0B' and '1B'. If a literal is specified in an object type definition, then this address will be constant for all instances of the object type.

Basic address expressions can be grouped together to form a record-like structure. Consider the list of integers '(1,2,3,4)'. In order to specify an object type for this particular instance, we group together basic address expressions of type **Character** and **Integer**. The character types are denoted by the literals '(', ',' and ')'. The integer types are denoted by the

```
Object Type Array_1
    subfield [4] → Integer
end

Object Type Array_2
    label → Integer
    dependents → Array_1
end

Object Type List_1
    str_1 → '('
    sub_1 → Integer
    str_2 → ','
    sub_2 → Integer
    str_3 → ','
    sub_3 → Integer
    str_4 → ','
    sub_4 → Integer
    str_5 → ')'
end

Object Type List_2
    label → Integer
    dependents → List_1
end
```

Figure 4.1: Sample object types.

name of their atomic type. Figure 4.1 shows object type List_1[7], which specifies the type discussed above. The other object types in this example are described later in this section and are used as *source* and *view* object types in the discussion of filter types later in this paper.

For succinctness, when multiple subparts of the same type are needed, we introduce the iteration address which has the form:

<address> [<iteration_factor>] → <object_type>

where <iteration_factor> specifies how many times this address should be replicated. The <iteration_factor> can be an integer constant or an arithmetic expression that evaluates to integer. Figure 4.1 shows object type Array_1 where 4 subfields are summarized as an iteration. An instance of this object type could be: '1 2 3 4', modelled as an array of integers. Figure 4.1 shows object types Array_2 and List_2 that reference not only atomic types but also previously defined object types, such as Array_1 and List_1.

To express the fact that object types can have variable structure we introduce the conditional address of the form:

(<union-condition>) : <address_expr>

where the <address_expr> exists only if the <union-condition> evaluates to true. The <union-condition> has to evaluate to type **Boolean**. The expressions used in the union-condition have to be bound in the current environment, which includes other addresses and properties of the object type. Figure 4.2 shows object types Array_3 and List_3 with a conditional address expression. To express an object type in terms of itself we need to introduce recursion. Recursion is specified by using the object type name of the current definition in the <object_type> specification of the address expression. Mutual recursion is also allowed. Figure 4.2 shows object types Array_4 and List_4, which are defined recursively. Note that recursion permits cycles in our data structures. To illustrate the data structures, Figure 4.3 shows graphs of some of the types defined so far.

This type specification for objects is closely related to well-formed terms [Ait-Kaci 84], but he has no constructs for iteration and condition in well-formed terms. Iteration can be

```
Object Type Array_3                 Object Type Array_4
    label → Integer                     label → Integer
    ( label = NIL ) :                   ( label = NIL ) :
        dependents → Array_1                dependents → Array_4
end                                 end

Object Type List_3                  Object Type List_4
    label → Integer                     label → Integer
    ( label = NIL ) :                   ( label = NIL ) :
        dependents → List_1                 dependents → List_4
end                                 end
```

Figure 4.2: Sample object types with condition and recursion.

[7] Note that this type does not capture the order of the fields the way an array would.

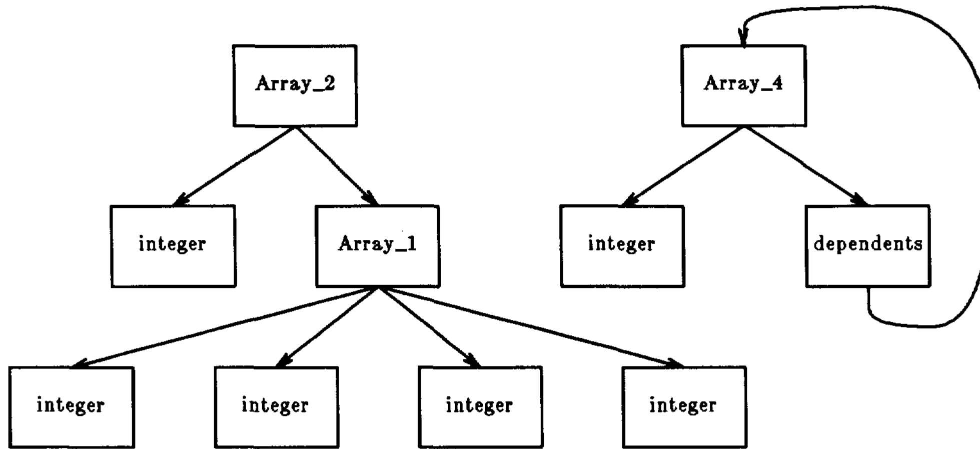

Figure 4.3: Graphs of sample object types.

thought of as pure notation simplification, except when the iteration factor depends on some other part of the object. Condition can be viewed as the union of two separate well-formed terms, one including the address expression of the conditional address, the other not. The first object type is a subtype of the second.

Subtyping allows us to build a type hierarchy. A type hierarchy is useful because it allows us to define general filters that not only can connect objects of a given type but also all their subtypes. An object type can implicitly be a subtype of another object type, by using the

```
Object Type Array_5
        size → Integer
        subfield [ size ] → Integer
end

Object Type Array_6
        inherit from Array_5
        name[10] → Character
end

Object Type Array_7
        inherit from Array_5
        constraint IntegerIdentity(size,4)
end
```

Figure 4.4: Sample object types using inheritance.

addresses and constructs in its definition, or it can be explicitly defined as a subtype of some other object type. If an object type definition names another object type in its **inherit from** statement, then all the components from that object type are inherited. Figure 4.4 shows object type Array_5, which defines an array of integer. The size of the array is stored within the definition. Array_1 of Figure 4.1 is an implicit subtype of Array_5. Array_6 of Figure 4.4 is an explicit subtype of Array_5. Operationally it does not matter whether an object type is an implicit or explicit subtype of another object type.

In the examples so far, we explained how to build structured objects. If we want to impose conditions on these structures we can use the **constraint** statement, which allows us to name filter types or constraints to insure the well-formedness of an object. This concept corresponds to the intra-object constraints discussed earlier. The constraint can name explicit addresses or addresses that are inherited from the supertype. In contrast to the condition address expression, the **constraint** statement does not directly affect the structure of the object type but imposes restrictions on the values that are stored in an object of such a type. Figure 4.4 shows Array_7, which is also a subtype of Array_5, but it names the constraint that the number of subfields is always four[8]. Array_7 denotes the same object type as Array_1 of Figure 4.1.

Once we have defined object types, we can create instances of it. We need instances of object types when we define source and view objects for a filter or as variables in a filter type specification (see Section 4.4). An object is instantiated by giving its name and an instantiation list. The instantiation list contains pairs of address and initial value. The object is then instantiated using the specified initial values.

An object is accessed by traversing a path of addresses. To access subparts of the objects we have to specify a path through the object structure by using the addresses in a dot notation: 'myList.label' for a variable 'myList' of type List_3 at address 'label'. We distinguish direct and delayed access. In direct access the structure of the object is traversed according to the path and the correct object is returned. In delayed access we store the object identity together with the path to allow access on need at a later time. Delayed access is needed because it is possible that an object does not comply with the given path at the time when the path is defined, but may be changed in a dynamic environment. It is also possible that objects along the path change, thus changing the result of the path evaluation.

4.3. Filter Atoms

A filter enforces a constraint between its *source* and *view* object. It serves as a control element between the two objects. The objects are constructed from atomic types. In order to control the objects, a filter has to constrain the relations of their atomic types and the relation of their structure. A filter atom connects two objects that are of atomic type or are of a type where the substructure does not matter[9]. A filter is therefore built from filter atoms, which are composed by using filter constructors as described in the next section on filter packs.

The idea is to build filter incrementally from filter atoms. Filter atoms are predefined or imported from an external constraint satisfaction system. Each of these filter atoms has a filter type. In order to use it in a filter pack definition it has to be named in the **make** statement using the syntax:

---

[8] Constraints name addresses. The integer constant four has no address. It is a value. A possible compiler for the filter description has to detect that and create an object that has a constant value of four and cannot be changed, i.e., is anchored during constraint satisfaction.

[9] As example consider the constraint filter atom in Figure 4.5.

<filter atom name> (<source object> , <view object>)

or

use <variable> **with** <signature> (<source object> , <view object>)

where <filter atom name> is the name of the filter, <variable> is a variable that will hold the name of the filter at the time when it is instantiated, <signature> gives the types of source and view object as a pair, and <source object> and <view object> are path expressions that can be evaluated to yield the corresponding objects for source and view when the filter is instantiated. The *source* and *view* objects are identified by position.

We distinguish four groups of filter atoms. There are:

- equality filter atoms
- conversion filter atoms
- constraint filter atoms
- implementation filter atoms

Each group represents a class of filter types. Note that we always deal with relations between types. In our filter type definition we are only concerned about types. What the filter type mechanism establishes is a framework of relations between object types that could be thought of as slots, which are filled when the filter is instantiated with objects of the appropriate type. This framework constrains the value of objects that fill the slots. Notice, we cannot constrain two integer objects to be the same, but we can build a filter framework that will constrain two integer addresses to hold the same value.

4.3.1. Equality Filter Atoms

For each of the atomic object types there is an equality filter atom. The filter atom represents an equality constraint. The following filters are predefined[10]:

- IntegerIdentity (**Integer** , **Integer**)
- CharacterIdentity (**Character** , **Character**)
- BooleanIdentity (**Boolean** , **Boolean**)
- BitIdentity (**Bit** , **Bit**)

Equality filter atoms ensure that their associated *source* and *view* objects hold the same value. They are implicitly defined on the participating type, e.g., 'IntegerIdentity' is defined as a filter from type **Integer** to type **Integer**. There is no directionality implied among the objects for dynamic changes. However, when a filter atom is first instantiated we will propagate the value from the *source* object to the *view* object if necessary. Changes on either side will be propagated to the other side.

4.3.2. Conversion Filter Atoms

Similar to equality filter atoms, conversion filter atoms represent constraints. Conversion filter atoms are given between the atomic types, such as:

- IntegerCharacterAtom (**Integer** , **Character**)
- IntegerBooleanAtom (**Integer** , **Boolean**)
- IntegerBitAtom (**Integer** , **Bit**)

- CharacterIntegerAtom (**Character** , **Integer**)
- CharacterBooleanAtom **Character** , **Boolean**)
- CharacterBitAtom (**Character** , **Bit**)

---

[10] Equality filters can be defined for arbitrary structured objects using the appropriate equality filter atoms and the filter constructors according to the object structure.

- BooleanIntegerAtom (**Boolean** , **Character**)
- BooleanCharacterAtom (**Boolean** , **Character**)
- BooleanBitAtom (**Boolean** , **Bit**)

- BitIntegerAtom (**Bit** , **Integer**)
- BitCharacterAtom (**Bit** , **Character**)
- BitBooleanAtom (**Bit** , **Boolean**)

It is obvious that the object types of these conversion filter atoms are restricted. E.g., the IntegerCharacterAtom will restrict its *source* object to one-digit numbers and it *view* object to the characters '0' to '9'. Note that there is a conversion filter atom for both directions, e.g., IntegerCharacterAtom and CharacterIntegerAtom, to allow initial propagation to be done in either direction.

4.3.3. Constraint Filter Atoms

As mentioned earlier, our filter specification represents constraints, but for certain geometric and computational constraints we want to provide a trap door to an external system. Constraint filter atoms are externally defined filter atoms for which there is a constraint-satisfaction technique known to an external system. These constraints are specified like subroutine calls, where the parameters identify the objects that are to be constrained. Constraint filter atoms could be implemented in terms of an interface to the Bertrand programming language [Leler 86] or to a system like ThingLab [Borning 79].

Figure 4.5 shows an arithmetic filter that defines the *view* to be the sum of the two constituents of the *source*. The *source* is of type IntegerPair, the *view* of type **Integer**. The syntax follows the rules for filter packs except that the **make** statement is replaced by a constraint specification. The 'PlusConstraint' call is a reference to an external constraint satisfaction mechanism. It has three parameters and constrains the third parameter to be the sum of the first and second.

In addition, the constraint-satisfaction mechanism needs some information on which object is the "anchor" or "preferred object" in order to resatisfy the constraint correctly, because we don't want it to respond to a change to an object by undoing it. This information will be provided by the interface mechanism and can be deduced from the *source* and *view* objects. In case of initial value propagation the *source* object is preferred. In the case of a

---

```
Object Type IntegerPair
        first → Integer
        second → Integer
end

Filter Type IntegerPlus ( source : IntegerPair , view : Integer )
        constraint
                PlusConstraint ( source.first , source.second , view )
end
```

Figure 4.5: Object and filter type for IntegerPlus constraint filter atom.

---

change, the "anchor" marking is inferred from the state of the objects (see Section 4.5 on semantics).

4.3.4. Implementation Filter Atoms

In addition to the filter atoms mentioned above, an implementation of the filter paradigm has to provide more primitives. Section 5 on implementation describes input and output primitives that are modelled as filter atoms.

4.4. Filter Packs

A filter pack is a higher-level filter that is constructed from atomic or previously defined filters. A filter-pack type definition represents a constraint between two objects. Throughout this paper these objects were called the *source* and *view* object.

The type definition follows this syntax:

```
Filter Type <Name> ( source : <source_type> , view : <view_type> )
        var
                <variable declaration list>
        make
                <filter construct list>
        merge
                <address_equals_variable list>
end
```

Where:

<Name>
: is a unique name for this type of filter. Subsequent filter definitions can use this name when constructing more complex filters. The name has to start with a capital letter.

<source_type>
: is labelled by the keyword 'source' and specifies the type of the *source* object of the filter that is defined. It has to be either atomic (**Integer**, **Boolean**, **Character**, **Bit**), or has to refer to an object definition.

<view_type>
: is labelled by the keyword 'view' and specifies the type of the *view* object of the filter that is defined. It has to be either atomic (**Integer**, **Boolean**, **Character**, **Bit**), or has to refer to an object definition.

<variable declaration list>
: is a list of variable names followed by their type. The types must be either atomic (**Integer**, **Character**, **Boolean**, **Bit**), or refer to a object type definition. These variables can be used to create intermediate objects to connect complex filter packs or to reduce the amount of text. Note that there is an important distinction to make: variables can be used to create intermediate objects to connect subfilters, and variables can be introduced for clarity reasons, which are later unified with components of either the *source* or *view* object. This unification process can be controlled in the **merge** statement.

<filter construct list>
: establishes the structure of the filter pack, by instantiating filter atoms or previously defined filter packs. Analogously to the object type definition, we introduce iteration, set,

condition and recursion. The valid filter constructs are:

- **iteration** <expression> **times** <variable>
- **set of**
- **condition** <condition>
- implicit recursion

These subfilters are established between components of the *source* and *view* objects or the variables that are introduced in the variable declaration list. A filter is instantiated by using its specific name and by identifying the appropriate *source* and *view* object, or by providing a variable, holding the filter name, plus a signature for *source* and *view* object that can be evaluated when the filter is instantiated (see Section 4.3).

The **set of** construct instantiates several filters of possibly different types with different arguments. This this can be used for side-by-side or end-to-end composition. In end-to-end composition we have to introduce a variable to serve as an intermediate object.

The **iteration** construct instantiates a certain number of same filters with arguments of type array. The <expression> defines the range of the <variable>. The <expression> is evaluated within this filter type definition. It can mention *source* or *view* object components or defined variables. Therefore, the iteration can depend on an object that is itself part of an instantiated filter.

The **condition** construct instantiates a filter only if the condition given is true. The <condition> is evaluated within the filter type definition and therefore the value can depend on an object that is part of another subfilter.

The recursion construct instantiates a filter of the same type as the one being defined, much like a recursive call in a conventional programming language.

<address_equals_variable list>

is a list of equations relating the addresses from the object definition to the variables used in the filter construction. A *source-view* relation can be established here if it was not possible or convenient to identify *source* and *view* objects of the subfilters in the **make** statement. The *source* and *view* objects are bound to objects of the lower-level filters, which in turn are bound to even lower levels. This binding proceeds down to the level of filter atoms, where *source* and *view* are slots for atomic objects. To express the fact that we are using unification, rather than type or token identity, we introduce the special symbol '=○='.

If any of these statements, such as **var**, **make** or **merge**, are not necessary, they can be omitted. With the given syntax we are able to define arbitrary filters.

The object types used in the following examples are all described in Section 4.2 on object type definition. Figure 4.6 shows a filter of type IterationExample where a filter type is constructed from four instantiations of the IntegerIdentity filter atom. This filter establishes an equality constraint between an integer array of size 4 and the components of a list. Note that the size 4 in this example is necessary since none of the participating objects contains information about the size of the arrays. It does not need to be a constant if the *source* or *view* object or a variable within the filter type definition could be used to express the iteration factor.

If we want to combine filters of different types, we use the **set of** construct. Figure 4.7 shows the SetExample where an IntegerIdentity filter atom and the IterationExample of the last example are composed. Note that the actual connection of *source* and *view* objects of the defined filter to the *source* and *view* objects of the instantiated filters is done in the **make** statement. This SetExample filter is defined for *source* and *view* of type Array_2 and List_2, respectively. But it is also defined for all subtypes of Array_2 and List_2.

```
Filter Type IterationExample ( source : Array_1 , view : List_1 )
        var
                v[4] → Integer
        make
                iteration 4 times i
                        IntegerIdentity ( source.subfield[i] , v[i] )
        merge
                view.sub_1 =O= v[1]
                view.sub_2 =O= v[2]
                view.sub_3 =O= v[3]
                view.sub_4 =O= v[4]
end
```

Figure 4.6: Filter type for IterationExample.

```
Filter Type SetExample ( source : Array_2 , view : List_2 )
        make
                set of
                        IntegerIdentity ( source.label , view.label )
                        IterationExample ( source.dependents , view.dependents )
end
```

Figure 4.7: Filter type for SetExample.

```
Filter Type ConditionExample ( source : Array_3 , view : List_3 )
        make
                set of
                        IntegerIdentity ( source.label , view.label )
                        condition source.label = NIL or view.label = NIL
                                IterationExample ( source.dependents , view.dependents )
end
```

Figure 4.8: Filter type for ConditionExample.

If the instantiation of a subfilter depends on components of *source* or *view* objects or defined variables, we can use the **condition** construct. Figure 4.8 shows the ConditionExample, which is similar to the IterationExample except that the instantiation of the IterationExample subfilter depends on the value of the first part of the *source* and *view* object. Figure 4.9 shows

```
Filter Type RecursionExample ( source : Array_4 , view : List_4 )
        make
                set of
                        IntegerIdentity ( source.label , view.label )
                        condition source.label = NIL or view.label = NIL
                                RecursionExample ( source.dependents , view.dependents )
end
```

Figure 4.9: Filter type for RecursionExample.

RecursionExample, which is the same as the SetExample except that it will instantiate the RecursionExample again recursively, depending on whether the label part of the *source* or *view* object are not NIL.

The last examples clarified our concept of composing a filter from subfilters using the different filter constructors. We used the terms "instantiation" and "establishing" to express the fact that a filter is created from its filter type specification. If a filter is established, it has a *source* and *view* object associated with it that are of known object type. We have to distinguish between the type given by the filter type for the *source* and *view* and the type of the objects filling these slots. The type of an object filling a slot has to be a subtype of the one defined by the filter type.

4.5. Informal Semantics

Now that we have defined the syntax of our filter type specification language and given some examples, we move on to specifying the meaning of the language. Note that this is only an informal discussion and that the details still have to be worked out. The language specifies objects in terms of object types built from atomic types using object constructors, and filter types built from filter atoms using filter constructors. A filter and its two associated *source* and *view* objects constitute an ***instantiated filter***.

In a programming language, the meaning of a program can be defined by a semantic function. This semantic function maps the domains 'program' and 'state' to the domain 'state' [Gordon 79], i.e., it defines how a program affects the state. We can define the meaning of the filter paradigm in a similar fashion. All *source* and *view* objects constitute the state. Instead of a program we have filters. The filters represent constraints that are defined for the objects. A constraint is a condition that must hold for some objects, plus a method to satisfy the condition by manipulating the participating objects. As filter specification is declarative, there should be no distinction between the semantic function denoting a well-formed specification, and the semantic function denoting behavioral aspects. However, because the notion of input to an instantiated filter is not yet clearly defined, we would like to distinguish between static and behavioral semantics for this informal discussion.

Let us first look at the static part of the semantics. Consider having two objects and a filter and wanting to combine them to an instantiated filter. The objects are of a certain object type, the filter has its filter type. The filter type specifies the composing subfilters and how the subobjects of the *source* and *view* are associated with them. The addresses of *source* and *view* objects have to be determined and are combined with the subfilters to instantiate the subfilters. We instantiate subfilters until we reach filter atoms. A filter atom cannot be decomposed any

further, but there are implicit constraints associated with it. For an identity filter atom, the implicit constraint is equality; for a conversion filter atom, it is a conversion constraint; and for a constraint filter atom, it is the defined constraint. So this constraint can either be satisfied or fail. Note that there are two places where failure can occur: first, in identifying the subparts of *source* and *view* objects when building an instantiated subfilter, and second, when satisfying the constraint of a filter atom. The first failure can be detected from the filter type specification using the object type definition of the *source* and *view* objects. The second failure will result in invoking the constraint satisfaction mechanism that is defined for the constraint. Thus, the meaning of our filter paradigm can be defined by a semantic function that maps the domains 'state' and 'filter' to the domain 'state' augmented with a value for failure. The type signature of such a semantic function expressing the well-formedness of an instantiated filter would be:

$$(\text{state} \times \text{filter}) \rightarrow (\text{state} + \text{'failure'})$$

The behavioral aspect of the semantics has to give meaning to dynamic changes in the instantiated filter when an object is changed. Let us impose the restriction that changes may occur only to one object of the instantiated filter at a given time, either the *source* or *view* object. As an example, consider a QuadArrayMirror filter connecting two QuadArrays with one sublevel of nodes. Figure 4.10 illustrates the QuadArrayMirror filter constructed from five

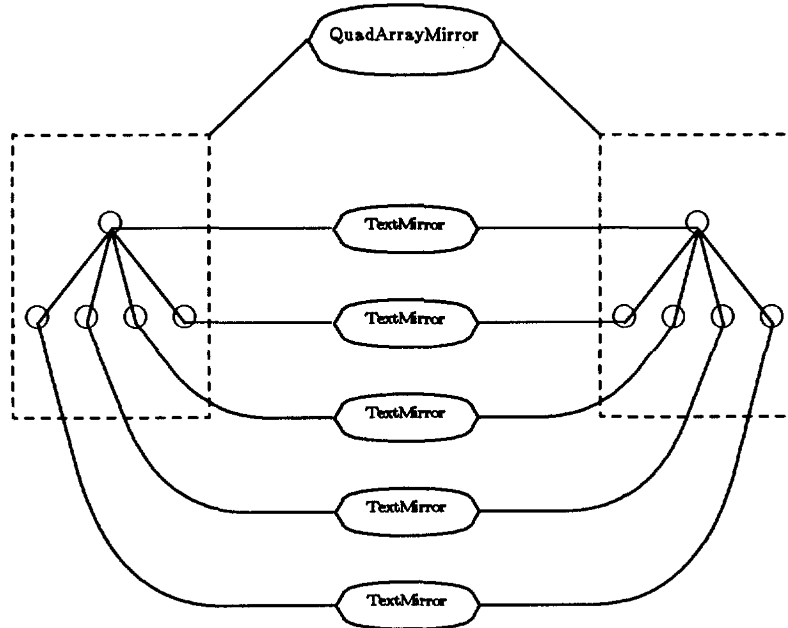

Figure 4.10: QuadArrayMirror for QuadArrays with one sublevel.

instances of the TextMirror filter[11]. There are two kinds of changes: changes that only affect the state of a node, such as changing the 'label' address of a QuadArray, and changes that affect the structure of the QuadArray, such as adding or deleting subnodes. The label is associated with the TextMirror subfilter. If the label is changed, the TextMirror filter updates the corresponding label. This update can be done by marking the changed label as "anchor" and satisfying the TextMirror constraint of the TextMirror filter. If the structure is changed, the **make** statement of the QuadArrayMirror filter has to be consulted again to check which subfilters have to be released or which new ones have to be added. For example, if four subnodes are added at a leaf node of the *source* object, the address 'subarrays' in the **make** statement of the QuadArrayMirror filter type definition (see Figure 3.8) is no longer NIL and four subfilters will be instantiated. This node will then be marked as "anchor" and the constraint satisfaction will also create four subarrays for the *view* object of the QuadArrayMirror filter.

As in the static case, there are two cases where failure can occur: the constraint for the filter atom cannot be satisfied, or the subparts, the addresses, of the *source* and *view* object cannot be identified. Hence, some changes are not allowable. The semantic function for the behavioral aspect maps the domain 'state', where some objects are marked as changed, and the domain 'filter' to the domain 'state' augmented with a label for failure. The type signature of such a semantic function expressing the change behavior of an instantiated filter would be:

$$(\text{state} \times \text{marking} \times \text{filter}) \rightarrow (\text{state} + \text{'failure'})$$

This semantic function expresses only the effect the change has on the objects. But as a result of the change, the filters can have changed, too. This is an area where we need to work on the details. Also, if the structure of objects is changed, the marking may not provide enough information for the constraint satisfaction. It has the meaning that only the structure at this level of the instantiated filter has to be conserved in constraint satisfaction, but that the subparts can be subject to changes due to dependencies in the chain of change propagation.

As we have seen, the static and dynamic semantic functions are related. One difference is that in the behavioral case we mark one object as special so it will not be affected in the constraint satisfaction. Further research in this area is under way.

5. Implementation

Although not the main issue in this phase of the research, the implementation of some prototype filters is useful to locate the critical issues. Two implementations were done: one for the QuadArray filters[12] described in Section 3, and one to explore the interactive behavior of filters described later in this section. The QuadArray example led to recognizing the necessity to control the structure of objects and the access to them. The other example, manipulating primitive graphical objects, clarified the issues of rendering objects on the bitmap display and sensing user action. Both implementations used a very basic constraint satisfaction mechanism that was sufficient for the examples. In order to allow the full expressiveness of our filter paradigm we plan to implement it on top of an existing constraint system, such as ThingLab [Borning 79].

[11] The TextMirror filter is assumed to be previously defined. See also Section 3.1.

[12] See Figures 3.1, 3.2 and 3.8.

5.1. Object Control

The language used is Smalltalk. Therefore, objects are instances of classes. Programming is done in terms of messages that are sent between objects, resulting in methods being executed. Filters are implemented as objects, named filter describing objects. Source and view objects have to be controlled in order to detect a change. The objects could be examined at a specific event [Ege 85], or the access to the objects could be controlled. To provide this control, access to objects is done through *object holders*, which behave like the actual objects but also monitor the access to it. Source and view objects are replaced by *object holders*. The *object holder* implements the same messages as the controlled object. Whenever a message is received, the holder forwards this message to the held object and sends messages to all objects that are interested in the access. The *object holder* takes registrations from filter describing objects and notifies them when an update to the source or view object has occurred. In order to do that, it keeps a list of all filter describing objects that are interested in the held object. The filter describing objects that have registered at the object holders for their source and view object, are notified when a change has occurred and update the other participants in the filter accordingly. Objects can be intermediate objects in a chain of end-to-end composed filters. The *object holder* then sends messages to several filter objects. The filter describing objects ensure that an update is not applied twice by using a simple synchronization technique. Note that in this implementation the filter describing objects handle the constraint satisfaction. The structure of the instantiated filter is hidden in dependency chains. Therefore only simple 'propagation of value' is used as constraint satisfaction method. Future implementations will use more sophisticated methods like in ThingLab [Borning 79].

5.2. Path Expressions

When a filter is instantiated, a source and view object is associated with it. These objects have to be retrieved. As mentioned in Section 4.2 objects can be accessed dynamically on need. To model this in Smalltalk path expressions are used. A path expression is a pair consisting of an anchor object and a list of selection messages. In order to access an object that is referenced by a path expression, the first selection message is sent to the anchor object yielding one of its subobjects. The second selection message is then sent to that subobject yielding another object, and so on for all selection messages in the list. The object resulting from the last selection message in the list is the result of the delayed access. If we view an object as a tree, the delayed access through a path expression represent a tree traversal by executing each selection message from the list. If one of the intermediate objects in the path becomes 'NIL', then the path evaluation returns 'NIL'. It is also possible to substitute the last object in the path traversal. In this case the last selection message is used as update message with the substitution object as parameter.

5.3. Constructing Filters

Filters are constructed from filter atoms. As described in Section 4, these filter atoms have to be provided by the basic implementation. In order to incorporate user interaction in the filter pack, we have to introduce filter atoms for accessing the display bitmap and the input devices of the user terminal. This access can be modelled in our paradigm.

The filter atom that renders a point on the screen is modelled as a filter connecting an object that consists of a X and Y coordinate and the screen bitmap. The bit representing the location expressed by the coordinates is turned on. This PointRender filter is a one-way filter atom. Change can occur only at the coordinates object. This filter atom can be used to define filter packs to render higher level graphical objects, such as line, box, etc., although we will probably include atomic filters for simple geometric objects.

```
Filter Type PointAtMouse ( source: Mouse , view: Display )
        var
                position → Point
        make set of
                PointRender ( position , view )
                PointSensor ( source, position )
end
```

Figure 5.1: Filter pack to display point at mouse position.

A filter atom that senses input from an input device is modelled analogously. As an example, consider a pointing device, the mouse. Any change in the mouse position affects the value of a mouse position object consisting of an X and Y coordinate. Such filter atom sensors have to be implemented for all the graphical input primitives [Mallgren 83]. Figure 5.1 shows these two filter atoms combined, resulting in a simple interface, where the mouse moves a point across the screen.

When manipulating graphical objects, basic arithmetic operations are needed[13]. Our implementation includes some basic constraints for solving arithmetic, such as addition, subtraction, multiplication and division for integers. They are implemented using a technique similar to ThingLab's satisfaction methods [Borning 79]. Figure 5.2 shows a more elaborate example, where a box is defined in terms of its corner points. The coordinates are constrained to ensure parallel edges using IntegerIdentity filter atoms. The lines connecting the corners are rendered

```
Filter Type BoxAtMouse ( source: Mouse , view: Display )
        var
                northWest, northEast, southWest, southEast → Point
        make set of
                IntegerIdentity ( northWest.x , northEast.x )
                IntegerIdentity ( northWest.y , southWest.y )
                IntegerIdentity ( southWest.x , southEast.x )
                IntegerIdentity ( northEast.y , southEast.y )
                LineRender ( (northWest,northEast) , view )
                LineRender ( (northWest,southWest) , view )
                LineRender ( (northEast,southEast) , view )
                LineRender ( (southWest,southEast) , view )
                PointSensor ( source , southEast )
end
```

Figure 5.2: Filter pack to display box at mouse position.

[13] E.g., to express point locations relative to a reference point. See the appendix for an example.

onto the display bitmap using the LineRender filter. The LineRender filter constrains the screen bitmap to display a line at the location specified by the coordinates of the endpoints of the line. The lower left corner (southEast) of the box is then connected to a PointSensor filter that binds it to the mouse location. Whenever the mouse is moved the coordinates of that point will

---

From Smalltalk-80 version T2.2.0, of March 13, 1986 on 28 August 1986 at 11:34:36 am

FilterExamples class methodsFor: 'input-output'

renderBox

```
"Example tracks mouse cursor with box"

|       northLine eastLine westLine southLine
        northWest northEast southWest southEast
        x1 x2 x3 x4 y1 y2 y3 y4                    |

"create variables"

        x1 ← IntegerHolder new. x2 ← IntegerHolder new.
        x3 ← IntegerHolder new. x4 ← IntegerHolder new.
        y1 ← IntegerHolder new. y2 ← IntegerHolder new.
        y3 ← IntegerHolder new. y4 ← IntegerHolder new.

        northWest ← ImpPoint x: x4 y: y4. northEast ← ImpPoint x: x3 y: y3.
        southWest ← ImpPoint x: x2 y: y2. southEast ← ImpPoint x: x1 y: y1.

        northLine ← ImpLine top: northWest bot: northEast.
        westLine ← ImpLine top: northWest bot: southWest.
        southLine ← ImpLine top: southWest bot: southEast.
        eastLine ← ImpLine top: northEast bot: southEast.

"instantiate filters"

        IntegerIdentity source: x1 view: x2.
        IntegerIdentity source: y1 view: y3.
        IntegerIdentity source: x3 view: x4.
        IntegerIdentity source: y2 view: y4.

        LineRender source: northLine view: Display.
        LineRender source: westLine view: Display.
        LineRender source: southLine view: Display.
        LineRender source: eastLine view: Display.

        PointSensor source: nil view: southEast.

"FilterExamples renderBox"
```

Figure 5.3: Smalltalk instantiation of BoxAtMouse filter pack.

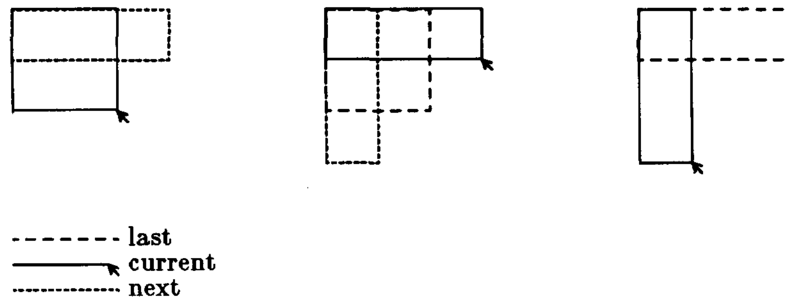

Figure 5.4: BoxAtMouse interface.

change. The coordinates are part of IntegerIdentity filter atoms that will be satisfied to ensure parallel edges by changing the coordinates of the adjacent points. All these points (southEast, southWest, NorthEast) are part of lines that will be redisplayed, thus reshaping the box. Figure 5.3 shows the Smalltalk code that was written to instantiate the filter pack. The objects that are values for the variables have to be created first. Points are created from X and Y coordinates. Lines are created from points[14]. The IntegerIdenity subfilters are instantiated for the coordinates. The lines are displayed by sending the LineRender class the message 'source:view:' and identifying the correct objects. The input sensor is instantiated by sending the same message to the PointSensor class. Figure 5.4 illustrates the screen representation. Three instances in time are recorded. The first part shows a rectangle. Then the cursor is moved right and up. The rectangle follows as shown in the middle part. Then the cursor is moved down and left. The rectangle follows as shown in the right part.

5.4. Condition and Iteration

The current implementation is also concerned with the notion of structural change. If the structure of the filter pack is to be modifiable by user input, then a more complex control of the filter instantiation mechanism has to be provided. The structure of a instantiated filter can change, because objects that are participating in the filter constructors, such as iteration and condition, are changed. Iteration and condition filters depend on the objects that are referenced in their iteration factor and condition expression, respectively. When one of these objects is changed the filter has to determine whether and how many subfilters to establish. This dependency can be viewed as a one-way filter where changes affect the iteration and condition filter. Whenever a subfilter is then instantiated, the source and view objects have to be reevaluated from their path expressions.

The next step in the implementation of a prototype interface is to provide filter atoms to create interface concepts, such as menu, form, window, etc., and to code more basic constraints, such as constraints for arithmetic. As mentioned at the beginning of this section, the current

[14] Note that the object types point and line are called ImpPoint and ImpLine, respectively, to avoid a conflict with Smalltalk classes point and line.

implementation uses a very basic constraint satisfaction mechanism. ThingLab [Borning 79] implements a powerful constraint satisfaction. Combining ThingLab and the filter paradigm by implementing the filter describing objects as ThingLab 'things' will probably yield a more usable system. Research in this area is under way.

6. Conclusion

This paper presented the filter paradigm. It is a new approach to model interfaces. The basic concepts of objects, constraints and filters were introduced and the filter specification language was defined. Critical issues in a possible implementation were discussed.

This research is a step towards the goal of automatic generation of user interfaces, which is subject of joint research at the University of Washington, the Oregon Graduate Center and the Tektronix Computer Research Laboratory [Borning 86]. The generator will be based on the filter paradigm. We think of a system where the interface designer specifies the interface graphically from subfilters, plugging existing parts together to build his system incrementally. This graphical description has to be transformed into the filter specification language given in this paper. The language is declarative and can be analyzed to optimize the interface. This language is then subject to compilation into constraints, which can be satisfied using a constraint-satisfaction system. Before we build a compiler for the filter specification language, translating them into constraints, we would like to build a prototype interface by implementing the filters directly in Smalltalk or on top of ThingLab [Borning 79] as discussed in Section 5.

7. Acknowledgement

Grateful acknowledgement goes to my advisor, Dr. David Maier, for supplying the original idea for this research and guiding me through the development process. I would also like to thank Dr. Alan Borning, who provided ThingLab that will be used in the next implementation of the filter paradigm. This work was done while the author held the Tektronix Fellowship for Computer Science and Engineering.

Appendix

8. Geometric Theorem

As in the related literature [Borning 81] [Leler 86], we will demonstrate the usability of our filter description. The following sections will describe step by step how to visualize a geometric theorem. The geometric theorem states that if we connect the midpoints of all edges of a quadrilateral, we will always get a parallelogram. We want to construct an interface for a graphical screen that allows us to manipulate the points of the quadrilateral and of the parallelogram by preserving their geometric properties. This example will highlight several features in our filter paradigm:

- I/O primitives as filter atoms
- type hierarchies
- constraints
- filter instantiation

8.1. I/O Primitives

The section on filter specification (see Section 4) introduced the need for filter atoms. The basic filter atoms are identity, conversion and constraint filter atoms. The section on implementation (see Section 5) added filter atoms for input and output. The table in Figure 8.1 gives some examples: The PointRender filter atom displays a pixel on the screen bitmap at the location given by the object of type Point. The LineRender filter atom displays a line of pixels

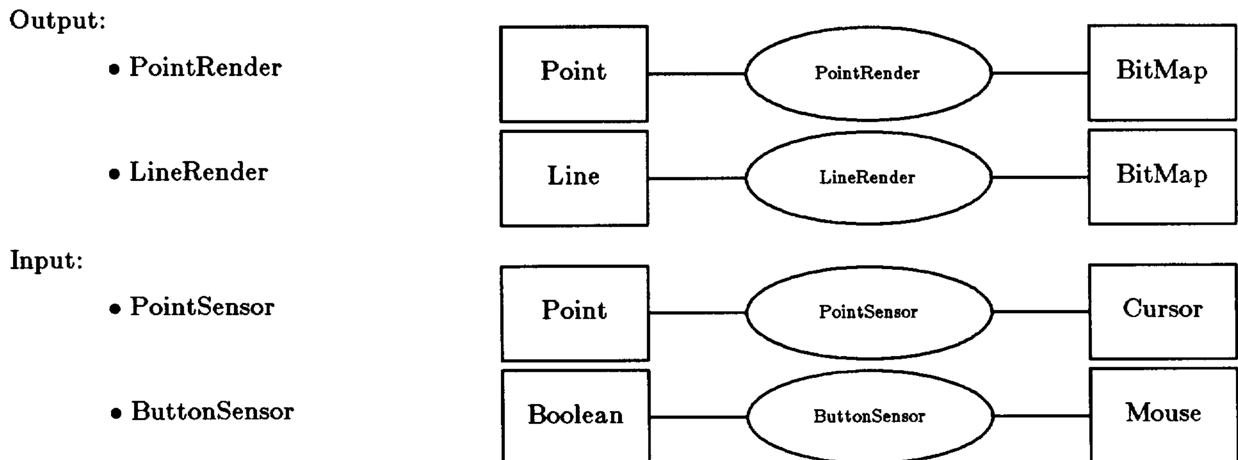

Figure 8.1: I/O Primitives.

```
Filter Type PointAtCursor(source: InputMedium, view: DisplayMedium)
        var
                position → Point
        make set of
                PointSensor ( position , view )
                PointRender ( position , source )
end
```

Figure 8.2: PointAtCursor Filter Type.

on the screen bitmap at the location given by the object of type Line. The PointSensor filter atom reflects the position of the cursor in an object of type Point. The ButtonSensor relates a mouse button to an object of type Boolean.

A simple interface can be constructed by connecting end-to-end a PointSensor and PointRender filter atom that agree on an intermediate object 'position' of type Point. This PointAtCursor filter (see Figure 8.2) will trace the cursor movement with a pixel on the display bitmap. (This filter type is basically identical to the one in Figure 5.1 and is included here only for completeness.)

8.2. Type Hierarchies

Our object type specification allows us to build type hierarchies. Consider as example the four object types: Polygon, Quadrilateral, Triangle and Box. Their object types are given in Figure 8.3. The figure also illustrates their type hierarchy. The object type Polygon is supertype of Quadrilateral, Triangle and Box. Quadrilateral is supertype of Box. Note that the subtypes do not add addresses to the type definition. Instead, they constrain inherited addresses.

We can use this type hierarchy in defining a filter type to display objects of these types on the display bitmap. The PolygonRender filter type will instantiate a LineRender subfilters for each edge of the source object. The number of edges is given by the *count* address of the source object. Figure 8.4 shows the resulting filter type definition. Notice that two adjacent points are combined to form an edge.

8.3. Constraints

As described in Section 5 on implementation, we need basic arithmetic constraints. Other constraints can be build from them. As an example (see Figure 8.5), consider a line with two points 'head' and 'tail'. We want a third point, 'mid', to be in the middle of the line from 'head' to 'tail'. If the two points of the line are moved, the midpoint should be adjusted to satisfy the constraint. If the midpoint is moved one of the endpoints should be moved accordingly.

We can express this relationship with a constraint equation:

$$\text{mid} = (\text{ head} + \text{tail}) / 2$$

If we want to decompose the constraint into subfilters, we introduce another point, 'add', that represents the addition of the two endpoints of the line. The 'add' point can then be divided by two to yield the midpoint. The addition and division are not simple arithmetic operations but

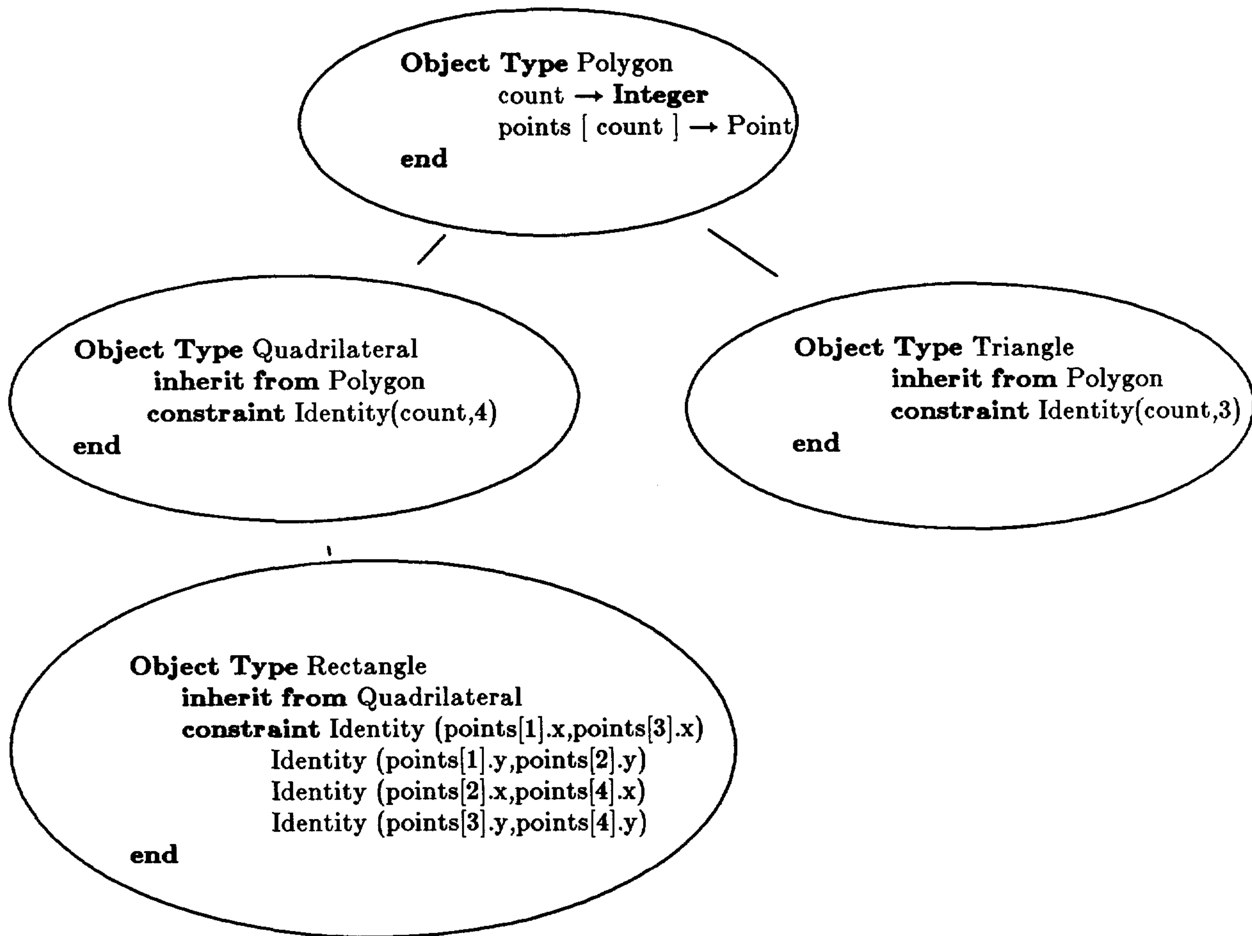

Figure 8.3: Type Hierarchy for Polygons.

```
Filter Type PolygonRender ( source : Polygon , view : DisplayMedium )
  make set of
    iteration source.count times i
      LineRender( (source.points[i],source.points[(i+1)%source.count]) , view)
end
```

Figure 8.4: PolygonRender Filter Type.

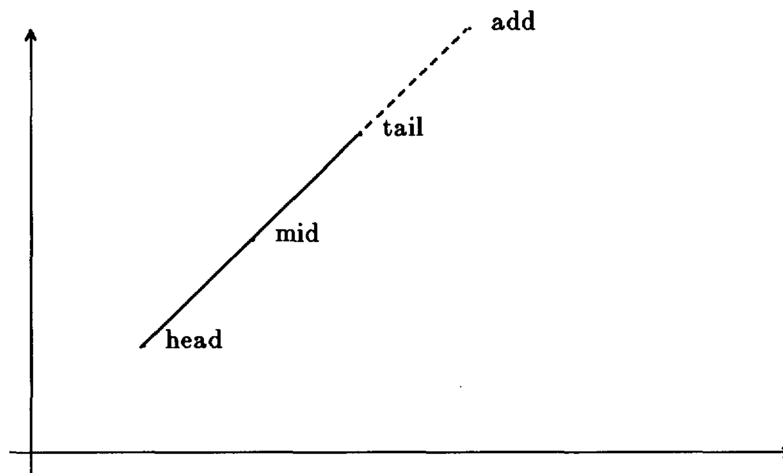

Figure 8.5: Midpoint on a Line.

represent sub-constraints that keep the relation of the points. These sub-constraints are further subdivided into basic arithmetic constraints for integer arithmetic that are filter atoms. We express this construction as a filter type in Figure 8.6.

8.4. Filter Instantiation

Filter instantiation is not static. As an example, consider the condition filter constructor. The subfilter is only instantiated when the associated condition is true. The filter type PointMenu (see Figure 8.7) instantiates a conditional subfilter PointSensor 'view.count' times. The condition depends on the variable 'selection' that is manipulated by the PopUpMenu subfilter. This PopUpMenu subfilter is supplied by the implementation as a filter atom. It sets the variable 'selection' according to the choice the user made. Only one PointSensor subfilter is

```
Filter Type MidPoint ( source : Line , view : Point )
        var
                add → Point
        make set of
                PointPlus ( ( source.head , source.tail ) , add )
                PointDivision ( ( add , 2 ) , view )
end
```

Figure 8.6: MidPoint Filter Type.

instantiated because only one of the conditions can be true. If the value of 'selection' changes, then the old subfilter is released and the new subfilter, which condition is now true, is instantiated.

8.5. Geometric Theorem

The previous examples can now be plugged together to illustrate the geometric theorem: connect the midpoints of the edges of a quadrilateral to yield a parallelogram. Figure 8.8 shows three examples of the graphical representation of the theorem[1]. The interface we built will display the quadrilateral and the parallelogram on the display bitmap and will provide a popup-menu to let us select one of the eight points to move it with the mouse. The graphical

```
Filter Type PointMenu ( source : InputMedium , view : Polygon )
   var
      selection → Integer
   make set of
      PopUpMenu ( selection, view.count)
      iteration view.count times i
         condition ( selection = i )
            PointSensor ( view.points[i], source )
end
```

Figure 8.7: PointMenu Filter Pack.

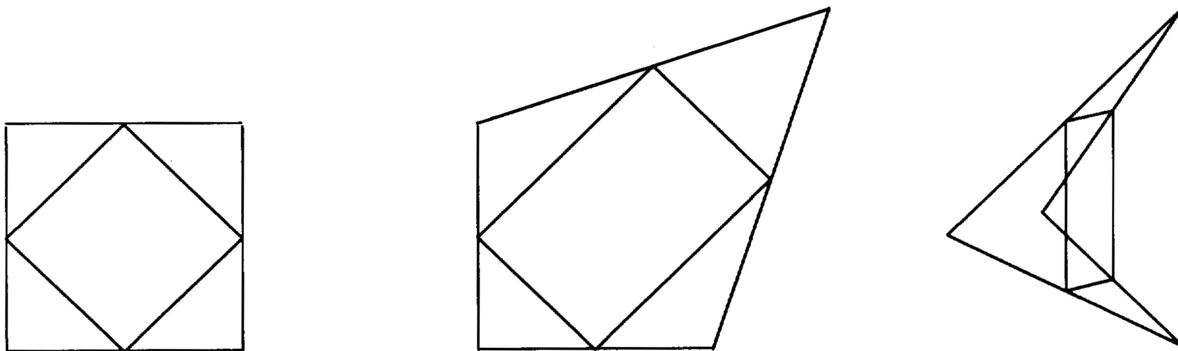

Figure 8.8: Illustrated Geometric Theorem.

[1] This figure was generated using the constraint-based IDEAL language.

```
Filter Type Theorem ( source : InputMedium , view : DisplayMedium )
  var
    quad , para → Quadrilateral
  make set of

    iteration 4 times i
      MidPoint((quad.points[i],quad.points[(i+1)%5]), para.points[i])

    PolygonRender ( quad , view )
    PolygonRender ( para , view )

    PointMenu ( source , (quad.points + para.points) )
end
```

Figure 8.9: Theorem Filter Pack.

display follows the mouse movement, always satisfying the midpoint constraint, thus illustrating the geometric theorem.

The filter type 'Theorem' (see Figure 8.9) relates an InputMedium to a DisplayMedium as source and view type. In the **var** statement the filter type defines two variables of type Quadrilateral (see Section 8.2). Since a Quadrilateral has four sides The **make** statement names four MidPoint subfilters (see Appendix Section 1.3), associating the lines of the 'quad' quadrilateral and the points of the 'para' quadrilateral. The two quadrilaterals are then displayed using two PolygonRender subfilters (see Section 8.2). We finally include a PointMenu subfilter (see Section 8.4) to bind the InputMedium to the eight points of the quadrilaterals.

This interface is implemented in Smalltalk-80 on a Tektronix 4400 machine. It allows the user to select any point of either the quadrilateral or the parallelogram. The selected point is then associated with the mouse. Mouse movement will cause the point to move, thus reshaping the graph as shown in Figure 8.8. Note that the filter type 'Theorem' describes this dynamic behavior purely declarative. The dynamic behavior is hidden in the constraint satisfaction.

Bibliography

[Ait-Kaci 84]
Ait-Kaci, H., *A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures,* PhD thesis, University of Pennsylvania, 1984.

[Albano 83]
Albano, A., Cardelli, L., Orsini, R., *Galileo: a strongly typed, interactive conceptual language.* Bell Labs Technical Memorandum TM 83-11271-2, 1983; also ACM Transactions on Data Base Systems 10(2), 1983.

[Atkinson 85]
Atkinson, M., Morrison, R., *Types, Bindings and Parameters in a Persistent Environment,* Persistent Programming Research Report 16, University of Glasgow and University of St. Andrews, August 1985.

[Borgida 84]
Borgida, A., Mylopoulos, J., Wong, H., *Generalization/Specialization as a Basis for Software Specification,* in: On Conceptual Modelling, Brodie, M., Mylopoulos, J., Schmidt, J. (eds), Springer-Verlag, New York, 1984.

[Borning 79]
Borning, A., *ThingLab - A Constraint-Oriented Simulation Laboratory,* PhD thesis, Stanford, 1979.

[Borning 81]
Borning, A., *The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory,* ACM Transactions on Programming Languages and Systems 3(4), October 1981.

[Borning 86]
Borning, A., Maier, D., London, R., *Automatic Generation of Interactive Displays,* proposed research, proposal to the National Science Foundation, 1986.

[Bournique 85]
Bournique, R., Treu, S., *Specifaction and Generation of Variable, Personalized Graphical Interfaces,* International Journal Man-Machine Studies, (1985) 22, 663-684.

[Chi 85]
Chi, U., *Formal Specification of User Interfaces: a Comparison and Evaluation of four axiomatic Matehods,* IEEE Transactions on Software Engineering, SE-11:8, August 1985.

[Ege 84]
Ege, R., *The Display Function in ALLEGRO.* Master's thesis, Oregon State University, July 1984.

[Ege 85]
Ege, R., *Entwicklung eines Systems zur vereinfachten alphanumerischen Ein/Ausgabe für die Programmiersprache Pascal (Development of a System for simplified alphanumerical Input/Output for the Programming Language Pascal).* Diplomarbeit, Institut für Informatik, Universität Stuttgart, August 1985.

[Goldberg 83]
Goldberg, A., Robson, D., *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Mass., 1983.

[Goldberg 85]
Goldberg, A., *Application Development Frameworks,* Oregon Graduate Center Colloquium, Video Tape, November 13, 1985.

[Gordon 79]
Gordon, M. *The Denotational Description of Programming Languages,* Springer-Verlag, New York, 1979.

[Kay 83]
Kay, A., *Novice Programming in the 1980's,* Programming Technology, Pergamon Infotech, 1983.

[Leler 86]
Leler, W., *Specification and Generation of Constraint Satisfaction Systems using Augmented Term Rewriting,* PhD thesis draft, The University of North Carolina at Chapel Hill, 1986.

[Maier 86]
Maier, D., Nordquist, P., Grossman, M., *Displaying Database Objects.* Proceedings First International Conference on Expert Database Systems, April 1986.

[Mallgren 83]
Mallgren, W., *Formal Specification of Interactive Graphics Programming Languages,* ACM distinguished dissertation 1982, MIT Press, Cambridge, MA, 1983.

[Milner 78]
Milner, R., *A Theory of Type Polymorphism in Programming,* Journal of Computer and System Science 17(3), 348-375, 1978.

[Morgenstern 83]
Morgenstern, M., *Active Databases as a Paradigm for Enhanced Computing Environments,* Proceedings 9th International Conference on Very Large Data Bases, Florence, Italy, October 1983.

[Myers 83]
Myers, B., *INCENSE: A System for Displaying Data Structures,* Computer Graphics 17(3), July 1983.

[Myers 84]
Myers, B., *Strategies for Creating an Easy to Use Window Manager with Icons,* Proceedings Graphics Interface '84, National Research Council of Canada, 1984.

[Nordquist 85]
Nordquist, P., *Interactive Display Generation in Smalltalk,* Master's thesis, Oregon Graduate Center, Technical Report CS/E 85-009, March 1985.

[Reis 86]
Reis, S., *An Object-Oriented Framework for Graphical Programming,* research report, Brown University, March 1986.

[Scofield 85]
Scofield, J., *Editing as a Paradigm for User Interaction.* PhD thesis, University of Washington, August 1985. Available as Computer Science Department Technical Report 85-08-10.

[Shaw 83]
Shaw, M., Borison, E., Horowitz, M., Lane, T., Nichols, D., Pausch, R., *Descartes: A Programming-Language Approach to Interactive Display Interfaces.* Proceedings SIGPLAN Symposium on Programming Language Issues in Software Systems, ACM, June, 1983.

[Shaw 86]
Shaw, M., *An Input-Output Model for Interactive Systems,* Proceedings CHI '86: Conference on Human Factors in Computing Systems, April 1986.

[Studer 84]
Studer, R., *Abstract Models of Dialog Concepts,* Proceedings 7th International Conference

on Software Engineering, IEEE, 1984, pp.420-429.

[Sutherland 63]
Sutherland, I., *Sketchpad: A Man-Machine Graphical Communication System.* PhD thesis, MIT, 1963.

[Takala 85]
Takala, T., *Communication Mediator - A Structure for UIMS,* in: User Interface Management Systems, (G. Pfaff, ed.), Springer-Verlag, Berlin, 1985.

[Van Wyk 81]
Van Wyk, C., *IDEAL User's Manual.* Computing Science Technical Report No. 103, Bell Laboratories, Murray Hill 1981.

[Van Wyk 82]
Van Wyk, C., *A High-Level Language for Specifying Pictures,* ACM Transactions on Graphics 1(2), April 1982.