Automatic Generation of Interfaces using Constraints

Raimund Karl-Heinz Ege

Vordiplom (B.Sc), Universität Stuttgart, Fed. Rep. Germany, 1981
M.S., Oregon State University, Corvallis, Oregon, USA, 1984
Diplom-Informatiker, Universität Stuttgart, Fed. Rep. Germany, 1985

A dissertation submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of
the requirements for
the degree of

Doctor of Philosophy
in
Computer Science

July, 1987

The dissertation "Automatic Generation of Interfaces using Constraints" by Raimund Karl-Heinz Ege has been examined and approved by the following Examination Committee:

David Maier
Associate Professor
Thesis Advisor

Richard Hamlet
Professor

Earl F. Ecklund, Jr.
Principal Scientist

Alan Borning
Assistant Professor

Dedication

To my parents,
Karl and Irmgard Ege,
for their never ending
support and encouragement
from far-away lands.

Danke.

Acknowledgement

It is my pleasure to thank my advisor, Dr. David Maier, for his support and direction. I would also like to thank Dr. Alan Borning for supplying various versions of the ThingLab system that I used to implement my work. I am also grateful to my fellow graduate students that helped me through many fruitful discussions.

Table of Contents

List of Figures

ABSTRACT

Automatic Generation of Interfaces using Constraints

Raimund Karl-Heinz Ege, Ph.D.
Oregon Graduate Center, 1987

Supervising Professor: David Maier

Interfaces play a crucial role in today's computer technology and much effort is spent to design and program user interfaces. This dissertation reports a new approach to this area of research that is based on the concept of separating the presentation from the data, and describing their relationship declaratively via filters. A filter is a package of constraints and associated typed objects that expresses that relationship of data and representation objects.

This dissertation 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. Object and filter types are implemented in an object-oriented language with the aid of a constraint-satisfaction system. A graphical tool for constructing filters is provided to build and test interfaces interactively.

Prologue

1.1. Introduction

Interfaces are a crucial part of any computer system, not only between users and the computer, but also between programs. The quality of an application is partly judged by the quality of its 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 with 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 interface, but a new architecture for building interfaces.

1.1.1. Object-Oriented Systems

The term "object-oriented" is used to denote a new direction in programming languages. This direction is based on the realization that people work with problem-domain concepts when they try to solve problems, while the computer works with concepts such as bits and operations [Cox 86]. Object-oriented systems try to model the concepts in the world that are germane to a problem so that people have an easier transition from their concepts to the concepts that the computer understands. In object-oriented programming languages, the computer understands concepts that are called "objects." An object models a concept that has a crisply defined boundary and encapsulates concept-specific properties and behavior. Objects are grouped into classes. A class describes what properties and behaviors its member objects share. Classes can be related: they can specialize or generalize other classes. Related classes create hierarchies where objects of one class inherit properties and behavior from more general classes.

An object in an object-oriented system is an entity that represents its properties as an aggregation of fields that form a local state, and its behavior as procedures that can affect the local state. The interaction between objects can be viewed as the sending of messages that result in execution of local procedures, or as remote procedure calls that invoke the local procedures remotely.

1.1.2. Interfaces For Object-Oriented Systems

In an object-oriented environment, objects that belong to an application and objects that belong to the user interface can be separated. These objects communicate with each other and with the user of the application. The user perceives these objects presented to him on the screen and interacts with them by using the input devices of a modern workstation. For example, the user of the interface sees a graphical representation of a tree; however the application knows of the tree as a data structure. Conceptually, the tree exists only once, but it may have aspects that are only relevant to the application or relevant to the interface. The application and the interface look at this tree object in the universe and model it in their

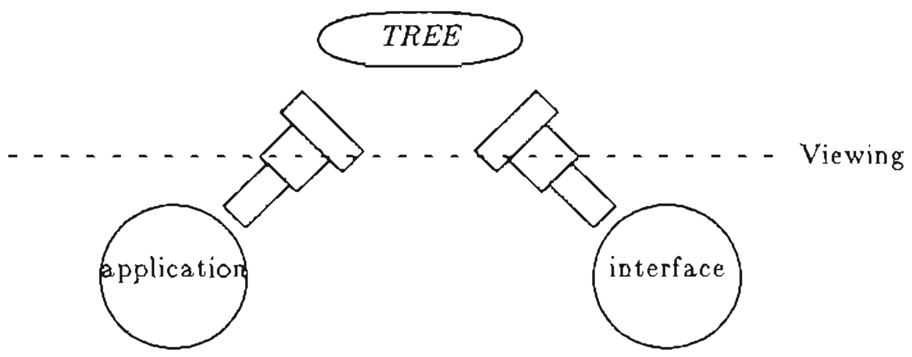


Figure 1.1: The Conceptual Model.

own world. The interface views the tree as a graphical image consisting of points and lines within a bitmap, while the application views the tree in terms of a nested collection of records. We can picture this abstraction as looking at an object in the universe through telescopes using different filters. Figure 1.1 illustrates this conceptual model. Interface and application each have their own view of the object in the universe.

1.1.3. Declarative Specification of Interfaces

In order to declartively specify interface we first define the two objects that result from the filtering. Then the object in the universe disappears and the two filtering mechanisms are combined into one two-way filter. Instead of having the interface and the application look at the same object using a filtering technique (Figure 1.1), we define two separate objects, one in the interface's reality and one in the application's reality, which are connected via a control mechanism (Figure 1.2).

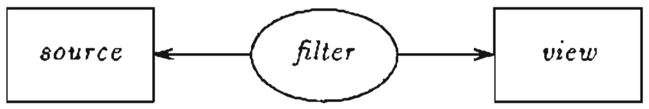


Figure 1.2: The Filter Paradigm.

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"[1].

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 or the interface. The filter constrains the source and view objects to be representations of the same conceptual object. Both objects belong to their respective environments, which could be the memory of a program in an application or the user's display screen in the interface. In general, a filter can connect any two objects. Bigger filters can be constructed from smaller filters using intermediate objects or by 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, then 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

[1] Our notion of "filter" should not be confused with filters in operating systems that are connected via pipes. Our filters could be viewed as "operating system" filters that work both directions.

representation on the screen, the memory of the program has to be 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 (numeric values) can be expressed by equations. Subsequent values can be defined in terms of previously defined values. In common spreadsheet programs, changes are only forwarded in one direction through the equations. In our notion, a change on either side of the equation is reflected on the other side.

Interfaces built according to the filter paradigm are constructed in a bottom-up fashion. They are assembled from atomic components that are constraints defined for two objects of specific types. The filters built from these atomic components also represent constraints. Since the atomic components are typed and the construction is done in a structured way, the resulting filter has a well-defined type for the source and view object. It can therefore be checked whether a filter is suitable for specific source and view objects.

Each constructed filter is available for use in other filter compositions. This availability encourages the reuse of previously defined components in the definition of new interfaces, making it easier to ensure that various interfaces in a system present a uniform appearance to the user.

1.2. Thesis Statement

This thesis explores a new approach to constructing interfaces. Interfaces in an object-oriented environment relate application objects and presentation objects. The goal of this work is to model this relationship in terms of constraints (filters) and to provide a tool to generate interfaces from them. This dissertation represents the first step towards the goal of automatic generation of interactive displays using the filter paradigm. The scope of the research that is reported in this dissertation is to provide

a specification language for objects and filters,

an implementation of filters,

an interface for constructing filters.

The interface, the *Filter Browser*, allows a designer to manipulate filter objects graphically. The main implementation of the filter browser is done in Smalltalk [Goldberg and Robson 83] using ThingLab [Borning 79] to perform the constraint-satisfaction. Sample interfaces have been constructed with it. The description of the Filter Browser in terms of filters will serve as demonstration of the feasibility of this new approach to building interfaces.

Chapter 2 of this dissertation introduces the basic concepts, which are objects, constraints and filters. Chapter 3 gives syntax and semantics for the object and filter specification language. Chapter 4 introduces the *Filter Browser*, a graphical tool to define and manipulate filters. Chapter 5 describes the implementation of the object type system and how filter types are modelled in this object type system. We conclude this dissertation with a summary of the work that has been completed and provide some ideas on future work on this topic. In the appendix we list the source code for the examples that are used throughout the dissertation.

1.3. 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 and reuse of display code. This area of research has drawn much attention and many results have been reported. The following sections will cover the four major issues that are involved: (1) abstractions or models that are being developed to describe interfaces, (2) methods and tools to specify interfaces and to generate them automatically, (3) constraints as a way to hide procedurality, (4) graphical specification of programs, as in visual programming.

1.3.1. Abstractions for Interfaces

Many abstractions or models have been proposed in order to grasp the complexity of a user interface. These models are usually called *user interface management systems (UIMS)*. Figure 1.3 shows a logical model of a UIMS [Green 85]. It shows the presentation, dialog-control and application interface model. The presentation component is responsible for the external representation of the user interface. This component generates the images that are perceived by the user. The dialog control component defines the structure of the dialog between the user and the

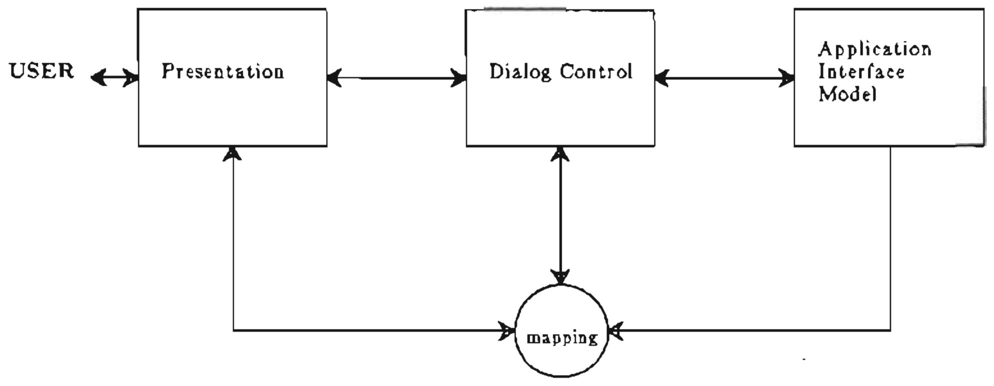


Figure 1.3: Logical Model of a UIMS.

application program. The dialog component can be viewed as mediator between the presentation and the application component. It interprets events in the presentation component and translates them into events for the application component and vice versa. The application-interface-model component defines the portion of the application domain that is visible to the user. The dialog-control component controls all communication between the presentation and the application component. It can set up a direct connection between them, as is shown by the lower links and the circle in Figure 1.3, to handle direct internal-external mappings.

This model is very general and has been accepted in the research community. The filter paradigm fits this classification since the source-filter-view triple can be mapped directly onto this logical model. View objects in the filter paradigm represent the presentation component, source objects the application-interface-model component. Filters represent constraints that control all communication between the source and view objects. Other approaches use similar ideas for abstracting the user interface [Coutaz 85, Takala 85]. Others have implemented user interface systems following their own abstractions [Buxton et al. 83, Shaw et al. 83, Shaw 86].

Our approach was guided by experience with the Smalltalk model-view-controller (MVC) paradigm [Goldberg and Robson 83]. This paradigm employs the idea that all data of an application are kept by a model. The presentation is kept in a view and a controller handles the interaction. 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 [Maier, Nordquist and Grossman 86, Nordquist 85]. One conclusion of SIG is that display procedures need type information about the objects they display and that Smalltalk does not provide this kind of typing.

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. The IMPULSE system [Schoen and Smith 83, Smith et al. 84, Smith, Dinitz and Bart 86] provides abstractions for objects as well as for interactions. Their main goal is to ensure consistency throughout the interface and application description, but they do not provide a declarative specification of the interface as we do for the filter paradigm.

A UIMS that uses the object-oriented programming paradigm is GWUIMS (George Washington User Interface Management System) [Sibert, Hurley and Bleser 86]. It consists of a variety of object classes representing different levels of abstractions. Representation objects (R_objects), interaction objects (I_objects) and application objects (A_objects) maintain a strict separation between the lexical/syntactic domain in the UIMS and the semantic domain of the application. The interface is then represented by a knowledge base. Our implementation of the filter paradigm as constrained objects also represents a knowledge base, but we also provide a tool, the Filter Browser, to maintain and manipulate the knowledge base. Other systems [Broverman and Croft 85, Hirsch et al. 86] also use knowledge-based approaches to building interfaces; others use graphical specifications of procedural knowledge for expert systems [Musen, Fagan and Shortliffe 86]. Yet another area of research uses windows as the main feature of its systems [Beach 84, Greenberg, Peterson and Witten 86, Myers 84, Sweetman 85, Williams 85] and places the window package at the center of the UIMS, thus ensuring uniformity of appearance to the user across all interfaces.

1.3.2. Specification and Generation of Interfaces

Wasserman [Wasserman 85] states some requirements for specifications of interfaces for interactive systems: The specification has to provide a formalism that is complete and comprehensible to both the system designer and the user. The formalism has to allow flexibility in order to accommodate a broad variety of styles and the formalism should be executable. Wasserman uses a state transition diagram approach to describe user interfaces. The diagrams are executable, so that the specified interface can be evaluated during the design phase.

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 interaction such as user input, error handling and undo operations. Several other theoretical approaches use algebraic techniques to specify user interfaces and interaction axiomatically [Chi 85]. These approaches seem to work well only for small example systems.

SYNGRAPH [Olsen and Dempsey 83] is a tool to generate graphical interfaces. It uses a grammar specification to drive the generation of interactive Pascal programs. From the grammar it deduces information about how to manage devices and detect and handle input errors. The authors conclude that after experience with developing such systems, they feel that more emphasis should be put on the display update problem [Olsen 83, Olsen 86], i.e., how to produce the code to update the display after each modification to the application data structure. Another formal specification of user interfaces [Bournique and Treu 85] uses a BNF-like syntax to generate personalized interfaces. The authors distinguish "action language agents" in their language that address *communicability*, which is how the user can express his wishes, and "display language agents" that address *perceptibility*, which is how the user perceives data presented to him. Common to the two approaches is that they automatically generate interfaces from a specification, although they are using different abstractions. The filter paradigm also provides the generation of interfaces from a specification, but we provide a tool, the filter browser, that lets a designer develop the specification of an interface interactively.

GUIDE is an interface development environment [Granor and Badler 85, Granor 86]. It is an interactive graphical system for designing and generating graphical user interfaces. The concepts it uses are "tool," "task" and "context." Tools are the techniques for graphical interaction; tasks are a set of tools; contexts describe the state of the system. The designer does not have to write interface code, instead he edits a user interface specification that consists of the above concepts. Procedurality is achieved by providing "action routines" that are invoked (parameterized) by the GUIDE-generated interface. The GUIDE system itself is an example of a graphical system generated with GUIDE.

The GUIDE system uses explicit procedurality, while the filter paradigm hides the procedurality in constraints. Various other approaches use forms [Bass 85], input-output tools [van den Bos, Plasmeijer and Hartel 83], menus [Rubin and Pato 84], and demonstration [Myers and Buxton 86] as their abstractions to describe, specify and generate user interfaces.

1.3.3. Constraints

Constraints are used in our system to express conceptual relationships. These relationships are declared and maintained by a constraint-satisfaction system. An early system that employed constraints to express graphical relationships was Sketchpad [Sutherland 63]. Graphical objects could be constructed by constraining other objects to form the new object. For example, a triangle is constructed from three lines by constraining their respective head and tail points to join. Other systems use constraints as their major construct, such as ThingLab [Borning 79, Borning 81], which is an extension to Smalltalk where constraints can be associated with classes. Constraints can be used as a basis for programming languages [Steele and Sussman 80]. The language Ideal, used in typesetting graphical pictures, is based on constraints and demonstrates their power and usefulness to define graphical pictures[2] [Van Wyk 80, Van Wyk 81, Van Wyk 82]. Bertrand [Leler 86] is a language that can specify and generate constraint satisfaction systems. It has been used to build graphics constraint languages.

An approach that is very similar to our filter description language is the VIVID language [Maleki 87]. VIVID is an object-oriented, interactive and declarative language for developing knowledge representation environments. It distinguishes constraints and constraint types. The constraint types are generic patterns defining the structure and behavior of a class of constraints. Constraints can be clustered where common variables are connected by equality relationships. VIVID's constraints and constraint types are very similar to filters and filter types in our filter paradigm.

Constraints and logic programming can be combined. The CLP framework [Jaffar and Lassez 87] of languages uses constraints to describe the basic components of a problem, while the problem itself is constructed in logic using the components. A similar idea is used to extend our filter paradigm to a user interface management system where the components of the system are described as filters that are combined using a distributed logic scheme [Grossman and Ege 87].

Our current implementation of the filter browser is based on ThingLab. Other extensions to ThingLab that define constraints involving time were proposed and

[2] Figure 1.1 of this chapter was produced using Ideal

implemented [Borning and Duisberg 86, Borning 86, Duisberg 86]. The "Human Interface Graphical Generation System (HIGGENS)" [Hudson and King 86] can generate user interfaces to an office information system. It can express constraints that are associated with conceptual data. It is notable that this system's constraints are satisfied using a data-flow computation and a demand-driven evaluation strategy. Constraints are used to specify relations and dependencies in a so-called *active* database interface system [Morgenstern 83]. In the context of a window management system [Cohen, Smith and Iverson 86] constraints can help reducing the complexity of controlling the layout and structure within the tiling process. An editor based on constraints [Carter and LaLonde 84] uses Steele's constraint system coupled with a syntax tree as the underlying mechanism for editing.

1.3.4. Graphical Programming

Graphical programming has gained momentum in recent years because the falling prices for graphical workstations have made it feasible to use pictures as means of communication between the user and a computer system. Shu [Shu 85] distinguishes four categories of languages with respect to graphical programming: (1) "visual programming languages" that allow users to actually program in a graphical environment; (2) "languages supporting visual interaction" that help to program a visual environment; (3) "visual aid languages" that deal with the graphical support in the run-time environment; (4) "languages for processing visual information" that deal with graphical information in the problem domain.

Our system, the Filter Browser, falls into category (1) in that it allows the user to design an interface by assembling units graphically. The Animus system

[Duisberg 86, London and Duisberg 85] falls into category (3) in that it displays underlying data structures while a program is running. The program itself is written in Smalltalk. The Alternate Reality Kit [Smith 86] is also based on Smalltalk and provides an animated environment for interactive simulations. The PECAN system [Reiss 85] also visualizes a program's progress at runtime. Based on experience with the PECAN system, the successor, the GARDEN programming environment [Reiss 86, Reiss, Golin and Rubin 86, Reiss and Pato 87, Reiss 87], is based on object-oriented programming and provides a framework to support a wide range of languages and graphics to support languages based on pictures. It also falls into category (1). Other systems [Jacob 85, Moriconi and Hare 85] are emerging that provide programming graphically. A survey of more graphical programming languages and techniques was done by Georg Raeder [Raeder 85].

1.4. Terminology

This section introduces the terms object, constraint, object type, constraint-satisfaction, filter, and filter type. We give a brief description for each term and its use in the filter paradigm. Chapter 3 will define these terms in detail.

Objects are identifiable units in an object-oriented system. Objects are either atomic, i.e., they have a value, or are structured, i.e., they have fields that reference other objects. Objects are the primitive elements of object-oriented programming and model all entities of concern. In the filter paradigm, objects model the source and view of filters.

A *constraint* is a Boolean predicate on the state of an object. The Boolean predicate is true if the object obeys the constraint. Constraints can be imposed on objects. In the filter paradigm, constraints are used to relate source and view objects in a filter.

An *object type* is a description of a set of objects that have similar structure and obey the same constraints. An object is an instance of a type if it conforms to the description of the structure and obeys the constraints. In the filter paradigm, object types are used to describe the objects that can be part of an interface.

Constraint-satisfaction is a mechanism to enforce constraints. A constraint is enforced by changing the state of the objects that have to obey the constraint. In the filter paradigm, constraint-satisfaction is used to provide the procedurality of an interface.

A *filter* is an object that represents constraints defined between two objects of specific types that are enforced by constraint-satisfaction. The two objects are the source and view of the filter. Filters can be atomic, i.e., they have no substructure, or they can be constructed from subfilters. In the filter paradigm, filters are used to relate objects to form an interface.

A *filter type* is a description of a set of filters. The description specifies the types for source and view object of the filters, the types of the subfilters and how the subfilters, if any, are connected to the source and view objects. In the filter paradigm, filter types are used to describe interfaces.

Chapter 2
Interfaces From Filters

The filter paradigm uses objects, constraints and filters to describe interfaces. This chapter will introduce these three concepts. An extended example will serve as an introduction and illustration of how we can use constraints to specify interfaces declaratively.

2.1. Example: Factory Simulation

Many modern object-oriented languages derive their features from the programming language Simula [Dahl and Nygaard 66]. Simula is a ALGOL-like language that was intended to support simulation systems. Smalltalk-80 [Goldberg and Robson 83] can also be used in simulating systems. We chose an event-driven simulation as an introductory example for how to build interfaces according to the filter paradigm. We want to simulate a situation of a factory, where unfinished products enter the system at the "producer," pass through two "stations," where they are refined and finished, and leave the system at the "consumer." Figure 2.1 shows the general flow of products. The producer produces goods at a constant rate, which can be varied. The stations can be manned with at most two workers. There

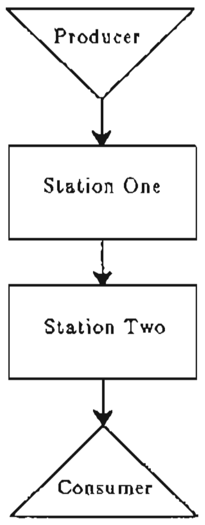


Figure 2.1: Flow Chart for Factory Simulation.

is one input queue for each station. The input queue of station one is filled by the producer; the input queue of station two is filled by station one. The finished products of station one are consumed by the consumer without being queued. This simulation has two variables: (1) the rate at which products are produced at the producer, and (2) the number of workers that work at each of the stations.

2.1.1. Constraints for Interfaces

We assume that this simulation exists in an object-oriented system. Our goal is to provide a user interface for it. The user interface has to portray the actions that are happening within the simulation and provide some means to manipulate it. Figure 2.2 shows a possible screen layout for the user interface. To the left is the producer; connected to it are the two stations with their respective input queues; to the right is the consumer. The simulation can be manipulated as it runs by adjusting the rate of products that are introduced into the system and by adding or removing workers from their stations.

2.1.2. Application Interface Model

According to the "Logical Model of a UIMS" [Green 85] an interface can be viewed as consisting of three components (see Figure 1.3): (1) the presentation component, (2) the dialog-control component, and (3) the application-interface-model component. The application interface model for our factory simulation contains all those objects that are used in the interface to the simulation. The model will include the following application objects: the producer, with its number of produced elements and the production rate; the two stations with their input queues and workers; and the consumer, with its amount of consumed elements. The simulation will modify the objects within the model as it progresses. Other information, like

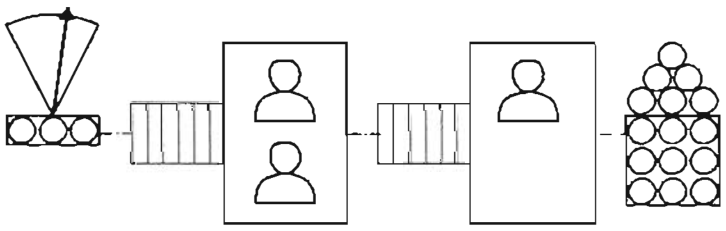


Figure 2.2: Screen Layout for Simulation Example.

the connection between the four elements in our simulation or details about event generation and timing, is not important to the interface and is local to the simulation application.

2.1.3. Display

The data structures for the objects in the application model will be mapped into the screen representations as shown in Figure 2.2. We can express these mappings with constraints. The screen bitmap is divided into four subparts by constraints to contain the presentations of the objects. We can view this constraint as a filter from the screen bitmap, the source, to a list of four smaller bitmaps, the view. The constraint specifies how the screen bitmap is divided up. The subparts are for: the producer, station one, station two, and the consumer. Each of the four subparts is then constrained to contain the display of its corresponding application part. The presentations of stations are further constrained to display the length of the input queues and icons for each worker. We use constraints instead of display functions to ensure that changes to the application data are reflected on the screen automatically, e.g., if the input queue shrinks or grows, or workers are added or removed from the stations, the display is updated immediately.

2.1.4. User input

This user interface can influence the simulation in two ways:

(1) The production rate of the source can be adjusted by pointing with the mouse at the gauge above the producer and pulling its needle within the markers. The productivity rate in the application interface model for the producer is constrained to

reflect the position of the needle within the gauge.

(2) Workers can be added or removed from the stations. This is done by moving the mouse cursor into the presentation area of one of either stations. If the mouse cursor points to an icon of a worker directly, then this worker can be removed by pressing the mouse button. Two types of workers are available to be added to a station: experts and apprentices. A menu is available for the mouse button to select what type of worker has to be added to the station.

The input actions affect the objects in the application model, which are constrained to be presented on the screen. After the application model has been

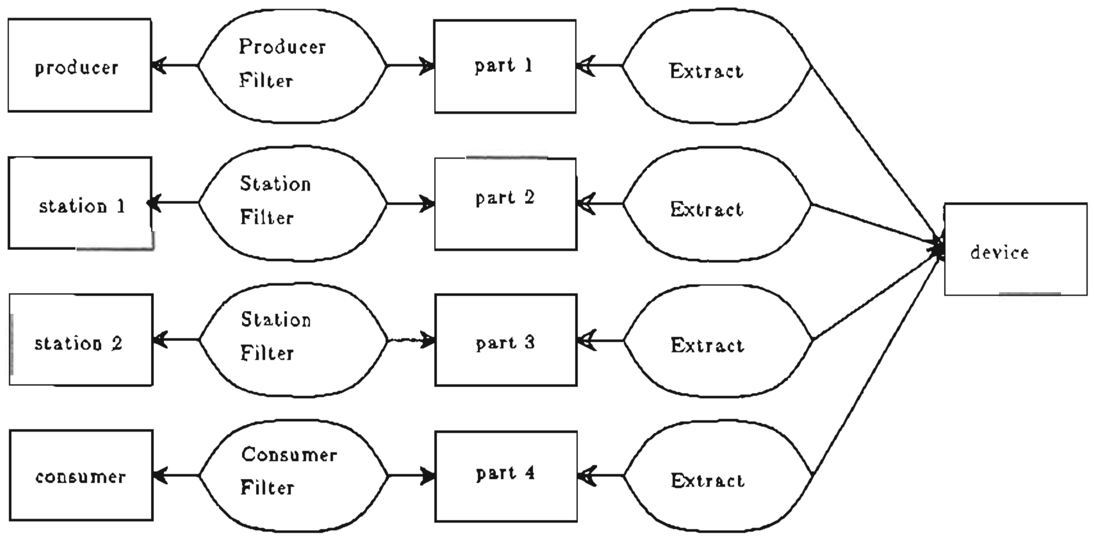


Figure 2.3: Filter Diagram for Factory Simulation Interface.

changed according to an input or the ongoing simulation, these changes are immediately visible on the screen. This example interface can be completely defined with constraints. The relations between model, representation and input media are declared. The procedurality of the interface is derived from the fact that the constraints are maintained by a constraint-satisfaction system.

Figure 2.3 shows the top level objects and constraints that are part of our simulation interface example. Filters represent the constraints that are defined for source and view objects. On its left side, the figure shows the four source objects from the application: producer, station 1, station 2, and consumer. On the right side it shows the view object that is the device used for this interface. For each of the source objects there is a filter that constrains it to be represented on a part of the view device. The "ProducerFilter" filter relates its source object, "producer," to the first part, "part 1," that is the extracted as the left-most part from the view "device." It displays the producer on the subpart of the screen and accepts input from the user to change the producer's productivity. Two "StationFilter" filters relate their source objects, "station 1" and "station 2," to the second and third part that are extracted as the middle parts from the view "device." They display the stations on the subpart of the screen and accept input from the user to add or remove workers. The "ConsumerFilter" filter relates its source object, "consumer," to the fourth part that is extracted as the right most part form the view "device." It displays the consumer on the subpart of the screen and does not accept user input. The "device" is the view object in four "Extract" filters that divide it into four source parts. An "Extract" filter constrains its source object to be mapped into a subpart of its view object.

2.2. Basic Concepts

The introductory example shows how the filter paradigm employs objects, constraints and filters. Objects have structure that is 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. The following three sections will define these concepts informally.

2.2.1. Objects

Objects are present at both sides of a filter. In order to define constraints on them we need information about their structure. We define an object to be an atomic value or a structured collection of fields. Fields consist of a name, called *address*, and a slot. The address is used to identify what fields are subject to constraints; the slot contains a reference to another object. Structured objects are

```
Object Type Station
      numberOfWorkers → Integer
      workers [numberOfWorkers] → Worker
      inNumber → Integer
      inQueue [inNumber] → Queue
      constraint LessThan (numberOfWorkers, 3)
end
```

Figure 2.4: Object Type for Station.

formed from fields, which can be iterated or conditional. Objects can be accessed by naming the address of a field. Subfields of fields can be accessed by concatenating addresses. A list of addresses, separated by a dot, is called an *access path* to an object.

An object type describes a set of objects that have similar structure. Objects have similar structure if their fields have the same addresses and reference objects of the same type. An object type is a subtype of another type if its elements are more specific, i.e., they have the same and more fields, than the elements of the other type. An object type is a supertype of another type if its elements have fewer fields than the elements of the other type. Constraints can be defined on fields to express restrictions on objects in an object type.

An example object type is `Station` (Figure 2.4). It contains four addresses: `numberOfWorkers`, `workers`, `inNumber` and `inQueue`. They name the four fields of type `Integer`, array of `Worker`, `Integer`, and `Queue`, respectively. The type `Integer` is atomic and we assume that the types `Worker` and `Queue` are already defined. The iteration in the `worker` address uses the address `numberOfWorkers` to express how many `workers` fields are defined for this type. The `numberOfWorkers` address is also used in the **`constraint`** statement, which constrains the `worker` field to hold at most two workers.

Subtyping is an important notion. An object type can be defined to be a subtype of another type. The fields of the supertype are then inherited by all elements of the subtype. Supertype and subtypes form a type hierarchy. A filter that is defined to accept objects of a specific type also accepts objects of their subtypes.

```
Object Type Worker
      inherit from Person
      salary → Integer
      throughput → Integer
end

Object Type Apprentice                  Object Type Expert
      inherit from Worker                     inherit from Worker
      level → Integer                         years → Integer
end                                     end
```

Figure 2.5: Subtypes of Worker.

The constraints in a filter are expressed with access paths that name addresses of fields. A subtype has at least all fields of its supertype, therefore the addresses in the access paths will exist.

Figure 2.5 shows the three object types `Worker`, `Apprentice` and `Expert`. Object type `Worker` is a supertype of `Apprentice` and `Expert`. Object type `Apprentice` specifies in the **`inherit from`** statement that it will inherit all fields from object type `Worker`, and it adds one more field, `level`, of type integer. Object type `Expert` also inherits all fields from `Worker` and adds a `years` field. Whenever an object of type `Worker` is required we can now use objects of type `Apprentice` or `Expert`.

We chose this object-type model because it lets us describe the structure of objects that are accepted for filters; it provides addresses, i.e., symbolic references to parts of the objects, to express constraints; it provides dynamic fields to describe objects of different structure with the type; and it distinguishes sub- and supertypes.

Our object types are sufficient to characterize the objects that occur in our filter paradigm.

2.2.2. Constraints

Constraints are the backbone and the basic building tool in our filter paradigm: filters represent constraints and are used to express interfaces; constraints are used to place conditions on objects as part of the object type definition. A constraint is a condition that is expressed with access paths. The condition must be kept true upon updates to objects. The mechanism to maintain constraints is called a constraint-satisfaction system.

2.2.3. Filters

A filter represents a package of constraints that have to be maintained between two objects. A filter is defined for specific types of objects. We have to distinguish atomic filters (filter atoms), which have to be provided by an implementation, and higher-level filters, which are constructed from atomic filters or other constructed filters.

Like an object type, a filter type describes a set of filters that have similar structure. The filter type defines the subfilter of a filter. Our filter specification language provides constructors to define filter types. The basic constructor declares an arbitrary collection of subfilters. The iteration constructor declares a variable number of identical subfilters. The condition constructor declares a conditional subfilter, i.e., the subfilter exists only if an expression is true. A subfilter is declared by naming its type and its associations to the source and view objects within the containing filter type. The set of subfilters of an instance of a filter type is called a configuration. The configuration can change with time since the iteration and condition constructors depend on other objects.

In general, we can distinguish end-to-end and side-by-side subfilter combination. In end-to-end combination, a filter is constructed using a chain of subfilters. The source object of the first subfilter is the source object of the constructed filter. The view object of the first subfilter is also the source object of the next subfilter. The view object of the last subfilter is the view object of the constructed filter. Two adjacent subfilters in the chain agree on a common intermediate object. Figure 2.6 shows how a filter is constructed from two subfilters. The source of the left subfilter is the source of the constructed filter. The view of the first subfilter is the source of the second subfilter. The view of the second subfilter is the view of the constructed filter. Note that this filter construction introduces intermediate variables as the connecting objects.

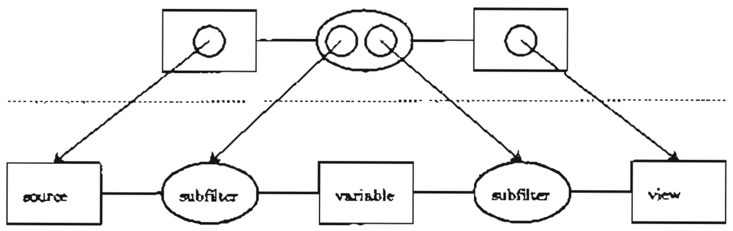


Figure 2.6: End-to-End Combination.

In side-by-side combination a filter is constructed using a set of two or more subfilters. The source and view objects of the subfilters are parts of the source and view object of the constructed filter. Figure 2.7 shows how a filter is constructed from two subfilters. The source of the first and second subfilter are part of the source of the constructed filter. The view of the first and second subfilter are part of the view of the constructed filter.

End-to-end and side-by-side combination are the most general form of constructing filters. Any specific constructed filter will probably represent a mixture of these two general forms.

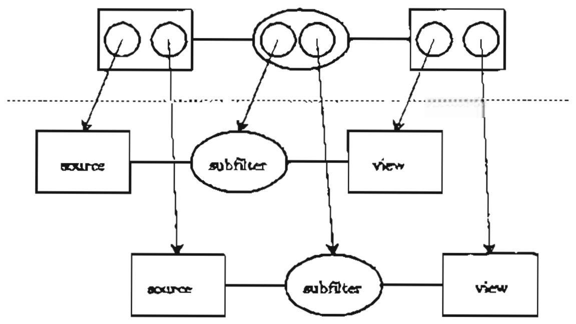


Figure 2.7: Side-by-side Combination.

2.3. Factory Simulation Filter

Using the concepts of "object" and "filter" we can now define our example interface in more detail. The application interface model for the factory simulation contains objects of type `Producer`, `Consumer` and `Station`. The structure of these objects is given by their object types (Figure 2.4 and 2.8). The objects for the simulation example are all aggregated into the object type `Factory` (Figure 2.8). Our factory has exactly one producer, two stations and one consumer. Since we want to describe an interface, we also have to supply an object type for the device we are using to communicate with the user. Object type `Device` has a field for an input medium of type `Mouse` and an output medium of type `Bitmap`.

Figure 2.9 shows the type definition for the `FactorySimulation` filter type. A `FactorySimulation` filter accepts an object of type `Factory` as source and an object of type `Device` as view. The **var** statement defines four intermediate

```
Object Type Producer                          Object Type Consumer
        productivity → Integer                        consumed → Integer
        produced → Integer                    end
end

Object Type Factory                           Object Type Device
        producer → Producer                           input → Mouse
        ws1 → Station                                 output → Bitmap
        ws2 → Station                         end
        consumer → Consumer
end
```

Figure 2.8: Object Types for Factory Simulation.

```
Filter Type FactorySimulation (source: Factory, view: Device)
var
      part[4] → Device
make set of
      ProducerFilter (source.producer, part[1])
      StationFilter (source.ws1, part[2])
      StationFilter (source.ws2, part[3])
      ConsumerFilter (source.consumer, part[4])
      iteration 4 times i
            Extract(part[i], view)
end
```

Figure 2.9: Filter Type for Factory Simulation.

variables with an iterated field, `part[4]`. The four variables are of type `Device` and are used to connect subfilters. The **make** statement defines the subfilters that are used to construct filters of this type. The first subfilter, `ProducerFilter`, constrains the `producer` part of the `source` factory to be displayed within the `part[1]` variable. This subfilter is connected with its source to the `producer` field of source, expressed with the access path "`source.producer`," and with its view to the `part[1]` variable.

Similarly, `StationFilter` subfilters relate the two stations of the factory to the second and third `part` variable. The `consumer` of the factory is rendered in its `part` using a `ConsumerFilter` subfilter. The **iteration** statement defines four subfilters to extract the four `part` variables from the `view` device. Figure 2.3 showed the filter configuration for the factory simulation filter.

The `FactorySimulation` filter decomposes the interface into subconstraints that are represented by subfilters. Each of these subfilters has to be

```
Filter Type StationFilter (source: Station, view: Device)
var
      left, part[2] → Device
      workerDetected [2] → Boolean
      selection → Integer
      expert → Expert
      apprentice → Apprentice
make set of
      QueueRender (source.inNumber, left)
      Extract (left, view)
      iteration 2 times i
            WorkerRender (source.worker[i], part[i])
            Extract (part[i], view)
            DetectCursor (part[i], workerDetected[i])
            condition workerDetected[i]
                  condition source.worker[i] isNil
                        PopUpMenu (selection, "Expert, Apprentice")
                        condition selection = 1
                              Equality (expert, source.worker[i])
                        condition selection = 2
                              Equality (apprentice, source.worker[i])
                  condition source.worker[i] notNil
                        Equality (NIL, source.worker[i])
end
```

Figure 2.10: Filter Type for Station.

defined separately using other constructed or atomic filters. We give here the definition of the `StationFilter` to show the expressiveness of filter types. A full definition of all the filter and object types for the factory simulation interface is listed in Appendix A.

Figure 2.10 shows the `StationFilter` filter type, which is defined for source objects of type `Station` and view objects of type `Device`. It displays the station for our simulation and allows the user to add or remove workers from it. As in the `FactorySimulation` filter, it defines variables and is constructed from subfilters.

The first subfilter constrains the number of products in the input queue of the station, inNumber, to be displayed within the left variable. The left variable is extracted from the view device.

Since the part variable and the worker field of the station are iterated fields, we can use an iteration constructor that ranges over the number of workers per station. Each worker is rendered as icon within the corresponding part variable with a WorkerRender subfilter, which is provided as a primitive. The part[i] variable is extracted from the view device. A Boolean variable (workerDetected) is used in a DetectCursor subfilter to detect whether the cursor is pointing at a worker within a station. The condition constructor (**condition**) is used to evaluate the Boolean variable. The constructor defines further subfilters that only exist if the condition is true.

If a worker field is not filled, i.e., it contains the value NIL, then a PopUpMenu subfilter is defined that causes a pop-up menu to appear on the screen. The user can select from two types of workers: expert or apprentice. They are both subtypes of Worker, so they can be referenced in the worker field of Station that is of type Worker. The worker field is related to expert or apprentice variable with an Equality filter.

If the worker field holds a worker, i.e., it does not contain the value NIL, then this worker is removed. The removal is done by defining an Equality subfilter with source NIL and view source.worker[i], which will set the i-th worker field to value NIL.

The `StationFilter` filter type uses all three subfilter constructors: set, iteration and condition. The set constructor defines a static configuration of subfilters, while the iteration and condition constructors define a dynamic configuration of subfilters that depends on the state of the source and view objects.

This introductory example showed how we can decompose interfaces using constraints, and tried to describe the concepts of the filter paradigm in a less formal way. Chapter 3 gives a complete definition of the filter and object specification language that was used to describe the objects and filters in this example.

Chapter 3
Filter Specification

This chapter formally defines the object and filter types that were introduced informally in the last chapter. We introduce types to group and distinguish different objects based on their structure and intended use. Section 3.1 defines our model for objects. The syntax for object types is given in Section 3.2. Section 3.3 incorporates filter types in our object type system, Section 3.4 gives the syntax for filter types, and Section 3.5 discusses the behavior of filters.

3.1. Object Model

Everything in our computation model is an *object*, which is a unit with a unique identifier. An object is an atomic value or a collection of fields. Atomic values are integer, character, boolean or bit data items. Fields consist of an address, which names the field, and a slot, which contains the object that is the value of the field.

We depict an object that is a collection of fields by a box labelled with the object identifier and subdivided into fields. A field is labelled with its address; its slot either contains an atomic value or a reference to another object, drawn as an

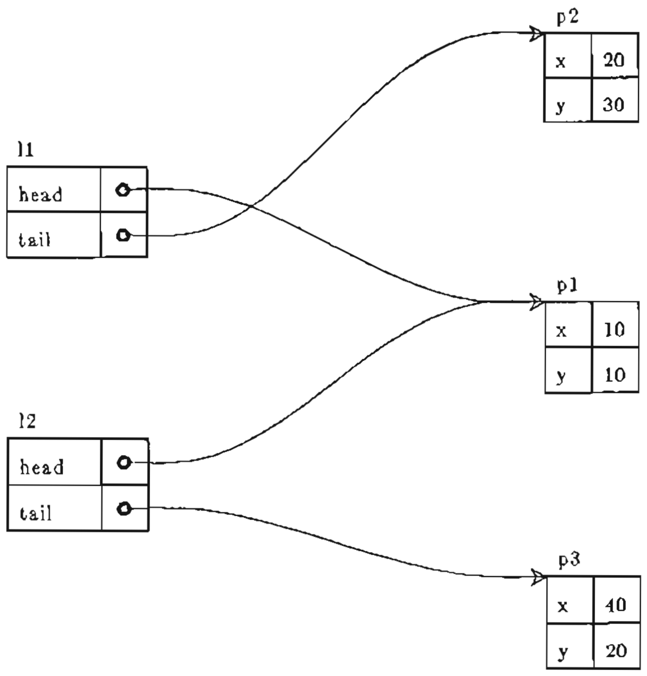


Figure 3.1: Two Lines Sharing a Point.

arrow. For example, Figure 3.1 shows the five objects `p1`, `p2`, `p3`, `l1` and `l2`. Object `l1` references object `p1` at the `head` address and object `p2` at the `tail` address. Object `l2` references `p1` at `head` and `p3` at `tail`. Objects `p1`, `p2` and `p3` all have the same structure. We could interpret them, based on their intended use, as points. The `x` field holds an integer value for the x coordinate; the `y` field holds an integer value for the y coordinate. Objects `l1` and `l2` also have the same structure as each other. We could interpret them as lines: a `head` field holding a point and a `tail` fields, also holding a point.

It is useful to group objects that model the same kind of thing in an application into sets, as there is information about such objects that can be shared. We can group atomic values into natural sets: the set of all integers, the set of all characters, the set of the elements `true` and `false`, and the set of the elements `0` and `1`. Objects that are a collection of fields have common structure if they have some fields that are similar. Fields are similar if they have the same address and their values are objects from the same set. Intuitively, objects with some similar fields should be grouped into the same set. The following paragraphs will explain how we describe these sets.

The structure of a set of objects is described by a *signature*. A signature **S** is a list of pairs, each pair consisting of an address and a set of objects. An object **o** *conforms* to **S** if for every address-set pair (**f**, **v**) in the signature, **o** has a field with address **f** with a value from **v** or the special atomic value `NIL`; Object **o** can have other fields that are not in **S**. An object can conform to more than one signature, because it can have more fields than each signature requires.

For example, objects `l1` and `l2` in Figure 3.1 conform to the signature

```
head, {p1, p2, p3}
tail, {p1, p2, p3},
```

since each has a `head` and `tail` field with a value from the set.

Consider an object `c1` that models a colored line. It has fields `head` and `tail` with references to `p1` and `p2`, plus a field with address `color`, where the slot contains a color code. Object `c1` also conforms to the signature above since it has fields with addresses `head` and `tail`, where the slots take values form {`p1`, p2, p3}.

We use a collection **D** of signatures to describe a set of objects. An object is a member of the set described by **D** if it conforms to some signature **S** in **D**. We use collections of signatures to give meaning to conditional and iterated fields used in the object type definition language in the next section.

For example, objects l1, l2 and c1 are members of the set that is described by the pair of signatures

```
head, {p1, p2, p3}
tail, {p1, p2, p3}
color, {1,2,3, ... }
```

and

```
head, {p1, p2, p3}
tail, {p1, p2, p3}.
```

Objects l1 and l2 conform only to the second signature, while c1 conforms to both signatures.

The *state* of an object is defined as a collection of fields, each field being a pair (**f**, **r**) consisting of an address **f** and a value **r**. Addresses are used to retrieve field values of an object. Addresses can be concatenated into access paths in order to access subfields of fields. An access path **A** is a list of addresses **f1**, **f2**, ... ,**fn**, denoted "**f1**.**f2**. ... .**fn**". For example, "head.x" accesses the x coordinate in the head point of a line object.

Access paths support retrieval and assignment operations on objects. The retrieval operation **Get** = (o, **A**) accesses object o according to access path **A** = (**f1**, **f2**, ..., **fi**, ... ,**fn**) and returns a field value. **Get** selects the value **r1** of field **f1** of o, then it selects the value of field f2 of r1, Get repeats this selection for all addresses in A. The value of field fn is returned as result of Get. An access path A *exists* for an object o, if for each address fi in A there is a field with address fi during the selection process. A retrieval operation Get is *valid* if A exists.

The assignment operation Put = (o, A, r) accesses object o with access path A and stores value r in the slot for the field specified by the access path. Put follows the path as for Get, but replaces the value of field fn with r. If A does not exist for o, then Put is invalid.

We also allow update operations that add or remove fields from an object, which may affect the signatures that an object conforms to. We will define their behavior after we have introduced object types.

Signatures are not the only means used to describe sets of objects. Objects can be subjected to restrictions on their states, called *constraints*, which are represented as Boolean predicates. A constraint is expressed with access paths and primitive predicates, such as basic Boolean operations, existence of access path, equality for atomic values, and external computational predicates, e.g., for arithmetic. An object *satisfies* a constraint if the evaluation of the predicate with the field values retrieved from the object yields true. Note, all the retrievals must be valid. For an object to continue to satisfy a constraint, some updates may be disallowed.

We extent the description of sets of objects with constraints to express types. An *object type* T = (D, C) is a collection of signatures D plus a collection of constraints C, where the sets in the signatures of D are object types or the predefined sets of atomic values. An object o is an *instance* of object type T = (D, C) if o conforms to *one* of the signatures in D and satisfies *all* constraints in C. Since the sets in signatures are now denoted by object types, the value of a field must be an instance of the appropriate object type. An object may be an instance of multiple object types.

For example, consider colored line `c1` again. It is an instance of the object type with the signature:

```
head,Point
tail,Point
color,Integer,
```

given that `Point` and `Integer` are other object types with appropriate definitions. The value `head` must be an instance of object type `Point`. Object `c1` is also an instance of the object type signature:

```
head,Point
tail,Point,
```

In order to determine whether an object conforms to an object type we define a *type map*. Given a collection **U** of all objects in our universe and a type system **TS** of all object types, the type map $\mathbf{m}\colon \mathbf{TS} \rightarrow 2^{\mathbf{U}}$ associates sets of objects in **U** with types in **TS**. The type map is *valid* for **U** if for each $o \in m(T)$, o is an instance of **T**, using $m(W)$ as the set denoted by **W**, for any type **W** used in a signature of **T**. Object o conforms to type $T = (D, C)$ if there is a valid mapping m with $o \in m(T)$.

Without a type map we would not always be able to efficiently check whether an object conforms to a type. Consider the following example: Object `o1` has field `f1` with value `o2`; object `o2` has field `f2` with value `o1`. Type `t1` has signature `f1, t2`; type `t2` has signature `f2, t1`. To see if `o1` is an instance of `t1`, we check whether it has a field `f1` with a value that is an instance of type `t2`. Now we have to check whether `o2` is an instance of `t2`, which leads us back to determining whether `o1` is an instance of `t1`. This check is cyclic. Given a valid type map **m** we can check what objects are instances of object type `t1` and `t2` relative to **m**. For our implementation, we will store a partial type map to make type checking easier.

For two object types A and B, B is defined to be a *subtype* of A if any object that is an instance of A is also an instance of B for a valid type mapping. All objects are instances of the universal type **U**. Thus, all other object types are subtypes of **U**. In general, it is hard to check the subtype relationship, because it involves determining if the constraints for B imply the constraints for A.

This section defined object types using signatures and constraints. The next section will give a language to express the signatures and constraints of an object type.

3.2. Object Types

We support aggregation and specialization as type definition mechanisms [Albano, Cardelli and Orsini 85] [Borgida, Mylopoulos and Wong 84]. With aggregation we build object types by enumerating the fields the object has. The field is specified by naming its address and the type for its slot. The field type is an object type that is already defined, the object type being defined, or one of the following predefined atomic types:

- `Integer`	for integers
- `Character`	for single characters
- `Boolean`	for truth values 'true' and 'false'
- `Bit`	for bit values '0' and '1'

A new object type can be defined by *inheriting* fields and constraints of existing object types and adding more fields or constraints. This inheritance (specialization) mechanism is only a notational convenience, however, it guaranties that the new object type is a subtype of the existing object types. We are using strict inheritance, which means that all fields of the supertypes are inherited by the new object type.

An object type may also list constraints.

The syntax is for object types is as follows:

```
Object Type <Name>
      inherit from <object types>
      <field definition list>
      constraint <constraint statement list>
end
```

Where

<Name>

is a unique name for the type. Subsequent object and filter definitions can use this name when constructing more complex objects or filters. The name must start with a capital letter.

<object types>

are the names of previously defined object types, from which components are inherited. If inherited fields have the same address in more than one supertype, then the supertypes are concatenated to the field. For example, A inherits fields from B and C; B and C both define a "x" field; then A has "B.x" and "C.x" fields.

<field definition list>

establishes the structure of the object type by defining the signatures. The list contains field definitions of basic, iteration and condition form, which are defined below.

<constraint statement list>

is a list of constraint statements. Each can name addresses of fields in the current object type definition or from supertypes. The constraint statement has to be in a form that can be understood by the constraint-satisfaction system.

The **`inherit from`**, <field definition list> or **`constraint`** statements can be omitted if not needed.

The <field definition list> is a list of one or more field definitions of the form:

<address> $\rightarrow$ <object type>

where <address> is the address for the field and <object type> is the type for values of the field. We also allow literals from the atomic types. 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`. A literal specifies that the field will contain a constant value for all instances of the object type. The value NIL is an element of all types and can be stored in any field slot. In Figure 3.2, the `ListOne` and `ListTwo` object types use basic field

```
Object Type ArrayOne
      array [4] → Integer
end

Object Type ArrayTwo
      label → Integer
      dependents → ArrayOne
end

Object Type ListTwo
      label → Integer
      dependents → ListOne
end

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

Figure 3.2: Sample Object Types.

definitions.

The iteration field definition format defines multiple fields of the same type:

<address> [<iteration factor>] → <object type>

where <iteration factor> specifies how many times this address should be replicated. The <iteration factor> can be an integer literal or the address of an integer field within this object type. Figure 3.2 shows object type `ArrayOne` where 4 array subfields are summarized as an iteration.

To express object types with variable structure we introduce the condition field definition:

(<condition expression>) : <field definition>

where the <field definition> exists only if the <condition expression> evaluates to true. The <field definition> can be of basic or iteration form. The <condition expression> uses field addresses of the object type and Boolean operators, and must evaluate to type Boolean. Figure 3.3 shows object types ArrayThree and ListThree, among others, each with a conditional field definition.

Recursion is specified by using the type name of the current definition in the <object type> specification of a field definition. Figure 3.3 shows object types

```
Object Type ArrayThree                  Object Type ArrayFour
      label → Integer                         label → Integer
      ( label = NIL ) :                       ( label = NIL ) :
            dependents→ ArrayOne                    dependents→ ArrayFour
end                                     end

Object Type ListThree                   Object Type ListFour
      label → Integer                         label → Integer
      ( label = NIL ) :                       ( label = NIL ) :
            dependents→ ListOne                     dependents→ ListFour
end                                     end
```

Figure 3.3: Object Types with Condition and Recursion.

ArrayFour and ListFour, which are defined recursively.

The iteration and condition field definitions allow an object to change its structure and still be an instance of the same object type. An object type with an iteration field can be described with multiple signatures plus a list of access path existence constraints. The path existence constraints specify that for a specific iteration factor i there have to be i paths in the iteration, one for each field. The number of signatures and constraints could be infinite, depending on the range of the iteration factor. If the iteration factor for an instance changes, then the number of fields in the instance changes, and the instance might now conform to a new signature from the object type. Figure 3.4 shows the object type ArrayFive, where the number of array fields is determined by the size field.

An object type with a condition field is described with two signatures, one including the field, the other not, plus the constraint that objects for which the condition is true must have a path to the field. If the condition is true, then the object

```
Object Type ArrayFive
      size → Integer
      array [size] → Integer
end

Object Type ArraySix
      inherit from ArrayFive
      constraint IntegerIdentity(size,4)
end
```

Figure 3.4: Object Types using Inheritance.

has to conform to the signature including the field; if not, the object conforms to the other signature.

Our specification language for object types cannot define all types, since the multiple signatures that describe iterated and conditional fields are not arbitrary.

The examples so far covered object structure. The **constraint** statement restricts the state of instances. A constraint is expressed with access paths that may contain addresses of fields and subfields for the object types. Figure 3.4 shows `ArraySix`, which is a subtype of `ArrayFive` that imposes the constraint that the number of `array` subfields is always four.

Once we have defined an object type, we can create instances of it. The syntax for object instantiation is:

<object type> (<initialization list>)

The <object type> is a type name. The <initialization list> contains zero or more <initialization pairs>. An <initialization pair> is:

<address> $\leftarrow$ <instance>

The <address> names the field to be initialized in the new object. The <instance> is a literal or another object instantiation expression that defines the value for the field. The instantiation expression returns a new instance of type <object type>. If a field is absent from the initialization list, then the field is `NIL`. Figure 3.5 shows how instances of type `ArrayThree` and `ArrayFive` are instantiated with initial values for all fields.

```
Object Type ArrayThree (
      label ← NIL
      dependents ← ArrayOne (
            array [1] ← 99
            array [2] ← 98
            array [3] ← 97
            array [4] ← 96 ) )

Object Type ArrayFive (
      size ← 2
      array [1] ← 99
      array [2] ← 98 )
```

Figure 3.5: Instantiation of Object Types.

In general, it is hard to check that an object is an instance of an object type, because we have to generate a type map. In order to make type checking tractable, we require that every object remain an instance of the type from which it is created. This type is called the *creation-time* type of the object. The creation-time types give a partial type map.

An object can be a field value of many other objects. Therefore, if the containing object has a type defined for the field, then the value must be of that type as well as its creation-time type. The constraints on an object derived from its creation-time type are called *creation-time* constraints, whereas the constraints on an object derived from object types whose instances reference the object are called *imposed* constraints.

```
Object Type Point
      x → Integer
      y → Integer
end

Object Type Line
      head → Point
      tail → Point
end

Object Type HorizontalLine
      inherit from Line
      constraint
            head.y = tail.y
end
```

Figure 3.6: Object Types for Point, Line, and HorizontalLine.

Consider the object types in Figure 3.6 and an object that has creation-time object type `Line`, which does not define any constraints. If this object is now referenced in a field of type `HorizontalLine`, then it must satisfy the imposed horizontal line constraint.

We have to distinguish the constraints that are imposed on a given object and the restrictions that are associated with a particular field slot in an object.

The *imposed* restrictions on a field slot in object `o` are the type given for the field by `o`'s creation-time type, plus the imposed constraints of `o` that affect the field. Some of the imposed restrictions are not expressible as constraints on `o` since they involve fields that are not accessible from `o`, but only from objects containing `o`. To determine the imposed restrictions for a slot in `o`, we have to check all references to o and find all access paths that traverse through o. If all objects in our universe conform to their creation-time types, then they satisfy all their imposed restrictions, because every imposed restriction is expressed as a creation-time constraint of some other object.

Consider the object type in Figure 3.7. The TwoLines type defines fields source and view of type Line and four constraints that relate the coordinates of head and tail points in the two lines. Figure 3.8 shows object f1, which is an instance of this type, where source references line object l1 with points p1 and p2, and view references line object l2 with points p3 and p4. Suppose object l1 has creation-time type HorizontalLine (Figure 3.6). Since the TwoLines type constrains its source and view fields, there are imposed restrictions on the view slot of object f1. Any object in this slot has to satisfy the imposed restrictions, i.e., in this special case, be a vertical line.

```
Object Type TwoLines
      source → Line
      view → Line
      constraint
            source.head.x = view.tail.y
            source.head.y = view.tail.x
            source.tail.x = view.head.y
            source.tail.y = view.head.x
end
```

Figure 3.7: Object Type for TwoLines.

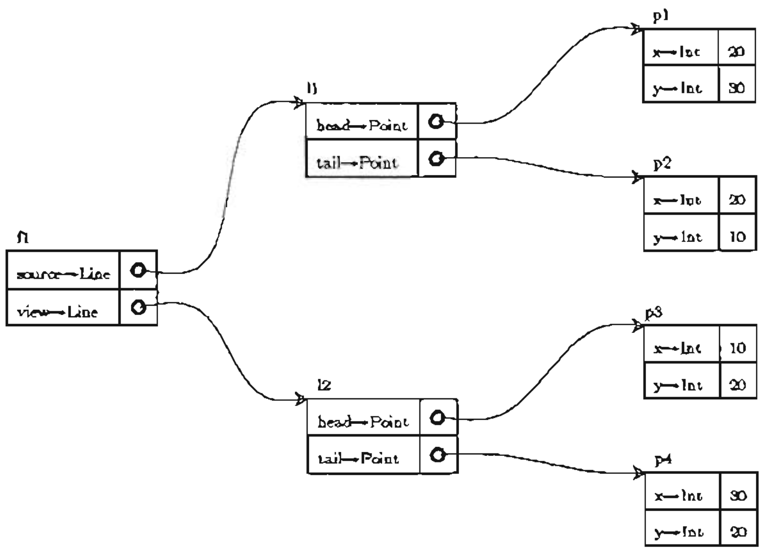


Figure 3.8: Instance of Object Type `TwoLines`.

Objects can be changed by storing values in their fields, or by adding or removing fields. The assignment operation has already been defined. The assignment operation **`Put`** = (`o`, **`A`**, `r`) is defined to be *consistent* if the imposed restrictions of the field slot specified by **`A`** = (**`f1`**, **`f2`**, ..., **`fn`**) within object `o` are satisfied by value `r`. To check whether an assignment is consistent we must check whether `r` conforms to the imposed restrictions for the field **`fn`** specified by **`A`**. To see whether `r` satisfies the imposed restrictions is hard, since we cannot easily find all objects that place constraints on the slot. (In general, it might be necessary to check all creation-time constraints.)

We only allow updates that add or remove fields for objects that have iterated or conditional field definitions in their creation-time types. After an assignment to a field that is either a condition in a conditional field or an iteration factor in an iterated field, the path existence constraints will add or remove a field to make the object conform to a different signature in its creation-time type. (The general constraint-satisfaction mechanism is defined later.) The new field is then accessible in assignment and retrieve operations.

Our object type system has a very powerful notion of type. As we saw in the discussion of the assignment operation, an implementation could be quite inefficient because of the constraint checking. The filters that are modelled within this object model will be structured in a way so they have no imposed constraints and therefore allow a more efficient implementation.

3.3. Filter Model

A filter is an object that represents constraints defined between two objects of specific types, where the constraints will be enforced by constraint-satisfaction. A filter type is a description of a set of filters. This section defines filters and their types in terms of the object model from Section 3.1.

Filters are objects with special structure and behavior. One aspect of the special structure is that all filters must have a source and view field for which constraints can be defined. Another aspect is that we distinguish atomic filters from constructed filters. Atomic filters represent primitive constraints, whereas constructed filters are built from atomic filters. Filters have special behavior in that they allow an assignment of a field value that violates a constraint. Filters try to accommodate the update by changing other field values or adding or removing fields to resatisfy the constraints. This mechanism is called constraint-satisfaction.

For example, consider an atomic filter with source and view fields containing integer values. The value of its source is constrained to be equal to the value of its view. After we change the source value, the constraint is violated. The filter accommodates this change by replacing the view value.

Now consider a constructed filter with a source and view fields that each contain a point. The filter is build from two atomic filters similar to the one in the previous example connected to the x and y coordinates of the two source and view points. The constructed filter represents the constraint that its source and view points have the same coordinates. After we change a coordinate in the source point of the filter the constraint is violated and the filter will resatisfy it by changing the corresponding coordinate in the view point.

We categorize filters that share common structure and constraints by types, as for objects, using terminology similar to that of Section 3.1. Signatures define addresses and types of fields and constraints impose restrictions on the existence and possible values of fields.

A filter atom is an object, whose creation-time type defines a signature with two address-type pairs and a collection of constraints. The addresses for the fields are always "source" and "view." A filter atom conforms to a filter type if its source and view values are instances of the appropriate types, and the constraints evaluate to true. Constraints for filter atoms are certain primitive predicates that can be satisfied directly; which primitive predicates are allowed is determined by the constraint-satisfaction used.

Consider again the atomic filter that constrains its source integer to be equal to its view. Its filter type has the signature

```
source, Integer
view, Integer,
```

and the constraint that the source field is equal to the view field.

Constructed filters also have a source and a view field, but also can have fields for subfilters and fields for variables. Variable fields reference intermediate objects that are used to connect the subfilters, e.g., as in end-to-end combination of subfilters. Subfilter and variable fields are special in that they do not have addresses accessible from outside the constructed filter. However, each has an address that can be used to express constraints within the filter.

Only path existence constraints and merge constraints are allowed for constructed filters. *Merge constraints* are used to attach the source and view fields of subfilters to the variables and the source and view fields of the constructed filter. Merge constraints specify that one object is referenced from two different field slots. It is expressed with two access paths that use addresses from the source, view, subfilter and variable fields.

A *filter type* **F** is a tuple (**ES** , **IS**, **C**) with an external signature **ES**, a collection **IS** of internal signatures, and a list of constraints **C**. **ES** is a signature that types just the source and view fields. **IS** = (**SS**, **VS**) has a collection **SS** of subfilter signatures and a collection **VS** of signatures for variable fields. Each signature in **SS** is a list of address-filter-type pairs where the address is internal to the constructed filter. Each signature in **VS** is a list of address-object-type pairs where the address is internal to the constructed filter. **C** is a list of constraints that are expressed with access paths that use addresses external and internal to the object.

A constructed filter conforms to a filter type **F** = (**ES** , **IS**, **C**) if (1) the values of the source and view fields conform to **ES**; and for **IS** = (**SS**, **VS**) (2) the values of subfilter fields conform to one of the signatures in **SS**; (3) the values of variable fields conform to one of the signatures in **VS**; and (4) the filter satisfies all constraints in **C**.

The signatures **ES** and **IS** could be combined into one set of signatures, but it is easier to consider the signatures broken into independent pieces.

Access to filters is restricted, since only source and view fields are visible externally. Only fields of the source and view objects (and their subfields) can be retrieved and updated externally.

Since subfilters do not have externally visible addresses and are only referenced from a single slot, they do not have imposed constraints, i.e., the creation-time type of a filter always matches the filter type for the slot it occupies.

The regular structure of constructed filters makes it easier to determine the constraints that are imposed on their source and view objects and variables, which helps to determine whether an assignment operation is consistent. The precise behavior of filters on assignments will be described after we have presented the language for filter types in the next section.

3.4. Filter Types

Filter types are specified by enumerating their subfilters and variables and by relating them to the source and view objects. Some filter types for filter atoms are predefined. Constructed filters are defined from other filter types.

3.4.1. Filter Atoms

A filter atom has a predefined filter type that declares the types for source and view objects and the constraints that hold for them. We distinguish three groups of filter atoms. There are:

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

Each group represents a class of filter types.

3.4.1.1. Equality Filter Atoms

For each of the atomic object types there is a filter atom representing an equality constraint (types that are accepted for source and view in parentheses):

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

Equality filter atoms ensure that their source and view fields hold the same atomic value. They do constraint-satisfaction by replacing, rather than modifying, the values in the slots they are attached to.

3.4.1.2. External Constraint Filter Atoms

Our filter specifications represent constraints, but we want to invoke certain geometric and computational constraints directly. Constraint filter atoms represent externally defined constraints for which the system knows a satisfaction technique.

3.4.1.3. Implementation Filter Atoms

Additional primitives are defined that allow an implementor to incorporate objects such as the display bitmap, the keyboard, or a pointing device in the object model and provide primitives that are used like filter atoms but are written in a procedural language. For example, consider a filter atom that maintains the position of the cursor in a point object. This filter atom can be used when constructing filters, thus allowing objects to be changed according to user input. Chapter 5 on implementation describes input and output primitives that are modelled as filter atoms.

3.4.2. Constructed Filters

Filter types for constructed filters define multiple signatures. In this section we describe their filter type definition.

The filter type definition follows the syntax:

```
Filter Type <Name> ( source : <source type> , view : <view type> )
        var
                <variable declaration list>
        make
                <filter construction list>
        merge
                <merge list>
end
```

Where:

<Name>

is a unique name for the type. Subsequent filter type definitions can reference this name when constructing other filters. The name must start with a capital letter.

<source type>

is labelled by the keyword `source` and specifies the object type for the source slot of the filter instance. The <source type> names an object type.

<view type>

is labelled by the keyword `view` and specifies the object type for the view slot of the filter instance. The <view type> names an object type.

<variable declaration list>

is a list of variable definitions of the form:

$$\langle\text{variable}\rangle \rightarrow \langle\text{object type}\rangle \ (\ \langle\text{initialization list}\rangle \ )$$

where <variable> is the name of the new variable, <object type> is the type for the variable, and <initialization list> is an initialization expression. The variable name can optionally be iterated (using "[ ]"). The iteration factor can be an integer literal or the address of an integer field within this filter type.

<filter construction list>

lists the subfilters that are declared for an instance of this filter type. In analogy to the object type definition, we provide a basic, iteration and condition format. These valid formats to declare subfilters are:

- **set of** <body>
- **iteration** <expression> **times** <variable> <body>
- **condition** <condition> <body>

where <body> is either a nested <filter construction list> or a declaration of subfilters in the format:

<filter type> (<source object> , <view object>).

Here <filter type> names the filter type for the subfilter, and <source object> and <view object> are access path expressions that are used to connect the subfilter to fields of source, view or variables within the filter type. (The connections are realized with merge constraints.)

The **set of** format declares several subfilters of possibly different types and arguments. It can be used for side-by-side or end-to-end composition. End-to-end composition requires a variable to serve as an intermediate object.

The **iteration** format declares a variable number of subfilters all of the same type. The <expression> defines the range of the <variable>, which can be used in the access path expressions for the source and view objects for the subfilter. The <expression> can contain an integer literal or an access path to an integer field in the source, the view or a variable. Therefore, the iteration can depend on another object that is itself part of an instantiated filter.

The **condition** format declares a subfilter that depends on a condition. The <condition> can mention access paths to fields, Boolean operations and comparisons of atomic values. The <condition> is evaluated within the instance of the filter type; therefore, the value can depend on an object that is part of another subfilter.

The **set of**, **iteration** and **condition** formats can be nested within each other, e.g., the **iteration** format can declare a set of subfilters with a **set of** format.

<merge list>

is a list of merge constraints for the filter type. The merge is expressed with two access paths and the special symbol '=○='. It says that the two paths must always lead to the same object. This merge constraint is the same as the merge constraints that are used to connect source and view of subfilters to source, view and subfilters of the constructed filter. The **merge** statement can be used to improve the clarity of the filter type by merging variables with fields of source and view objects; it is provided as a syntactic convenience.

Any of the statements **var**, **make** or **merge** can be omitted if not needed. With the given syntax we are able to define a powerful set of filter types. However, certain sets of signatures are not expressible, since multiple signatures are only derived from conditional and iterated fields.

Figure 3.9 shows a filter type `IterationExample` constructed from four subfilters of type `IntegerIdentity`. It uses object types defined in Figure 3.2. This filter establishes an equality constraint between an integer array of size 4 and the components of a list. Note that the literal 4 in this example is necessary since none of the participating objects contain information about the size of the arrays. If the source or view object or a variable within the filter type definition could be

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

Figure 3.9: Filter Type with Iteration Constructor.

used to express the iteration factor, the literal is not needed. The **merge** statement merges the four fields of the view with the four fields of the variable. Because we used a variable that is an array, we were able to define the four subfilters with an iteration format.

If we want to combine filters of different types, we use the **set of** format. Figure 3.10 shows the `SetExample` type where an `IntegerIdentity` filter

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

Figure 3.10: Filter Type with Set Constructor.

atom and the IterationExample of the last example are composed (object types were defined in Figure 3.2). Note that the connection of subfilters to subparts of source and view objects of the defined filter establishes merge constraints.

Figure 3.11 shows the ConditionExample, which is similar to the IterationExample except that the existence of the IterationExample subfilter is conditional on the value of the label address of the source and view object (object types were defined in Figure 3.3).

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

Figure 3.11: Filter Type with Condition Constructor.

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

Figure 3.12: Filter Type with Recursion.

Figure 3.12 shows the RecursionExample type, which is the same as the SetExample except that it instantiates the RecursionExample again recursively, depending on whether the label address of either source or view object is NIL. It is possible to define infinitely recursive filter structures. The recursion in filters usually follows the structure of some tree-like object, and terminates at the leaves of the object.

3.5. Filter Behavior

Each filter type describes a large universe of different possible objects and atomic constraints, but each instance of a filter type has exactly one configuration at any time. The different configurations possible for a filter are described by multiple signatures in the filter type, which arise from conditional and iterated subfilters and iterated variables. The initial configuration of a filter is built when the filter is instantiated and may change to another configuration when objects within the filter instance change. The configuration is never changed directly from without the filter. Rather, filters change their objects and their configuration in response to changes in their source and view objects when attempting to resatisfy their constraints. This section will describe two aspect of filters: filter instantiation and changes to objects that are part of a filter.

A filter object is instantiated from a filter type (**ES**, **IS**, **C**). It will accept objects for its source and view fields that conform to the **ES** signature. There are no further imposed constraints, therefore, this initial assignment is consistent. When a filter is instantiated it follows this procedure:

```
(1) instantiate declared variables from their instantiation lists.
(2) for all declared subfilter constructors do:
        (a) if the subfilter constructor is sequence, then
            for all subfilters in body of sequence do:
                determine the merge constraints for the subfilter;
                retrieve subfields from source, view or variable
                according to access path in merge constraint;
                instantiate subfilter with values of subfields as source and view.
        (b) if the subfilter constructor is condition, then
                evaluate condition, if true, then perform (2)
        (c) if the subfilter constructor is iteration, then
                evaluate iteration factor and perform (2) for each
                iteration of the body in the constructor.
```

This procedure is invoked recursively until it reaches a filter atom, which is instantiated like an object with values for the source and view field. If one of the objects is not completely specified, i.e., a slot does not contain a value or a reference, then it will be initialized according to the constraints. This behavior of filter atoms is called *initial value propagation*, and is not necessarily directed from either source to view or view to source. After initial value propagation the constraints for the filter atoms are checked. If there is a conflict, then the source object overrides the view object. After all filter atoms are instantiated and connected together with merge constraints, initial constraint-satisfaction is invoked for the complete filter instance. If the initial constraint-satisfaction fails, then the instantiation procedure fails.

For example, consider the filter type `Example` in Figure 3.13, with source and view objects of type `Point`. The `x` coordinates are kept equal with an `IntegerIdentity` filter atom, while the `y` coordinates are only kept equal if the `x` coordinate of the source point is greater than zero. If we instantiate a filter from

```
Filter Type Example (source: Point, view: Point)
make set of
      IntegerIdentity ( source.x, view.x )
      condition source.x > 0
            IntegerIdentity (source.y, view.y)
end
```

Figure 3.13: Filter Instantiation.

this filter type with the point (x: 10, y: 10) as source object and an unspecified point object (x: NIL, y: NIL) for the view field, the procedure is as follows. The first subfilter in the `set of` constructor is a filter atom; it is instantiated like an object with the x coordinate of the `source` point as source object and NIL as view object; initial value propagation will set the value of the view field to 10. Then, the expression in the `condition` constructor is evaluated and the second `IntegerIdentity` filter atom is instantiated with the y coordinate of the `source` point as source object and NIL as view object; initial value propagation will set the value of the view field to 10. The resulting configuration of the filter instance consists of two filter atoms.

The only objects that can be accessed from outside a filter are the source and view objects and their subfields. Such an object can be changed in a way that temporarily violates its creation-time and imposed constraints. The filter tries to accommodate the change by changing the value of other fields within the filter or by changing the configuration of the filter. The only constraints that are considered are local to the filter, therefore we do not have to search for all references to objects, but can restrict the search to within the filter. If the configuration changes, then new filters are instantiated with the same procedure as described above. Change in a part of the source and view object may mean that a subfilter must reconnect to a new object. Therefore, the merge constraints hold dynamically; all paths that are merged with a merge constraints have to be considered when the value along one of them is changed. If the constraints cannot be satisfied for an update to an object, then the update is disallowed.

For example, consider the filter from the previous example. If the value of the x coordinate of the `source` point is changed to -10, then constraint-satisfaction can accommodate this change by changing the value of the x coordinate of the `view` point. Moreover, since the x coordinate of the `source` point is now less than zero, the second `IntegerIdentity` filter atom will be removed. Any change to the y coordinate of one point will no longer affect the y coordinate of the other.

Constraint-satisfaction is inherently non-deterministic. A given constraint can be resatisfied in multiple ways. Also constraint-satisfaction systems have different levels of power, as we will describe in Chapter 5 on implementation. A method that tries to resatisfy the constraints locally may not find a solution, whereas a method that considers the constraints globally might succeed. The exact behavior of a filter therefore depends on the constraint-satisfaction mechanism that is used in an actual implementation; but any implementation must respect the constraints of the filter's type.

Chapter 4
Defining Interfaces Graphically

Filter types can describe interfaces. In order to define interfaces graphically we represented filter types in an object-oriented system and provided a graphical tool that manipulates the representation. Filter types could also be defined with the filter specification language described in Chapter 3 and translated by a compiler into code that describes the appropriate classes in an object-oriented system. The code produced by such a compiler does not necessarily have to be object-oriented. We chose to provide a graphical tool, since we are mainly interested in the flexibility of the filter paradigm, but are aware that a compiler will eventually be needed to produce optimized and efficient interfaces. Our tool, the *Filter Browser*, can define, manipulate and test filter types graphically. The filter browser has been implemented in Smalltalk-80[1] on a Tektronix 4405 AI workstation.

This chapter describes the filter browser. The first section outlines the general organization of this tool to define interfaces graphically. Sections 4.2, 4.3 and 4.4 explain the features of the filter browser. Figures 4.3, 4.4 and 4.5 show the filter browser as we step through the definition of the filter type `ProducerFilter` from our introductory example in Chapter 2. Section 4.5 explains some possible

[1] Smalltalk-80 is a trademark of Xerox Corporation

enhancements that would improve the filter browser. The last section concludes with a comparison of the filter specification language and the filter browser.

4.1. The Filter Browser

A filter type describes a set of filter instances. As discussed in Chapter 3, types are described with signatures and constraints. Filter types can be represented as classes in an object-oriented system that supports types for instance variables and constraints. The filter browser creates the classes that represent filter types. It also maintains a sample filter instance (prototype) that can be used to instantiate and test the filter type. Object types can be implemented directly in such a system as classes where instances of classes are instances of the object type. The filter browser cannot be used to define object types; Chapter 5 gives the details of the implementation.

The filter browser is similar to the Smalltalk [Goldberg 84], ThingLab [Borning 79] and Animus [Duisberg 86] browsers, which also browse classes. Except for the name of a new filter type, the filter browser specifies filter types entirely graphically, using menus, icons and a pointing device.

In defining filter types, we distinguish the external and internal parts of the definition. Figure 4.1 shows a filter type. Externally, a filter type is identified by its name (`ProducerFilter`) and the types of its source (`Producer`) and view (`Device`) object. Internally, a filter type declares subfilters (e.g., `IntegerEquality`) and variables (`gauge`). These details are encapsulated inside the filter type. A ses-

```
Filter Type ProducerRender (source: Producer, view: Device)
var
      gauge → Gauge
make set of
      IntegerEquality (source.productivity, gauge.number)
      Render (gauge, view.output)
      PointSensor (gauge.needle.point2, view.input)
end
```

Figure 4.1: ProducerFilter Filter Type.

sion with the filter browser has three different steps. Step one represents the external, step two the internal definition, and step three tests the filter type. In step one, the name of the filter type and the type of source and view objects are given. In step two, the variables and subfilters that participate in filter constructors, such as sequence, iteration and condition are specified. In step three, a constructed filter type is instantiated and exercised. The designer of a filter type will first proceed from step one to step two and then test the filter type in step three. Then he can go back to step two and add or delete subfilters and variables. At any time he can test the filter type in step three. If he goes back to step one and changes the source or view object type he will invalidate the internal parts of the filter type and has to redo step two from the beginning.

Figure 4.2 shows the general screen layout of the filter browser interface to filter types. The same layout is used for steps one and two. Area 1 displays the list of existing filter types. Filter types can be selected; a pop-up menu is available that

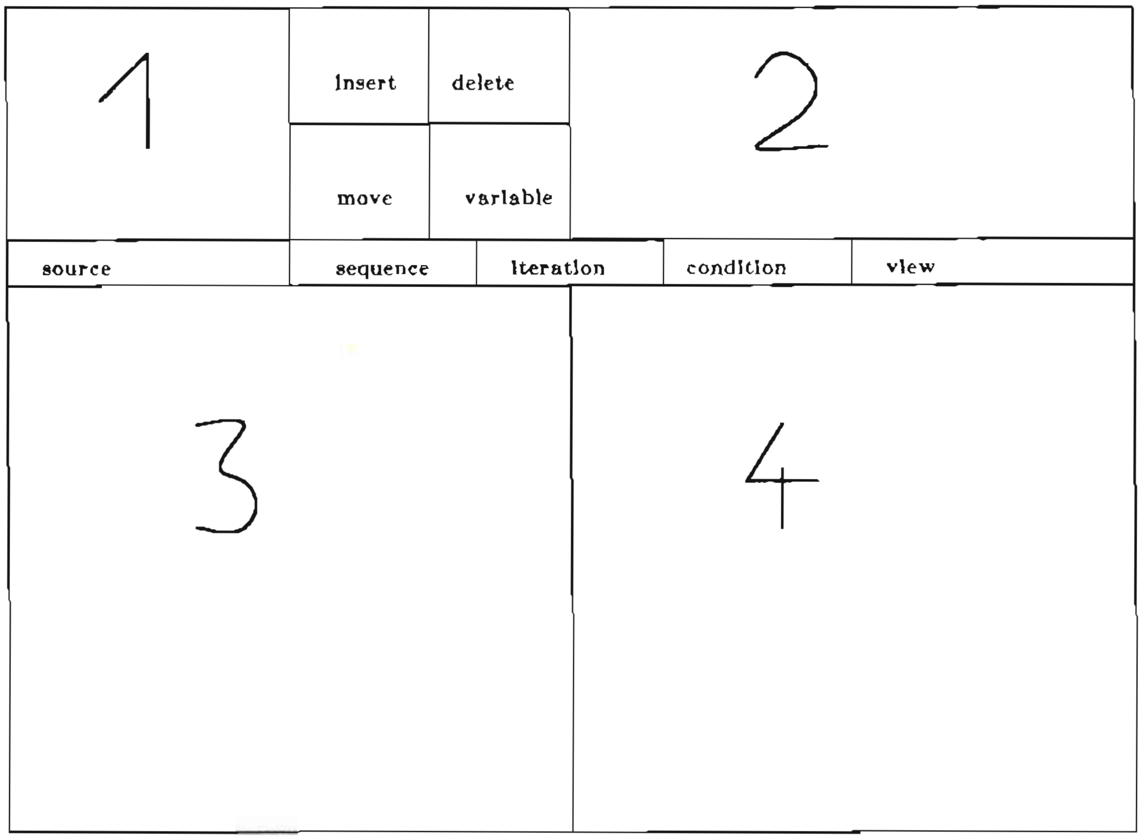


Figure 4.2: Filter Browser Layout.

can be used to create a new filter type, delete an existing filter type, instantiate a filter type, i.e., proceed to step three, or file out the description of the filter type onto the external file system in a form that is understood by the implementation. Area 2 is not used in step one. In step one, area 3 displays a list of all object types that can be defined as source types for the currently selected filter type. Area 4 contains the same list for view types. After either of the object types has been selected a pop-up menu is available to inspect the object type definition.

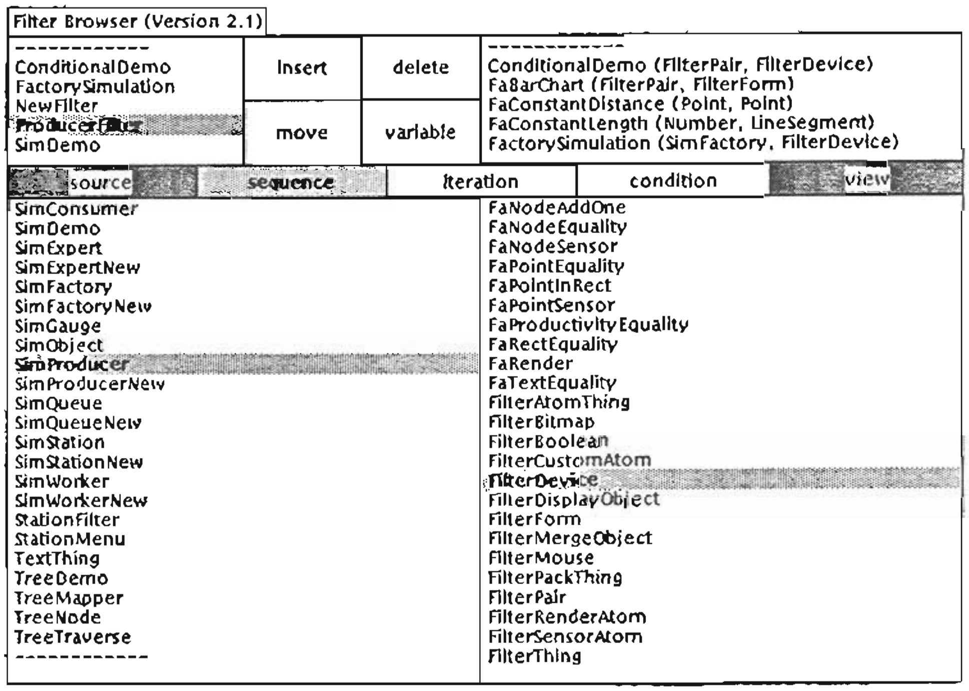
Below area 1 is a button, labelled *source*, that when selected will cause the filter browser to switch from step one to step two, or vice versa; the *view* button below area 2 has exactly the same function, which is just a convenience for the user. To the right of area 1 is a block of 4 actions (insert, move, delete, variable) that can be selected. If the *insert* action is selected, then area 2 lists all filter types that are available for insertion. This list includes filter atoms, but the list in area 1 does not, because filter atoms are predefined and cannot be changed with the filter browser. If the *variable* action is selected then area 2 lists all object types that are available for variables. The buttons labelled *sequence*, *iteration* and *condition*, select the filter constructor. In step two, areas 3 and 4 are combined to display a graphical representation of the variables, subfilters and their connections within the defined filter type. How subfilters and variable are inserted is described in Section 4.3.

In step three, the whole filter browser window is replaced. It now displays only two panes as shown in Figure 4.5. The left pane shows the bitmap that serves as the display screen for the filter type that is instantiated. The right pane contains a list of source, view and variable objects that can be inspected and changed, thus influencing the filter and the display in the left pane.

The next three sections will describe how to create a filter type using the filter browser. The `ProducerFilter` filter type will serve as an example. Figure 4.1 shows its filter type definition. Filters of this type constrain a producer object to be displayed on a device screen with a gauge and to accept user input to change its productivity. The definition of all object and filter types for the complete example is listed in the Appendix A.

4.2. External Definition

In this step only the upper left, lower left and lower right panes are used. The upper left pane shows a list of all filter types that are known to the system. The user can add a new filter type or select an existing filter type for modification. The current subject of the filter type definition, i.e., the filter type that will be modified



```
------------
ConditionalDemo
FactorySimulation
NewFilter
ProducerFilter
SimDemo
```

```
------------
ConditionalDemo (FilterPair, FilterDevice)
FaBarChart (FilterPair, FilterForm)
FaConstantDistance (Point, Point)
FaConstantLength (Number, LineSegment)
FactorySimulation (SimFactory, FilterDevice)
```

```
SimConsumer          FaNodeAddOne
SimDemo              FaNodeEquality
SimExpert            FaNodeSensor
SimExpertNew         FaPointEquality
SimFactory           FaPointInRect
SimFactoryNew        FaPointSensor
SimGauge             FaProductivityEquality
SimObject            FaRectEquality
SimProducer          FaRender
SimProducerNew       FaTextEquality
SimQueue             FilterAtomThing
SimQueueNew          FilterBitmap
SimStation           FilterBoolean
SimStationNew        FilterCustomAtom
SimWorker            FilterDevice
SimWorkerNew         FilterDisplayObject
StationFilter        FilterForm
StationMenu          FilterMergeObject
TextThing            FilterMouse
TreeDemo             FilterPackThing
TreeMapper           FilterPair
TreeNode             FilterRenderAtom
TreeTraverse         FilterSensorAtom
------------         FilterThing
```

Figure 4.3: Filter Browser Step One.

or newly defined, is highlighted. The filter browser displays two lists of all object types that are available for source and view types in the two panes below. The user can inspect all object types and select one for source and one for view. In Figure 4.3 the `ProducerFilter` filter type is defined on source objects of type `SimProducer`[2] and on view objects of type `FilterDevice`; these object types are highlighted in the lists.

4.3. Internal Definition

In the second step, the upper left pane still displays the list of filter types with the current selection, `ProducerFilter`, emphasized. To the right there are four panes to select the action that is to be performed on the current filter type. The user can *insert* subfilters, add *variables*, *move* or *delete* subfilters or variables in the picture pane below. The picture pane replaces the two lists of object types from step one. If the *insert* action is selected, then the upper right pane shows a list of all filter types and the types of their source and view objects. This list also contains the filter type currently being defined to allow a recursive definition. The selected element in this list names the filter type to be inserted as subfilter if the *insert* action is selected. If the *variable* action is selected, then the upper right pane shows a list of all available object types from which the user may select. The kind of subfilter constructor (sequence, iteration, condition) is selected with one of the three panes in the middle of the filter browser.

[2] The type `SimProducer` represents the object type `Producer` In the actual implementation of the simulation we preceded all object types with the prefix "`Sim`".

The picture pane is used to display the subfilters, variables and their connections. A variable is placed in the picture pane by selecting the *variable* action and selecting an object type from the list in the upper right-hand pane. The variable appears as a box that shows its name and type and the addresses its fields with their types; the types of subfields are expanded to show their fields. A unique name for the variable is generated from its type, e.g., the variable of type `SimGauge` has name `simGauge4`. The access path is abbreviated so that only the last field name is shown. The access paths to fields of subfields are also displayed. Conditional and iterated fields are not displayed, since our current implementation does not support them for objects. The user selects a location and places the variable. The variable is then inserted into the current filter type definition. A subfilter is placed in the picture pane by selecting the *insert* action and a filter type from the upper right pane. The added subfilter is then connected (linked) to paths in either the source, view or variable objects by pointing at their location on the screen. For iteration or condition constructors, an iteration or condition object has to be selected by pointing to a path that represents the iteration factor or the condition. After the subfilter is placed in the picture pane it is inserted into the current filter type definition.

Figure 4.4 shows the filter browser as the user inserts the `FaIntegerEquality`[3] subfilter into the `ProducerFilter` filter type. A variable of type `SimGauge` has already been defined and connected to an `FaPointSensor` and `FaRender` subfilter. The `FaRender` filter atom renders the gauge object on the

[3] All filter atom names are preceded by a "`Fa`" prefix.

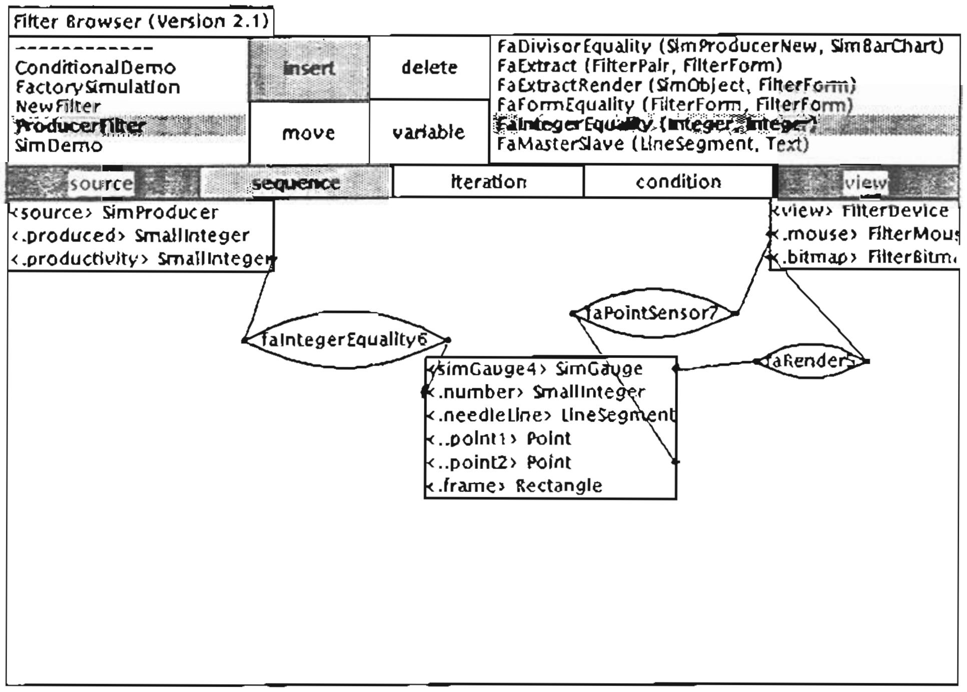
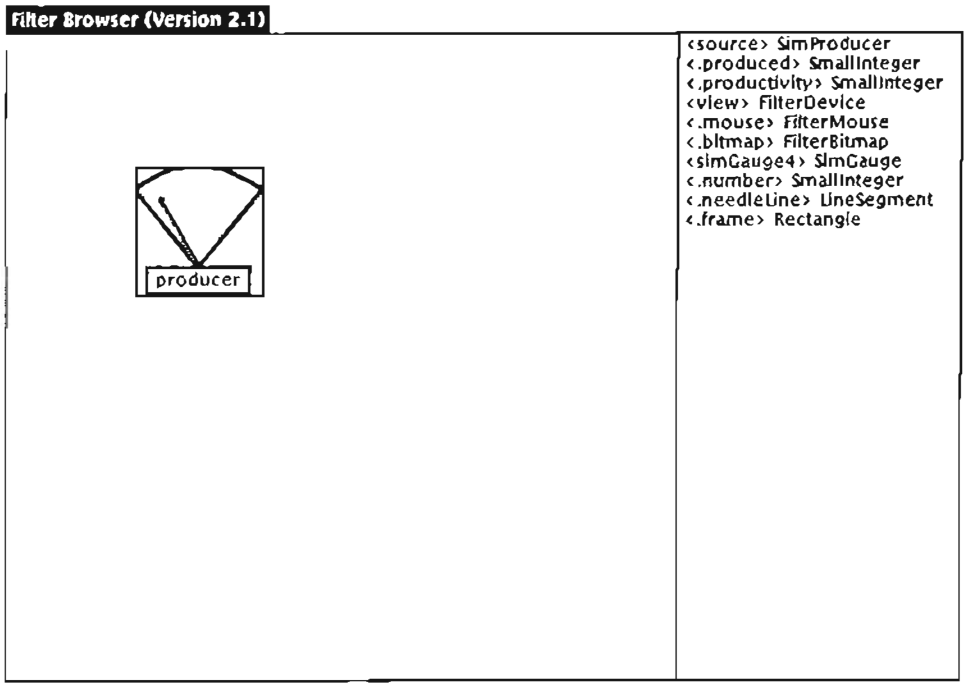


Figure 4.4: Filter Browser Step Two.

display bitmap. The `FaPointSensor` filter atom moves the second point of the gauge needle according to the location of the mouse input device. After the *insert* action has been selected and the user moves the mouse cursor into the picture pane a lozenge for the `FaIntegerEquality` filter appears. The source link is connected to the `<.productivity>` address of the source object and the view link is connected to the `<.bitmap>` address of the view object. After insertion it is displayed with its name, `faIntegerEquality6`, that is also generated from its filter type.

The picture pane represents the network of subfilters. The absolute position of the subfilters in the picture pane is not important, but it will be stored to redraw the subfilter network in the same way as it was defined. The *move* action is available to move subfilters or variables in the picture pane to enhance the appearance of the network. Subfilters or variables can be deleted from the filter type with the *delete* action. If a subfilter is deleted, then all its connections are also removed.

An important issue in connecting subfilters is typing. The external definition of a subfilter specifies the object type for its source and view object. Thus, when a source or view link of a subfilter is connected to addresses of source, view or variable objects of the currently defined filter type, it is necessary to check the corresponding types. These source and view links can be connected to object types that are of the same type as, or are subtypes of, their specified object types. For example, the source of the `FaIntegerEquality` filter has to be of type `Integer` or one of its subtypes. The `productivity` field of the source `Producer` is of type `SmallInteger`, which is a subtype of `Integer`. The filter browser checks the types of a subfilter when it is inserted. If the type checking fails, then the subfilter is not inserted and an error is reported to the user.



```
<source> SimProducer
<.produced> SmallInteger
<.productivity> SmallInteger
<view> FilterDevice
<.mouse> FilterMouse
<.bitmap> FilterBitmap
<simGauge4> SimGauge
<.number> SmallInteger
<.needleLine> LineSegment
<.frame> Rectangle
```

Figure 4.5: Filter Browser Step Three.

4.4. Filter Instantiation

Instantiating a filter type means creating an instance of the class that represents the filter type. When instantiating a filter, object instances of the correct object types have to be supplied for source, view and variables. This instantiation operation is accessible from a pop-up menu in the upper left pane of the browser, where the filter type has been selected. The left pane of the filter browser simulates the display bitmap for the filter and all input is controlled by the filter browser, so it is possible to switch back to the previous steps. The right pane of the filter browser shows the participating instantiated objects. They can be selected, inspected and updated. Any change to participating objects may immediately change the display in the left pane. Figure 4.5 shows the instantiated `Producer-Filter` filter type for a prototype `Producer`. The `FilterDevice` is simulated by the filter browser. The pane on the left displays a sample producer object with a gauge. The gauge needle can be moved with the mouse cursor. Appendix A lists the complete filter and object type definitions.

4.5. Possible Enhancements

Our experience with the initial version of the filter browser has shown that additional textual output would be helpful to the designer of an interface. Filter types can be described with the filter description language and there is an almost one-to-one mapping from filter types as defined by the language and their representation as classes in an object-oriented system. Therefore the filter browser could produce this language representation to give the interface designer additional feedback.

Another area for enhancements is the display and manipulation of the network of subfilters and variables in step two. More functionality, such as hiding and clustering of detail or lookup of subfilters by zooming, would be desirable.

4.6. Filter Browser vs. Specification Language

The filter browser was meant to provide the same expressibility as the filter specification language described in Chapter 3. In this section we will show that for each feature of the language there is a corresponding feature in the filter browser, but that there is no exact one-to-one structural mapping between the two.

The filter specification language has three aspects: objects, filter atoms and constructed filters. The goal of the filter browser was to be able to specify filter types for constructed filters. Types for objects and filter atoms were assumed to exist.

A filter type for a constructed filter specifies six properties: (1) the name of the filter type; (2) the object types for source and view; (3) the variables of the filter type and their types; (4) the subfilter constructors such as sequence, iteration and condition; (5) the subfilters of the filter type with their types and their connections to source, view and variables; (6) the additional merges, which can be viewed as notational convenience. The following paragraphs will explain how each of these properties is specified with the filter browser.

(1) The name of the filter type is defined when creating it. This name identifies the filter type when it is used as subfilter in subsequent filter type definitions.

(2) The source and view object types are selected from the list of all object types that are available. This object type is later used to check whether the filter type can be inserted as subfilter in another constructed filter.

(3) Variables can be defined by selecting the *variable* action in step two, and selecting an object type for the variable from the list of all object types in the

top-left pane. The variable is then displayed in the lower pane of the filter browser in step two. The filter specification language allows the initialization of fields within variables. The filter browser cannot express this initialization, but values for fields in variables can be set in step three where all participating objects are listed and accessible in the right pane.

(4) Filter constructors are selected with one of the three buttons in the middle of the filter browser that are labelled "sequence", "iteration" and "condition".

(5) Subfilters can be defined by selecting the *insert* action in step two, and selecting a filter type for the subfilter from the list of all available filter types in the top-left pane. The subfilter is then displayed in the lower pane of the filter browser and the user has to connect it to fields within source, view or a variable according to the selected subfilter constructor. All possible access paths are available, since the object types for all fields of the source, view and variable objects are expanded.

The condition subfilter constructor in the filter specification language allows a conditional expression whereas the filter browser uses only a single object of type Boolean for the condition. In order to have a conditional expression for the condition subfilter constructor, the user may first use constraint filter atoms that evaluate a Boolean expression into a variable of type Boolean. Such constraint filter atoms have to be provided by the implementation. This Boolean variable can then be used in the condition subfilter constructor. The iteration subfilter constructor allows an iteration factor of type `Integer` exactly as defined in the language.

Another difference to the filter specification language is that it allows nesting of subfilters within constructors. The filter browser can only define a flat sequence of subfilters where each subfilter may be conditional, iterated or basic. However, arbitrary nesting can be achieved by defining separate filter types for sequences of subfilters and then using these filter types as subfilters within the appropriate constructors.

(6) Additional merges cannot directly be expressed with the filter browser. However, the implementation can provide a "merge" filter atom that can be inserted as a subfilter into the filter type. It would be possible to add the direct support of merges to the filter browser by adding another action button that would work the same way as the *insert* button but would select the special "merge" filter atom automatically. We do not think that the absence of merges is a serious limitation, since they are only provided in the language as a notational convenience.

Overall, we can say that the filter browser matches but does not exceed the expressibility of the filter specification language, but offers more flexibility by letting the user test his design immediately and allowing interactive changes.

Chapter 5
Implementation

The goal of our research is to provide an architecture to generate interfaces interactively. The filter specification language provides a way to specify interfaces. In addition, we wanted to implement these ideas in an object-oriented system on a graphical workstation. In the course of our research we implemented the filter paradigm twice. However, the first version did not include an implementation of the filter browser. The first system mapped the filter types that describe interfaces onto Smalltalk-80 classes [Goldberg and Robson 83] and attempted to provide a hand-coded constraint-satisfaction. The constraint-satisfaction strategy was local and therefore very limited in that it could only handle forward propagation of values. Fortunately, we had available ThingLab [Borning 79], which extends Smalltalk with constraints. We used ThingLab for the second implementation of our filter paradigm. This implementation has two aspects: (1) filter and object types are represented as Smalltalk classes and ThingLab "things," and (2) the filter browser is an interface to these classes that can create and manipulate objects that represent filter types.

The first section in this chapter describes our first implementation of the filter paradigm. Section 5.2 reports how filters would be represented as objects in an ideal object-oriented system that provides constraint-satisfaction. Section 5.3 describes the implementation of filter types and the filter browser in the Smalltalk and ThingLab environment. Section 5.4 compares the different features of the two implementations. The last section concludes with a few examples of filter types that have been defined, including the filter browser itself.

5.1. First Implementation

The first implementation of the filter paradigm was done in Smalltalk without a pre-existing constraint-satisfaction mechanism. It was used to explore the features of the filter paradigm. The strategy of the implementation was to map object and filter types into classes and represent constraints as hand-coded methods. This implementation is also described in a technical report [Ege 86].

In order to detect updates to objects, the classes that represent the object and filter types implement a monitoring protocol that controls the access to the objects and filters. Instances of these classes that hold the actual objects and filters are called *object holders.* The object holder accepts registrations from objects that use the held object in constraints. Any update of the held objects is done through the object holder. Whenever an update message is received the holder forwards this message to the held object and notifies the objects that had registered their interest. An object is installed in an object holder by using Smalltalk's *becomes* message that changes the object holder's identity to the identity of the held object.

Filter types are represented as classes that implement methods to map the source object into the view object, and vice versa. The idea is that the filter knows what constraints are to be enforced for the source and view objects. When a filter is created, it has to be supplied with instances for source and view object. The filter instance registers with the source and view objects, which are held by object holders. The filter is then notified when source or view objects are changed and examines their values to make other updates if necessary. Connection of subfilters for a filter is done in methods defined for the filter by creating instances of classes that represent object types and creating instances of subfilters with them.

Constructed filters are built by defining instance variables to hold subfilters and variables in the class that represents the filter type. Subfilter instances are not directly referenced in instance variable, but indirectly through a subfilter descriptor object. Three classes of subfilter descriptors are available, for basic, iterated and conditional subfilters. All subfilter descriptor objects have an instance variable to reference the subfilters. The basic subfilter descriptor references the subfilter instance; the iterated subfilter descriptor registers with the object holder that holds the iteration factor and references a collection of subfilters; the conditional subfilter descriptor registers with the object holder that holds the condition object and a subfilter. The subfilter descriptors also store symbolic information on how to access the source and view objects for the subfilters. This symbolic information is similar to the access paths described in Chapter 3. The class for the iterated subfilter constructor implements the behavior for when the iteration factor is updated and the subfilter constructor is notified, i.e. the constructor adds or removes subfilters from the collection of subfilters using the symbolic information on the source and view objects for the subfilters. The class for the conditional subfilter constructor implements a similar behavior in that it creates or deletes the subfilter depending on the value of the condition object.

This first implementation is able to support the full expressibility of our filter description language. However, the constraint-satisfaction is local to the filter objects and requires hand-coding of the constraints. The only method of resolving conflicts is propagation of values. The restricted constraint-satisfaction limits the usability of this implementation for complex interfaces. Therefore, we did not try to provide a tool to specify and generate interfaces automatically for this implementation.

5.2. Filters as Objects

Our initial approach to implement filters and filter types in an object-oriented environment is that we assume that we have an implementation of objects and object types as defined by our object model. This imaginary implementation will provide object types with multiple signatures and sets of constraints, and creation-time types for objects. We also assume a special assignment operation that allows temporary violation of constraints and tries to accommodate the change of objects by changing other objects or their configuration using constraint-satisfaction.

This section describes how filters and filter types can be represented as objects and object types by mapping filter types to object types. How filter types are mapped into object types is described in the next two subsections with some examples for filter atoms and constructed filters.

In general, a filter type is modelled as an object type with at least two fields, for its source and view object type. Object types for filter atoms declares their constraints directly, while object types for constructed filters have additional fields for variables and subfilters, and constraints to merge the subfilters with source and view objects.

5.2.1. Filter Atoms

Filter atoms are used as the lowest-level components when building interfaces. As described in Section 3.4.1, there are different types of filter atoms. Filter types for filter atoms can be represented directly as object types. The object type defines the signature for the source and view field and defines the constraints that are used for the filter atom: Equality filter atoms have equality constraints; constraint filter atoms name their external constraints; implementation filter atoms don't name a constraint, but they will be recognized by the implementation and their behavior will be modelled as procedures.

For example, consider the `IntegerIdentity` filter atom. Figure 5.1 shows its object type definition. The fields for `source` and `view` reflect the types defined for the `IntegerIdentity` filter atom. The constraint is a primitive equality predicate on atomic integer values and says that the value stored in the source field is identical to the value stored in the view field.

```
Object Type IntegerIdentity
      source → Integer
      view → Integer
      constraint
            source = view
end
```

Figure 5.1: Object Type for Filter Atom.

5.2.2. Constructed Filters

Constructed filters are built from subfilters using the three different subfilter constructors: basic, condition and iteration. They can also have variables. The variables are added as fields to the object type representing the filter type. Subfilters are also defined as fields in the object type. Subfilters are then connected by placing constraints on their source and view fields that merge them with the appropriate fields within the filter type. These constraints are the merge constraints as defined in Section 3.3, and we assume that they are supported by our ideal implementation of our object model.

The basic sequence constructor is modelled by listing the subfilters and equating their source and view fields with fields of the filter type. For example consider the `SetExample` filter type in Section 3.4 (Figure 3.10). Figure 5.2 shows its object type definition. The symbol "==" denotes a merge constraint. The merge constraints equate the source and view fields of the subfilters with fields of source and view of the containing filter.

```
Object Type SetExample
      source → ArrayTwo
      view → ListTwo
      subfilter1 → IntegerIdentity
      subfilter2 → IterationExample
      constraint
            subfilter1.source == source.label
            subfilter1.view == view.label
            subfilter2.source == source.dependents
            subfilter2.view == view.dependents
end
```

Figure 5.2: Object Type for Sequence Filter Constructor.

The condition constructor can be represented by an object type that lists the merge constraints within the conditional field. A conditional field that is declared for an object type produces two different signatures (see Section 3.1), one listing the additional field and merges, the other not. Figure 5.3 shows the `ConditionExample` described in Section 3.4 (Figure 3.7) as an object type with the merge constraints placed only if the access path to the field `subfilter2` exists. The `PathExistence` constraint has to be provided by our ideal implementation.

The iteration constructor is either of static or dynamic nature depending on whether the filter type specifies a constant for the iteration factor or a field. If the iteration factor is a constant, then the iteration can be unfolded to yield a sequence, which will be represented by a sequence constructor. If the iteration factor is a field value, then the number of subfilters is dynamic, and so is the number of merge constraints to be generated. The mapping of filter type to object type with iterated fields is similar to the mapping as shown for conditional subfilters.

```
Object Type ConditionExample
      source → ArrayThree
      view → ListThree
      subfilter1 → IntegerIdentity
      (source.label = NIL or view.label = NIL)
            subfilter2 → IterationExample
      constraint
            subfilter1.source == source.label
            subfilter1.view == view.label
            PathExistence('subfilter2') and
                  (subfilter2.source == source.dependents)
            PathExistence('subfilter2') and
                  (subfilter2.view == view.dependents)
end
```

Figure 5.3: Object Type for Condition Filter Constructor.

The iteration subfilter constructor can also be modelled by using recursion and the mapping for the conditional subfilter constructor. Figure 5.4 shows a filter and object type definition where the dynamic iteration is converted into a condition and a recursive invocation. This example defines an object type whose instances have as many subfilters as specified in the `factor` field. The example assumes that an iterated field (`<some array>`) can respond to the selectors `first` and `rest` to retrieve the first and remaining elements in the array like a list.

```
Filter Type DynamicIteration (source:<some array>,view:<some array>)
var
        factor → Integer
make
        iteration factor times i
                IntegerIdentity (source[i], view[i])
end

Object Type DynamicIteration
        source → <some array>
        view → <some array>
        factor → Integer
        (factor > 0)
                subfilter → IntegerIdentity
                restfilter → DynamicIteration
        constraint
                (factor > 0) and (subfilter.source == source.first)
                (factor > 0) and (subfilter.view == view.first)
                (factor > 0) and (restfilter.source == source.rest)
                (factor > 0) and (restfilter.view == view.rest)
end
```

Figure 5.4: Object Type for Iteration Filter Constructor.

5.3. ThingLab Implementation

The last section showed how the filter paradigm could be implemented easily by mapping the filter types to object types under the assumption of an ideal (for our filter paradigm) object system with a constraint-satisfaction system that handles our notion of dynamic types and constraints. The implementation that is described in this section uses ThingLab as the object and constraint-satisfaction system. ThingLab does not fulfill all our needs, but we show how we realized an almost complete implementation of our filter paradigm.

ThingLab is an extension to the Smalltalk programming environment. Smalltalk groups objects into classes that are part of a class hierarchy. Objects that belong to a class are instances of that class. Classes define *instance variables*, i.e., the fields an instance has, but there is no type information on what kind of object can be stored in them. Classes also define *class variables* that hold values that are common to all instances of a class.

ThingLab is a constraint-satisfaction system that extends Smalltalk classes to Things that can have constraints defined for them. An instance of a Thing obeys the constraints that are defined for the class that represents the Thing. A detailed description of ThingLab can be found in Alan Borning's Ph.D. Thesis [Borning 79].

Instance variables for Things are called "parts". Parts extend Smalltalk instance variables, in that the class stores field descriptions in class variables for each part. The field description contains the field name of the instance variable[6]. The field description is static, and does not feature iterated or conditional parts. The field descriptions are used to support access paths and to provide information about the fields content that can be used during constraint-satisfaction planning.

A *prototype* is a distinguished sample instance of a Thing. The prototype is used to infer the type of instance variables in instances of the Thing. Each Thing must have a prototype and can be changed only through its prototype. The field descriptions and the prototype establish a signature (see Section 3.1) that instances of Things conform to: the field name is stored in the field description, the type of the

[6] The field description also contains a reference to a class that represents the type of the part, but this field is not maintained correctly in the current version of ThingLab

field slot is deduced from the object that is stored in the corresponding instance variable of the prototype. Therefore, we can only represent object types with a single signature. The conditional and iterated fields that can be defined for object types establish a union of signatures. This feature cannot be implemented directly in the current version of ThingLab.

Constraints are defined for classes that represent Things and stored in class variables. A constraint is defined with a Boolean predicate and fragments of pseudo-code that specify how to maintain the constraint. The Boolean predicate and the code fragments are expressed in terms of access paths and Smalltalk messages. All instances of a Thing have to obey the defined constraints. If an instance is updated, then the code fragments are used to compile a Smalltalk method that is executed to enforce the constraints. These constraints are static, and cannot be defined conditionally or for a dynamic set of objects.

ThingLab distinguishes merges and constraints. Merges can be defined for Things. Merges are expressed with access paths and are stored in class variables for classes that represent Things. They express the fact that the two paths always have to access the same object. ThingLab implements merges by sharing of objects.

Not all constraints that are used when mapping filter types onto object types can be modelled by the current version of ThingLab. Conditional and iterative constraints are not supported in the current version of ThingLab, so we have provided mechanisms to implement them directly in Smalltalk.

5.3.1. Object Types as Things

This section describes the representation of object types in ThingLab. The creation-time type of an object is modelled as a ThingLab class. Object fields are modelled as instance variables, parts, of a Thing. Constraints and merges can be defined on parts of a Thing. If a Thing has superclasses, then it inherits their parts, constraints and merges. For example, consider the `SetExample` Thing in Figure 5.5. It lists the fields of the object type (see Figure 5.2) as parts. Our merge constraints are expressed directly as merges for ThingLab classes.

Iterated fields are not available for object types but could be simulated by chosing an appropriate class for parts, e.g., class "Array." A conditional field can be modelled by a part that exists but contains only a valid value if the condition is true. ThingLab supports recursive Things, but because it requires a prototype for

```
Class SetExample
        Superclass
                FilterPackThing
        Parts
                source: an ArrayTwo
                view: a ListTwo
                subfilter1: an IntegerIdentity
                subfilter2: an IterationExample
        Merges
                subfilter1 source == source label
                subfilter1 view == view label
                subfilter2 source == source dependents
                subfilter2 view == view dependents
```

Figure 5.5: Thing Definition for SetExample.

each Thing, the depth of the recursion is always explicit. For example, it is possible to define a tree Thing, but since its type is defined by the prototype it will be a tree that has a specific height (0, 1, 2, ...).

Since type information is inferred from the prototype, there is another problem: if the prototype stores NIL in an instance variable, it means that anything can be stored in it, i.e., it is of type "Object," which has all possible types as subtypes, whereas the type of NIL is actually "UndefinedObject," which has no subtypes.

Therefore, we cannot use the prototype to check the possible types for source and view objects of a filter. Instead, we store the names of admissible types for source and view in the Smalltalk class definition for a filter type. When a subfilter is inserted into a constructed filter and connected to fields of source, view and variables, we check whether the source and view of the subfilter are of the correct type, i.e., are supertypes of the object types of the fields to which they are to be connected. Smalltalk has an explicit type hierarchy and can list all subtypes of a type (types are classes in Smalltalk). With the explicit type definition for source and view we produce a list of admissible types. We then check whether the type of the field, inferred from the prototype, is an element in the list of admissible types.

The representation of object types as classes in ThingLab is a restricted version of what we defined in Chapter 3. We could not model multiple signatures that describe object types and are only using the explicit hierarchy information provided by Smalltalk to type check the composition of filters.

5.3.2. Filter Types as Things

Filter atoms can be viewed as instances of object types. These object types can be represented by Things as shown in the previous section. In addition to equality and constraint filter atoms we provide implementation filter atoms that model input and output. Input is modelled by filter atoms that sense input from the keyboard, detect the click of a mouse button, or locate the cursor. Output is modelled by filter atoms that render objects onto the display screen. Additional filter atoms can be defined easily in ThingLab to complete the list of interaction primitives [Mallgren 83], such as picking, button selection, valuators and locators.

Filter types can be viewed as object types as described in Section 5.2 and then represented as Things. The types that are allowed for source and view objects are stored explicitly in the class definition for filters as references to Smalltalk classes. When a subfilter is inserted into a filter type in step two of the filter browser, then the types of the source and view objects are checked. The Smalltalk class hierarchy is used to check the admissible types. This mechanism is provided by the `FilterPackThing` class, which is a superclass for all Things that represent filter types. Figure 5.5 shows the `SetExample` Thing with its superclass `FilterPackThing`.

Since ThingLab does not provide a mechanism to handle conditional constraints, we had to implement a mechanism that simulates conditional merges. If a subfilter is inserted in a filter type that uses a conditional subfilter constructor, then code is compiled automatically that controls access to the object that makes up the conditional expression. The compiled code checks the value of the conditional expression whenever the value is accessed, and activates or deactivates the merges accordingly. This mechanism is rather slow since insertion or deletion of a merge causes ThingLab to forget all previously compiled constraint satisfaction methods; these methods have then to be recompiled.

ThingLab provides special mapping mechanisms that can impose a constraint on dynamic objects. Such a mapping mechanism has three parameters: (1) a source object, which is dynamic, i.e., its type is defined with a union of signatures; (2) a view object, which is dynamic also and matches the structure of the source object; (3) a filter type, instances of which are to be created for specific fields in the source and view objects. Such a mapping mechanism really models two constraints: one that enforces corresponding structure of the source and view object; the other that connects the same fields in corresponding elements of the source and view object.

The `TreeMapper`, for example, relates `label` fields in two objects of type `BinaryTree` with a given filter. It takes dynamic changes of either object into account. If a subtree is added or deleted in one tree, then it is also deleted in the other tree and the corresponding filter is added or deleted. Figure 5.6 shows the filter type definition for a tree mapper that relates the integer labels in two binary trees with `IntegerIdentity` filter atoms. This filter type is implemented with a `TreeMapper` where the first and second parameter are the source and view objects, and the third parameter is an `IntegerIdentity` filter atom.

In the current version of ThingLab, only `TreeMappers` are provided, but other mapping mechanisms, like reversing a tree or mapping two arrays of dynamic length, could be easily added by writing the appropriate Smalltalk code. We understand that the mapping mechanism limits the expressibility of the filter types. We

```
Object Type BinaryTree
      label → Integer
      left → BinaryTree
      right → BinaryTree
end

Filter Type TreeMapper (source: BinaryTree view: BinaryTree)
make
      condition (source notNil or view notNil)
            IntegerIdentity (source.label, view.label)
            TreeMapper (source.left, view.left)
            TreeMapper (source.right, view.right)
end
```

Figure 5.6: Filter Type for TreeMapper.

have a version of the filter browser that can construct iterated subfilters in step two, but it cannot proceed to step three to instantiate the filter, since ThingLab does not support dynamically adding or removing parts. Instead, our current version of the filter browser uses the mapping mechanism. If the iteration subfilter constructor is selected in step two (labelled "mapper" in this version), then the inserted subfilter is imbedded in a mapping mechanism according to the type of the source and view objects.

5.3.3. Manipulating Filter Types

The filter browser is an interface to the objects and Things that represent filter types. It can create a filter type with its source and view object types, insert variables and subfilters, and instantiate filter types to yield a test interface. The filter browser is written in Smalltalk-80 using its Model-View-Controller paradigm for user interfaces. The MVC paradigm has many problems, the biggest of which is its lack of documentation. Other problems include [Deutsch 86]: complicated display updating; no strict separation of model, view and controller; insufficient coordinate transformations and event handling. Smalltalk, however, guided the modularization of our implementation into small pieces of code that are easy to maintain. Much of previously written code from the Smalltalk or ThingLab browser was reused. Slight disadvantages of Smalltalk are that (1) the dependencies among program components are not always apparent, and (2) that system code that has been changed by a user is not distinguishable from original system code. The second point is a problem, since the user can with an inadvertent change made to system code cause the Smalltalk system to behave totally different, but he and others are not easily able to locate that change.

The filter browser is described in Chapter 4. Step one creates a Thing class, a subclass of FilterPackThing, and stores the source and view types in class variables. When creating a class in ThingLab the prototype is also created. Step two inserts fields in ThingLab field descriptions for the subfilters and variables. Fields that are added to the class definition are also added to the prototype. The field descriptions have been subclassed with special descriptors for variables and subfilters to distinguish them from parts. When a subfilter is inserted, the type checking is done according to the subfilter's source and view types and the prototypes of the fields that it is inserted in. If the type checking is successful, then the merges are defined for the Things that represent the filter type. For the "sequence" and "iteration" ("mapper") subfilter constructors, the merges are inserted directly into the definition; for the "condition" constructor, code is compiled that will insert them at run-time.

Step three works on prototypes of the selected filter type and its source and view object types. In step one and two the filter browser builds the prototype for the Thing that represents the filter type. Special implementation filter atoms, renderers and sensors are used to provide visualization and modification of a filter. Renderer filter atoms are defined with a `Bitmap` as the view object type. Sensor filter atoms are defined with an `InputMedium` as the source object type. When the filter browser switches to step three, it examines all subfilters of the selected filter and detects all renderer and sensor filter atoms. The left pane of the filter browser is used to simulate an instance of the `Bitmap` object type; the renderer filter atoms display their source there. The mouse and keyboard of the workstation are used to simulate an instance of `InputMedium`. If the user modifies his input medium, e.g., pushes a button on the mouse, the sensor filter atom changes its view field, thus invoking the constraint-satisfaction mechanism. After the constraints are satisfied the renderer filter atoms examine their source objects again and display them in the left pane of the filter browser to show their new values.

The right pane of the filter browser gives access to source, view and variable parts within the selected filter type. Any part can be selected and inspected. The inspection of a part opens a pane that shows the current value of the part. The value can be changed, possibly invoking constraint-satisfaction. After the constraints are satisfied, the renderer filter atoms examine their sources again and update the left pane of the filter browser.

5.3.4. ThingLab Constraint Satisfaction

Each instance of a Thing obeys the constraints that are defined for it. If a part within a Thing is changed then ThingLab compiles and executes a Smalltalk method that enforces the constraints. This satisfaction method is saved and run immediately whenever the same part is updated again. After the structure of a Thing has been changed, i.e., parts, constraints or merges are added or deleted, all previously defined satisfaction methods are deleted. This "compile-on-need" mechanism leads to a rather slow behavior of the filter browser since it interactively adds or deletes variables or subfilters within the filter type, causing any satisfaction method to be discarded at each change.

The hand-coded conditional constraint causes the satisfaction methods to be recompiled because it adds and removes merge constraints. Future versions of ThingLab should address this problem, maybe by examining dependencies between constrained objects to determine what satisfaction methods need not be recompiled when the structure of a Thing is changed.

5.4. Comparison: First and Second Implementation

Section 5.1 described our initial implementation. Our initial implementation represented filter and object types as classes in Smalltalk. It did not provide a graphical tool to manipulate this representation. Our main implementation represented filter and object types in ThingLab and provided a graphical tool, the filter browser, to build and maintain filter types. Both implementations represented and maintained constraints. This section tries to highlight their differing features.

The first implementation represented object types as Smalltalk classes without constraints. Therefore objects could not have creation-time constraints enforced directly. Filter types defined constraints on source and view objects by providing update methods in case one of them was changed. Therefore to only way a constraint could be enforced on an object was that it had to occupy the source or view slot of a filter or be a subpart of a source or view object.

Access control was done with object holders. Thus, changes to objects could come from anywhere, and did not have to expressed as path down from the top-level filter. This implementation could deal with any sharing of objects as values of multiple fields. Filter types did not have to define merge constraints, since the source and view fields of subfilters were connected explicitly to existing objects when those filters were created.

An object could have imposed constraints if it was part of a filter. Constraint-satisfaction was done locally for each filter and the only global constraint-satisfaction was done by propagation of values throughout a network of connected filters.

A positive aspect of this implementation was that it could handle conditional and iterative subfilters with dynamic creation and deletion of subfilters, since the local constraint-satisfaction methods of the filters encoded this behavior explicitly. A major drawback of this implementation was the lack of global planning for the constraint-satisfaction, so that certain networks of filters could not be satisfied, even so there existed a solution. Also, the filter types did not keep any type information about their source and view objects; therefore, there was no type checking. The constraint-satisfaction procedure was slow, since only the local satisfaction methods were compiled, while the propagation of values was achieved with dependency chains kept by the object holders.

The ThingLab implementation represents object and filter types as Things, thus allowing creation-time constraints for objects. All constraints, creation-time and imposed, are satisfied by a by a global constraint-satisfaction mechanism that is very powerful. Sharing of objects as values for multiple fields have to be expressed with merge constraints. The merge constraints to connect subfilters are maintained by ThingLab and are used during constraint-satisfaction planning. Changes to objects have to be expressed via access paths. Constraints are satisfied for all objects that are reachable from where the access path starts. Our implementation relied on the fact that all constraints in a constructed filter are local to the filter, and that all changes are expressed with access paths that start at the top-level filter. ThingLab's approach to constraint-satisfaction is global and monolithic, i.e., for a given change it compiles all necessary resulting changes at once into satisfaction methods. Once a satisfaction method is compiled, then the constraint-satisfaction is very fast.

A drawback of the second implementation is that ThingLab does not support dynamic types and constraints for conditional and iterated fields in objects and filters. Our environment, the filter browser, managed to provide reasonable substitutes.

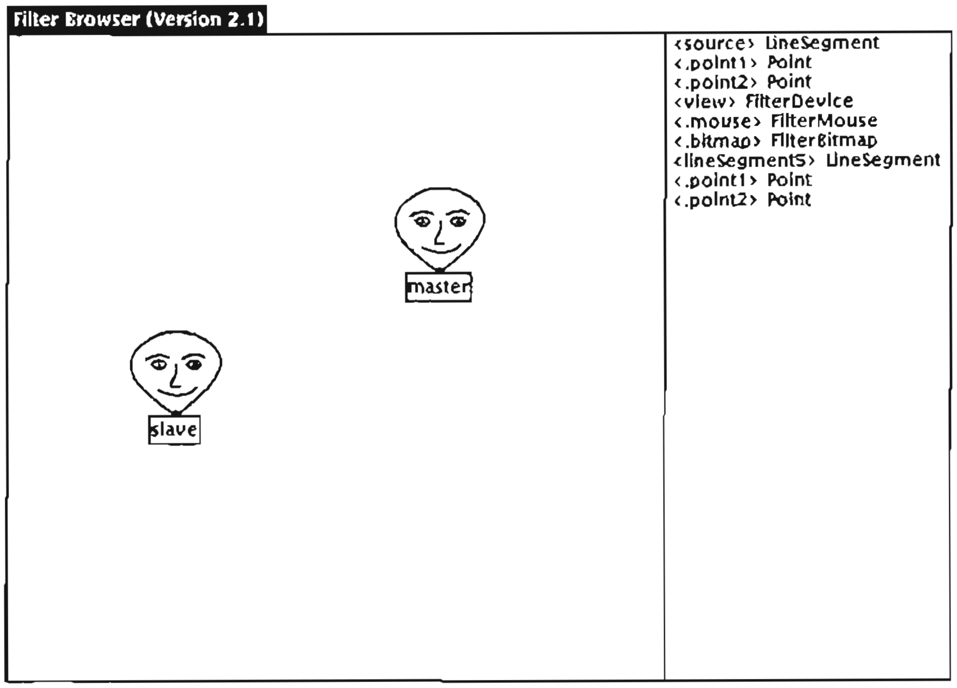
Summarizing, we can say that the first implementation lacked a global strategy to constraint-satisfaction and a notion of type. It provided dynamic constraints and showed that they are feasible. The second implementation lacks dynamic types and constraints, but provided a very powerful global constraint-satisfaction. The second implementation also provided a graphical tool to specify and automatically generate interfaces. Some of the interfaces produced with the second implementation are described in the next section.

5.5. Filters Defined With the Filter Browser

The following sections describe examples that have been defined with filter types and the filter browser. They highlight different features of the filter paradigm, such as dynamic behavior, input sensors and output renderers, the mapping mechanism, the independence of interface and application, and the ability of the filter paradigm to define complex interfaces, such as the filter browser itself. The appendix lists the complete filter and object type definitions for all examples.

5.5.1. Master-Slave Filter

The Master-Slave filter demonstrates the use of the condition subfilter constructor and the filter atoms for sensing input and rendering output to the display bitmap. Figure 5.7 shows the filter browser in step three. The left pane displays two icons on the screen that are labelled with the text "Master" and "Slave". The possible labels on each are "Master", "Slave" or "Colleague". The two icons can be moved with the mouse. The icon labels change according to the relative position of



```
<source> LineSegment
<.point1> Point
<.point2> Point
<view> FilterDevice
<.mouse> FilterMouse
<.bitmap> FilterBitmap
<lineSegments> LineSegment
<.point1> Point
<.point2> Point
```

Figure 5.7: Master - Slave Interface.

the two icons. The higher of the two icons displays "Master", the icon below "Slave". If both icons are at the same level, then the label reads "Colleague" for both of them.

We represent a person with an object of type `LineSegment` where the first point in the line models the location where the person will be displayed, and the second point models the location of his colleague. Figure 5.8 shows the object and filter type definitions. There are three filter types: `MasterSlave`, `PersonAtPoint` and `Relativity`. The `MasterSlave` filter is defined for source objects of type `LineSegment` and view objects of type `Text`. It uses three

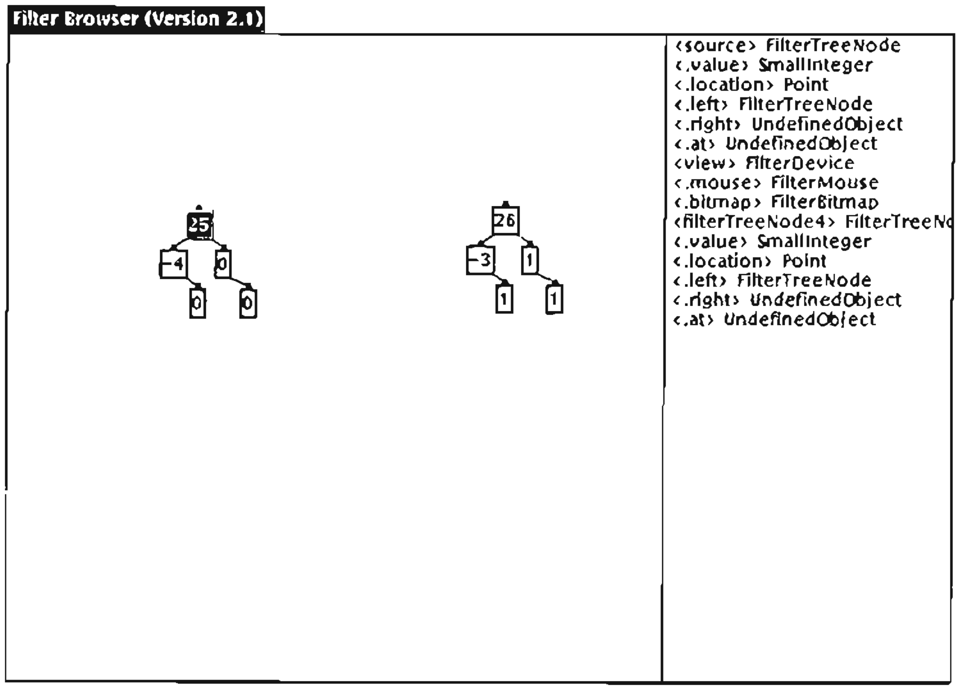
```
Object Type CenteredText                 Object Type PersonIcon
      text → Text                              icon → Form
      location → Point                         location → Point
end                                      end

Filter Type MasterSlave (source: LineSegment, view: Text)
make
      (source.point1 > source.point2)
            TextEquality ( "Master", view)
      (source.point1 = source.point2)
            TextEquality ( "Colleague", view)
      (source.point1 < source.point2)
            TextEquality ( "Slave", view)
end

Filter Type PersonAtPoint (source: LineSegment, view: FilterDevice)
var
      ct → CenteredText
      pi → PersonIcon
make
      PointEquality (source.point1, ct.location)
      PointEquality (source.point1, pi.location)
      MasterSlave (source, ct.text)
      PointSensor (source.point1, view.mouse)
      Renderer (ct, view.bitmap)
      Renderer (pi, view.bitmap)
end

Filter Type Colleagues (source: LineSegment, view: FilterDevice)
var
      ls → LineSegment
make
      PersonAtPoint (source, view)
      PersonAtPoint (ls, view)
      PointEquality (source.point1, ls.point2)
      PointEquality (source.point2, ls.point1)
end
```

Figure 5.8: Master - Slave Example.

condition constructors to set the view of type Text to either "Master", "Colleague" or "Slave". Only one of the `TextEquality` subfilters is ever active, since the conditions are mutually exclusive. The PersonAtPoint filter is defined for source objects of type LineSegment and view objects of type FilterDevice. It declares variable ct of type CenteredText and variable pi of type PersonIcon. The PersonAtPoint filter uses the PointSensor filter atom to set the first point in the source LineSegment, which is the location where the person will be displayed, to the location of the mouse. This filter also uses Renderer filter atoms to copy the centered text and the person icon into the display bitmap, and a MasterSlave subfilter to select the text for the icon label. The PersonAtPoint filter is used to define the Relativity filter for two persons, each represented as a LineSegment. The Relativity filter also defines PointIdentity subfilters to identify one persons location as the location of the colleague.

5.5.2. Tree Manipulation Filter

The "Tree Manipulation" filter shows the dynamic behavior that can be achieved with filters. Two binary trees with integer node labels are displayed on the screen. Figure 5.9 shows the filter browser in step three displaying the two binary trees. There is a filter defined between the two trees that expresses the constraints that both trees have the same structure and that the integer values for corresponding labels differ by 1. For example, if the root node of the first tree holds the integer 25 then the root node of the second tree holds the integer 26. This Add1Filter is mapped over the tree using the TreeMapper construct in ThingLab. Both trees can be moved on the screen by selecting a new location for the root node. The left tree can be traversed. Initially the top node of the left tree is selected. A pop-up menu is available that select one of the following actions for



```
<source> FilterTreeNode
<.value> SmallInteger
<.location> Point
<.left> FilterTreeNode
<.right> UndefinedObject
<.at> UndefinedObject
<view> FilterDevice
<.mouse> FilterMouse
<.bitmap> FilterBitmap
<filterTreeNode4> FilterTreeN
<.value> SmallInteger
<.location> Point
<.left> FilterTreeNode
<.right> UndefinedObject
<.at> UndefinedObject
```

Figure 5.9: Tree Manipulation Filter.

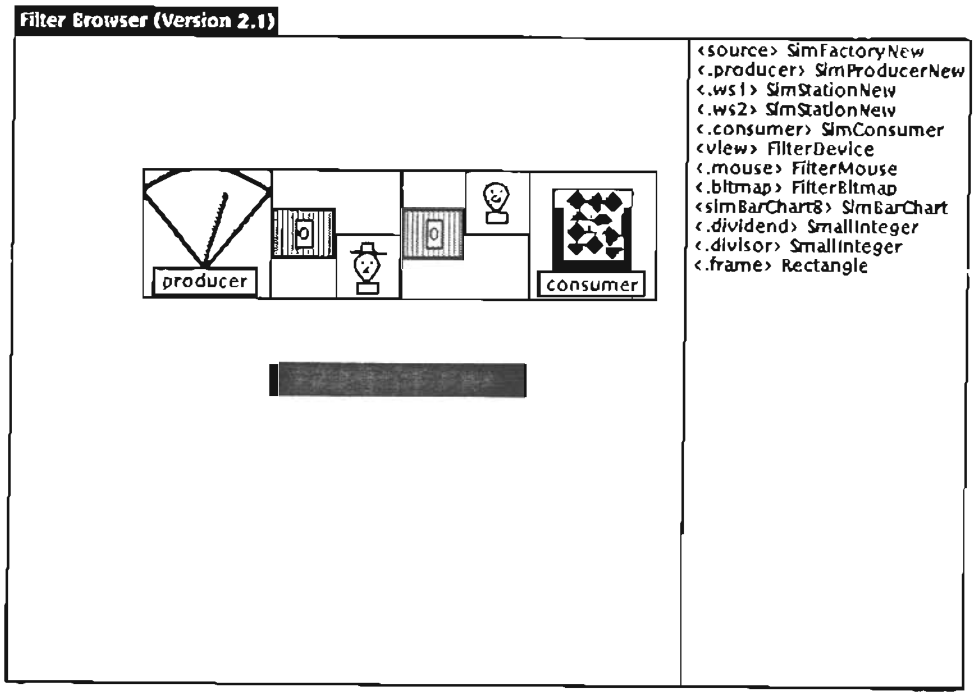
the selected node:

- decrease value of node by 1,
- delete or create left and right subtrees,
- select father node, left or right subtree.

As values within the left tree are changed, the right tree is also updated. As subtrees are added to or deleted from the left tree the right tree is also changed accordingly. The right pane of the filter browser in Figure 5.9 displays the parts of the Thing that represents the filter type. All parts can be inspected and changed, thus changing the display of the two trees in the left pane. The right tree can only be changed in the right pane of the filter browser.

5.5.3. Factory Simulation Filter

This filter has already been described in Chapter 2. It visualizes the simulation of some workstations in a factory. This interface demonstrates that a filter can be "plugged" onto existing data structures without disturbing the application. This simulation application, however, is imbedded into the ThingLab class hierarchy in order to be able to define constraints for the simulation objects. Figure 5.10 shows the filter browser in step three, displaying in the left pane the producer, the two workstations and the consumer. The right pane lists the participating objects



```
<source> SimFactoryNew
<.producer> SimProducerNew
<.ws1> SimStationNew
<.ws2> SimStationNew
<.consumer> SimConsumer
<view> FilterDevice
<.mouse> FilterMouse
<.bitmap> FilterBitmap
<simBarChart8> SimBarChart
<.dividend> SmallInteger
<.divisor> SmallInteger
<.frame> Rectangle
```

Figure 5.10: Factory Simulation Filter.

that can be inspected and changed.

5.5.4. Filter Browser as Filter Type

The filter browser is a user interface itself. It provides access to the Smalltalk class hierarchy that models the filter types through the input and output media of a graphical workstation. The application objects are the list of all filter types and the list of all object types in the system. The presentation objects are the appearance of these types on the workstation in the three steps. The filter browser can be described by a filter, making it an instance of a filter type. This filter type is defined to accept the Smalltalk and ThingLab class hierarchies as source and the workstation with its input and output parts as view.

```
Object Type FilterMetaType
      filterName → String
      sourceType → Class
      viewType → Class
end

Object Type FilterAtomMetaType
      inherit from FilterMetaType
end

Object Type FilterPackMetaType
      inherit from FilterMetaType
      variables [] → FieldDescription
      subfilters [] → SubfilterDescription
end
```

Figure 5.11: Filter Meta Types.

Figure 5.11 shows some of the object types that have to be defined for the filter type hierarchy as it is constructed by the filter browser. We included "meta" in their object type names to distinguish them from the classes that they describe. Newly defined filter types are instances of the FilterPackMetaType, which is a subtype of the FilterMetaType. FilterMetaType is the object type for all filter types. The FilterPackMetaType has fields for variable and subfilter descriptions. The "[]" notation is a shorthand for an iterated field with variable length. The FilterPackMetaType inherits the fields filterName, sourceType and viewType from its supertype. These fields store the name of the filter type and its admissible source and view object types. These fields are also inherited by the FilterAtomMetaType for the meta type of filter atoms. Any filter type that is a filter atom will be a subtype of FilterAtomMetaType and will add constraints that describe its atomic behavior.

Figure 5.12 shows the field descriptions for filter types. FieldDescription is the common supertype, which is defined and used in ThingLab. SequenceConstructor, ConditionConstructor and IterationConstructor are subclasses of SubfilterDescription and their instances can be inserted in the subfilters field of instances of type FilterPackMetaType. The SequenceConstructor type is only used to distinguish filter field descriptions from ThingLab field descriptions. The types for the field descriptions declare fields to hold the merge constraints that are of type AccessPath. An AccessPath is a list of field names. The merge constraints for a sequence constructor identify an access path for the source and for the view field of the subfilter. For condition and

```
Object Type FieldDescription
      fieldName → String
      fieldClass → Class
end

Object Type SubfilterDescription
      inherit from FieldDescription
end

Object Type SequenceConstructor
      inherit from SubfilterDescription
      sourceMerge [] → AccessPath
      viewMerge [] → AccessPath
end

Object Type ConditionConstructor
      inherit from SubfilterDescription
      sourceMerge [] → AccessPath
      viewMerge [] → AccessPath
      conditonMerge [] → AccessPath
end

Object Type IterationConstructor
      inherit from SubfilterDescription
      sourceMerge [] → AccessPath
      viewMerge [] → AccessPath
      factorMerge [] → AccessPath
end

Object Type AccessPath
      fields [] → String
end
```

Figure 5.12: Filter Constructor Types.

```
Filter Type FilterBrowser ( source: ClassList, view: FilterDevice)
var
      filterType → FilterPackMetaType
      step → Integer
      selection → Integer
make
      PopUpMenu (step, 'Step 1/Step 2/Step 3')
      (step = 1)
            PopUpMenu (selection, 'create/select')
            (selection = 1)
                  NewFilter (filterType, view)
                  AddToList (source.filters.filterPacks, filterType)
            (selection = 2)
                  SelectFilter (filterType, view)
            ModifyTypes ((source, filterType), view)
      (step = 2 & filterType notNil)
            ModifyFilter ((source, filterType), view)
      (step = 3 & filterType notNil)
            InstantiateFilter (filterType, view)
end
```

Figure 5.13: FilterBrowser Filter Type.

iteration constructors there is an additional field to identify the condition or iteration factor.

Figure 5.13 shows the definition of the `FilterBrowser` filter type. It is defined for a source object of type `ClassList`, which lists all existing object and filter types, and a view object of type `FilterDevice`, which represents a workstation. Using a pop-up menu, one of the three steps of the filter browser can be selected. A pop-up menu filter is a implementation filter atom that presents a menu to the user when a mouse button is pressed. It constrains its source object to reflect the selection from the list given by its view object. In step 1, either a new filter is created or an existing filter is selected and unified with the `filterType` variable. Step 1 can also change the admissible source and view object types for the selected filter. Step 2 modifies the selected filter by adding or removing variables or subfilter. Step 3 instantiates the selected filter.

As the filter type definition for the filter browser shows, we assume the existence of many subfilters. These subfilters have to capture the behavior of the filter browser, e.g., how it creates and modifies objects. Many of these subfilters have to be implemented as filter atoms. Since the current implementation of ThingLab has some performance problems, the filter browser was not re-implemented in itself. The description here served to show that we can build filter types that model such a complex interface as the filter browser. The appendix contains more filter and object types for this filter type.

Epilogue

6.1. Summary

This thesis explored a new approach to constructing interfaces. It introduced the filter paradigm, which uses the concepts of object, constraint and filter as building blocks. An object and filter model was defined that gives semantics to our specification language for objects and filters, and that provides a terminology for declaratively constructing structured interfaces. The language distinguishes object and filter types, but for the implementation we showed how to represent filter types as special cases of object types. Object types were implemented in an object-oriented environment that supports constraints. Object types that represent filter types can be manipulated with a graphical tool, the filter browser, which is an interface to these special object types. Several examples were built using the filter browser, showing the feasibility of this new approach to building interfaces.

6.2. Conclusion

Our filter and object model that incorporates constraints is very powerful. It was shown that it is expressive enough to declaratively specify interfaces and hide the complex procedurality found in user interfaces in constraint-satisfaction. Interfaces are of dynamic nature: this has to be supported by the constraint system. Our first implementation attempted to do the constraint-satisfaction directly in Smalltalk but was limited in the strategies used for constraint-satisfaction. We soon learned that a more complex solution was necessary.

Our current implementation uses ThingLab as an extension to Smalltalk. ThingLab is an elegant system that has a powerful constraint-satisfaction mechanism. However, there were some drawbacks:

(1) ThingLab has a more limited notion of type than our filter paradigm. The type information in ThingLab is represented by prototypes, which are static instances and which cannot express type constructions such as condition, dynamic iteration and recursion. Therefore, object types are limited to only one signature for describing the set of all its instances. Filter types thus implemented are therefore limited to only one configuration of the constraints they represent. These limitations are the reason why our implementation using ThingLab cannot express all the dynamic features of the filter paradigm.

(2) ThingLab compiles satisfaction methods on demand, i.e., when they are invoked. This mechanism is slow. When the structure of the object types changes, which is common when using the filter browser, these compiled satisfaction methods have to be recompiled. Dynamic constraints are not handled

directly, which leads to handcoded additions that slow down the constraint satisfaction even more.

(3) Each atomic constraint that is defined for ThingLab has to give its satisfaction methods. ThingLab uses these method fragments to compile satisfaction code for constraints that are build from these atomic constraints. ThingLab uses some heuristics to deduce dependencies between different parts of an object for which the constraints are defined. These dependencies are important when filter atoms are defined. However, these dependencies are not stated explicitly and require knowledge of the ThingLab implementation.

Based on our experience with the current version of ThingLab we propose the following extensions to ThingLab:

(1) A facility to define iterated and structured parts, where the type information is stored explicitly.

(2) Constraints that can be placed on a dynamic collection of instances and constraints that are satisfied depending on the dynamic value.

(3) Reuse of previously compiled satisfaction methods that are still valid after the structure of a Thing has been changed.

Overall, our work has shown that constraints can be used to define interfaces. The implementation of the filter paradigm showed the limits in our constraint-satisfaction techniques. The next section will discuss whether the filter paradigm could serve as the general interface mechanism in an object-oriented system.

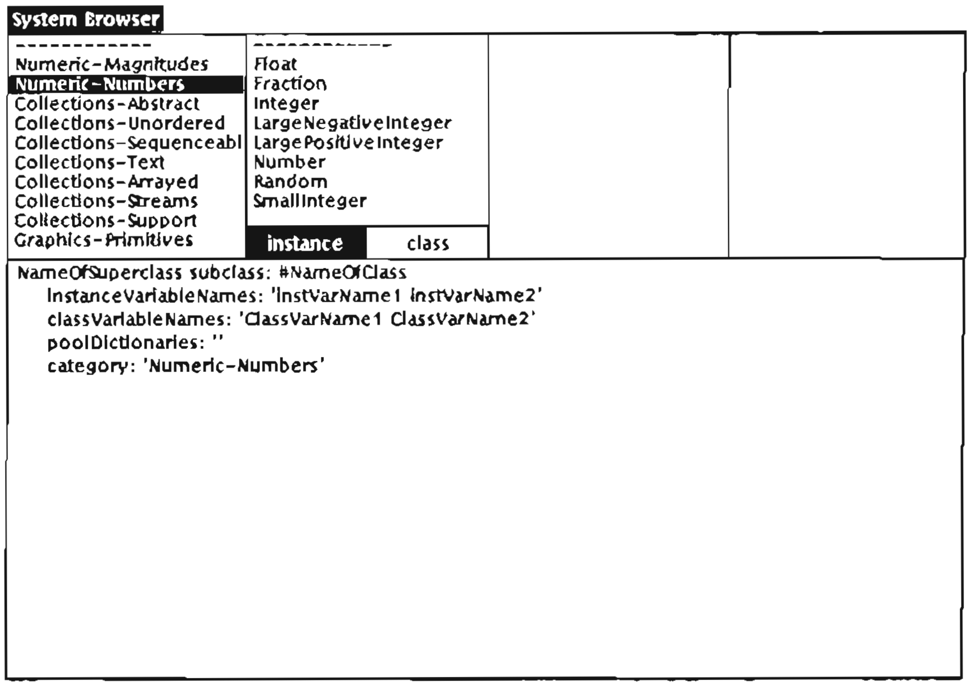
6.2.1. The Filter: A Paradigm for Interfaces ?

We believe that our filter paradigm is powerful enough to express complex interfaces. The specification of the filter browser interface as a filter type showed that an interface can be decomposed using the filter paradigm, but that some of the procedurality of the interface has to be captured by filter atoms. What we will discuss in this section is whether the filter paradigm could replace an existing interface paradigm, such as the Smalltalk Model-View-Controller paradigm [Goldberg and Robson 83]. The Smalltalk class browser [Goldberg 84] will serve as an example of an interface that is implemented according to the MVC paradigm.

The MVC paradigm represents interfaces as a triple of a model, a view and a controller. The model represents the application, the view the presentation, and the controller the dialog control component. Communication between them is done with certain standard Smalltalk messages, such as "update" that is send by the controller to the view in order to redisplay it, or "aspect" that is sent by the view to the model in order to retrieve to current model data. The kind of messages that are sent depends on the nature of the interface that is modelled with an MVC triple.

An interface is represented as one top-level MVC triple, which is decomposed into sub-triples. All these MVC triples build a hierarchy where the leaf nodes implement the specific behavior of the interface. There is a collection of basic triples available in the Smalltalk environment, e.g., for views that display text, views that display lists, or views that detect cursor clicks for switches.

An example of an interface that is built according to the MVC paradigm, let us consider the Smalltalk class browser. It represents an interface to all classes in Smalltalk. Its presentation on the screen, the view, shows different levels of details about the classes: the classes are organized in a tree structure; classes that are related belong to a category; and classes have a collection of methods that are grouped into protocols. Figure 6.1 shows the Smalltalk class browser view. It is decomposed into five panes. The top left pane lists all categories of classes. A category can be selected and is then highlighted. In Figure 6.1 the category "Numeric-Numbers" is selected. The next pane to the left shows the classes for the

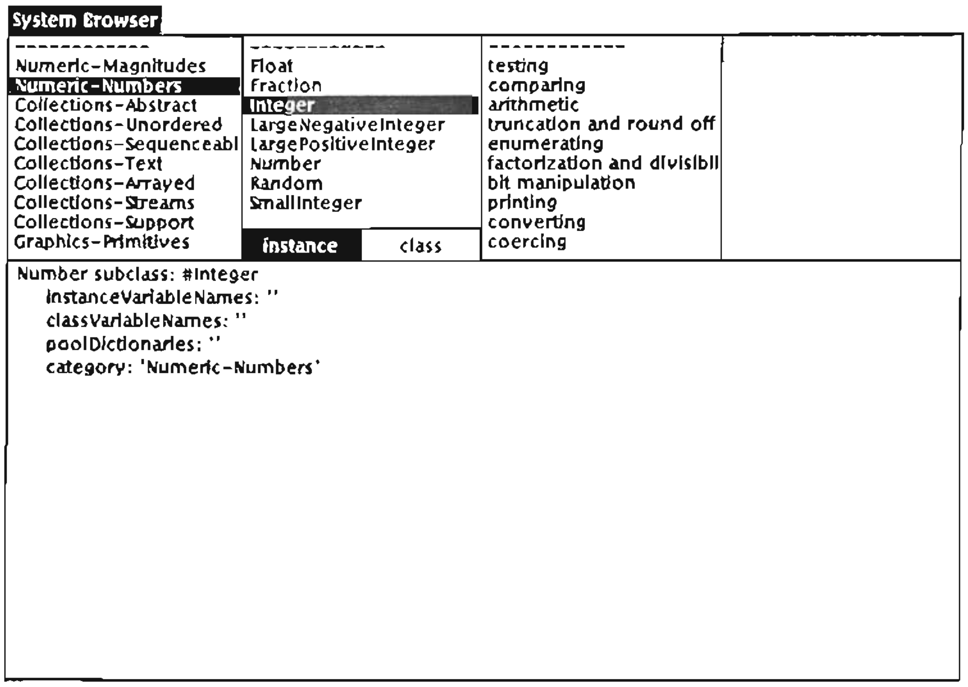


```
NameOfSuperclass subclass: #NameOfClass
    instanceVariableNames: 'instVarName1 instVarName2'
    classVariableNames: 'ClassVarName1 ClassVarName2'
    poolDictionaries: ''
    category: 'Numeric-Numbers'
```

Figure 6.1: Smalltalk Class Browser.

selected category, which, in this case, are all classes that are related to numbers. The other three panes display their default content, which is either blank or a template for a class definition, as in the bottom pane.

The Smalltalk class browser has a model that represents the organization of all Smalltalk classes. The controller handles the input from the user and communicates with model and view by sending messages to them. The top-level MVC triple is decomposed into triples for the five panes of the browser presentation. The model of the second pane is the list of classes within the selected category and an indicator



```
Number subclass: #Integer
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Numeric-Numbers'
```

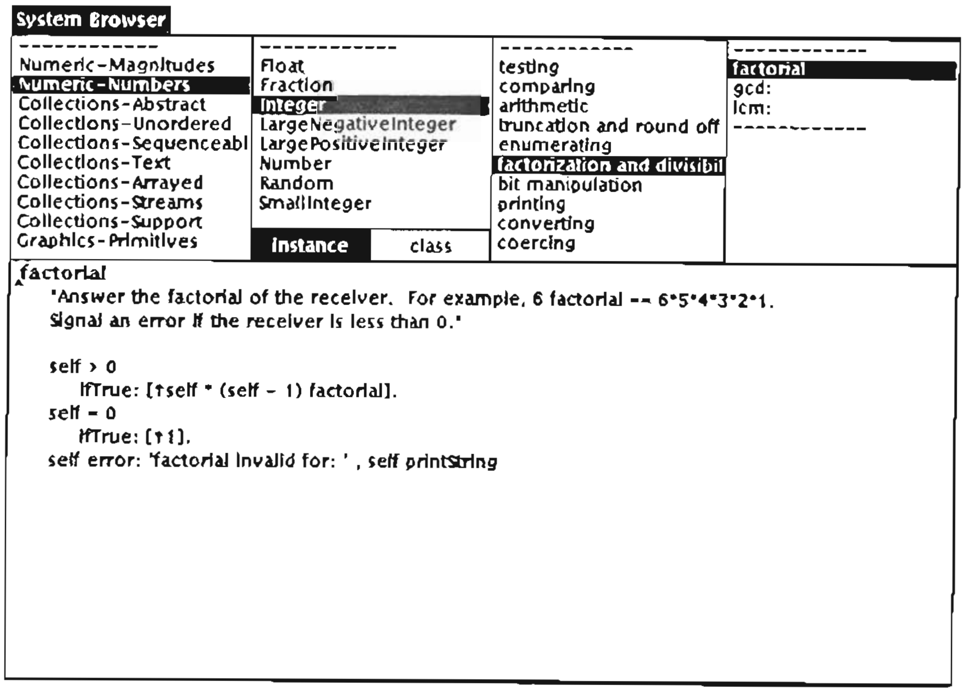
Figure 6.2: Class Selection.

that tells which element in the list is selected. Initially there is no selection made. If the user wishes to select a class within this list, he places the cursor on the appropriate line and clicks a mouse button. The controller detects this input and sends a message to the model indicating the user's selection. The model then changes its selection indicator and notifies its dependents, e.g., its view, that it has changed. The view then reexamines the model and redisplays the pane.

Figure 6.2 shows that the user has selected the "Integer" class. The selection of the class caused changes in other parts of the interface. The third pane in the top row now shows all the different protocols that are defined for the "Integer" class. There are dependencies between different panes in the Smalltalk browser that are maintained explicitly by the model by notifying depending views after a change occurred.

Figure 6.3 shows the class browser where a method has been selected. The bottom pane now displays the code for the selected method. The code can be changed by the user, which causes the invocation of the Smalltalk compiler after the user has accepted his changes. The compiler is not really part of the interface, but could be considered as part of the application that the Smalltalk class browser represents, i.e., defining classes.

If we want to represent such an interface with the filter paradigm we have to consider these four aspects: (1) the decomposition of the interface into application, control and presentation; (2) the decomposition of the interface into subcomponents, e.g., for the five different panes; (3) the dependencies between the different components; and (4) the application behavior that is part of the interface. The



```
factorial
    "Answer the factorial of the receiver.  For example, 6 factorial -- 6*5*4*3*2*1.
    Signal an error if the receiver is less than 0."

    self > 0
        ifTrue: [↑self * (self - 1) factorial].
    self = 0
        ifTrue: [↑1].
    self error: 'factorial invalid for: ' , self printString
```

Figure 6.3: Method Definition.

following four paragraphs will address each of the four aspects.

(1) The filter paradigm represents the application (source), the control (filter), and the presentation (view) as objects. The Smalltalk environment does not really provide all aspects of the model as object, but rather calculates them as they are requested by the controller or view. If we want to represent the Smalltalk browser with filters, we would have to provide all aspects of the model as objects. Smalltalk also provides a set of view classes that handle aspects such as coordinate transformations or highlighting of certain areas within the view automatically. For the filter paradigm we could provide a similar set of

predefined object types where constraints could define aspects such as coordinate transformations and highlighting.

(2) Filters can be constructed from subfilters to create a hierarchy. Leaf nodes in this hierarchy are filter atoms that implement the specific behavior of the interface. Some complex input and output behavior can be further decomposed into subfilters as was shown by the simulation interface example in Chapter 2. However, basic input/output sequences have to be modelled as filter atoms. For example, consider a user who defines a new protocol for a class with the Smalltalk browser. He selects the "add protocol" option from a pop-up menu; the browser then prompts him for the name of the new protocol; the user then types in the name and accepts the new name by typing a carriage return character; the browser then adds the new protocol to the list of defined protocols. Such a behavior cannot be modelled by further decomposing filters and has to be provided as a filter atom.

(3) In the filter paradigm source and view components of the interface do not have to specify their dependencies explicitly. The dependencies are captured by constraints that are maintained by the constraint-satisfaction mechanism. For example, if the selection indicator for the class list changes, then the filter that is responsible for displaying the protocol of classes will detect a change in its source object and will redisplay the correct list of protocols. Filters declare the dependencies explicitly with constraints, therefore, the source component does not have to anticipate its view as has to be done in the MVC paradigm.

(4) Specific behavior of an application should be factored out as much as possible from the interface. However, the invocation of application specific functions is part of an interface. The filter paradigm does not handle such procedural behavior, since the constraints defined with filters can change only the state of the source component, but not invoke procedures defined for the source component. However, the filter paradigm offers the possibility to express application behavior with constraints. It is conceivable to represent the Smalltalk compiler as a constraint from a text to its machine-code representation. Such constraints are certainly beyond the scope of this research.

In conclusion, we can say the filter paradigm has the potential to be used as the major interface model in an object-oriented system, but that we need an implementation that supports its full expressibility. The current implementation lacks the handling of dynamic lists, which is a major feature of the Smalltalk browser. Also, in order to produce reasonable interfaces it also needs a well though-out set of filter atoms and object types, and an extented filter browser.

6.3. Future Work

Some aspects of this research deserve further investigation. The areas are:

- textual extensions to the filter browser
- graphical extensions to the filter browser
- extension to the object type system
- different systems for constraint-satisfaction

The filter browser only provides one view of a filter type, but it could be extended to include a textual view for the filter specification language. This view could accept the language as input to form the internal object structure and deduce a graphical layout for it. The language could also be decompiled after the internal object structure has been defined graphically in step two of the filter browser. This textual view of the filter type representation is useful to the designer of an interface, because it will give him a more structured view of the interface he is designing. In addition, this textual view could be combined with a compiler that would optimize the filter type representation.

Graphical extensions to the filter browser in step two are necessary. The network of subfilters and variables becomes obscure as more elements are added. Techniques that help manage the semantic information could be developed, such as automatic layout of subfilters and variables, removal of line crossings, panning and zooming of detail and look-up of participating filter and variable definitions. Also, providing nesting for subfilter constructors would help manage the complexity of filter types.

The object type system is implemented in the Smalltalk class hierarchy. The type information for the source and view types of filters are stored explicitly. Other type information is not stored but inferred from sample instances, prototypes. Instead of adding more explicit type information in terms of class references, we would like to represent the typing information as constraints: a field would be constrained to hold only objects of a specific type. The type information and the type checking would then be handled by the constraint system.

Since our filter paradigm is dynamic in nature we will have to explore other constraint systems. There are two relevant approaches published in the literature.

One is based on term-rewriting [Leler 86], the other uses logic programming to solve constraints [Jaffar and Lassez 87]. Both approaches have features to express constraints conditionally.

Bibliography

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

[Albano, Cardelli and Orsini 85]
Albano, A., L. Cardelli and R. Orsini, Galileo: a Strongly Typed, Interactive Conceptual Language, *ACM Transactions on Database Systems 10,* 2 (1985), 230-260.

[Bass 85]
Bass, Leonard J., An Approach to User Specification of Interactive Display Interfaces, *IEEE Transactions on Software Engineering SE-11,* 8 (August 1985), 686-698.

[Beach 84]
Beach, Richard J., Experience with the Cedar Programming Environment for Computer Graphics Research, *Proceedings Graphics Interface'84 Conference,* 1984, 65-74.

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

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

[Borning 81]
Borning, Alan, The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, *ACM Transactions on Programming Languages and Systems 3,* 4 (October 1981), 353-387.

[Borning and Duisberg 86]
Borning, Alan and Robert A. Duisberg, Constraint-Based Tools for Building User Interfaces, *ACM Transactions on Graphics 5,* 4 (October 1986), 345-374.

[Borning 86]
Borning, Alan, Graphically Defining New Building Blocks in ThingLab, *Human-Computer Interaction 2,* 4 (1986), 269-295.

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

[Broverman and Croft 85]
Broverman, Carol A. and W. Bruce Croft, A Knowledge-Based Approach to Data Management for Intelligent User Interfaces, *Proceedings of Conference Very Large Data Bases*, Stockholm, Sweden, 1985, 96-104.

[Buxton et al. 83]
Buxton, W., M.R. Lamb, D. Sherman and K.C. Smith, Towards a Comprehensive User Interface Management System, *ACM Computer Graphics 17*, 3 (July 1983), 35-42.

[Carter and LaLonde 84]
Carter, Christopher A. and Wilf R. LaLonde, The Design of a Program Editor Based on Constraints, Technical Report SCS-Technical Report-50, School of Computer Science, Carleton University, Ottawa, Canada, May 1984.

[Chi 85]
Chi, Uli, Formal Specification of User Interfaces: a Comparison and Evaluation of four axiomatic Methods, *IEEE Transactions on Software Engineering SE-11:8* (August 1985), 671-685.

[Cohen, Smith and Iverson 86]
Cohen, Ellis S., Edward T. Smith and Lee A. Iverson, Constraint-Based Tiled Windows, *IEEE Computer Graphics and Applications*, May 1986.

[Coutaz 85]
Coutaz, Joelle, Abstractions for User Interface Design, *IEEE Computer 18*, 9 (September 1985), 21-34.

[Cox 86]
Cox, Brad J., *Object Oriented Programming - An Evolutionary Approach*, Addison Wesley, Reading, Mass., 1986.

[Dahl and Nygaard 66]
Dahl, O.-J. and K. Nygaard, Simula - An Algol-based Simulation Language, *Communications of the ACM 9*, 9 (September 1966), 671-678.

[Deutsch 86]
Deutsch, L. Peter, Panel: User Interface Frameworks, *OOPSLA '86 Conference Proceedings*, Portland, OR, September 1986.

[Duisberg 86]
Duisberg, Robert A., *Constraint-Based Animation: The Implementation of Temporal Constraints in the Animus System*, Ph.D. Thesis, Department of Computer Science, University of Washington, 1986.

[Ege 84]
Ege, Raimund K., The Display Function in ALLEGRO, Master's Thesis, Oregon State University, July 1984.

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

[Ege 86]
Ege, Raimund K., The Filter - A Paradigm for Interfaces, Technical Report No. CSE-86-011, Oregon Graduate Center, Beaverton, OR, September 1986.

[Ege, Maier and Borning 87]
Ege, Raimund K., David Maier and Alan Borning, The Filter Browser: Defining Interfaces Graphically, *Proceedings European Conference on Object Oriented Programming*, Paris, France, June 1987, 155-165.

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

[Goldberg 84]
Goldberg, Adele, *Smalltalk-80: The Interactive Programming Environment*, Addison Wesley, Reading, MA, 1984.

[Goldberg 85]
Goldberg, Adele, Application Development Framework, Oregon Graduate Center Colloquium, Video Tape, Beaverton, OR, November 13, 1985.

[Granor and Badler 85]
Granor, Tamar Ezekiel and Norman I. Badler, GUIDE: Graphical User Interface Development Environment, Department of Computer and Information Science Technical Report MS-CIS-85-19, University of Pennsylvania, Philadelphia, PA, May 1985.

[Granor 86]
Granor, Tamar Ezekiel, *A User Interface Management System Generator*, Ph.D. Thesis, University of Pennsylvania, Philadelphia, PA, May 1986.

[Green 85]
Green, Mark, Report on Dialog Specification Tools, in *Workshop on User Interface Management Systems (1983: Seeheim-Jugenheim, Germany)*, Guenther E. Pfaff (ed.), Springer Verlag, Berlin, 1985, 9-20.

[Greenberg, Peterson and Witten 86]
Greenberg, Saul, Murray Peterson and Ian Witten, Issues and Experiences in the Design of a Window Management System, *Proceedings of the Canadian Information Processing Society National Conference*, Edmonton, 1986.

[Grossman and Ege 87]
Grossman, Mark and Raimund K. Ege, Logical Composition of Object-Oriented Interfaces, *OOPSLA '87 Conference Proceedings*, Orlando, FL, October 1987.

[Hirsch et al. 86]
Hirsch, Peter M., William Katke, Michael Meier, Steven Snyder and Richard E. Stillman, Interfaces for Knowledge-base Builders' Control Knowledge and Application-specific Procedures, *IBM Journal of Research and Development 30*, 1 (January 1986), 29-38.

[Hudson and King 86]
Hudson, Scott E. and Roger King, A Generator of Direct Manipulation Office Systems, *ACM Transactions on Office Information Systems 4*, 2 (April 1986), 132-163.

[Jacob 85]
Jacob, Robert J.K., A State Transition Diagram Language for Visual

Programming, *IEEE Computer 18*, 8 (August 1985), 51-59.

[Jaffar and Lassez 87]
Jaffar, Joxan and Lean-Louis Lassez, Constraint Logic Programming, *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, 1987.

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

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

[London and Duisberg 85]
London, Ralph L. and Robert A. Duisberg, Animating Programs Using Smalltalk, *IEEE Computer 18*, 8 (August 1985), 61-71.

[Maier, Nordquist and Grossman 86]
Maier, David, Peter Nordquist and Mark Grossman, Displaying Database Objects, *Proceedings First Int. Conference on Expert Database Systems*, Charleston, South Carolina, April 1986.

[Maleki 87]
Maleki, Jalal, VIVID, The Kernel of a Knowledge Representation Environment Based on the Constraints Paradigm of Computation, *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences 1*(January 1987), 591-600.

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

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

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

[Moriconi and Hare 85]
Moriconi, Mark and Dwight F. Hare, Visualizing Program Designs Through PegaSys, *IEEE Computer 18*, 8 (August 1985), 72-85.

[Musen, Fagan and Shortliffe 86]
Musen, Mark A., Lawrence M. Fagan and Edward H. Shortliffe, Graphical Specification of Procedural Knowledge for an Expert System, *Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages*, Dallas, Texas, Juli 1986, 167-178.

[Myers 83]
Myers, Brad A., INCENSE: A System for Displaying Data Structures,

Computer Graphics 17(3) (July 1983), 115-125.

[Myers 84]
Myers, Brad A., Strategies for Creating an Easy to Use Window Manager with Icons, *Proceedings Graphics Interface'84 Conference*, 1984, 227-233.

[Myers and Buxton 86]
Myers, Brad A. and William Buxton, Creating Highly-Interactive and Graphical User Interfaces by Demonstration, *ACM SIGGRAPH'86 Conference Proceedings*, Dallas, Texas, August 1986, 249-258.

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

[Olsen and Dempsey 83]
Olsen, Dan R. and Elizabeth P. Dempsey, SYNGRAPH: A Graphical User Interface Generator, *ACM SIGGRAPH Computer Graphics 17,* 3 (January 1983), 43-50.

[Olsen 83]
Olsen, Dan R., Automatic Generation of Interactive Systems, *ACM SIGGRAPH Computer Graphics 17*, 1 (January 1983), 53-57.

[Olsen 86]
Olsen, Dan R., Editing Templates: A User Interface Generation Tool, *IEEE Computer Graphics and Applications 6*, 11 (November 1986), 40-45.

[Raeder 85]
Raeder, Georg, A Survey of Current Graphical Programming Techniques, *IEEE Computer 18,* 8 (August 1985), 11-25.

[Reiss 85]
Reiss, Steven P., PECAN: Program Development Systems That Support Multiple Views, *IEEE Transactions on Software Engineering 11*, 3 (March 1985), 276-285.

[Reiss, Golin and Rubin 86]
Reiss, Steven P., Eric J. Golin and Robert V. Rubin, Prototyping Visual Languages With the Garden System, *Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages*, Dallas, Texas, Juli 1986, 81-90.

[Reiss 86]
Reiss, Steven P., An Object-Oriented Framework for Graphical Programming, *SIGPLAN Notices Notices 21,* 10 (October 1986), 49-57.

[Reiss 87]
Reiss, Steven P., A Conceptual Programming Environment, *Proceedings 9th International Conference on Software Engineering*, Monterey, CA, March 1987, 225-235.

[Reiss and Pato 87]
Reiss, Steven P. and Joseph N. Pato, Displaying Program and Data Structures, *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences 2*(January 1987), 391-401.

|Rubin and Pato 84]
Rubin, Robert V. and Joseph N. Pato, MFE: A Syntax Directed Editor for Interaction Specification, *Proceedings Graphics Interface'84 Conference*, 1984, 265-269.

|Schoen and Smith 83]
Schoen, Eric and Reid G. Smith, Impulse: A Display Oriented Editor for STROBE, *Proceedings National Conference on Artificial Intelligence*, August 1983, 356-358.

|Shaw et al. 83]
Shaw, Mary, Ellen Borison, Michael Horowitz, Tom Lane, David Nichols and David Pausch, Descartes: A Programming-Language Approach to Interactive Display Interfaces, *Proceedings ACM SIGPLAN Notices Symposium on Programming Language Issues in Software Systems*, June 1983, 100-111.

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

[Shu 85]
Shu, Nan C., Visual Programming Languages: A Dimensional Analysis, *Proceedings of International Symposium on New Directions in Computing*, Trondheim, Norway, August 12-14, 1985, 326-334.

[Sibert, Hurley and Bleser 86]
Sibert, John L., William D. Hurley and Teresa W. Bleser, An Object-Oriented User Interface Management System, *ACM SIGGRAPH'86 Conference Proceedings*, Dallas, Texas, August 1986, 259-268.

[Smith et al. 84]
Smith, Reid G., Gilles M.E. Lafue, Eric Schoen and Stanley C. Vestal, Declarative Task Description as a User-Interface Structuring Mechanism, *IEEE Computer 17*, 9 (September 1984), 29-38.

[Smith, Dinitz and Bart 86]
Smith, Reid G., Rick Dinitz and Paul Bart, Impulse-86: A Substrate for Object-Oriented Interface Design, *OOPSLA'86 Conference Proceedings*, Portland, OR, September 1986.

[Smith 86]
Smith, Randall B., THE ALTERNATE REALITY KIT - An Animated Environment for Creating Interactive Simulations, *Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages*, Dallas, Texas, Juli 1986, 99-106.

|Steele and Sussman 80]
Steele, Jr., G.L. and G.J. Sussman, CONSTRAINTS, A Language for Expressing Almost-Hierarchical Descriptions, *Artificial Intelligence 14*(1980), 1-39.

[Stefik and Bobrow 86]
Stefik, Mark and Daniel G. Bobrow, Object-Oriented Programming: Themes and Variations, *The AI Magazine*, 1986, 40-62.

[Studer 84]
Studer, R., Abstract Models of Dialog Concepts, *Proceedings 7th International Conference on Software Engineering*, 1984, 420-429.

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

[Sweetman 85]
Sweetman, Dominic, A Modular Window System for Unix, in *Methodology of Window Management*, F.R.A. Hopgood, D.A. Duce, E.V.C Fielding, K. Robinson and A.S. Williams (ed.), Springer Verlag, Berlin, 1985, 73-79.

[Takala 85]
Takala, T., Communication Mediator - A Structure for UIMS, in *Workshop on User Interface Management Systems (1983: Seeheim-Jugenheim, Germany)*, Guenther E. Pfaff (ed.), Springer-Verlag, Berlin, 1985, 59-66.

[Van Wyk 80]
Van Wyk, C., *A Language for Typesetting Graphics*, PhD Thesis, Stanford University, 1980.

[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), 163-182.

[van den Bos, Plasmeijer and Hartel 83]
van den Bos, Jan, Marinus J. Plasmeijer and Pieter H. Hartel, Input-Output Tools: A Language Facility for Interactive and Real-Time Systems, *IEEE Transactions on Software Engineering SE-9*, 3 (May 1983), 247-259.

[Wasserman 85]
Wasserman, Anthony I., Extending State Transition Diagrams for the Specification of Human-Computer Interaction, *IEEE Transactions on Software Engineering SE-11*, 8 (August 1985), 699-713.

[Williams 85]
Williams, Tony, A Comparison of Some Window Managers, in *Methodology of Window Management*, F.R.A. Hopgood, D.A. Duce, E.V.C Fielding, K. Robinson and A.S. Williams (ed.), Springer Verlag, Berlin, 1985, 15-33.

Appendices

The following appendices include the complete filter type definitions that were used in examples throughout this dissertation. Appendix A describes the filter type for the factory simulation interface described in Chapter 3. Appendix B defines the filter type that was implemented with the tree mapper mechanism as described in Chapter 5. Appendix C defines the filter and object types that define the filter browser as an interface to the filter class hierarchy. Appendix D lists the object types that are common to all of our examples. Appendix E contains a list of filter atoms that are provided with our current implementation. Appendix F contains a listing of the MasterSlave filter type (Chapter 5) that was produced with the "file-out" mechanism of the filter browser.

A. Factory Simulation

The interface to a factory simulation was used as introduction to the concepts of our filter paradigm. This interface visualizes the flow of products from a producer through two work stations to a consumer. The user can manipulate the simulation by varying the rate of productivity of the producer and by adding or removing workers from work stations. The `FactorySimulation` filter type is the top level filter that defines this interface. It decomposes the source and view objects and establishes subfilters, like `StationFilter`, `ProducerFilter`, `Consumer-Render` and `BarChart` for them.

```
Filter Type FactorySimulation ( source: Factory, view: Device)
var
        part[4] → Device
make set of
        ProducerFilter (source.producer, part[1])
        StationFilter (source.ws1, part[2])
        StationFilter (source.ws2, part[3])
        ConsumerFilter (source.consumer, part[4])
        iteration 4 times i
                Extract (part[i], view)
end

Object Type Producer
        productivity → Integer
        produced → Integer
end

Object Type Station
        numberOfWorkers → Integer
        workers [numberOfWorkers] → Worker
        inNumber → Integer
        inQueue [inNumber] → Queue
        constraint LessThan (numberOfWorkers, 3)
end
```

```
Object Type Consumer
       consumed → Integer
end

Object Type Factory
       producer → Producer
       ws1 → Station
       ws2 → Station
       consumer → Consumer
end

Object Type Worker
       inherit from Person
       salary → Integer
       throughput → Integer
end

Object Type Apprentice
       inherit from Worker
       level → Integer
end

Object Type Expert
       inherit from Worker
       years → Integer
end
```

```
Filter Type StationFilter (source: Station, view: Device)
var
        left, part[2] → Bitmap
        workerDetected [2] → Boolean
        selection → Integer
        expert → Expert
        apprentice → Apprentice
make set of
        QueueRender (source.inNumber, left)
        Extract (left, view)
        iteration 2 times i
                WorkerRender (source.worker[i], part[i])
                Extract (part[i], view)
                DetectCursor (part[i], workerDetected[i])
                condition workerDetected[i]
                        condition source.worker[i] isNil
                                PopUpMenu (selection, "Expert, Apprentice")
                                condition selection = 1
                                        Equality (expert, source.worker[i])
                                condition selection = 2
                                        Equality (apprentice, source.worker[i])
                        condition source.worker[i] notNil
                                Equality (NIL, source.worker[i])
end

Object Type Gauge
        number → Integer
        needle → LineSegment
end

Filter Type ProducerRender (source: Producer, view: Device)
var
        gauge → Gauge
make set of
        IntegerEquality (source.productivity, gauge.number)
        Render (gauge, view.output)
        PointSensor (gauge.needle.point2, view.input)
end
```

```
Filter Type ConsumerFilter (source: Consumer, view: Device)
var
       ct → CenteredText
make set of
       Render (source, view.output)
       IntegerStringConversion (source.consumed, ct.text)
       Render (ct, view.output)
end
```

B. Manipulating Trees

Two trees are used to illustrate the tree mapper mechanism. The trees are defined recursively and have an integer label at each node. The filter type defines that all nodes from one tree are connected to the corresponding nodes in the other tree with `Add1Filter` filters. The two trees are displayed on the bitmap of the workstation and can be moved with the mouse cursor. The left tree can be traversed with a `TraverseTree` filter that walks the tree depending on selections from a pop-up menu and can add or delete subtrees. This filter type was implemented with the filter browser using the mapper mechanism.

```
Object Type BinaryTree
       value → Integer
       location → Point
       left → BinaryTree
       right → BinaryTree
end

Filter Type TreeManipulation (source: BinaryTree, view: BinaryTree)
var
       device → Device
make set of
       TreeRender (source, device.output)
       TreeRender (view, device.output)
       PointSensor (source.location, device.input)
       PointSensor (view.location, device.input)
       TreeNodeAdder (source, view)
       TreeTraverse (source, device)
end
```

```
Filter Type TreeRender (source: BinaryTree, view: Bitmap)
make set of
      NodeRender (source, view)
      condition source.left notNil
            LineRender ((source.location, source.left.location), view)
            NodeRender (source.left, view)
      condition source.right notNil
            LineRender ((source.location, source.right.location), view)
            NodeRender (source.right, view)
end

Filter Type NodeRender (source: BinaryTree, view: Bitmap)
var
       ct → CenteredText
make set of
      IntegerTextConversion (source.value, ct.text)
      PointEquality (source.location, ct.location)
      Render (ct, view)
end

Filter Type TreeNodeAdder (source: BinaryTree, view: BinaryTree)
make
      condition source notNil and view notNil
            Add1Filter (source.value, view.value)
            TreeNodeAdder (source.left, view.left)
            TreeNodeAdder (source.right, view.right)
end
```

```
Filter Type TreeTraverse (source: BinaryTree, view: Device)
var
      newTree → BinaryTree (value ← 0)
      selection → Integer
make set of
      condition source notNil
            PopUpMenu (selection, 'Left/Right/Add Left/Add Right/
                                  Delete Left/Delete Right')
            condition selection = 1
                  TreeTraverse (source.left, view)
            condition selection = 2
                  TreeTraverse (source.right, view)
            condition selection = 3
                  Equality (source.left, newTree)
            condition selection = 4
                  Equality (source.right, newTree)
            condition selection = 5
                  Equality (source.left, NIL)
            condition selection = 6
                  Equality (source.right, NIL)
end
```

C. Filter Browser Filter Type

The FilterBrowser filter type defines an interface to the object and filter class hierarchy as implemented in ThingLab and Smalltalk. The object type definitions use the shorthand "[]" to denote an iterated field of variable length where the length is stored in an implicit `count` field.

```
Object Type ClassList
       objects [] → Class
       filters → FilterClassList
end

Object Type FilterClassList
       filterAtoms [] → FilterAtomMetaType
       filterPacks [] → FilterPackMetaType
end

Object Type FilterMetaType
       filterName → String
       sourceType → Class
       viewType → Class
end

Object Type FilterAtomMetaType
       inherit from FilterMetaType
end

Object Type FilterPackMetaType
       inherit from FilterMetaType
       variables [] → FieldDescription
       subfilters [] → SubfilterDescription
end

Object Type FieldDescription
       fieldName → String
       fieldClass → Class
end
```

```
Object Type SubfilterDescription
       inherit from FieldDescription
end

Object Type SequenceConstructor
       inherit from SubfilterDescription
       sourceMerge [] → AccessPath
       viewMerge [] → AccessPath
end

Object Type ConditionConstructor
       inherit from SubfilterDescription
       sourceMerge [] → AccessPath
       viewMerge [] → AccessPath
       conditonMerge [] → AccessPath
end

Object Type IterationConstructor
       inherit from SubfilterDescription
       sourceMerge [] → AccessPath
       viewMerge [] → AccessPath
       factorMerge [] → AccessPath
end

Object Type AccessPath
       fields [] → String
end
```

```
Filter Type FilterBrowser ( source: ClassList, view: FilterDevice)
var
       filterType → FilterPackMetaType
       step → Integer
       selection → Integer
make
       PopUpMenu (step, 'Step 2/Step 2/Step 3')
       condition (step = 1)
              PopUpMenu (selection, 'create/select')
              condition (selection = 1)
                     NewFilter (filterType, view)
                     AddToList (source.filters.filterPacks, filterType)
              condition (selection = 2)
                     SelectFilter (filterType, view)
              ModifyTypes ((source, filterType), view)
       condition (step = 2)
              ModifyFilter ((source, filterType), view)
       condition (step = 3)
              InstantiateFilter (filterType, view)
end

Filter Type NewFilter (source: FilterPackMetaType, view: FilterDevice)
var
       filter → FilterPackMetaType (filterName ← 'NewFilter'
                             sourceType ← Bottom
                             viewtype ← Bottom)
make
       FilterEquality (filter, source)
       StringSensor (source.filterName, view.input)
end
```

```
Filter Type ModifyTypes(source:(ClassList,FilterPackMetaType),view:FilterDevice)
var
      uppers, lowers → Bitmap
      selection → Integer
      part[] → Bitmap
      detected[] → Boolean
make
      ExtractHorizontal ((view.output,(0,0,1)), uppers)
      ExtractHorizontal ((view.output,(0,1,1)), lowers)
      Render (source.second.filterName, uppers)
      RenderList (source.first.objects, (lowers,parts))
      iteration source.first.count times n
            DetectCursor (part[n], detected[n])
            condition detected
                  PopUpMenu (selection, "source/view")
                  condition selection = 1
                        Equality(source.second.sourceType,source.first.objects[n])
                  condition selection = 2
                        Equality(source.second.viewType,source.first.objects[n])
end
```

```
Filter Type ModifyFilter(source:(ClassList,FilterPackMetaType),view:FilterDevice)
var
        left, middle, right → Bitmap
        subfilters[], variables[], selectables[] → Bitmap
        tobeInserted, sourceVar, viewVar, iterVar, condVar → Class
        selected[], subSelected[], varSelected[] → Boolean
        selection → Integer
        newField → FieldDescription
make
        ExtractVertical ((view.output,(0,0.3)), left)
        ExtractVertical ((view.output,(0.3,0.6), middle)
        ExtractVertical ((view.output,(0.6,1), right)
        RenderList (source.second.subfilters, (left, subfilters))
        RenderList (source.second.variables, (middle, variables))
        RenderList (source.first, (right, selectables))
        iteration selectables.count times i
                DetectCursor (selectables[n], selected)
                Equality (source.first[i], tobeInserted)
                condition tobeInserted isFilterType
                        PopUpMenu (selection, "source/view/iteration/condition")
                        iteration variables.count times n
                                DetectCursor (variables[n], varSelected[n])
                                condition varSelected[n]
                                        condition selection = 1
                                                Equality(source.first.objects[n], sourceVar)
                                        condition selection = 2
                                                Equality(source.first.objects[n], viewVar)
                                        condition selection = 3
                                                Equality(source.first.objects[n], iterVar)
                                        condition selection = 4
                                                Equality(source.first.objects[n], condVar)
                        CreateSubfilterConstructor(
                                (sourceVar,viewVar,iterVar,condVar), newField)
                        AddToList (source.second.subfilters, newField)
                condition tobeInserted isObjectType
                        CreateFieldDescription (tobeInserted, newField)
                        AddToList (source.second.variables, newField)
        iteration subfilters.count times n
                DetectCursor (subfilters[n], subSelected[n])
                        condition subSelected[n]
                                PopUpMenu (selection, "delete")
                                condition selection = 1
                                        RemoveFormList(source.second.subfilters,
                                                subfilters[n])

        iteration variables.count times n
```

```
            DetectCursor (variables[n], varSelected[n])
                  condition subSelected[n]
                        PopUpMenu (selection, "delete")
                        condition selection = 1
                              RemoveFormList(source.second.variables,
                                    variables[n])
end

Filter Type CreateSubfilterConstructor(source:Object[4],view:SubfilterDescription)
make set of
      condition source[3] notNil
            CreateObjectInstance ("IterationConstructor", view)
            Equality (source[3], view.factorMerge)
      condition source[4] notNil
            CreateObjectInstance ("ConditionConstructor", view)
            Equality (source[4], view.conditionMerge)
      condition source[3] isNil and source[4] isNil
            CreateObjectInstance ("SequenceConstructor", view)
      Equality (source[1], view.sourceMerge)
      Equality (source[2], view.viewMerge)
end

Filter Type InstantiateFilter (source: FilterPackMetaType, view: FilterDevice)
var
      filterInstance, sourceInstance, viewInstance → Object
make
      CreateObjectInstance (source, filterInstance)
      CreateObjectInstance (source.sourceType, sourceInstance)
      CreateObjectInstance (source.viewType, viewInstance)
      CreateFilterInstance (sourceInstance, viewInstance)
end
```

D. Object Types

Objects are constructed from atomic values that can be of type Integer, Character, Boolean or Bit. Object types String and Text are represented as an array of characters. In addition our implementation gives the types Mouse and Bitmap. The following common object types were used in the previous sections. These object types are defined in the current implementation of the filter browser. They are represented as Things in ThingLab.

```
        A device models the workstation and has an input and output medium.
        The input medium is a mouse, the output medium a display bitmap.

Object Type Device
        input → Mouse
        output → Bitmap
end

        A centered text centers a text around a location when it is
        copied into a bitmap.

Object Type CenteredText
        location → Point
        text → Text
end

        An interval has two elements.

Object Type Interval
        low → Integer
        high → Integer
end
```

E. Implementation Filter Atoms

The following filter atoms were used in the previous examples. They are defined as Things in ThingLab and have constraints associated with them.

```
Filter Type IntegerEquality (source: Integer, view: Integer)
Filter Type PointEquality (source: Point, view: Point)
       equality filter atoms

Filter Type PointSensor (source: Point, view: Mouse)
       reflects the location of the mouse in a point
Filter Type TextSensor (source: Text, view: InputMedium)
Filter Type StringSensor (source: String, view: InputMedium)
       inputs a text into source

Filter Type Render (source: Object, view: Bitmap)
       copies a source object into the view bitmap
Filter Type LineRender (source: LineSegment, view: Bitmap)
       copies a source line into the view bitmap
Filter Type RenderList (source: Object[], view: (Bitmap[], Bitmap))
       copies a list of source objects into the subparts
       of the second part of the view pair

Filter Type IntegerTextConversion (source: Integer, view: Text)
       converts integers into text and vice versa

Filter Type IntegerDivide (source: (Integer, Integer), Integer)
       divides first element of source pair by second giving view
Filter Type Add1Filter (source: Integer, view: Integer)
       view is one more than source

Filter Type AddToList (source: Object[], view: Object)
       adds view object to list of source objects
Filter Type RemoveFromList (source: Object[], view: Object)
       removes view object from list of source objects

Filter Type TextConcat (source: (Text, Text), view: Text)
       concatenates the source text pair into the view text
Filter Type ExtractHorizontal (source: (Bitmap, Interval), view: Bitmap)
       horizontally extracts part from bitmap in source
       that is specified by the source interval into view bitmap
Filter Type ExtractVertical (source: (Bitmap, Interval), view: Bitmap)
       vertically extracts part from bitmap in source
```

that is specified by the source interval into view bitmap

Filter Type PopUpMenu (source: Integer, view: Text)
selects option from list specified in view text

Filter Type DetectCursor (source: Bitmap, view: Boolean)
indicates whether the cursor is within bitmap

Filter Type GaugeSensor (source: Gauge, view: Integer)
sets view integer according to location of gauge needle

Filter Type BarChart (source: (Integer, Integer), view: Bitmap)
displays barchart in view bitmap specified by source

Filter Type CreateFieldDescription (source: Object, view: FieldDescription)
creates a field description view object for the source object

Filter Type CreateObjectInstance (source: Class, view: Object)
creates an instance (prototype) of the source class

F. Master-Slave Filter

This listing of the MasterSlave filter type was filed out from the filter browser. If this text is filed in by Smalltalk, then it will produce the same filter type that can be used by the filter browser.

```
"From Smalltalk-80 version T2.2.0, of March 13, 1986 on 15 July 1987 at 1:56:07 pm"!

"Filed out from Filter Browser (Version 2.1) of Saturday, June 27, 1987"!


ThingLabObject subclass: #CenteredText
        instanceVariableNames: 'text center '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Prototypes'!


!CenteredText methodsFor: 'showing'!

showPicture: medium

| para cr |

        para _ text asParagraph.
        cr _ para compositionRectangle.
        para displayOn: medium at: center - (cr width / 2@ -4).
        (cr expandBy: 2) showPicture: medium! !

CenteredText prototype parts: 'center text '!

#Text lookupClass!
CenteredText prototype instVarAt: 1 put:
        (Text string: 'text' runs: (RunArray runs: #(4 ) values: #(1 )))!

#Point lookupClass!
CenteredText prototype instVarAt: 2 put: (Point x: 25 y: 10)!

Constraint owner: CenteredText prototype
        rule: 'self stay'
        methods: #(
                'self stay')
        priority: (ConstraintPriority at: #default)!

CenteredText prototype performAllMerges!



ThingLabObject subclass: #PersonIcon
        instanceVariableNames: 'point '
        classVariableNames: 'PictureForm '
        poolDictionaries: ''
        category: 'Prototypes'!



!PersonIcon methodsFor: 'quickCompile'!

enclosingFrameOrNil
        | t1 t2 |
        t1 _ point + PictureForm offset extent: PictureForm width @ PictureForm height.
        t2 _ super enclosingFrameOrNil.
        t2 == nil
                ifTrue: [^t1]
                ifFalse: [^t1 merge: t2]!

showPicture: t1
        super showPicture: t1.
        PictureForm
                displayOn: t1
                at: point
                rule: Form paint! !


PersonIcon prototype primitives: 'point '!

#Point lookupClass!
PersonIcon prototype instVarAt: 1 put: (Point x: 80 y: 80)!

Constraint owner: PersonIcon prototype
        rule: 'self stay'
        methods: #(
                'self stay')
        priority: (ConstraintPriority at: #default)!

PersonIcon prototype performAllMerges!
```

```
FilterAtomThing subclass: #FaMasterSlave
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Filter-Atoms'!


!FaMasterSlave methodsFor: 'all'!

check

        ((source point1 y) > (source point2 y)) ifTrue: [
                ^self view = 'slave' asText
        ].
        ((source point1 y) = (source point2 y)) ifTrue: [
                ^self view = 'colleague' asText
        ].
        ((source point1 y) < (source point2 y)) ifTrue: [
                ^self view: 'master' asText
        ].
        ^false!

getText

        ((source point1 y) > (source point2 y)) ifTrue: [
                ^'slave' asText
        ].
        ((source point1 y) = (source point2 y)) ifTrue: [
                ^'colleague' asText
        ].
        ((source point1 y) < (source point2 y)) ifTrue: [
                ^'master' asText
        ].! !

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

FaMasterSlave class
        instanceVariableNames: ''!


!FaMasterSlave class methodsFor: 'initialize class'!

initializePrototype

"initialize the prototype"
```

```
        self sourceType: #LineSegment.
        self viewType: #Text.
        self prototype parts: 'source view '.
        self prototype primitives: 'picture '.
        self prototype instVarAt: 1 put: (10@10 line: 50@50).
        self prototype instVarAt: 2 put: 'master' asText.
        self prototype instVarAt: 3 put: nil.

        Constraint owner: self prototype
        rule: 'self check'
        methods: #('self primitiveSet.view: self getText'
                    'source point1 referenceOnly'
                    'source point2 referenceOnly').

        self prototype performAllMerges.

"FaMasterSlave initializePrototype"! !



FilterPackThing subclass: #PersonAtPoint
        instanceVariableNames: 'centeredText4 personIcon5 faPointEquality6
                faPointEquality7 faPointSensor8 faRender9 faRender10 faMasterSlave11 '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'FilterPrototypes'!

PersonAtPoint sourceType: #LineSegment!
PersonAtPoint viewType: #FilterDevice!

PersonAtPoint prototype parts: 'source view '!
PersonAtPoint prototype variables: 'centeredText4 personIcon5 '!
PersonAtPoint prototype subfilters: 'faMasterSlave11 faPointEquality6
                faPointEquality7 faPointSensor8 faRender10 faRender9 '!
PersonAtPoint prototype primitives: 'picture '!

PersonAtPoint prototype instVarAt: 1 put:
        (LineSegment basicNew instVarAt: 1 put:
                (Point x: 270 y: 143); instVarAt: 2 put: (Point x: 50 y: 20); yourself)!

PersonAtPoint prototype instVarAt: 2 put:
        (FilterDevice basicNew instVarAt: 1 put:
                (FilterMouse basicNew yourself); instVarAt: 2 put: nil; yourself)!

PersonAtPoint prototype instVarAt: 3 put:
        ((Dictionary new)
        add: (Association basicNew instVarAt: 1 put: 4; instVarAt: 2 put:
                (Point x: (215/634) y: (176/359)); yourself);
        add: (Association basicNew instVarAt: 1 put: 5; instVarAt: 2 put:
                (Point x: (236/317) y: (121/359)); yourself);
        add: (Association basicNew instVarAt: 1 put: 6; instVarAt: 2 put:
                (Point x: (211/634) y: (75/359)); yourself);
        add: (Association basicNew instVarAt: 1 put: 7; instVarAt: 2 put:
                (Point x: (251/634) y: (134/359)); yourself);
        add: (Association basicNew instVarAt: 1 put: 8; instVarAt: 2 put:
                (Point x: (135/317) y: (28/359)); yourself);
        add: (Association basicNew instVarAt: 1 put: 9; instVarAt: 2 put:
                (Point x: (359/634) y: (67/359)); yourself);
        add: (Association basicNew instVarAt: 1 put: 10; instVarAt: 2 put:
                (Point x: (232/317) y: (87/359)); yourself);
        add: (Association basicNew instVarAt: 1 put: 11; instVarAt: 2 put:
                (Point x: (50/317) y: (140/359)); yourself); yourself)!

PersonAtPoint prototype instVarAt: 4 put:
        (CenteredText basicNew instVarAt: 1 put:
                (Text string: 'master' runs: (RunArray runs: #(6 ) values: #(1 )));
                instVarAt: 2 put: (Point x: 270 y: 143); yourself)!

PersonAtPoint prototype instVarAt: 5 put:
        (PersonIcon basicNew instVarAt: 1 put: (Point x: 270 y: 143); yourself)!

PersonAtPoint prototype instVarAt: 6 put:
        (FaPointEquality basicNew instVarAt: 1 put:
                (Point x: 270 y: 143); instVarAt: 2 put:
                (Point x: 270 y: 143); instVarAt: 3 put: nil; yourself)!

PersonAtPoint prototype instVarAt: 7 put:
        (FaPointEquality basicNew instVarAt: 1 put:
                (Point x: 270 y: 143); instVarAt: 2 put:
                (Point x: 270 y: 143); instVarAt: 3 put: nil; yourself)!

PersonAtPoint prototype instVarAt: 8 put:
        (FaPointSensor basicNew instVarAt: 1 put:
                (Point x: 270 y: 143)  instVarAt: 2 put:
                (FilterMouse basicNew yourself); instVarAt: 3 put: nil; yourself)!

PersonAtPoint prototype instVarAt: 9 put:
        (FaRender basicNew instVarAt: 1 put:
                (CenteredText basicNew instVarAt: 1 put:
                (Text string: 'master' runs: (RunArray runs: #(6 ) values: #(1 )));
                instVarAt: 2 put: (Point x: 270 y: 143); yourself); instVarAt: 2
                put: nil; instVarAt: 3 put: nil; yourself)!
```

```
PersonAtPoint prototype instVarAt: 10 put:
        (FaRender basicNew instVarAt: 1 put: (PersonIcon basicNew instVarAt: 1 put:
                (Point x: 270 y: 143); yourself); instVarAt: 2 put: nil;
                instVarAt: 3 put: nil; yourself)!

PersonAtPoint prototype instVarAt: 11 put:
        (FaMasterSlave basicNew instVarAt: 1 put:
                (LineSegment basicNew instVarAt: 1 put: (Point x: 270 y: 143);
                instVarAt: 2 put: (Point x: 50 y: 20); yourself); instVarAt: 2 put:
                (Text string: 'master' runs: (RunArray runs: #(6 ) values: #(1 )));
                instVarAt: 3 put: nil; yourself)!

Constraint owner: PersonAtPoint prototype
        rule: 'self stay'
        methods: #(
                'self stay')
        priority: (ConstraintPriority at: #default)!

Constraint owner: PersonAtPoint prototype
        rule: 'source stay'
        methods: #(
                'source stay')
        priority: (ConstraintPriority at: #default)!

Constraint owner: PersonAtPoint prototype
        rule: 'view stay'
        methods: #(
                'view stay')
        priority: (ConstraintPriority at: #default)!

Constraint owner: PersonAtPoint prototype
        rule: 'picture stay'
        methods: #(
                'picture stay')
        priority: (ConstraintPriority at: #default)!

PersonAtPoint prototype merge: #( 'source point1' 'faPointEquality6 source')!

PersonAtPoint prototype merge: #( 'centeredText4 center' 'faPointEquality6 view')!

PersonAtPoint prototype merge: #( 'source point1' 'faPointEquality7 source')!

PersonAtPoint prototype merge: #( 'personIcon5 point' 'faPointEquality7 view')!

PersonAtPoint prototype merge: #( 'source point1' 'faPointSensor8 source')!
PersonAtPoint prototype merge: #( 'view mouse' 'faPointSensor8 view')!

PersonAtPoint prototype merge: #( 'centeredText4' 'faRender9 source')!

PersonAtPoint prototype merge: #( 'view bitmap' 'faRender9 view')!

PersonAtPoint prototype merge: #( 'personIcon5' 'faRender10 source')!

PersonAtPoint prototype merge: #( 'view bitmap' 'faRender10 view')!

PersonAtPoint prototype merge: #( 'source' 'faMasterSlave11 source')!

PersonAtPoint prototype merge: #( 'centeredText4 text' 'faMasterSlave11 view')!

PersonAtPoint prototype performAllMerges!


FilterPackThing subclass: #Relativity
        instanceVariableNames: 'lineSegment4 personAtPoint5 personAtPoint6
                faPointEquality7 faPointEquality8 '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'FilterPrototypes'!

Relativity sourceType: #LineSegment!
Relativity viewType: #FilterDevice!



Relativity prototype parts: 'source view '!
Relativity prototype variables: 'lineSegment4 '!
Relativity prototype subfilters: 'faPointEquality7 faPointEquality8
                personAtPoint5 personAtPoint6 '!
Relativity prototype primitives: 'picture '!

Relativity prototype instVarAt: 1 put: (LineSegment basicNew instVarAt: 1 put:
        (Point x: 30 y: 30); instVarAt: 2 put: (Point x: 50 y: 20); yourself)!

Relativity prototype instVarAt: 2 put: (FilterDevice basicNew instVarAt: 1 put:
        (FilterMouse basicNew yourself); instVarAt: 2 put: nil; yourself)!

Relativity prototype instVarAt: 3 put: ((Dictionary new)
        add: (Association basicNew instVarAt: 1 put: 4; instVarAt: 2 put:
                (Point x: (73/634) y: (200/359)); yourself);
        add: (Association basicNew instVarAt: 1 put: 5; instVarAt: 2 put:
                (Point x: (579/1268) y: (79/359)); yourself);
        add: (Association basicNew instVarAt: 1 put: 6; instVarAt: 2 put:
                (Point x: (637/1268) y: (144/359)); yourself);
        add: (Association basicNew instVarAt: 1 put: 7; instVarAt: 2 put:
                (Point x: (205/634) y: (114/359)); yourself);
        add: (Association basicNew instVarAt: 1 put: 8; instVarAt: 2 put:
                (Point x: (66/317) y: (152/359)); yourself); yourself)!

Relativity prototype instVarAt: 4 put:
        (LineSegment basicNew instVarAt: 1 put:
        (Point x: 30 y: 30); instVarAt: 2 put: (Point x: 50 y: 20); yourself)!

Relativity prototype instVarAt: 5 put: (PersonAtPoint basicNew instVarAt: 1 put:
        (LineSegment basicNew instVarAt: 1 put: (Point x: 30 y: 30); instVarAt: 2 put:
        (Point x: 50 y: 20); yourself); instVarAt: 2 put:
        (FilterDevice basicNew instVarAt: 1 put: (FilterMouse basicNew yourself);
        instVarAt: 2 put: nil; yourself); instVarAt: 3 put: nil; instVarAt: 4 put:
        (CenteredText basicNew instVarAt: 1 put: (Text string: 'master' runs:
        (RunArray runs: #(6 ) values: #(1 ))); instVarAt: 2 put:
        (Point x: 270 y: 143); yourself); instVarAt: 5 put: (PersonIcon basicNew
        instVarAt: 1 put: (Point x: 270 y: 143); yourself); instVarAt: 6 put:
        (FaPointEquality basicNew instVarAt: 1 put: (Point x: 30 y: 30); instVarAt: 2 put:
        (Point x: 270 y: 143); instVarAt: 3 put: nil; yourself); instVarAt: 7 put:
        (FaPointEquality basicNew instVarAt: 1 put: (Point x: 30 y: 30); instVarAt: 2 put:
        (Point x: 270 y: 143); instVarAt: 3 put: nil; yourself); instVarAt: 8 put:
        (FaPointSensor basicNew instVarAt: 1 put: (Point x: 30 y: 30); instVarAt: 2 put:
        (FilterMouse basicNew yourself); instVarAt: 3 put: nil; yourself); instVarAt: 9 put:
        (FaRender basicNew instVarAt: 1 put: (CenteredText basicNew instVarAt: 1 put:
        (Text string: 'master' runs: (RunArray runs: #(6 ) values: #(1 ))); instVarAt: 2 put:
        (Point x: 270 y: 143); yourself); instVarAt: 2 put: nil; instVarAt: 3 put: nil;
        yourself); instVarAt: 10 put: (FaRender basicNew instVarAt: 1 put: (PersonIcon basicNew
        instVarAt: 1 put: (Point x: 270 y: 143); yourself); instVarAt: 2 put: nil;
        instVarAt: 3 put: nil; yourself); instVarAt: 11 put: (FaMasterSlave basicNew
        instVarAt: 1 put: (LineSegment basicNew instVarAt: 1 put: (Point x: 30 y: 30);
        instVarAt: 2 put: (Point x: 50 y: 20); yourself); instVarAt: 2 put:
        (Text string: 'master' runs: (RunArray runs: #(6 ) values: #(1 )));
        instVarAt: 3 put: nil; yourself); yourself)!

Relativity prototype instVarAt: 6 put: (PersonAtPoint basicNew instVarAt: 1
        put: (LineSegment basicNew instVarAt: 1 put: (Point x: 30 y: 30); instVarAt: 2
        put: (Point x: 50 y: 20); yourself); instVarAt: 2 put: (FilterDevice basicNew
        instVarAt: 1 put: (FilterMouse basicNew yourself); instVarAt: 2 put: nil; yourself);
        instVarAt: 3 put: nil; instVarAt: 4 put: (CenteredText basicNew instVarAt: 1 put:
        (Text string: 'master' runs: (RunArray runs: #(6 ) values: #(1 ))); instVarAt: 2
        put: (Point x: 270 y: 143); yourself); instVarAt: 5 put: (PersonIcon basicNew
        instVarAt: 1 put: (Point x: 270 y: 143); yourself); instVarAt: 6 put:
        (FaPointEquality basicNew instVarAt: 1 put: (Point x: 30 y: 30); instVarAt: 2 put:
```

```
        (Point x: 270 y: 143); instVarAt: 3 put: nil; yourself); instVarAt: 7 put:
        (FaPointEquality basicNew instVarAt: 1 put: (Point x: 30 y: 30); instVarAt: 2 put:
        (Point x: 270 y: 143); instVarAt: 3 put: nil; yourself); instVarAt: 8 put:
        (FaPointSensor basicNew instVarAt: 1 put: (Point x: 30 y: 30); instVarAt: 2 put:
        (FilterMouse basicNew yourself); instVarAt: 3 put: nil; yourself); instVarAt: 9 put:
        (FaRender basicNew instVarAt: 1 put: (CenteredText basicNew instVarAt: 1 put: (Text
        string: 'master' runs: (RunArray runs: #(6 ) values: #(1 ))); instVarAt: 2 put:
        (Point x: 270 y: 143); yourself); instVarAt: 2 put: nil; instVarAt: 3 put: nil;
        yourself); instVarAt: 10 put: (FaRender basicNew instVarAt: 1 put: (PersonIcon
        basicNew instVarAt: 1 put: (Point x: 270 y: 143); yourself); instVarAt: 2 put: nil;
        instVarAt: 3 put: nil; yourself); instVarAt: 11 put: (FaMasterSlave basicNew
        instVarAt: 1 put: (LineSegment basicNew instVarAt: 1 put: (Point x: 30 y: 30);
        instVarAt: 2 put: (Point x: 50 y: 20); yourself); instVarAt: 2 put: (Text
        string: 'master' runs: (RunArray runs: #(6 ) values: #(1 )));
        instVarAt: 3 put: nil; yourself); yourself)!

Relativity prototype instVarAt: 7 put:
        (FaPointEquality basicNew instVarAt: 1 put:
        (Point x: 30 y: 30); instVarAt: 2 put:
        (Point x: 50 y: 20); instVarAt: 3 put: nil; yourself)!

Relativity prototype instVarAt: 8 put:
        (FaPointEquality basicNew instVarAt: 1 put:
        (Point x: 50 y: 20); instVarAt: 2 put:
        (Point x: 30 y: 30); instVarAt: 3 put: nil; yourself)!

Constraint owner: Relativity prototype
        rule: 'self stay'
        methods: #(
                'self stay')
        priority: (ConstraintPriority at: #default)!

Constraint owner: Relativity prototype
        rule: 'source stay'
        methods: #(
                'source stay')
        priority: (ConstraintPriority at: #default)!

Constraint owner: Relativity prototype
        rule: 'view stay'
        methods: #(
                'view stay')
        priority: (ConstraintPriority at: #default)!

Constraint owner: Relativity prototype
        rule: 'picture stay'
        methods: #(
                'picture stay')
        priority: (ConstraintPriority at: #default)!

Relativity prototype merge: #( 'source' 'personAtPoint5 source')!

Relativity prototype merge: #( 'view' 'personAtPoint5 view')!

Relativity prototype merge: #( 'lineSegment4' 'personAtPoint6 source')!

Relativity prototype merge: #( 'view' 'personAtPoint6 view')!

Relativity prototype merge: #( 'source point1' 'faPointEquality7 source')!

Relativity prototype merge: #( 'lineSegment4 point2' 'faPointEquality7 view')!

Relativity prototype merge: #( 'source point2' 'faPointEquality8 source')!

Relativity prototype merge: #( 'lineSegment4 point1' 'faPointEquality8 view')!

Relativity prototype performAllMerges!
```

Biographical Note

The author was born on May 20, 1958, in Hechingen, Federal Republic of Germany. He attended public schools in Hechingen from 1964 until 1977.

After high school and before entering the University of Stuttgart in 1978 he served in the West German Army for 15 months as an artillery Corporal. At the University of Stuttgart he pursued the field of computer science and a minor in the field of transportation.

In 1981 he earned his undergraduate degree (Vordiplom) in computer science and continued as a graduate student, which is customary in Germany. During the academic year of 1983 to 1984 he was an exchange student from the University of Stuttgart to Oregon State University, Corvallis, Oregon, where he earned his Master of Science degree in computer science. After returning to Germany in 1984, he completed the requirements for the German graduate degree in computer science (Diplom-Informatiker) in 1985.

The author came to the Oregon Graduate Center in September 1985 to pursue his doctoral degree. During his first year he held the Tektronix Fellowship for Graduate Studies. He leaves OGC to accept a faculty position at the Florida International University in Miami, Florida.