Proceedings of the 1988 Winter Simulation Conference
M. Abrams, P. Haigh, and J. Comfort (eds.)

Constraint-based user interfaces for simulations

Raimund K. Ege

Florida International University
University Park
Miami, Fla. 33199

eger@servax.bitnet

Abstract

Many simulation environments lack modern user interfaces. This paper describes an approach to user interfaces that is very well suited for object-oriented simulation environments. Our approach views user interfaces as constraints that are maintained between separate objects. The contraints and the participating objects are modelled with a type language that captures the important aspects of an interface. Some of these aspects are the relationships and their types between various presentation objects. The type language can be produced graphically with the help of an interactive interface generator that can define and manipulate user interfaces to typed objects. The implementation of this interface generator does not disturb a running simulation and is based on constraint-satisfaction.

Keywords: simulation environment, user interfaces, object-oriented, constraints.

1 Introduction

The principles of encapsulation and abstraction are germane to the field of simulation. Recently these ideas have become fashionable under the heading of object-oriented programming. Many simulation systems follow these principles [Kre86] without necessarily calling themselves "object-oriented." In this paper we introduce our ideas of how to apply those principles to user interfaces for simulation environments. We extend the common notion of object-oriented programming with the concept of "constraints." Constraints are a natural extension to abstraction and encapsulation, they allow us to state simple facts about objects and their relationships.

Many modern object-oriented languages derive their features from the programming language Simula [DMN68]. Simula is a ALGOL-like language that was intended to support simulation systems. Smalltalk-80 [GR83], the prototypical object-oriented language, can also be used in simulating systems. ThingLab [Bor81], a constraint-oriented simulation laboratory, extended object-oriented Smalltalk with constraints. Animus [Dui86], an animation kit, extended ThingLab to provide notions of time and continuity for simulation animation. There are many other constraint languages and systems available, a survey of constraint languages and systems can be found in [Lel87].

We start this paper with a simulation example that serves as an introduction to our "Filter Paradigm" and illustrates the usage of constraints to specify interfaces declaratively. The second section of this paper explains the major building blocks of our interface paradigm: objects, constraints and filters. The third section re-examines the introductory example and shows how such an animated interface can easily be constructed in our system. The fourth section briefly explains our environment that allows us to interactively specify and manipulate such interfaces. Our system is implemented using ThingLab, the last section discusses some of its implementation aspects.

2 Introductory Example: Factory Simulation

We use 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 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 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.

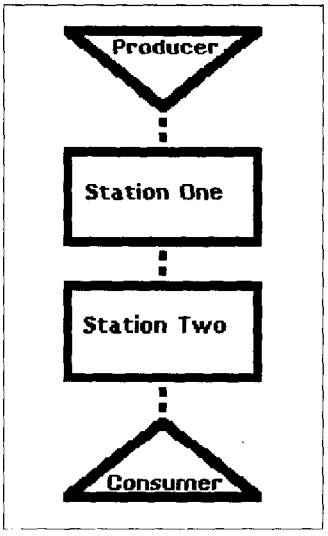


Figure 1: Flow Chart for Factory Simulation

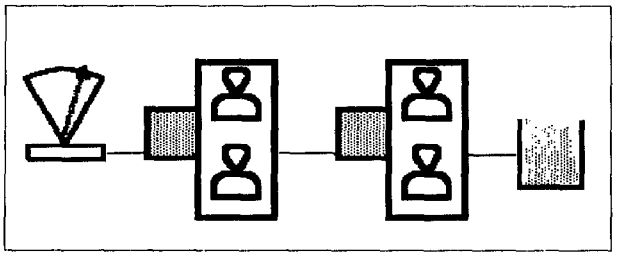


Figure 2: Screen Layout for Simulation Example

2.1 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 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.

2.2 Application Interface Model

According to the "Logical Model of a UIMS" [Gre85] an interface can be viewed as consisting of three components (see Figure 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 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.

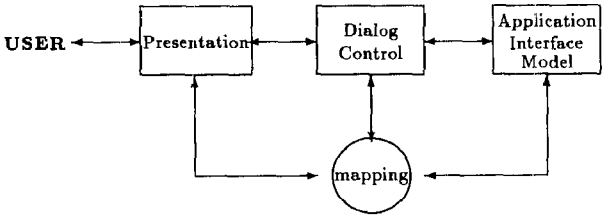


Figure 3: User Interface Management System

2.3 Display

The data structures for the objects in the application model will be mapped into the screen representations as shown in Figure 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.

2.4 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 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

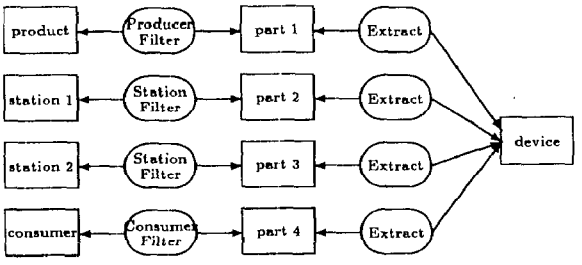


Figure 4: Filter Diagram for Factory Simulation Interface

the interface is derived from the fact that the constraints are maintained by a constraint-satisfaction system.

Figure 4 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.

3 Building Blocks

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.

3.1 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 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.

```
Object Type Station
        numberOfWorkers -> Integer
        workers [numberOfWorkers] -> Worker
        inNumber -> Integer
        inQueue [inNumber] -> Queue
        constraints LessThan (numberOfWorkers, 3)
end
```

Figure 5: Object Type for Station

An example object type is `Station` (Figure 5). 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 **constraints** 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. 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 6 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

```
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
```

Figure 6: Subtypes of Worker

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.

3.2 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.

3.3 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 7 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.

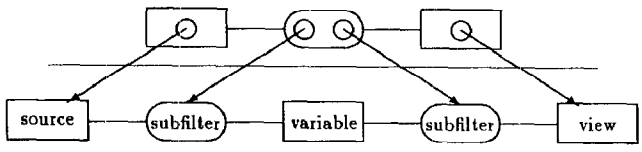


Figure 7: End-to-End Combination

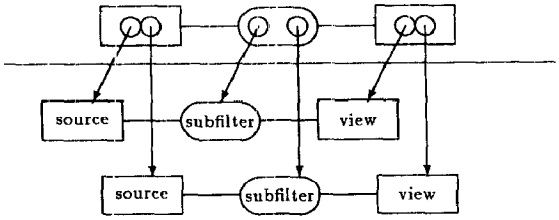


Figure 8: Side-by-side Combination

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 8 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.

4 Factory Simulation: Revisited

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 5 and 9). The objects for the simulation example are all aggregated into the object type `Factory` (Figure 9). Our factory has exactly one

```
Object Type Producer
       productivity -> Integer
       produced -> Integer
end

Object Type Consumer
       consumed -> Integer
end

Object Type Factory
       producer -> Producer
       ws1 -> Station
       ws2 -> Station
       consumer -> Consumer
end

Object Type  Device
       input -> Mouse
       output -> Bitmap
end
```

Figure 9: 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 10: Filter Type for Factory Simulation

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 10 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 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 4 showed the filter configuration for the factory simulation filter.

The `FactorySimulation` filter decomposes the interface into sub-constraints that are represented by subfilters. Each of these subfilters has to be defined separately using other constructed or atomic filters. We give here the definition of the `StationFilter` to show the expressiveness of filter types.

Figure 11 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 con-

```
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 11: Filter Type for Station

structed 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.

5 Defining Interfaces Graphically

The introductory example showed how we can decompose interfaces using constraints. Filter types are packages of constraints that describe interfaces. In order to define interfaces graphically we represent filter types in an object-oriented system and provided a graphical tool that manipulates the representation. Our tool, the *Filter Browser* [EMB87], can define, manipulate and test filter types graphically. The filter browser has been implemented in Smalltalk-80.

The filter browser creates the classes within Smalltalk 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, they are modeled as regular classes within Smalltalk. 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. A session 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.

The appendix includes three screen snapshots of the *Filter Browser* as the user interface for the factory simulation example is generated, as well as a screen snapshot of the resulting user interface.

ThingLab [Bor81], an extension to Smalltalk, is used to do the constraint satisfaction. ThingLab extends the Smalltalk class definition with constraints, types for instance variables and dynamic access. Smalltalk does not keep information on the type of instance variables, so ThingLab augments the class definition to hold the type (reference to another object type class) for each instance variable. Constraints that are defined within the object type are also stored in the class definition. An instance of an object type refers back to its class definition, so the constraint-satisfaction mechanism can retrieve the defined constraints. Note, that a subtype also inherits all constraints that are defined for its supertypes. Thus, our implemented typing mechanism does not allow an instance of a subtype to be stored in a typed instance variable. This is clearly an undesirable limitation, which we plan to remove (see [Ege87][BDF*87][Ege88]). Removing it is not trivial, however, since depending on how the instance variable is used, we may need to ensure that the constraints on a subtype are not more restrictive than those on the specified type.

All objects have to exist within ThingLab. Objects outside ThingLab are the display bitmap, the keyboard, specialized input devices (mice), or existing complex objects in an application. These object are incorporated by either providing a special object in ThingLab that holds the outside object and controls all accesses to it (object holder), or by providing special filters that link existing objects within ThingLab to those external objects (implementation filter atoms). Graphical primitives, such as line rendering or input sensoring [Ege86], are examples for filter atoms to incorporate I/O-objects.

6 Conclusion

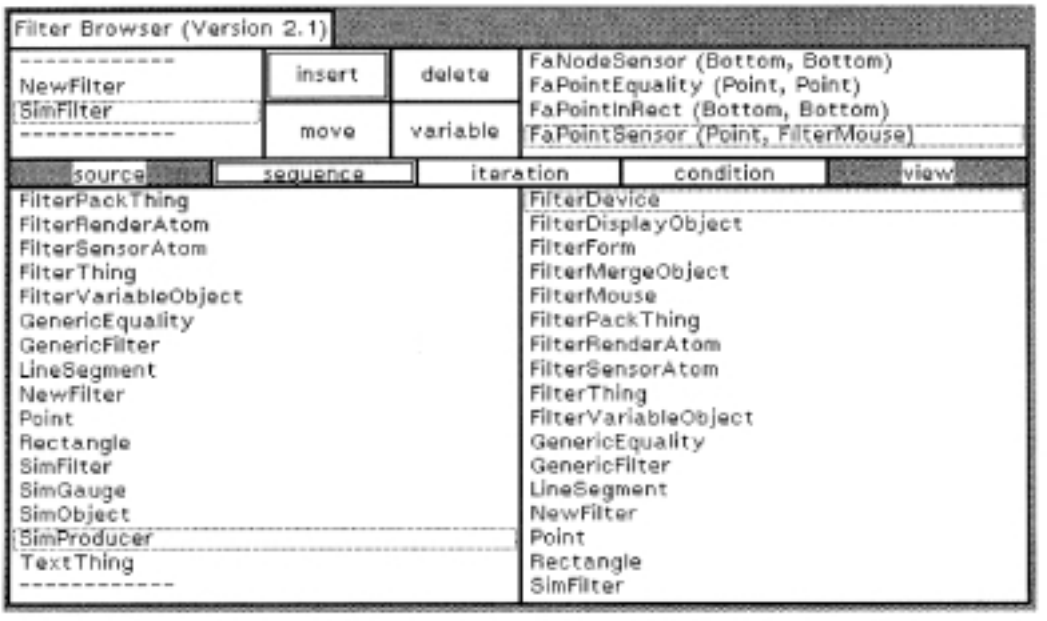
Our object-oriented paradigm for user interfaces based on constraints represents a new approach to interfaces in a simulation environment. Constraints are used as the basic building block for interfaces. Constructors are provided to allow building of structured interfaces in a declarative way. The feasibility of the filter paradigm is shown by providing a working interface generation tool. A video tape that shows the animation and manipulation of the simulation as well as the generation and manipulation of the user interface itself is available. The major drawback of the current implementation is that, while it does

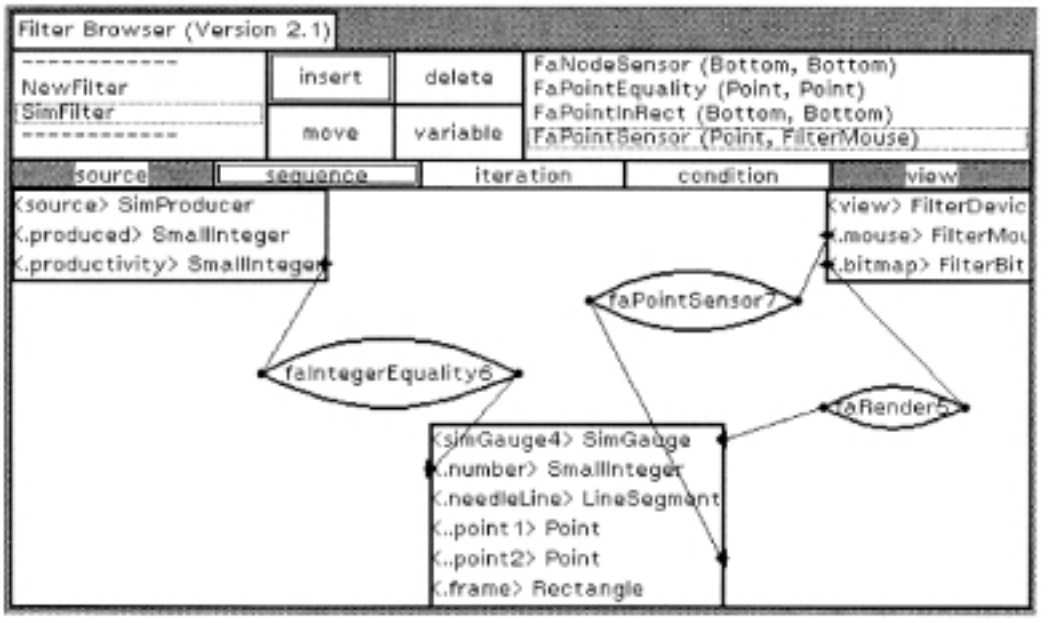
not influence the simulation process directly, it consumes most of the computing power of the workstation, which eventually has an indirect influence on the simulation. We are currently working on a more efficient implementation of the constraint-satisfaction mechanism.

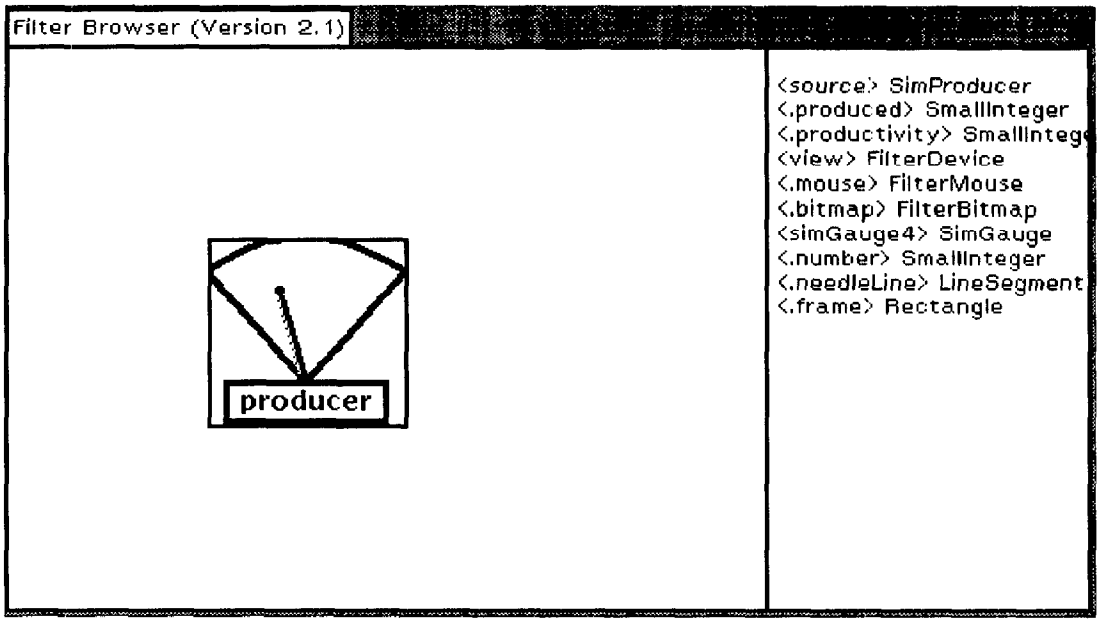
Acknowledgements

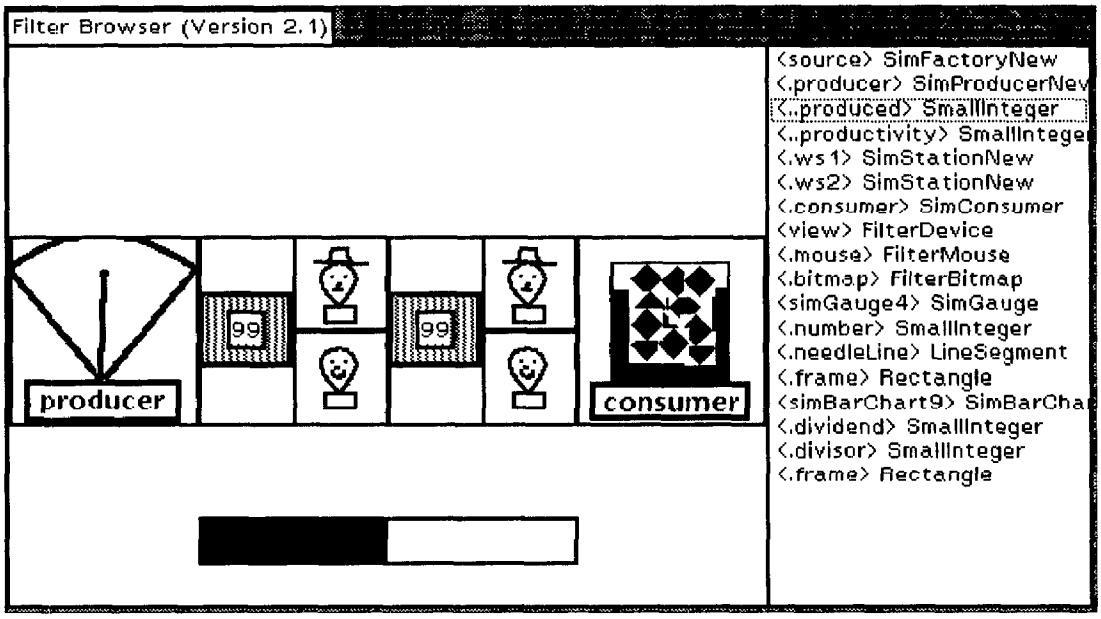
This research has been funded by the National Science Foundation under Grants No. IRI-8604923 and IRI-8604977.

Appendix






Filter Browser (Version 2.1)
producer
<source> SimProducer
<.produced> SmallInteger
<.productivity> SmallInteg
<view> FilterDevice
<.mouse> FilterMouse
<.bitmap> FilterBitmap
<simGauge4> SimGauge
<.number> SmallInteger
<.needleLine> LineSegment
<.frame> Rectangle


Filter Browser (Version 2.1)
99
99
producer
consumer
<source> SimFactoryNew
<.producer> SimProducerNe
<..produced> SmallInteger
<..productivity> SmallInteg
<.ws1> SimStationNew
<.ws2> SimStationNew
<.consumer> SimConsumer
<view> FilterDevice
<.mouse> FilterMouse
<.bitmap> FilterBitmap
<simGauge4> SimGauge
<.number> SmallInteger
<.needleLine> LineSegment
<.frame> Rectangle
<simBarChart9> SimBarCha
<.dividend> SmallInteger
<.divisor> SmallInteger
<.frame> Rectangle

References

[BDF*87] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In *Proceedings of OOPSLA'87 Conference*, pages 48–60, Orlando, FL, October 1987.

[Bor81] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[DMN68] O.J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA 67 Common Base Language.* Technical Report, Norwegian Computing Center, 1968.

[Dui86] Robert A. Duisberg. *Constraint-Based Animation: The Implementation of Temporal Constraints in the Animus System.* PhD thesis, University of Washington, 1986.

[Ege86] Raimund K. Ege. *The Filter - A Paradigm for Interfaces.* Technical Report No. CSE-86-011, Oregon Graduate Center, Beaverton, OR, September 1986.

[Ege87] Raimund K. Ege. *Automatic Generation of User Interfaces Using Constraints.* PhD thesis, Oregon Graduate Center, 1987.

[Ege88] Raimund K. Ege. Defining constraint-based user interfaces. *IEEE Data Engineering, Special Issue on Whatever Happened to Semantic Modeling*, 11(2), 1988.

[EMB87] Raimund K. Ege, David Maier, and Alan Borning. The Filter Browser: Defining interfaces graphically. In J. Bézivin et al., editor, *Proceedings of European Conference on Object Oriented Programming (Springer Verlag: Lecture Notes in Computer Science No. 276)*, pages 155–165, Paris, France, June 1987.

[GR83] Adele Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison Wesley, Reading, Mass., 1983.

[Gre85] Mark Green. Report on dialog specification tools. In Guenther E. Pfaff, editor, *Workshop on User Interface Management Systems (1983: Seeheim-Jugenheim, Germany)*, pages 9–20, Springer Verlag, 1985.

[Kre86] Wolfgang Kreutzer. *System Simulation - Programming Styles and Languages.* Addison Wesley, 1986.

[Lel87] Wm Leler. *Constraint Programming Languages.* Addison Wesley, 1987.

Biographical Note

Dr. Ege received his Ph.D. degree in computer science and engineering from the Oregon Graduate Center in 1987, where he was a research assistant to Prof. David Maier. He is currently Assistant Professor of Computer Science at the School of Computer Science at Florida International University. His general area of research is object-oriented concepts. He has investigated the application of these concepts to user interfaces and software engineering and has published several papers in these areas. The School of Computer Science at FIU has an extensive object-oriented programming laboratory that features many of the current object-oriented programming languages.

Raimund K. Ege
School of Computer Science
Florida International University
University Park
Miami, Fla. 33199
(305) 554-2744
eger@servax.bitnet