Logical Composition of Object-Oriented Interfaces

Mark Grossman

University of Hawaii at Hilo
Hilo, Hawaii 96720

Raimund K. Ege

Florida International University
Miami, Florida 33199

Abstract

This paper describes an approach to object-oriented interface design that goes beyond mere object decomposition. In our user interface management system we use logic and filters to declaratively specify and control a space of ways that objects may be composed to create interfaces. A filter is a package of constraints and associated typed objects that express the relationship of data and representation objects.

Conceptually our system is completely based on constraints. Filters provide the high bandwidth constraints to maintain the components of the direct-manipulation interface while the logic forms the low bandwidth constraints to combine and provide communication between these components. The use of Horn-clause logic to compose separate interface objects facilitates both the distribution of computation onto multiple processors and the generation of multiple views of data. Intelligent backtracking implemented in the logic allows for user- and system-initiated *undo* operations to correct errors and/or try alternative approaches to a problem. We illustrate the power and flexibility of this approach by describing a floor layout and design system.

1. Motivation

How can people and machines best solve problems is the focus of our research. Computer Science has made great progress in providing algorithms to help the machine solve its problems. The cost of hardware and software no longer justify the restrictions computer systems place on the way people solve problems. We seek to provide a user-oriented system that will encourage people to use their creativity and imagination to find better solutions.

How do people go about solving a *real-world* problem? The most natural methods are reflected in the structure of our organizations. Typically, a manager is assigned overall responsibility for a problem. Most managers decompose a complex problem into subtasks which he or she then delegates to other personnel. These people go off and work on their assigned area of the problem. Some subtasks affect and/or are affected by other subtasks thus, a person might need to communicate with other people working on different aspects of the overall problem. Solutions to subproblems are sent back up the hierarchy. The resulting answer consists of a select subset of all the information generated. A difficult problem, because of the complex interrelationships between its subproblems, often will require an iterative approach to generate a satisfactory solution. Therefore, a person evaluates the result and decides whether to accept it or generate another solution. Our proposed system is oriented toward these human ways of problem-solving.

Work done while both authors were at the Oregon Graduate Center; Beaverton, Oregon 97006.

Work supported by NSF grants IST-83-51730 and IRI-86-04977, cosponsored by Tektronix Foundation, Intel, Mentor Graphics, DEC, Servio Logic Corp., IBM, Xerox and Beaverton Chamber of Commerce.

1.1. Introduction

This paper describes an approach to object-oriented problem-solving that goes beyond mere object decomposition. Our approach uses a constraint-based paradigm to compose a specific interface and adds a logical component to specify and manage relations between different interfaces. The result is a flexible user-interface management system that is purely declarative.

We will illustrate the features of our system with the following floor layout, *construct*, example. In this example the task of designing a floor within a house is decomposed into subtasks solved by one or more persons using graphical workstations. An architect creates the structure of the floor by inserting walls, doors and windows into the layout. An interior designer populates that structure, the rooms, with furniture such as desks, chairs, and closets. A list of all furniture must be generated and the price of the overall building calculated. In addition, checks are to be made to ensure that manipulating the wall structure or adding furniture does not degrade the overall stability of the structure. It is not necessary, or even desirable, to have either the work of the architect or the designer completely precede the other. The designer may wish to start before the floor layout is completed, and the architect could benefit by taking furniture, included by the designer, into consideration when she draws a wall. The architect and the designer require different interfaces to a common data structure that will allow them to work in parallel. If more than one architect or designer is working on the same floor plan then the system should accommodate them on additional workstations. If the floor layout problem can be handled by one person then the system should provide an alternative approach that incorporates the functions of architect and designer on a single workstation. Figure 1 illustrates this example, the *construct* problem. It shows two different workstation screens. The upper screen lets the architect manipulate the physical structure of the floor plan while the designer may add and move furniture within the walls, via the lower screen.

A system like this is feasible using todays technology of user-interfaces and distributed systems. But it would take great programming effort to build this special system. As Winograd [Winograd 79] and Cox [Cox 86] have pointed out, the problem lies in the difficulties of organizing a complex system. It is hard to achieve a system incorporating the functionality of the above *construct* example, and almost impossible to create this system in a way that is easy to change or modify: adding workstations, moving displays from one workstation to another, combining functionality of interfaces. We propose an object-oriented system that separates the interface and the logical control components. Object-oriented programming is augmented with constraints to provide flexible and user-oriented systems.

Our interface components are built from constraints following the *Filter Paradigm* for constructing interfaces.

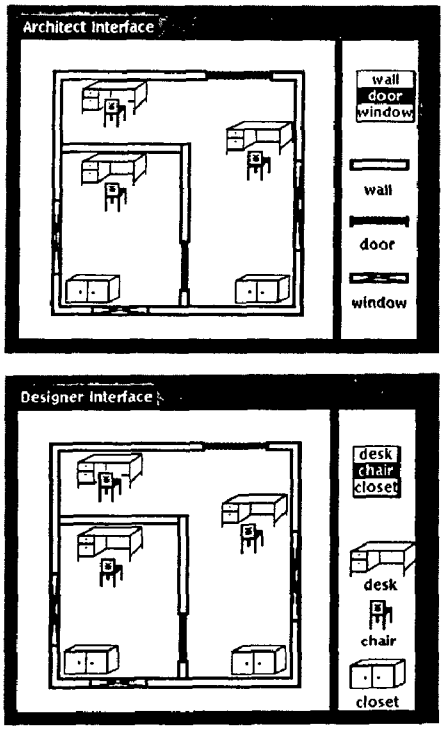

Figure 1: The *construct* problem.

A filter is a package of constraints and associated typed objects that expresses the relationship between data and the data's representation objects. For example, the relationship of a rectangle on a display screen and a piece of memory, both representing a wall, is modelled by a filter. A constraint-satisfaction system maintains the constraints that are expressed within the filter. If a user moves the rectangle on the screen, she directly manipulates the wall object in memory.

The logic part of our system views these interface components as special evaluable predicates that are implemented as independent processes. Logic resolution, is used to control the execution and enables processes to interact. When satisfied with the wall's position the user can cause a logical variable to be instantiated with the wall object's data. Other interfaces that are logically constrained to this variable have access to this information and can thus, make the wall visible on their screens. The non-deterministic flow of control within the logic evaluation provides for different approaches to solving a problem within our user-interface management system. For example, the configuration of the interfaces could be chosen based upon the number of workstations that are available at run-time. Intelligent backtracking implemented in the logic allows for user- and system-initiated *undo* operations to correct errors and/or try alternative approaches to solving a problem.

1.2. Overview of the paper

Our system consists of a Task Interaction and Control System (TICS) that is used to logically compose

processes. TICS is further described in the next section. Special processes called *filters,* implement the different user-interfaces. The user-interfaces, are specified declaratively and draw their procedurality from a constraint-satisfaction system. Filters are discussed in Section three. How filters and TICS communicate is explained in Section four. We conclude by describing a solution for the *construct* problem that illustrates the features of our system.

2. Logical Composition

We will utilize the Task Interaction and Control System (TICS) to model problem-solving as the decomposition of a problem into subtasks. TICS provides a declarative and executable specification of a such a model through the use of Horn clause logic. The logic clearly expresses assumptions and rules that control the composition and interaction of the procedures that solve the subtasks. The power of logic programming to represent a hierarchical search for task solutions is augmented in TICS by *evaluable predicates.* Evaluable predicates are solved by external procedures whose internals are hidden from TICS. TICS' *And-parallel* resolution engine allows these procedures to execute concurrently and thus, interact with each other to solve their subtasks. From the bottom up a solution can be viewed as a composition of facts that are either initially assumed, asserted by procedures or logically inferred. In this paper we only describe those aspects of TICS relevant to the solution of the problem. For a more detailed description the reader is referred to [Grossman 87].

2.1. What Makes TICS Tick

The key to a TICS implementation is its database that incorporates special-purpose functionality. The database is the conductor of a TICS symphony. It directs and controls the flow of music (data) that is being created and read by the instrumental players (evaluable predicates' external processes) according to the composition (Horn clause specification). The database contains the system's specification, provides dynamic working storage and implements a flexible *deduction engine.* Everything in TICS, except the external processes for evaluable predicates, is contained within the database process.

Subtasks specified by evaluable predicates are solved by external procedures that are implemented as separate processes. These concurrent processes communicate and synchronize via messages to and from the database access manager (DAM). The filter-based interface components for the architect and designer are implemented as such processes. Figure 2 is an overview of how our *construct* example can be solved in an environment that supports multi-tasking and interprocess communication.

2.1.1. Deduction Engine

The *deduction engine* is based upon a method of resolution called *plan-based deduction* [Cox and Pietrzykowski 81] [Forsythe and Matwin 84] [Matwin and Pietrzykowski 85]. *Plan-based deduction* differs from Prolog and other stack-based logic programming schemes by allowing unresolved goals to be solved in any order and by tracking actual data unification dependencies to implement intelligent versus blind backtracking.

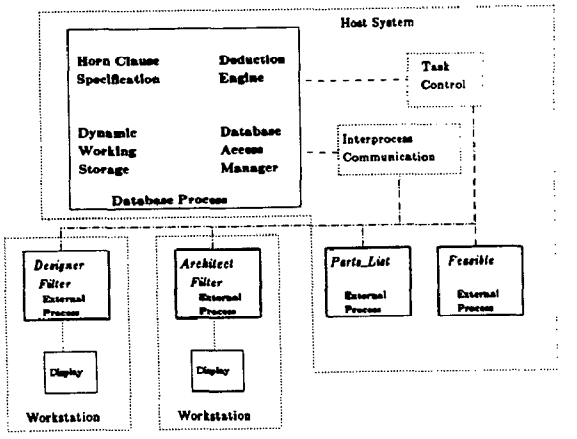

Figure 2: TICS *Construct* Overview.

TICS extends *plan-based deduction* by incorporating *And-parallel* resolution. All unsolved goals of a clause can be solved concurrently. Synchronization and communication are provided via logical variables. TICS provides an executing process with the ability to dynamically examine the evolving environments of other processes with which it shares common variables, i.e., the ability to interact. In our example the architect and interface processes are specified as evaluable predicates that share variables. TICS provides storage and access to these variables via its database and DAM. Each process can read and write these variables via messages to TICS and thus, information about walls and furniture objects can be shared.

3. Interfaces from Constraints

Interface processes are specified as evaluable predicates in TICS. This section presents a new approach to building these interfaces in an object-oriented environment. In such an environment, all entities of interest are represented as objects, so all aspects of the user interface are modelled as objects. In the Smalltalk model-view-controller (MVC) Paradigm [Goldberg and Robson 83], for example, the interface consists of model, view and controller objects. The model and view are basically two different representations of the same conceptual entity [Deutsch 86]. In Smalltalk's MVC paradigm, the model and view have procedural components that allow the controller to manage the interface correctly.

3.1. The Filter Paradigm

Our approach is to abandon procedural specification of user interfaces and relate the model (*source*) and *view* with a declarative interface

specification. The idea is to use constraints to specify the conceptual equivalence between the *source* and *view* objects. For example, the relationship between an employee object and a bitmap object on a screen can be represented by constraints. The constraints state that the bitmap object always displays the employee object. The constraints hide the procedurality of the interface. If the bitmap object on the screen is changed, then the constraint-satisfaction will ensure that the employee object is changed accordingly. If the employee object changes, then that change is reflected on the screen.

A *filter* is an object that describes and maintains these special constraints between objects in an interface. For example, consider our *construct* problem. The designer can select furniture items and place them on the floor plan. Let's say she has selected a desk and wants to move it, using a mouse locator device, to a location within the floor plan. The desk can be placed anywhere except on top of walls, windows, doors or other furniture. This sub-problem can be expressed with constraints: First, the location of the desk is constrained by the location of the mouse. Second, the desk is constrained not to overlap with any of the existing structures.

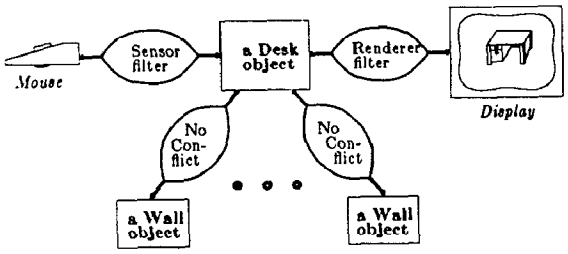

Figure 3: Manipulating a desk object.

Figure 3 shows a diagram of how this subpart of the designer interface can be modelled with filters. The figure contains three types of filters (shown as ellipses). The "Sensor filter" connects the mouse to a desk object and represents the constraint that the desk object stores the location of the mouse. The "Renderer filter" connects a desk object to the display and represents the constraint that the desk is displayed at the given location. The "No Conflict" filter connects the desk object with all the existing walls and represents the constraint that the desk does not overlap with any of the walls in the floor plan. All three types of constraints have to be maintained. If the mouse is moved, then the desk object changes its location value, thus changing its location on the display. If the mouse is moved on top of an existing wall, then the "No Conflict" constraint will prohibit that move. As more walls are added to the floor plan more such "No Conflict" filters are added dynamically. Of course, Figure 3 only shows a small subpart of our *construct* example. Some details are missing in order to keep the figure simple, e.g. the walls would also be shown on the same display as the desk using similar "Renderer filters".

The definition of what types of source and view objects are allowed for the filter and how the subfilters are connected to them is given by the *filter type*. Filter types specify how filters are built from atomic filters using set, iteration and condition constructors. A filter is instantiated from its filter type definition. Atomic filters, like sensors or renderers, are provided by the implementation. The filter and object types are described by a filter specification language (*FiSpeL*) [Ege 86]. *FiSpeL* is a theoretical tool to compose filters, a compiler and optimizer for it are planned. The *Filter Browser* is a tool to construct filters graphically. The *Filter Browser* lets the interface designer create filters by defining and manipulating filter types. Subfilters are added interactively by connecting them with the various constructors to the object types that are displayed in the browser. The *Filter Browser* also allows the designer to instantiate a filter with sample objects to test the constructed interface. For a more detailed discussion see [Ege, Maier and Borning 87].

4. Objects

The logic and interface components of our system communicate via objects. The logical component (TICS) views these objects as logical variables. The interface component (filters) views them as instances of constraint object types.

4.1. Logical Variables as Objects

External processes, e.g. filter-based interface components, are started by TICS' deduction engine to solve evaluable predicates. These processes are invoked with the database identifiers of the logical variables they can access. These variables are accessed only via database read and write messages sent to and received from the DAM in a manner similar to that used in the Humanizer framework [Maier, Nordquist and Grossman 86].

When an external process issues a read command for a variable in its environment the DAM responds with the data type and value derived by unifying the variable's constraint graph, i.e., the current value of the variable. If the value is unacceptable to the process then the process can invoke system backtracking by terminating with a failure status.

When a process sends a write command, the database's DAM enters that value into the constraint graph of the variable to be written. If the database can't unify all variables in the constraint graph intelligent backtracking is invoked. Unifiability is restored by selecting a set of predicates to be undone.

Each process is responsible for acquiring any data it needs. When a process requires data from a logical variable that is not yet instantiated, the process can issue a request to be notified when that variable changes and then suspend itself. This mechanism permits external processes to be started even if the data they require is not currently available. *Mode* declarations and other annotations to provide data synchronization are neither permitted nor required in the logic. Logical variables need never be fully instantiated nor even accessible, as in the case of infinite structures and non-terminating computations, if they are not accessed by a procedure.

Type checking of its parameters is the responsibility of the individual external process. Data types can be

protected from TICS' type-free logic by being specified as being of type private. This technique is used in "Persistent Prolog" [Gray, Moffat and Boulay 85] [Moffat and Gray 86]. Private variables can not be examined by TICS' logic because such a variable can only be unified with another term of that type or with a free variable. All access to the internals of this type of variable must be done by external procedures. TICS' logic can only be used to pass private variables from one predicate to another or to create compound variables, e.g., lists, made up of private types.

A data record can be represented by a logical term. In the *construct* example, the frame can be specified by:

```
frame( origin (X1, Y1), corner (X2, Y2) )
```

In this record `X1` and `Y1` represent the x and y position of the origin of the floor's frame while `X2` and `Y2` represent the far corner. The variables in this term can be written and read by all the evaluable predicates, e.g. filter-based interfaces, that contain the frame record in their environment.

The filter-based process that provides the architect's interface, when invoked will read the value of the frame record to determine the initial values, if any, of the frames `origin` and `corner` points. The architect will, via the interface, directly manipulate the frame until satisfied. When he commits, the process will send a write message to instantiate the appropriate variables. Should he later change his mind, the process can cause backtracking to occur. Backtracking will undo the bindings of the variables and thus, allow the architect to respecify the frame.

4.2. Instances of Object Types: Objects

The logical terms with variables are viewed as instances of object types in the interface component of our system. If we want to build interfaces by composing filters from subfilters, connecting objects of different kinds, it is necessary to type the objects. All entities in our filter paradigm are ultimately implemented by objects, so we put much effort in providing a comprehensive type system.

4.2.1. Object Types

The object type system supports the notions of aggregation and specialization. With aggregation we can build structured objects from components. Specialization allows us to refine existing objects via a hierarchy of object types and inheritance. We view object types as records. A record is a collection of typed fields. The fields have names called *addresses*. There are constant fields, which are constant for all instances of a type, and there are data fields that are local to an instance of an object. Fields can be iterated by specifying an iteration factor; fields can be conditional by specifying a condition that must be true for the field to exist; and fields can specify its type recursively. In addition, an object type can inherit fields from other object types and can place constraints on all fields.

Figure 4 shows the `Frame` object type as defined in *FiSpeL*. It names two fields with addresses, `origin` and `corner`, of type `Point`. The object type `Point` has two addresses, `x` and `y`, of type `Integer`. The logical term

```
frame( origin (X1, Y1), corner (X2, Y2) )
```

would be represented as an instance of type `Frame` holding two instances of type `Point` with the coordinates, `x` and `y`, not yet instantiated.

Objects are used in the filter type definition to describe source, view and variables that are needed to connect subfilters. The filter specification language (*FiSpeL*) provides mechanisms to initialize and reference instances of object types.

```
Object Type Frame                Object Type Point
     origin → Point                   x → Integer
     corner → Point                   y → Integer
end                              end
```

Figure 4: Frame object type.

4.2.2. Filter Types

The filter type system defines the structure of filters. Filters represent constraints between two objects. The filter type defines the types of the source and view objects it relates. The filter type also declares the subfilters that compose the filter. In addition, the filter type can define variables to be used as intermediate objects when subfilters are combined. A filter that is not further decomposed is called a *filter atom* and is provided by the implementation. For example, filter atoms are used for low-level input/output, data conversion or error handling, and to handle primitive user commands, such as commit, undo or fail. A filter that has subfilters is called a *filter pack*. Subfilter constructors are: sequence, iteration and condition. The sequence constructor (**`set of`**) declares several subfilters of possibly different types; the iteration constructor (**`iteration n times`** i) declares a certain number of filters of the same type; the condition constructor (**`condition`**) declares a subfilter only if a given condition is true. It is possible to declare a filter with a subfilter of the same kind as the one being defined, much like a recursive procedure call in a conventional programming language.

Figure 5 shows the `PlaceUnits` filter type. `PlaceUnits` is used by the `Architect` filter. Instances of it allows the user to constrain source objects of type `Floor` to view objects of type `Workstation`. `PlaceUnits` composes the subfilters `PopUpMenu`, `GetObject` and `AddToList`. The `PopUpMenu` subfilter manipulates a `selection` variable to indicate the type of furniture to be added to the floor plan. `GetObject` instantiates an appropriate `fpo` object of type `rectangle` to be added to the floor plan with the `AddToL-`

ist subfilter, Condition constructors are used to select the correct list for the new furniture item. A more detailed description of our *construct* problem can be found in the appendix.

```
Filter Type PlaceUnits(source: Floor, view: Workstation)
var
     selection → String
     fpo → Rectangle
make
     PopUpMenu((selection, 'Wall/Door/Window'),
                              (view, redButton))
     GetObject((floor, fpo), view)
     condition selection = 'Wall'
          AddToList(source.floorPlan.walls, fpo)
     condition selection = 'Door'
          AddToList(source.floorPlan.doors, fpo)
     condition selection = 'Window'
          AddToList(source.floorPlan.windows, fpo)
end
```

Figure 5: Sample filter type.

4.2.3. Commit and Undo

When a filter is invoked it receives the identifiers of the variables in its environment. Filters write and read these variable objects via messages to and from TICS' database access manager. A filter commits information, i.e. makes local data available to the rest of the system, by writing to variables. A filter can undo this information by sending a fail message that initiates TICS' backtracking mechanism. Backtracking by TICS results in selected predicates receiving an abort/suspend message. The bindings of these predicate's variables are undone. A filter, while suspended, can preserve its local state so that if TICS later reactivates that filter to generate another solution the filter can examine its past!

5. Illustrative Solution

The logical component of our system uses Horn clause logic to specify and execute the composition of processes and provides a powerful and understandable model for both the system designer and end-user. The reader is referred to Clocksin and Mellish [Clocksin and Mellish 84] for the syntactic conventions used and for a detailed technical description of Horn clauses and their relationship to logic in general. (Variables begin with uppercase letters while constants and structure names begin with lowercase.) Horn clauses allow us to declaratively specify the facts and rules of a system that succinctly describe what constitutes a solution. The problem-reduction strategy for Horn clauses is identical with the the procedural interpretation of Horn clauses and naturally represents the decomposition of a task into subtasks. The procedural interpretation is described by Kowalski [Kowalski 82]. In our example an implication of the form

```
construct(Frame, Limit, WArch, WDesign) :-
     architect(source(Frame, FloorPlan,
                FloorObjects), view(WArch)),
     designer(source(Frame, FloorPlan,
                FloorObjects), view(WDesign)),
     parts_list(FloorObjects, WDesign),
     feasible(Frame, FloorPlan, FloorObjects,
                Limit, WArch).
```

is interpreted as reducing the task `construct` to subtasks, i.e. logical predicates, `architect`, `designer`, `parts_list` and `feasible`. Each of the subtasks is in turn reduced by other implications, or solved by facts or external processes. An external process can be viewed as a dynamic generator of fact(s).

To apply a clause to a predicate, i.e. subtask, unification may require the instantiation of variables. Instantiating variables in the clause can be regarded as transmitting input information from the predicate to the clause. Instantiating variables in the predicate can be viewed as transmitting output information from the clause to the predicate (and thereby to other predicates which shares variables). For example if we use the above clause to satisfy the following initial request, i.e. query:

```
?- construct(F, L, tty1, tty2).
```

we would be providing `tty1` and `tty2` as input for the values of the architect and designer workstations respectively.

Each subtask has a local binding environment, specified by Horn clause logical variables. The local environment contains a subset of the data that defines the overall state of the system. The `parts_list` environment contains information about the floor objects used in the construction and the workstation to display its results on.

Each subtask can view and modify its local binding environment and thus, examine and change its specified part of the system's state. Subtasks can exchange information with each other through communication channels, i.e., variables in each of their local environments that are constrained to contain the same value. If `designer` writes a value to its variable `Frame` then both `architect` and `feasible` can examine their `Frame` variable to read this value.

Non-determinism allows more than one clause, i.e., method of solution, to be applicable to a given subtask. We could also include the following clause to provide an alternate solution to the *construct* problem that uses only a single process on one workstation to perform both the architect and design functions.

```
construct(Frame, Limit, W, W) :-
     architect_designer(source(Frame, FloorPlan,
                FloorObjects), view(W)),
     parts_list(FloorObjects,W),
     feasible(Frame, FloorPlan, FloorObjects,
                Limit, W).
```

Most people approach a task by decomposing the problem into a limited number of subtasks that are solved by powerful, sometimes cooperating, processes. The internals of the process is not part of the decomposition of the problem and is therefore, not of interest to the problem solver. In addition, logic is not ideal for handling numeric computation and manipulating complex data objects. Thus, TICS extends the power of logic programming to represent a hierarchical search for task solutions by incorporating filters and other external processes. The internals of these processes and their parameters can be be hidden and isolated from the logic. Therefore, these processes can be implemented in any language and in any operating environment capable of interfacing to a TICS system.

Artificial constraints are not imposed by the logic. Solving the *construct* problem does not inherently require that any one subtask be started before the other. It is not necessary for the architect to install all the walls and fixtures that define the floor plan before the designer starts including floor objects. In fact the architect might want to use some of this information about floor objects before finalizing the floor plan. It is important to note that different people may prefer to reach an overall solution via different strategies.

Logic variables can allow indeterminism in which subtask supplies a value for a variable. We could start the *construct* problem with the following query that provides a value for the `frame` record.

```
?- construct(frame(origin(0,0),corner(100,100)),
                    L, tty1, tty1).
```

Alternatively, we could use the query

```
?- construct(F, L, tty1, tty2).
```

and leave it up to the either the `architect` or `designer` filter to instantiate the value.

There are many reasons a person may want to undo and change previous actions. To remove a constraint violation a person can choose between possible changes that will resolve the problem. For example, if the `feasible` subtask cannot succeed because of the combination of floor objects and floor plan then the user(s) may want to change either the floor objects or floor plan or both. A user cannot be expected to have complete knowledge of the interrelationships of a complex system and thus, may require an iterative approach. The ability to change answers provides a person with the capability to solve a task by trying out different choices and exploring different solutions. An architect might want to experiment with the location of a wall and thus learn about situations that cause `feasible` to fail. In addition, individuals sometimes prefer to start with existing prototypical solutions and modify them rather than starting from scratch.

As Donald Norman said, "Error is the natural result of a person attempting to do a task". Therefore, it should be as easy as possible to undo previous actions. Work done to solve parts of a problem that are independent of a modification should not be lost. If we are forced to undo the results of the `architect` filter we should not need to redo the `parts_list` process.

Logic is a good framework for tracking dependencies. The reading of a value by a subtask makes that subtask causally dependent upon the subtask that instantiated that value. If the instantiating subtask later fails then the reading subtask must be undone. The writer has, in effect, *caused* a change in the reader's environment. TICS' database tracks these causality dependencies. For example, if the `feasible` process read the value for `FloorPlan` that was instantiated by the `architect` filter, then if we fail `architect` we must fail `feasible`. However, if `feasible` has not gotten around to reading the value of `FloorPlan` then there is no reason to fail it. We would want `feasible` to be able to read the most current unified value of `FloorPlan` because it can use that knowledge to notify the user of any violations as soon as they occur. TICS extended version of *plan-based deduction* maintains information about the history of the resolution, the unification constraints between variables, and the causality relationships between binding environments. With this information TICS is able to provide user- and system-initiated intelligent dependency-directed backtracking. Intelligent backtracking detects and acts upon the exact source of failure as opposed to exhaustive blind backtracking which treats all the subtasks as equally probable sources of failure. TICS allows alternative solution paths to be tried and errors corrected with only those solutions affected by the change needing to be redone.

Logical variables are used by TICS to handle the communication between different subtasks. Unification and intelligent backtracking are a way of implementing the basic equality constraints that are present if the same variables are mentioned in more than one subtask. The interface component of our system uses constraints also. Constraints have been proven to be very useful for graphical applications [Van Wyk 81].

The interfaces in our *construct* problem are built from constraints. These special constraints are called filters and have been described earlier in this paper. The interface that presents the layout of a floor to a designer is represented by a filter from an object of type `Floor` to an object of type `Workstation`. This giant constraint is decomposed into subfilters using the filter constructors of the Filter Paradigm [Ege 86].

As an example, consider again the subfilter to position a desk on the floor layout as shown in Figure 3. Figure 6 shows the designer interface with the designer about to position a desk on the floor plan. The constraints ensure that the desk display follows the mouse cursor and that the desk does not conflict, i.e. overlap, with any existing structures. If the designer likes the location of the desk she clicks a mouse button and continues by selecting other furniture items to be included into the floor plan.

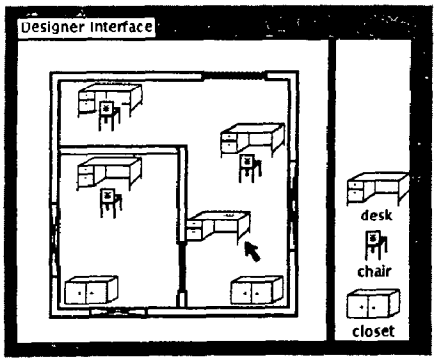

Figure 6: Manipulating a desk object.

If the user is satisfied with her work she can commit, thus triggering the transmission of the changed objects through logical variables to other tasks or interfaces. This communication only takes place if a user commits her choices. After a commit the logical component of our system checks for conflicts within the variables and may initiate backtracking. For example, the designer may place a desk on the floor plan and then commit her choice. The desk is then included in the list of furniture items that is also shared by the "parts_list" subtask. This task may determine that the price limit has been exceeded and therefore fail causing backtracking. Note that before the user commits, violations of the constraints represented by the logical variables are possible.

After a commit, if the user does not like her choice she can fail the subtask, thus causing all changes to the logical variables to be undone via backtracking. The filter subtasks maintain local information even if they are aborted/suspended by backtracking. This allows the filter to reduce the amount of work the user has to redo if the same filter subtask is re-invoked. For example, if the architect decides to remove a wall after he has committed this information he can cause the `architect` filter to fail. Backtracking is invoked and the variable `FloorPlan` containing the wall information is unbound. When the `architect` filter is re-invoked to generate another solution the filter uses local information so that the architect need not re-enter all previous `FloorPlan` data.

Our framework encourages and provides for user-oriented problem solving. The user interfaces, implemented by filter processes, are separate from the application processes. Isolating interface functionality facilitates the development of different versions of an interface to accommodate diverse users with varying proficiency levels and tastes. In addition modularization also allows interface programmers to implement the interactive displays iteratively and in parallel with the development of the rest of the program, a goal espoused by Norman and Draper [Draper and Norman 84].

6. Implementation

We have completed separate prototype implementation for both the TICS and the filter system. The next step is to implement the communication between the two systems as described in section 4.

The TICS system uses C++ and is running under the Berkeley UNIX[1] operating system on a VAX.[2] The prototype incorporates extended *Plan-based deduction* with intelligent backtracking. The UNIX *signal* and *fork* facility are used to execute and control concurrent processes that implement external procedures for evaluable predicates. Processes communicate with TICS by sending messages that read, write or request notification of changes in logical variables in their environment via UNIX *sockets*.

The filter system, together with the *Filter Browser* is written in Smalltalk-80[3] and runs on a Tektronix 4400 machine. The constraint satisfaction is performed using ThingLab [Borning 79]. ThingLab (and therefore Smalltalk) has been extended to handle the types for objects and filters. Dynamic constraints for conditions and iterations were added to ThingLab's constraint-satisfaction mechanism. The operating system used on the 4400 machine is Uniflex. The UNIX and Uniflex systems are connected via Ethernet.

7. Related Research

In the area of User Interface Management Systems (UIMS), dialogue management systems have been developed to coordinate the interaction between modules. Non-declarative approaches have included versions of state transition diagrams and event language. Examples of these techniques are in Wasserman [Wasserman 85] and Green [Green 85] respectively. We feel that there is no reason for the problem solver to anticipate or have knowledge of all possible event orderings. This detailed information will only confuse. We believe that to understand a complex concurrent program it is better to simply know what must be done. The step-by-step details of all the possible ways of how to achieve a goal are better ignored, i.e., abstracted away. Logic provides an effective way to declaratively describe the solution space of a problem. Logic was used by Roach and Nickson [Roach and Nickson 83] for their air traffic control system's dialogue specification. However, their choice of Prolog, with its depth-first strategy and naive backtracking unnecessarily constrains the user.

The problem of task definition and support in the office environment is addressed in both the POISE [Croft and Lefkowitz 84] [Broverman and Croft 85] and HIGGENS [Hudson and King 86] systems. Unlike these systems, we use Horn clause logic to specify the task and theorem proving with intelligent backtracking to control its resolution. We feel our techniques provide a clear and robust declarative model of complex problem

[1]UNIX is a trademark of Bell Laboratories

[2]VAX is a trademark of Digital Equipment Corporation

[3]Smalltalk-80 is a trademark of Xerox Inc.

environments that can be directly executed in a user-oriented manner.

Our approach to the interface component was guided by experience with the Smalltalk MVC paradigm [Goldberg and Robson 83]. 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. Constraints are used to specify relations and dependencies in Morgenstern's *active* database interface system [Morgenstern 83]. Other systems use constraints as their major construct, such as ThingLab [Borning 79], which allows constraints to be expressed in a graphical manner. The Animus system [Duisberg 86], an extension to ThingLab, provides constraints that involve time. An early system that employed constraints to express graphical relations was Sketchpad [Sutherland 63]. The language Ideal [Van Wyk 81], used in typesetting graphical pictures, is based on constraints and demonstrates their power and usefulness. Bertrand [Leler 86] is a term rewriting language that can specify constraint satisfaction systems. In its current implementation, however, it is not interactive and therefore not well suited for our problem. Constraints have also been used in the layout mechanism of a window management system [Cohen, Smith and Iverson 86].

8. Conclusions

We have given an example to illustrate the power of our problem-solving environment that incorporates a direct manipulation style of interface. Constraints are used as the basic building block for interfaces. Low-bandwidth constraints are handled in the logic; high-bandwidth constraints are handled in the interface filters.

In summary, our system provides the following user-oriented features: Horn clause logic provides a clear and concise way to decompose a task. Logic allows subtasks to be solved in any order consistent with the inherent nature of the task. Backtracking provides an intelligent user- and system-undo facility. The non-determinism of the logic provides alternate ways to solve a subtask. The filter paradigm provides a declarative approach to providing direct manipulation interfaces in an object-oriented environment.

In addition, the following designer-oriented features are provided: The ability to separate the user-interface filters from application processes and to logically compose them into an integrated system. Evaluable predicates provide the designer the flexibility to incorporate a wide variety of tools, languages and environments and the potential to distribute the solving of subtasks to multiple processors.

Bibliography

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

[Broverman and Croft 85]
Broverman, C. A. and W. B. Croft, A Knowledge-Based Approach To Data Management For Intelligent User Interfaces, *Proceedings of VLDB 85*, Stockholm, 1985, pp. 96-104.

[Clocksin and Mellish 84]
Clocksin, W.F. and C.S. Mellish, second edition, Springer-Verlag, Berlin, 1984.

[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.

[Cox and Pietrzykowski 81]
Cox, Philip T. and Tomasz Pietrzykowski, Deduction Plans: A Basis for Intelligent Backtracking, *IEEE Transactions on Pattern Analysis and Machine Intelligence Vol. PAMI-3*(January 1981), pp. 52-65.

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

[Croft and Lefkowitz 84]
Croft, W. B. and L. S. Lefkowitz, Task Support in an Office System, *ACM Transactions on Office Information Systems Vol. 2* , No. 3 (July 1984), pp. 197-212.

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

[Draper and Norman 84]
Draper, Stephen W. and Donald A. Norman, Software Engineering For User Interfaces, *Proceedings of the Seventh International Conference on Software Engineering*, Orlando, Florida, March 1984.

[Duisberg 86]
Duisberg, Robert A., Animus: A Constraint Based Animation System, *Proc. of the Conf. on Human Factors in Computing Systems*, 1986.

[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, *Proc. European Conf. on Object Oriented Programming*, Paris, France, June 1987.

[Forsythe and Matwin 84]
Forsythe, Kenneth and Stanislaw Matwin, Implementation Strategies For Plan-Based Deduction, in *International conference on automated deduction. Proceedings of the 7th conference (Napa, 1984) [Lecture notes in computer science; n.170]*, R.E. Shostak (ed.), Springer-Verlag, New York, 1984.

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

[Gray, Moffat and Boulay 85]
Gray, P.M.D., D.S. Moffat and J.B.H. du Boulay, Persistent Prolog: A Secondary Storage Manager for Prolog, *Persistence and Data Types Papers for the Appin Workshop*, University of Glasgow, Glasgow, August 1985, pp. 353 - 368.

[Green 85]
Green, Mark, The University of Alberta User Interface Management System, *ACM SIGGRAPH'85 San Francisco Vol. 19*, No. 3 (July 1985), pp. 205-213.

[Grossman 87]
Grossman, Mark, *Task Interaction and Control System (TICS)*, PhD Thesis, Oregon Graduate Center, 1987.

[Hudson and King 86]
Hudson, Scott E. and Roger King, A Generator of Direct Manipulation Office Systems, *ACM Transactions on Office Information Systems Vol. 4*, No. 2 (April 1986), pp. 132-163.

[Kowalski 82]
Kowalski, R.A., Logic As A Computer Language, in *Logic Programming*, K.L. Clark and S.A. Tarnlund (ed.), Academic Press, London, 1982, pp. 3-16.

[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.

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

[Matwin and Pietrzykowski 85]
Matwin, Stanislaw and Tomasz Pietrzykowski, Intelligent Backtracking in Plan-Based Deduction, *IEEE Transactions on Pattern Analysis and Machine Intelligence Vol. PAMI-7*(November 1985), pp. 682-692.

[Moffat and Gray 86]
Moffat, D.S. and P.M.D. Gray, Interfacing Prolog to a Persistent Data Store, *3rd International Conference on Logic Programming [to be published by Springer Verlag, editor E. Shapiro]*, London, July 1986.

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

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

[Roach and Nickson 83]
Roach, J. W. and M. Nickson, Formal Specifications For Modeling And Developing Human/Computer Interfaces, *Proceedings of the CHI 1983 Conference on Human Factors in Computer Systems*, December 1983, pp. 35-39.

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

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

[Wasserman 85]
Wasserman, A., Extending State Transition Diagrams for the Specification of Human-Computer Interaction, *Transactions On Software Engineering Vol. SE-11*, No. 8 (August 1985), pp. 699-713.

[Winograd 79]
Winograd, Terry, Beyond Programming Languages, *Comm. ACM 22*, 7 (July 1979), pp.391-401.

APPENDIX

```
/* TICS' executable specification of the construct problem */

/* method to provide an interface for one person to solve the architect and
      designer subtasks on a single workstation. */

construct(Frame, Limit, W, W) :-
      architect_designer(source(Frame, FloorPlan, FloorObjects), view(W)),
      parts_list(FloorObjects,W),
      feasible(Frame, FloorPlan, FloorObjects, Limit, W).

/* method to provide an interface for two persons to solve the architect and
      designer subtasks on separate workstations. */

construct(Frame, Limit, WArch, WDesign) :-
      architect(source(Frame, FloorPlan, FloorObjects), view(WArch)),
      designer(source(Frame, FloorPlan, FloorObjects), view(WDesign)),
      parts_list(FloorObjects, WDesign),
      feasible(Frame, FloorPlan, FloorObjects, Limit, WArch).

/* Note: architect, designer, parts_list and feasible predicates are specified
      as being evaluable via the $pred system predicate which is further described
      in [Grossman 87]. */

/* FiSpeL [Ege 86] specification of the construct problem */

/* object types for Floor, PlanUnits, ObjectUnits and Worstation */

Object Type Floor
      frame → Rectangle
      floorPlan → PlanUnits
      floorObjects → ObjectUnits
end

Object Type PlanUnits
      walls → Rectangle
      doors → Rectangle
      windows → Rectangle
end

Object Type ObjectUnits
      desks → DeskForm
      chairs → ChairForm
      closets → ClosetForm
end

Object Type Workstation
      input → InputMedium
      output → OutputMedium
end


/* filter type for Architect, displays the floorplan on the workstation */
/* and allows the user to modify it                                      */

Filter Type Architect (source: Floor, view; Workstation)
var
      temp → Floor
make
      condition source.frame = nil
            RectangleFromUser(temp.frame, view)
      condition source.frame ≠ nil
            PlaceUnits(temp, view)
            DisplayFloor(temp, view.output)
      PopUpMenu((selection,'Refresh/Commit/Undo/Close'), (view, blueButton))
      condition selection ≠ 'close'
            Equality(source, temp)
      condition selection = 'close'
            Succeed
      condition selection = 'commit'
            Transmit
      condition selection = 'undo'
            Fail
end
```

```
/* filter type for PlaceUnits, units can be selected with a popup menu  */
/* and are then added to the appropriate list                           */

Filter Type PlaceUnits(source: Floor, view: Workstation)
var
       selection → String
       fpo → Rectangle
make
       PopUpMenu((selection, 'Wall/Door/Window'), (view, redButton))
       GetObject((floor, fpo), view)
       condition selection = 'Wall'
              AddToList(source.floorPlan.walls, fpo)
       condition selection = 'Door'
              AddToList(source.floorPlan.doors, fpo)
       condition selection = 'Window'
              AddToList(source.floorPlan.windows, fpo)
end

/* filter type for DisplayFloor, all units are rendered onto the display */

Filter Type DisplayFloor (source: Floor, view: OutputMedium)
make
       Render(source.frame, view)
       iteration source.floorPlan.walls.size times i
              Render(source.floorPlan.walls[i], view)
       iteration source.floorPlan.doors.size times i
              Render(source.floorPlan.doors[i], view)
       iteration source.floorPlan.windows.size times i
              Render(source.floorPlan.windows[i], view)
       iteration source.floorObjects.desks.size times i
              Render(source.floorObjects.desks[i], view)
       iteration source.floorObjects.chairs.size times i
              Render(source.floorObjects.chairs[i], view)
       iteration source.floorObjects.closets.size times i
              Render(source.floorObjects.closets[i], view)
end

/* filter types for GetObjects, AddToList and Designer are included  */
/* here to show their respective source and veiw types               */

Filter Type GetObject (source: (Floor, Rectangle) view: Workstation)
make
       ... get rectangle from user that does not overlap existing structure

end
Filter Type AddToList (source: List, view: Element)
make
       ... adds element to list
end
Filter Type Designer (source: Floor, view: Workstation)
make
       ... analagous to Architect
end

/* other filter types and filter atoms that are not mentioned here are  */
/* provided by the implementation.                                      */
```