Siri:
A Constrained-Object Language for Reactive Program Implementation

Bruce Horn

June 1991

CMU-CS-91-152

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Siri is a small language intended to simplify the design and implementation of programs with graphical user interfaces such as those available for the Apple Macintosh. The interfaces for such programs are usually based on three concepts: accurate and appealing visual metaphors, interaction by direct manipulation, and immediate reflection of changes in the data. Programs based on these concepts are called reactive, after Alan Kay's Reactive Engine [Kay69]. A well-designed reactive program maintains an internal model that is kept consistent with an end user model. The end user's model is created and modified by direct manipulation, using an input device such as a mouse to alter a visual representation. This puts the user in control, rather than the computer.

Implementing reactive programs is difficult, due to computational mechanisms that are rarely supported in existing programming languages. Siri addresses these needs in a simple and uniform way. The declarative nature of Siri bypasses many issues of control flow; dependency networks and constraint satisfaction will transparently sequence updates. Siri's single abstraction mechanism, the constraint pattern, supports multiple object views simultaneously; separate hierarchies for visual layout, encapsulation, and class/subclass; and multiple concurrent objects. In addition, Siri will provide a straightforward framework for creation and composition of Siri programs using direct manipulation.

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

Keywords: Programming languages, object oriented programming, user interfaces, constraints, rewriting systems, incremental computation.

Table of Contents

1. Introduction

1.1 Reactive Programming

The widespread use of advanced user interfaces has raised the expectations of program users to new heights. Personal computers, such as the Apple Macintosh [Apple85], provide highly graphical, interactive programs with easy-to-use, intuitive interfaces. The best of these interfaces are based on three basic concepts: accurate and appealing visual metaphors, consistency maintenance, and interaction by direct manipulation. Programs based on these three concepts are called reactive programs, after Alan Kay's Reactive Engine [Kay69], one of the first systems which combined these concepts.

The goal of the Siri project is to design and implement a simple new programming language which embodies features that were found to be useful in the implementation of reactive interfaces. Siri is primarily declarative, which will help bypass many issues of control flow in reactive interface implementations. In addition, dependency networks and constraint satisfaction transparently sequence updates to the system. Siri's single abstraction mechanism supports multiple concurrent object views; separate hierarchies for visual layout, encapsulation, and class/subclass; and multiple threads of control. While some of these□□□□□dres are provided by existing special purpose systems, the systems were typically unsuitable for general programming requirements due to their complexity or narrowness of focus.

1.2 Origins

Reactive programming originally began with the use of computers for real-time simulation, in particular, for flight simulators. However, these simulators had a built-in model: the flight parameters of the aircraft and the simulated area that the airplane would be flying in, and the model was not changeable by the user. The first true reactive program was Sketchpad [Sutherland63]. This program pioneered many of the techniques used today: object templates, clicking and dragging to move and change objects, and so on. Sketchpad was well ahead of its time, partially due to the advanced hardware it used; other systems would not duplicate Sketchpad's functionality for over 15 years.

Doug Engelbart's Augment system [Engelbart62] was one of the first reactive text and idea processors; the original mouse was designed for use in this system, and was used to point to entities and commands on the screen. Building on this work, Alan Kay's Reactive Engine introduced the concept of a computer that would maintain a model that the user developed, and which would react to the user's actions rather than prompt the user for input. This way of looking at computation put the user in control, rather than the computer, which was a necessary presupposition for the design of direct manipulation interfaces. Kay's vision of a Dynabook, a portable computer that would serve the user's information, design, and modeling needs, was designed with these principles in mind. The Interim Dynabook, more widely known as the Alto [Thacker79], was an early platform for experimenting with these ideas.

The Smalltalk system was intended to be the foundation on which the Dynabook was built. It was one of the first computing environments to provide overlapping windows to simulate sheets of paper on a desk, with modeless, direct manipulation text editing. The Smalltalk language itself, being object oriented, was particularly suited to the implementation of direct manipulation concepts. In addition to overlapping windows and modeless text editing, Smalltalk pioneered the BitBLT graphics primitive (Bitwise BLock Transfer, also known as RasterOp), popup and hierarchical menus, analog scroll bars, browsers for navigating through hierarchies, object inspectors, document-based multimedia editing, and more.

The Xerox Star office system [Smith82] first offered the desktop metaphor in a commercial system, where documents appeared as non-overlapping sheets of paper on a desk, with in- and out-boxes, printers, and other office objects represented as icons. Star was only partially direct manipulation: the mouse was used to point out objects of interest, while keys on the keyboard were used to specify operations such as move, copy, and so on. Star provided document based editing, in which graphs, pictures, tables, and such were edited directly in the document itself, rather than pasted in from a graph- or table-building application as is done in other systems.

The Apple Lisa and Macintosh continued to bring the ideas from Xerox PARC into the commercial world. These machines provided the desktop metaphor, direct manipulation of documents and applications, modeless text editing, overlapping windows, and pull-down menus for selecting commands. While not as advanced as some of the systems produced at Xerox PARC (they did not have document-based editing, for example), the Apple systems were optimized for small machines, and as such provided higher performance.

1.3 Reactive Programming Is Difficult

The design and implementation of reactive interfaces can be quite complicated. Visual displays must be laid out in aesthetically pleasing and consistent ways; objects must behave as closely to their intended metaphors as possible; changes must be incorporated quickly, and the consequences of those changes reflected immediately; and the user must never feel stuck in a mode, where actions have surprising and undesired results.

There are several reasons that the implementation of such systems is difficult. The most important include the need for a large set of basic objects, user-centered design, mapping user actions, and global consistency maintenance.

Basic Objects

To implement a reactive interface, many different basic objects are required, such as buttons, scroll bars, dialog boxes, windows, menus, and so on, as well as a number of interfaces for manipulating these objects. The functionality of these objects is often fairly complicated, and their interactions with the program itself more complicated still. Gone are the days when user interfaces consisted solely of text written to the terminal, and text input by the user.

User Centered Design

By designing the program around the user, rather than forcing the user to respond to the program, program implementers must anticipate everything that a user might want to do in every context. Instead of prompting the user for input in a program-driven interface, the program must maintain an internal model which corresponds to the model presented to the user, and allow the user to modify it in a variety of direct ways by clicking, dragging entities, typing, and so on. The structure of user-oriented programs tends to be very different from the structure of typical, prompt-then-act programs, and so it can be hard to fully implement the reactive model.

Mapping User Actions

In addition, since the program must react to actions rather than simply prompt for input at well-defined places in the program, they can be difficult to interpret, since the actual meaning is dependent on a large number of different pieces of information. The current window, the selected object or objects, the state of the program, the mouse location, the action itself, and many other context markers are typically used to determine the meaning of a particular action. The programmer must not only take all of these context markers into account, but must also change the context information appropriately when necessary: selecting and deselecting objects, changing the active window, and so on.

Global Consistency Maintenance

Finally, because different parts of the program can be interrelated in surprisingly intricate ways, maintaining consistency of the program model during changes can be a challenge. In reactive interfaces, one of the most important abilities a program can have is to display information in several forms, or views. People gain an understanding of a concept through seeing it in different ways, and in different contexts; showing multiple views of an object simultaneously improves the user-computer communication dramatically. For example, modifying a table in a spreadsheet must cause dependent cells to be recomputed; charts to be redrawn with possibly new axes, entries, and data; menu choices to be enabled and disabled depending on the current selection; scroll bars must reflect the location of the insertion point; and so on. There can be hundreds of dependency relationships in a typical reactive program, and these must be specified directly. Because of this, the likelihood that a particular dependency is overlooked can be high.

1.4 Approaches For Supporting Reactive Interfaces

These difficulties have led many designers and researchers to support graphical and reactive interfaces in various ways, including user interface toolboxes, application frameworks, user interface management systems (UIMS's), and comprehensive user interface design systems.

Toolboxes are libraries of useful user interface routines and prebuilt objects; examples of toolboxes include the Macintosh toolbox [Apple85], the Andrew toolkit [Palay88], the X Window System and X Toolkit, and Interviews [Linton89]. Their graphics support components usually include routines for drawing rectangles, circles, lines, polygons, text, and arbitrary regions, and clipping to areas on the screen. The toolbox proper supports creation and manipulation of textual and graphic menus; controls such as scroll bars, buttons, checkboxes, scrolling and type-in fields; windows and dialog boxes; and mechanisms for handling user and system events, such as mouse movement, clicking, and keyboard input. More advanced features can include support for multimedia documents with dynamic loading and linking of code, as in Andrew, and special facilities for laying out objects, using "glue" to arrange them relative to their parents, as in Interviews. Toolboxes are very flexible; they deliver a constant factor improvement in the development of interfaces, by providing the basic objects and action classification mechanisms. However, they typically lack support for reactive programming in that they do not provide a structuring for user-centered design; it is unfortunately still as easy to write a non-reactive interface as a reactive one. In addition, extensions to the toolbox are difficult, and any actions that fall outside of the domain of the predefined window or menu objects must be handled by the programmer without help from the toolbox. Finally, there is no mechanism in most toolboxes for helping the programmer maintain global consistency; this must be specifically implemented by the programmer.

More ambitious systems address these problems by providing a complete application framework which define the user interface interaction, display layout, control flow, and available tools in a more rigorous fashion. One of the first such systems was Apple's MacApp, followed by systems such as ET++ [Weinand88], and UniDraw [Vlissides89]. These systems relieve the programmer of tedious event handling, and instead leave open points in the application where the programmer may define a specialized routine to be executed when a particular condition occurs. For example, if a window needs to be refreshed, an application framework would call the programmer's window update routine. (MacApp's motto is "Don't call us; we'll call you.") In this sense, frameworks are inverted toolboxes, in that they maintain control and only pass it to the programmer's routines when required. One drawback is that they often restrict the programmer to a single approved interface. If the programmer needs to extend this interface in any way not anticipated by the original designer, the entire structure of the framework must be understood, in particular the details of the control flow, which can be a daunting task. Some systems do extend the framework to special classes of reactive programs. For example, Unidraw is designed for the implementation of object-oriented layout programs. Even so, there will always be a frontier beyond which the programmer will need to provide routines for handling primitive events in the extended objects, without help from the framework. Furthermore, global consistency maintenance is still not considered to be part of application frameworks, and must still be implemented through explicit code by the programmer.

User Interface Management Systems (UIMS's) are programming systems in which the application code itself, and the code for the displaying of information, are kept separate. Systems such as Sassafras [Hill86] and the Apollo ADM package [Schulert85] are representative UIMS's. How the information is displayed is usually specified at a higher level of abstraction than the application itself, in order to allow different interfaces to be built for the same application. UIMS's, by design, limit the kinds of interfaces that can be built, and they are difficult to upgrade when new interface techniques become available. Their main disadvantage, however, is that they provide little or no support for reactive interfaces; the tight coupling required for most direct manipulation interactions precludes the use of a UIMS.

Finally, complete integrated systems, combining a toolbox, a mechanism for maintaining global consistency, and an application framework are beginning to be developed. These systems attempt to provide for all of the needs of reactive program implementation; Garnet [Myers89a] is one of the most comprehensive. Garnet's predecessor, Peridot [Myers87], allowed the user to specify interfaces by example, creating constraint code as the user "drew" the menu, dialog box, or scroll bar. Some systems provide additional support for the programming process itself; Garnet in particular provides a sub-system, Jade, for automatic, style-independent layout of dialog boxes.

1.5 Programming Language Support

Because of the structure of reactive interfaces, the process of building a reactive interface can be made easier by programming languages which are based on certain key concepts: object oriented structuring and constraint management.

Object Oriented Programming

Ever since the Smalltalk language [Goldberg83] and environment [Goldberg84] provided a graphical user interface with an object oriented implementation, designers realized that an object oriented language supports the graphical, entity-based model of interaction more directly than a conventional language. Objects on the display are represented simply by object definitions in the language, and user actions map to messages sent to each object. The noun-verb model for direct manipulation (first select an object, then choose an action) is modeled naturally by object oriented programming, since each object knows itself how to perform the requested action.

Object oriented programming has other advantages as well. The class/subclass hierarchy allows programmers to implement the different kinds of objects presented to the user, while sharing behavior specifications between similar objects. For example, desktop objects in the Macintosh Finder all have several visual representations (icon, small icon, or textual); they can all be moved within windows, dropped into new windows, or dropped into container icons; and they all can be opened, closed, their contents printed out or duplicated. Each type of icon is slightly different from the class of icons in various behaviors, however: some icons, such as folders, disks, and the trash can, are receptacles which can have other icons dropped into them; trash cans can't be duplicated or moved off of the desktop; and each icon displays different information depending on its type. Still, they all share to a great extent the particular behavior of desktop objects, and this common behavior is easily implemented in a single class in an object oriented language. The MacApp application framework takes particular advantage of the class/subclass hierarchy, implementing a variety of predefined classes for programmer use.

Constraint-Based Programming

Because of the need to maintain consistency between parts of a program, there has been quite a bit of activity in constraint systems and languages for graphics and reactive interface support. Sketchpad implemented the first direct manipulation user interface based on constraints. It provided the ability to draw lines with certain constraint relationships. The user could also assemble lines into an aggregate object, and make copies of that object, while changing some of its attributes, if desired. This aggregate was a prototype for its instances; one could edit the prototype, and each of the instances would change appropriately. Sketchpad's bidirectional constraints were solved numerically, but the system was not extensible; only the built-in constraints, and compositions of those constraints, could be handled.

ThingLab [Borning79], a constraint-oriented, direct-manipulation simulation laboratory built on top of Smalltalk, provided a language and a graphical interface for describing collections of related objects. Originally it did not directly address the user interface issue, although Thinglab is currently being extended for that use. ThingLab's constraints are bidirectional, but the programmer must define satisfying methods for each direction manually. Borning extended this work by integrating the constraint and functional programming paradigms in [Borning85].

Other languages were developed and demonstrated to be useful for simple user interface tasks. IDEAL [VanWyk80] used equation solving via graph reduction for the satisfaction of constraints on hierarchical objects; it was designed for layout of graphics in documents, and as such ran only in batch mode. The GROW system [Barth86] provided constraints as an intrinsic part of an object-oriented, graphical user interface toolkit. CONSTRAINTS [Sussman80] provided for the specification of hierarchical constraint equations for electrical circuits using local propagation. CONSTRAINT [VanderZanden88] is a language with which graphical objects and subobjects may be displayed and modified, while satisfying a set of constraint equations, ordering these equations for optimal update to ensure real-time interaction. Coral [Szekely88] is a declarative language for declaring aggregate objects and expressing unidirectional constraints between those objects. Part of the Garnet system is a unidirectional constraint language called KR built on Common Lisp. Bertrand [Leler88], on which Siri is partially based, is a general-purpose constraint language, and has been used for the specification of pictures, but it is not interactive, and has not been used for the implementation of user interfaces. Finally, Kaleidoscope [Freeman-Benson90] is the first of the constraint-imperative programming (CIP) languages, of which Siri is a member. Kaleidoscope allows the programmer to freely mix imperative statements with constraints that are to be maintained by the system.

Besides easing the job of maintaining relationships between objects, these languages have the advantage that constraint-based code is often more concise and easier to read than the corresponding code in a non-constraint language. However, these systems had several drawbacks: they were usually not general purpose languages, usually having to drop down to their host languages to compute things that they cannot; some only provided unidirectional constraints; they were often interpreted, and thus slower than desired; and they were too complicated for production use, and difficult to integrate into practical applications.

1.6 Summary

Each of these systems supports important requirements in the design and implementation of reactive user interfaces. An application framework written in an object oriented language with a constraint mechanism, including a complete graphical toolbox, would appear to be the ideal. Unfortunately, using an existing programming language with which to build such a system can lead to excessive complexity, due to the lack of support for the intrinsic requirements of reactive interfaces.

There are other ways which have not yet been explored that could accelerate the development of reactive interface. For example, one would also like to use direct manipulation in the programming process itself. Some systems provide a limited facility for programming using direct manipulation (such as the Macintosh's ResEdit, and dialog layout systems like NeXT's Interface Builder). Garnet's Lapidary subsystem [Myers89b] allows the definition of graphical as well as behavioral aspects to be specified using direct manipulation. Languages such as Juno [Nelson85] allowed the programmer to specify picture definitions using constraint mechanisms; Juno in particular provided for editing of a textual description of the picture as well as a graphical display, but was limited to a fixed set of primitive graphical objects and operations. The Miró system [Heydon89] is used to specify security constraints using direct manipulation of a graphical representation of the security relationships. These tools are typically restricted and specialized. In particular, only a small part of the programming process can be addressed, and they are only used by the programmer at development time, not by the program itself at runtime.

More directly, one could interpret the actions of the user as specifying or modifying a piece of a program. For example, Adobe Illustrator generates a PostScript program from direct manipulation of graphical entities, including text strings, splines, rotated, scaled and stretched objects, and so on, although Illustrator does not interpret PostScript itself. Similarly, drawing in MacDraw, which involves the creation and modification of objects and the application of simple constraints to those objects, can be considered programming; dragging out a rectangle and choosing a pattern from a palette is equivalent to writing a piece of code which specifies a rectangle, its size and location, and its pattern. If the language which implemented the MacDraw application could also be used recursively to specify, and interpret, the actual document objects and relationships as well, then much duplicated work could be avoided.

2. Project Goals

There are many different design and implementation principles which need to be followed in order to produce a good reactive program. Many approaches have been used to make reactive program development easier and faster: user interface toolboxes, application frameworks, UIMS's, and integrated systems. By providing a new model of computation for reactive programs, and a programming language that supports this model, many of the features of reactive programs can be integrated directly into the programming language itself; this is the major goal of this project.

2.1 A Reactive Programming Model

In most reactive programs, computation can be modelled by a change from one complex consistent state to another consistent state. Most direct manipulation interfaces are written using a model like Alan Kay's Reactive Engine, in which the user causes an event to occur, after which the system makes itself consistent while incorporating the new event. For example, a text editing system takes editing events (including keystrokes and menu selections) and incorporates them into the document, making the document consistent with respect to invariants such as line breaks, page size, line height, line width, word hyphenation, and the like. Events can be extended to include inputs from other sources than the user, such as the network, non-user I/O devices, and so on, allowing this model to suffice for general computing requirements.

Object-oriented programming lends itself well to the event mechanism for handing user interaction. The messages defined for an object correspond to events that change the state of a system. In particular, user interface entities on the screen, such as icons, windows, or menus, are typically coded as programming language objects in a straightforward manner, and events caused by the user are mapped directly to object messages. By designing a system as a hierarchy of objects, all based on the message-passing/event fielding model, the system as a whole can be driven by high-level events, which cascade down the object hierarchy.

The dependency maintenance requirements for reactive interfaces suggest that a program can also be viewed as a database of objects with particular relationships to each other. There are many kinds of relationships; however, they may be reduced to two major classes of relationships: form and content. Form relationship is defined by the language itself, in terms of how objects may be combined into complex entities. In an object oriented system, this typically includes such relationships as the structures of objects (what subparts each object has) and the class-subclass hierarchy. People who work with database systems are typically concerned with form specification and consistency, and several mechanisms have been created for dealing with this problem, most notably NIAM (Nijssen Information Analysis Method). Content relationships are those where particular, programmer-defined relations must hold (text editor invariants, etc.); these are of particular interest in reactive systems.

Informally, a system is consistent if all of these relationships are satisfied. A system has integrity if consistency is maintained under all defined perturbations and modifications of the system. Events perturb the system, indicating a change that should be made; the effect of computation is to re-satisfy the system, making it consistent again, after such an event.

Unfortunately, object oriented languages (much less non-object oriented languages) do not support this model well. When an object receives an event, it must accept the event and, if necessary, change its internal state to maintain consistency. Since objects consist of state that is maintained by a distributed set of methods or object access routines, any content consistency requirements that may exist for the relationships between the various parts of the object must be encoded in the methods. Each method, then, has code that exists specifically to do whatever is necessary to keep the object internally consistent given the particular event that the method handles. Assertions (as well as pre- and post-conditions) are sometimes considered as mechanisms for checking object consistency, but they are defined on a method level, and thus the actual consistency specification remains distributed among all the methods of the object. In addition, it is quite possible to define a method that only partially updates an object; while atomic in use, it is not atomic in effect, and thus leaves the object in an inconsistent state.

To make matters worse, the object consistency requirements are rarely specified formally, or even written down informally in object comments (which are seldom kept up to date, in any case); nevertheless, these requirements are coded in the methods, and any programmer modifying the behavior of the object must know them. The fact that the consistency maintenance is distributed among the methods of the object's description makes it quite often difficult to maintain. In addition, there is no centralized declaration of the invariants that the object maintains, and often the invariants are incorrectly implemented in the methods. The language Eiffel [Meyer88] makes steps toward resolving these problems; in addition to pre- and post-conditions, Eiffel has class invariants, statements that describe certain consistency requirements on instances of the class, which must be preserved by every method. However, these invariants are optional, as are the pre- and post-conditions, and they are only used in the debugging phase.

Another aspect of consistency and integrity involves the system dependency relationships. For example, there may be arbitrarily many different graphs of different parts of a spreadsheet, which are objects in their own right, as well as other dependent objects which are created at run time by the user. Although this dynamic structuring is in some sense "outside" of any object, the dependency relationships are still there, and they must be maintained to keep the system consistent. Some systems provide facilities for support of dependencies; in particular, Smalltalk's Model-View-Controller metaphor is used to separate the object itself (the model) from how it is displayed (the view) and modified (the controller). The communication from the model to its dependencies, typically the views, is done manually through a dependency list in the object. When a model is changed, it notifies all dependents to change if necessary, by traversing a dependents list and sending each object on the list a changed message. The dependents have the responsibility to determine how much has changed, and how to update as efficiently as possible.

2.2 Supporting The Model

In order to support this model of reactive program implementation, there needs to be a language which is simple and complete. It must be object oriented, and needs to support constraints within objects to maintain internal object consistency, and between objects to maintain global dependency relationships. It should be usable for the implementation of the required toolbox of interface objects, and the execution speed should be adequate for most user interface applications. The turnaround time should be fast, so as not to interfere with the design process. Finally, the language should be flexible enough to allow program definition and modification at runtime, to support the building of direct manipulation tools for the programming process itself. Given a programming language which implements these features, reactive programming can be done much more easily, and faster.

2.3 Siri

In this project, I plan to implement these requirements by distilling the best ideas from previous systems into a declarative programming language called Siri. Siri is an object oriented language with a straightforward, event based, multiway constraint satisfaction mechanism for consistency maintenance. While other systems have provided constraint satisfaction, this feature is typically only used in a small part of the system having to do with the display of objects on the screen, and the set of objects which can be manipulated by the system usually has been a small fixed set. Consistency maintenance in a system requires more than just constraint maintenance of display objects, and as such should be available uniformly. By encapsulating the fundamental structure of direct manipulation interfaces in a programming language, the concepts are more widely available in the programming process, and more easily communicated to, and used by, a programmer. One goal is to make it easier for a programmer to build a direct manipulation interface in Siri than an old-fashioned, prompt-then-act interface.

Simplicity is important in a language, for many reasons: first, it is easier to implement a language with a few powerful features, and second, it will be easier for a programmer to learn a smaller set of language concepts. In order to help support the concept of programming reactive applications with as simple a language as possible, Siri provides a single abstraction for object description, modification, and evaluation, called a constraint pattern. All kinds of objects will be created and modified in the same way using constraint patterns. Scroll bars, dialog boxes, other graphical aggregates, and entire documents, will share the same containment structure and modification mechanisms; the code to support building a dialog box will be exactly the same code as needed to support drawing figures in a document. As in Juno, the act of creating an object within a dialog box will cause corresponding Siri code to be created and added to the dialog box code. The semantics of that object can be specified either by using tools in the direct manipulation interface, or by programming in Siri directly. Changes made using direct manipulation will be visible in the Siri code, and conversely, changes in the Siri code will be reflected in the dialog box display. Dependency relationships between objects will be displayed graphically as well.

Siri's implementation is based on concepts from several different systems: Bertrand [Leler88], a batch constraint system which introduced the concept of augmented term rewriting; Beta [Kristensen89], a programming language which consists of a single object abstraction, the pattern; Linda [Gelernter83], which demonstrated the utility of communication through tuple spaces; and incremental computation mechanisms, demonstrated by Pugh [Pugh88] in his thesis. Siri is a symmetric language, meaning that execution occurs over the entire space of a program simultaneously, and that the same structures that are used for data encapsulation are also program structures; it is similar to Symmetric Lisp [Gelernter89] in this respect. These concepts are unified in the constraint pattern mechanism.

Siri will be a hybrid system, partially interpreted and partially compiled, which will allow fast turnaround along with reasonable performance. Constraint patterns will be able to be defined, modified, and evaluated at runtime by other constraint patterns, to ease the implementation of visual programming tools.

Finally, Siri will provide for multi-threaded control and concurrent objects, which will enhance the flexibility of Siri interfaces. Often, in current systems, once a dialog box comes up, the user has no choice but to reply to the dialog box, even though other tasks could be performed; other continuously-executing actions, such as animations, typically come to a stop. One exception is Squeak [Cardelli85], which is a special purpose language for handling events, such as key strokes or mouse clicks, using multiple tasks which execute concurrently. Squeak is not a general purpose language; it compiles the parallel specifications into serial C code, which is then linked in with a host application. Siri's declarative nature supports this distributed model, allowing separate processes to handle primitive events locally, while communicating anonymously through shared constraint patterns in a manner similar to Linda tuple spaces.

Smalltalk demonstrated the power of designing an entire system based on a single idea: uniform object-oriented programming. The idea of a constrained object system is also very powerful; in the Siri system, the entire programming environment, including the toolbox, will be written in Siri, taking advantage of the event based constraint satisfaction model. The declarative nature of the system will allow the programmer to bypass many of the details of control flow in the system, leaving them to be handled by the constraint satisfier and dependency mechanism.

Siri is a general purpose programming language; as such, anything that can be written in any other language should be able to be written in Siri. Siri's constraint mechanism will make it easy to develop a large class of WYSIWYG programs. However, computationally intensive WYSIWYG programs, such as text formatters and editors, may not fit into the Siri model. Although one can write a sequence of constraints that define how text should be laid out on a page, the number of constraints to be satisfied (potentially several per character) may be too large to evaluate efficiently.

3. Programming with Siri

Programming in Siri consists simply of specifying objects in an object-oriented manner, with internal consistency constraints, and event patterns by which objects may be changed. Although Siri is first a constraint system, it is not stateless like many constraint systems are; each object can have persistent information, and it can be modified by the receipt of events. Information flows in two ways: directly by sending events from object to object, and indirectly through the dependency matrix. Programming in Siri is similar to programming a spreadsheet with macros: the spreadsheet provides a constraint matrix, and the macros provide events which perturb the spreadsheet and cause it to be resatisfied.

Siri's syntax is based on that of Bertrand, but generalized and extended. Many of the following examples will be recognizable to anyone familiar with Bertrand.

Everything in Siri is implemented with a single abstraction mechanism, called a constraint pattern.

3.1 The Constraint Pattern

A constraint pattern consists of an optional label, the pattern body, and an optional type specifier:

```
label: { body } 'type;
```

The body is a group of expressions which evaluate to an object, which is assigned the given type. The expressions live inside the pattern, or in pattern space. All the expressions in pattern space are evaluated; if an object results, then the pattern has as its value the object. Labels are used to refer to objects which are evaluated in this way, as well as to refer to the pattern itself. Thus, evaluation of expressions consists of repeatedly finding labels which represent objects to replace with the actual object. For example,

```
myNumber: { 3 + 4 } 'number;
```

evaluates `3 + 4` to an object of type `'number`. The expression `myNumber+myNumber` would evaluate to `7+7` initially by replacing the label `myNumber` with its value; the final addition occurs as a primitive operation.

Types in Siri are optional tags on objects, which can be used to specify what kinds of objects are allowed in particular contexts. An object may have several types. In Siri, types are predicates, and they do not partition the set of objects.

If expressions in pattern space are labeled constraint patterns themselves, they are evaluated in parallel. Their names are visible to each other, and objects outside of the pattern space can refer to them via the enclosing pattern's label. For example, the constraint pattern

```
myThreeSums:  {
      firstSum:   { 3 + 4 };
      secondSum:  { 5 + 6 };
      thirdSum:   { 10 + 2 };

      firstSum+secondSum+thirdSum
};
```

evaluates to an object whose value is `30`, but which also has three sub-objects with values `7`, `11`, and `12`. They can be referenced from outside the object by using the dot operator to access objects within the pattern space; `myThreeSums.secondSum` is the object `11`. We can add the first two sums in `myThreeSums` by simply entering the object using the dot operator: `myThreeSums.(firstSum+secondSum).`

Labeling an existing object creates a copy of that object for local use. For example, the primitive object aNumber is defined to be a number which is not specified. Writing `x: aNumber` simply creates an instance of a number, without setting `x`'s value; further constraints will be required to determine `x`. We can define the sum of three unspecified numbers by just declaring them each as `aNumber`, and then specifying the sum:

```
aTriSum:  {
      a, b, c: aNumber;
      a+b+c
};
```

We can then use `aTriSum` as an underconstrained object in further constraints:

```
test: {
      x: aTriSum;
      x.(a=10; b=12; c=4); x
}
```

`test` then has the value `26`.

3.2 Operators

A label is in fact a nullary operator. Labels may also accept one or more parameters. A parameterized label consists of operators and parameter specifiers, which are variable names appended with (again, optional) types. The variables in the parameter specifier are typically referenced in the pattern body.

Siri allows definition and use of arbitrary arity operators, prefix, postfix, or infix. An operator may be explicitly defined as follows:

```
_+_: anOperator 200 leftAssociative commutative;
```

The underbars specify the location of parameters, while the arguments to `anOperator` specify precedence, associativity, and commutativity. Once `+` is defined, we can use it in a pattern:

```
x'integer+y'integer: { ... };
```

This pattern accepts two integers, and (presumably) returns their sum. In reality, the constraint pattern evaluates to an object immediately. However, because `x` and `y` are not specified, the object is underconstrained; it will evaluate to the sum of its two parameters when they are provided.

Not all operators need to be explicitly defined. If it is clear by the use in a pattern's body what an operator's usage is, and it is used consistently in that way, a definition will be automatically generated. For example, we could rewrite `aTriSum` to take 3 parameters and return their sum:

```
aTriSum first'number second'number third'number:  {
      first+second+third
};
```

Siri will assume that `aTriSum` is a prefix operator, with arity 3.

If we wanted to keep state in the object, we could define `aTriSum` as

```
aTriSum first'number second'number third'number:  {
      a, b, c: aNumber;
      a = first; b = second; c = third;
      a+b+c
};
```

which would allow us to reference `aTriSum.a`, `aTriSum.b`, and so on.

3.3 Functions

Functions are easily specified using constraint patterns with parameterized labels. Since constraint patterns can reference themselves, we can perform recursion:

```
factorial n'integer: {
      if n=1 then 1 else n*(factorial (n-1))
};
```

Here, `factorial_` is a prefix unary operator, and `if_then_else_` is an intrinsic Siri pattern.

3.4 Constraint Satisfaction by Equation Solving

The expressions in a pattern space are objects related by operators. A special operator, the semicolon, indicates that the previous expression should be asserted as a constraint. The last expression, if unasserted, is returned as the value of the constraint pattern. Expressions in pattern space are evaluated by matching with labels of patterns (whether underconstrained or nullary) and then replacing the matched subexpression with a copy of the computed object.

Siri's intrinsic patterns provide basic algebra simplification and solving reductions, by rewriting complex algebraic expressions to simpler, ordered expressions. Matching on semicolons is used to determine what is being asserted, and what result is required. For example, one of Siri's intrinsic patterns is

```
a*x+b = 0; x: { a != 0; -b/a };
```

This pattern matches any assertion of the form `a*x+b = 0`, where the result required is `x`. To solve this equation we assert that `a` must not be equal to zero, and return `-b/a` as the result. Using a labeled subpattern to hold result values, we could use Siri's equation solving patterns to evaluate the following:

```
simultaneousEquations x'number y'number z'number: {
   a, b, c: aNumber;
    2*a + 4*b - 3*c = x;
    4*a - 5*b + 4*c = y;
   -4*a +   b - 8*c = z;
};
```

No result is given. Instead, the answers are stored as state in the object, in subpatterns `a`, `b`, and `c`. To use this pattern we could write

```
computeEquations: {
      result: simultaneousEquations 8 2 -6;
      (result.a, result.b, result.c)
};
```

In the pattern `computeEquations`, we have declared a subpattern (an object), called `result`, which is a copy of the `simultaneousEquations` object with the additional parameters set as `8`, `2`, and `-6`. The object result is a group of three numbers which satisfy the simultaneous equations specified in the object `simultaneousEquations`. The value of `computeEquations` is simply the list `(1.85, 1.04, -0.04)`.

Note that Siri's constraint system saved us a lot of work here. In a non-constraint language, we would have had to write the equivalent of

```
simultaneousEquations x'integer y'integer z'integer: {
   a: { ((9/23)*x + (29/92)*y + (1/92)*z)/2 };
   b: {  (2/23)*x - (7/46)*y -  (5/46)*z    };
   c: { -(2/23)*x - (9/92)*y - (13/92)*z    };
};
```

which is much less understandable than the first representation, as well as more difficult to maintain and modify.

A simpler example is a temperature object, which can return its temperature in Fahrenheit or Celsius. It would typically be coded as follows (with double-dashes indicating a comment running to the end of the line):

```
aCFTemp: {
      --A temperature with Celsius and Fahrenheit values
      C, F: aNumber;    -- aNumber is a label which refers to a
      F = 9/5*C + 32;   -- number-creating constraint pattern.
};
```

In this pattern we have two state variables, `C` and `F`, and a single constraint that relates them. We can use this object to compute `F` given `C`: `aCFTemp.(C=100;  F);` or `C` given `F`: `aCFTemp(F=50; C)`, or make instances of it to store temperature values.

If Siri is unable to solve an equation, because it cannot match any reducible expressions, it is an error and is flagged by the system. This kind of error is similar to using a procedure without defining it in a conventional language. Often these kinds of errors can be resolved by adding a special constraint pattern that simplifies the expression that Siri was able to reduce to, allowing Siri to work further to solve the equation.

3.5 Prefixing Patterns

We can also use `aCFTemp` as a prefix to another pattern, to extend its functionality. If we want to also be able to compute temperatures in Kelvin, given we already have `aCFTemp` we can do the following:

```
aKCFTemp: aCFTemp {
      K: aNumber;       -- The additional subpattern
      C = K - 273;      -- and a constraint for K in terms of C
};
```

This pattern is semantically equivalent to

```
aKCFTemp: {
      C, F, K: aNumber;
      F = 9/5*C + 32;
      C = K - 273;
};
```

but it is simpler, and a better factoring, to have the two patterns separate.

Multiple prefixes may be specified as well; any subpatterns of the prefixes that have the same name are assumed to be equivalent. Say two patterns, `pat1` and `pat2`, each have a subpattern named `clyde`. If we prefix a new pattern, `patX`, with `pat1` and `pat2`, Siri will assume that the two `clydes` are the same, and will add the implied constraint

```
pat1.clyde = pat2.clyde;
```

to `patX`'s pattern. When Siri evaluates the pattern, this assertion will be checked; if in fact `pat1.clyde` is not equal to `pat2.clyde`, a contradiction will occur, and Siri will flag it as an error.

3.6 Prefixes As Class Definitions

A prefix can be considered to be a class definition, by defining the basic attributes of an object in a constraint pattern, while leaving degrees of freedom for each instance to constrain themselves. We could define points, a point equality operator, and lines in a way similar to the examples in [Leler88] as follows:

```
aPoint: {
   --Definition of a two-dimensional point.
   x, y: aNumber;
} 'point;

pt1'point = pt2.point: {
   pt1.x = pt2.x & pt1.y = pt2.y
};
```

Points are just two numbers, representing rectangular coordinates. They have no constraints relating them.

```
aLineSegment: {
   --Definition of a line, in two dimensions.
   pt1, pt2: aPoint;
   --Attributes.
   horizontal:    { pt1.y = pt2.y };
   vertical:      { pt1.x = pt2.x };
   length:        { sqrt ((pt1.y-pt2.y)^2 + (pt1.x-pt2.x)^2) };
   slope:         { if vertical then INFINITY
                    else (pt1.y - pt2.y) / (pt1.x - pt2.x) };
   angle:         { arctan slope };
   --Constraints.
   pt1 ~= pt2;
} 'lineSegment;
```

A line segment is simply two unequal points in the plane, with various attributes defined in terms of those points. Given generic points and lines, we could, for example, create a new kind of line segment (a subclass) that is always constrained to be diagonal:

```
aDiagonal: aLineSegment {
      angle = 45 | angle = 135 | angle = 225 | angle = 315;
};
```

We could also use prefixing directly, by creating instances of `aLineSegment` and further constraining it so that it is fully determined:

```
x: aLineSegment {
      pt1.x = 20; pt1.y = 100; length = 50; angle = 30;
};
```

This is very much like initializing the instance, but we are free to initialize it in any way we like, rather than just by setting the object state.

In Siri, superclasses (prefixes) may not be overridden, but only extended.

3.7 Sequencing

It is sometimes desirable to force a given execution ordering, in particular when dealing with the outside world. Siri provides a sequencing operator, _->_, which causes the first argument to be evaluated before the second.

3.8 Object Modification

Say we want to define an upright rectangle (horizontal and vertical sides), and a way to move and resize it. What is necessary is to define the object's subparts, how they are related, and how they are allowed to change when the object is moved and sized. For simplicity will define moving as moving the top left corner, and resizing as moving the bottom right; we will assume the input handling is taken care of elsewhere in the program. First we define line connectivity:

```
firstLine'lineSegment connectsTo secondLine'lineSegment: {
      firstLine.pt2 = secondLine.pt1;
};
```

This simply specifies that lines are connected to each other in a particular way. By connecting four lines and defining constraints and attributes, we get a rectangle:

```
aRectangle: {
      line1, line2, line3, line4: aLineSegment;
      width:      { line1.length };
      height:     { line2.length };
      area:       { width * height };

      -- Event patterns
      origin p1'point toCorner p2'point:> {
            line1.pt1 <- p1; line3.pt1 <- p2; };

      moveTo p'point:> {
            width is fixed; height is fixed; line1.pt1 is p; };

      cornerTo p'point:> {
            line1.pt1 is fixed; line3.pt1 is p; };

      -- Constraint expressions
      line1 connectsTo line2; line2 connectsTo line3;
      line3 connectsTo line4; line4 connectsTo line1;
      line1.horizontal; line2.vertical;
      line3.horizontal; line4.vertical;
} 'rectangle;
```

The rectangle consists of four line segments which are connected to each other, end to end, and they have requirements to be horizontal and vertical. `width`, `height`, and `area` are attributes defined in terms of the lengths of two of the lines.

We change the state of the rectangle by using one of three event patterns, which specifically bind values using the assignment operator, `<-`. Event patterns are indicated by the `:>` label.

First dragging out the rectangle sends the message `origin_toCorner_` which specifies its origin (the top left point) and its corner (the bottom right). These are then asserted to be bound in the rectangle. Siri then computes the appropriate values for the rest of the lines, given the rectangle constraints. Moving the topleft corner specifies a new binding for `line1.pt1`, but also states that `width` and `height` are fixed, and may not be changed. This reduces the number of degrees of freedom explicitly, so that the appropriate values can be computed for the change. Unlike other constraint systems, Siri has no heuristics for determining the "best" values under a particular perturbation, but requires the programmer instead to expressly pin those attributes that must remain unchanged.

3.9 The Graphical Model

Because Siri is primarily declarative, the graphical model shields the user from any need to know about the details of drawing graphics. Special constraint patterns are defined to automatically display the contained objects; any changes in the objects, such as color, position, and so on, are noted, and the objects redrawn. A 2 1/2D ordering is maintained for each object, and the pattern implementation takes care of only updating the visible portions of each object, clipping as appropriate.

To add an object to a window, a reference to that object is simply added to the window's constraint pattern. Updating occurs automatically, and the user is spared the details of when to draw an object, whether it is fully or partially obscured, and so on.

3.10 Concurrent Object Communication

Constraint patterns can also be considered as object repositories. In a way very similar to Linda, objects may be added to and removed from constraint patterns. Siri uses the existing label mechanism for this purpose: constraint patterns are used to match on objects and expressions in the shared pattern, to add, modify, or remove them.

3.11 Other Features

Basic boolean operations, numeric operators and types such as integers, complex numbers, rationals, and floating point numbers, and control structures are provided as intrinsic patterns in the Siri system.

4. Design Overview

Constraint patterns play multiple roles: they define a namespace for object references, they are a repository for subobjects, they provide for the description of composite objects, and they are used to evaluate other constraint patterns. Labels on constraint patterns specify what kinds of subexpressions in pattern space can be matched, and they are then replaced by the object specified by the constraint pattern. Thus the evaluation of a constraint pattern occurs by repeated matching and replacement; this is accomplished through Bertrand's augmented term rewriting mechanism.

4.1 Augmented Term Rewriting

All constraint patterns are evaluated, in no particular order, by reducing them using augmented term rewriting. Augmented term rewriting is standard term rewriting with the ability to bind values to variables within a namespace in each rule. In Siri, the role of a rule is played by a constraint pattern.

Labels and expressions are flattened into a string by traversing them in preorder, creating nodes that consist of an operator and an annotation which specifies how to find the next node in the tree. All constraint pattern labels are then compiled into tables, organized by pattern namespace contours. These tables will be used by the pattern matcher, which repeatedly matches against expressions in unreduced constraint patterns; an instantiator will replace the matched expression with the matching constraint pattern's evaluated object. A fast algorithm, such as Aho-Corasick, will be used for matching pattern labels to expressions. This algorithm allows them to be found very quickly: the time is proportional to the length of the expression being matched, and independent of the number of potential labels.

Several restrictions are made on the form of constraint labels and patterns. First, labels must obey strict left-sequentiality, in order to allow the fastest pattern matching algorithms to be used. As in standard term rewriting, constraint patterns must be restricted from introducing free variables, and their labels and bodies must satisfy similarity and non-overlapping restrictions in order to preserve confluence. These restrictions are checked by the Siri environment, when possible, and illegal structures flagged as errors when they are entered.

4.2 Linear Equation Patterns

As in Bertrand, the actual constraint satisfaction process is realized through a set of constraint patterns which transform equations into a linearized normal form using standard algebraic rules. The single assignment semantics of the augmented term rewriting system allow slightly-nonlinear, simultaneous equations to be solved, by binding an expression to a variable and replacing that variable's occurrences with its equivalent expression.

A constraint pattern may define several degrees of freedom; for each variable being solved for, all other variables are considered anchored, and a satisfaction method is executed which use the other variables as parameters. When changing a variable, the other variables are considered free (unless specifically anchored, using the fixed keyword); any or all variables can be changed in order to satisfy the object's consistency requirements.

4.3 Global Consistency Maintenance

When an object is changed, due to an event changing its internal state or a modification of its defining constraint pattern, it must not only resatisfy its own internal state, but also notify dependents that it has changed. Objects resatisfy their internal state by re-reducing their pattern with the changed part of the state held constant, leaving the others variable, again unless they are specified to be fixed.

In most cases, constraint patterns which are used directly in other constraint patterns are included in-line; when this is done, the complete details of the included pattern are available for optimization and re-evaluation of the composite pattern, and no communication is required. Any definitional changes cause the enclosing pattern to be re-evaluated. However, when a constraint pattern is used indirectly, through a reference, only the interface is available to the client, and no optimization is possible. In this case, the constraint pattern is considered a dependent which is notified of the change of its host object, and it is requested to resatisfy itself.

4.4 Incremental Computation

The rereduction process can be very expensive. Function caching and fast equality testing, mechanisms from Pugh's incremental computation thesis are used to minimize this cost; only those subpatterns which are affected by the change need to be rereduced. Dependents are notified by either traversing a known dependency tree (via references), or, if the structure is too large, by scanning the heap for objects which have references to the changed object. A special object format will allow the detection of object boundaries quickly while scanning the heap. This is rarely done; scanning is most often used for definitional changes.

4.5 Interpretation And Compilation

Execution occurs when a constraint pattern is reduced; the value of the constraint pattern is the object which results. Eventually, however, some low-level computation must take place. This occurs when primitive patterns, such as addition, match expressions in a constraint pattern. These primitive patterns can be either defined as an interpreter escape (a primitive instruction), or as in-lined machine code.

As a side-effect, reduction terminates either in a string of primitive instructions, which are implemented by the Siri kernel, or in a string of compiled machine code instructions, either of which evaluate to the constraint pattern's value when given parameters. The original pattern is maintained in case it is redefined or needs to be rereduced in a new context (another object, for example). Patterns can be defined for optimizing primitive or machine code sequences, by matching on instruction sequences and rewriting them to more efficient ones.

4.6 Open Design Issues

There are several design issues which remain open. First, input handling has not been defined as yet, and this promises to be a very interesting aspect of the design of the Siri environment. There are several different approaches for the design of the toolbox and program structuring; I intend to take advantage of Siri's unification of data and program structures to develop a unique and useful framework. Finally, there are issues still to be resolved in the area of dependency network optimization and the sharing of structures for space optimization.

5. Expected Contribution

The main contribution of this project will be an event based, constrained object language, Siri, for direct manipulation graphical user interface design and implementation. This language will be as simple as possible, while incorporating desirable features from user interface toolkits, application shells, and user interface management systems. While previous work addressed each requirement individually with separate modules or programs, Siri will address them all with a single language model. The unification of representation for objects, aggregates of objects, documents, and so on in a single abstraction provides a level of generality and adaptability not found in existing systems. In addition, Siri's straightforward mechanism for editing and composing these objects graphically will be useful in a variety of contexts, including the editing of graphical documents.

By distilling out the concepts used in previous systems, Siri will provide a clear example for basic reactive interfaces: not only what one must do in order to have an acceptable interface, but how that interface can be implemented in a transparent, understandable way. Finally, through the creation of an augmented term rewriting system with incremental computation techniques, Siri will demonstrate how large constraint-based systems can be efficient enough for practical use.

5.1 Criteria for Success

The main criteria for success will be a set of existence proofs. First, I plan to create a small, simple MacDraw-like object oriented drawing editor, written in Siri. This editor will include several additional features, such as a definable palette. However, its main feature will be its program size: I should be able to write the complete program over a period of only a few weeks. The editor will also demonstrate programming using direct manipulation, by building a parallel Siri program as the graphics document is edited. A very small enhancement to the program will result in a direct manipulation dialog box editor. Second, a file manipulation program similar to the Macintosh Finder will be developed. This program will test Siri's ability to map external structures to Siri structures at runtime. The final test will be the implementation of the Siri environment directly in Siri.

Although writing bad FORTRAN code is possible in any language, no matter how advanced, I hope that using Siri will encourage programmers to build reactive interfaces because it will be actually easier than building traditional ones. The Siri system should be learnable, and effectively usable, in a much shorter time than is normally required.

6. Status

The design of the Siri language is nearly complete, with preliminary implementation currently in progress. The environment design is proceeding in parallel with the implementation of the language. The Siri kernel and prototype environment is being written in Think C for the Macintosh. Once the basic system is functional, the environment will be ported back to Siri, with only the kernel routines left in C.

7. Summary

The design and implementation of graphical, direct manipulation programs is a difficult process. Many different approaches have been taken to make this process simpler and more direct, from graphical toolbox routines, to application shells, to complete user interface design and implementation systems.

In this project, the best ideas from the previous approaches will be distilled into a simple, complete programming language, based on the event/resatisfaction model of user interface computation. Embodying the appropriate concepts into a programming language will help implementers grasp the basics of direct manipulation programming more easily, so that they may create interfaces more quickly and with fewer errors.

Several powerful language concepts have the potential of making the process of programming direct manipulation interfaces much easier. From Bertrand, augmented term rewriting and ordered linear equations provide a simple and fast mechanism for constraint satisfaction, with the potential for compilation. From Beta, the idea of an object pattern as the single abstraction mechanism makes the design of aggregate objects more consistent and general. From Linda comes the tuple-space communication method, which simplifies the communication between concurrent objects. Finally, incremental computation techniques, developed by Pugh, allow a large system to be be modified quickly, by limiting redundant computation.

These concepts are unified in a small, simple, but powerful language, Siri. The goal of this project is to show that programming in Siri can make the design and implementation of practical, advanced user interfaces faster and easier. It will be possible to create Siri programs by using direct manipulation as well: just as editing a graphics document in Adobe Illustrator creates a Postscript program that images the graphic on a printer, editing a graphic in Siri will create a Siri program that specifies the objects; evaluating that program will display the objects in a document. This will be done by using the rewriting system to modify a document object directly, in response to editing actions by the user, while the incremental computation mechanism will be used to minimize recomputation after such edits. For specification tasks that cannot be done using the direct manipulation interface, the programmer will be able to edit the Siri program directly and see the changes on the display immediately.

It will be easy to define many threads of control which will execute simultaneously. Siri's declarative semantics will shield the programmer from the complicated flow of control that is typical of advanced user interfaces. These threads will communicate through a shared pattern which will act as a tuple-space, for objects to store and retrieve messages and other data. Again, this will be done using the rewriting system to provide a lookup mechanism in the shared pattern, and to add, change, and remove objects.

The Siri system is currently being implemented in Think C for the Macintosh. An eventual goal is to rewrite all but the kernel routines directly in Siri.

8. References

[Apple85] Apple Computer, Inc. Inside Macintosh. Addison-Wesley, New York, NY, 1985.

[Barth86] Barth, P. An Object Oriented Approach to Graphical Interfaces. ACM Transactions on Graphics 5(2):142-172, April 1986.

[Borning79] Borning, A.. Thinglab, a Constraint-Oriented Simulation Laboratory. Xerox PARC technical report SSL-79-3, Palo Alto, CA., October 1981.

[Borning85] Borning, A.. Constraints and Functional Programming. University of Washington Computer Science Department TR 85-09-05, September 1985.

[Cardelli85] Cardelli, L., and Pike, R. Squeak: A Language for Communicating with Mice. Computer Graphics:Siggraph '85 Conference Proceedings, 19, 3, 199-204, July 1985.

[Cardelli87] Cardelli, L.. Building User Interfaces by Direct Manipulation. Report 22, Digital Corporation Systems Research Center, Palo Alto, CA., 1987.

[Engelbart62] Engelbart, D.C. Augmenting Human Intellect: A Conceptual Framework. Summary Report, Stanford Research Institute, on Contract AF 49(638)-1024, October 1962.

[Freeman-Benson90] Freeman-Benson, B. Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. OOPSLA/Sigplan Notices 25(10): 77:88, October 1990.

[Gelernter83] Gelernter, D. Generative Communication in Linda. Yale University Research Report TR-294, 1983.

[Gelernter89] Gelertner, D., and Jagannathan, S. A Symmetric Language. Yale University TR YALEU/DCS/RR-568, May 1989.

[Goldberg83] Goldberg, A., and Robson, D. Smalltalk-80: The Language and its Implementation. Addison-Wesley, Menlo Park, CA., 1983.

[Goldberg84] Goldberg, A. Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, Menlo Park, CA., 1984.

[Heydon89] Heydon, A., Maimone, M., et al. Miró: Visual Specification of Security. Carnegie-Mellon University School of Computer Science, TR CMU-CS-89-199, December, 1989.

[Hill86] Hill, R.D. Supporting Concurrency, Communication and Synchronization in Human Computer Interaction--The Sassafras UIMS. ACM Transactions on Graphics 5, 3: 279-210, July 1986.

[Kay69] Kay, A. C. The Reactive Engine. Ph.D. Thesis, University of Utah Computer Science Department, September 1969.

[Kristensen89] Kristensen, B. B., Madsen, O.L, Møller-Pedersen, B., and Nygaard, K., 1989. Object-Oriented Programming in the Beta Programming Language. Norwegian Computing Center, Oslo, and Computer Science Department, Aarhus University, Aarhus, Denmark.

[Leler88] Leler, Wm. Constraint Programming Languages, Their Specification and Generation. Addison-Wesley, Menlo Park, CA., 1988.

[Linton89] Linton, M., Vlissides, J., Calder, P. Composing User Interfaces with Interviews. IEEE Computer, February 1989.

[Meyer88] Meyer, B. Object Oriented Software Construction. Prentice Hall Inc., Englewood Cliffs, NJ., 1988.

[Morris86] Morris, J., Satyanarayanan, M, et al. Andrew: A Distributed Personal Computing Environment. Communications of the ACM, 29, 3, March 1986.

[Myers87] Myers, B. Creating User Interfaces by Demonstration. Ph.D. Thesis, University of Toronto Computer Science Department, TR CSRI-196, 1987.

[Myers89a] Myers, B., Giuse, D., et al. The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp. Carnegie-Mellon University School of Computer Science TR CMU-CS-90-117, March 1990.

[Myers89b] Myers, B., Vander Zanden, B., Dannenberg, R. Creating Graphical Objects by Demonstration. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg, VA., 95:104, November 1989.

[Nelson85] Nelson, G. Juno: A Constraint Based Graphics System. Computer Graphics:Siggraph '85 Conference Proceedings, 19, 3, July 1985, 235-243.

[Palay88] Palay, A., Hansen, W., et al. The Andrew Toolkit--An Overview. Carnegie-Mellon University TR, 1988.

[Pugh88] Pugh, W. Incremental Computation and the Incremental Evaluation of Function Programs. Ph.D. Thesis, Cornell University Computer Science Department, TR 88-936, August 1988.

[Schulert85] Schulert, A., et al. ADM--A Dialog Manager. Proceedings of the ACM CHI: 177-183, April 1985.

[Smith82] Smith, D.C. et al. The Star User Interface: An Overview. In Proc. AFIPS Conf., 515-528, 1982.

[Sussman80] Sussman, G. J. and Steele, G. L., Jr. CONSTRAINTS--A Language for Expressing Almost-Hierarchical Descriptions. Artificial Intelligence, 14, 1-39, 1980.

[Sutherland63] Sutherland, I. Sketchpad: A Man-Machine Graphical Communication System. Ph.D. Thesis, MIT Computer Science Department, TR No. 296, January 30, 1963. Reissued May 19, 1965.

[Szekely88] Szekely, P., and Myers, B.A. A User Interface Toolkit Based on Graphical Objects and Constraints. OOPSLA/Sigplan Notices 23(11): 36:45, November 1988.

[Thacker79] Thacker, C. Alto: A Personal Computer. Xerox Palo Alto Research Center Report CSL-79-11, 1979.

[VanderZanden88] Vander Zanden, B. Incremental Constraint Satisfaction and Its Application to Graphical Interfaces. Cornell University TR 88-941, October 1988.

[VanderZanden89] Vander Zanden, B., and Myers, B. Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces. ACM Chi '90 Proceedings, Seattle, WA., 27:34, April 1990.

[VanWyk80] Van Wyk, C. A Language for Typesetting Graphics. Ph.D. Thesis, Stanford University Computer Science Department, June 1980.

[Vlissides89] Vlissides, J., and Linton, M. Unidraw: A Framework for Building Domain-Specific Graphical Editors. Stanford University Computer Systems Laboratory TR CSL-TR-89-380, July 1989.

[Weinand88] Weinand, A., Gamma, E., and Marty, R. ET++--An Object Oriented Application Framework in C++. OOPSLA/Sigplan Notices 23(11): 36:45, November 1988.