Report Number: WUCS-92-41

1992-10-01

Hyperflow: A Visual Programming Language for Pen Computers

Takayuki Dan Kimura

This paper presents the design philosophy of the Hyperflow visual programming language. It also gives an overview of its semantic model. The primary purpose of language is to provide a user interface for a pen-based multimedia computer system designed for school children. Yet it is versatile enough to be used as a system programming language. The concept of visually interactive process, vip in short, is introduced as the fundamental element of the semantics. Vips communicate with each other through exchange of signals, either discrete or continuous. Each vip communicates with the user through its own interface box by displaying... Read complete abstract on page 2.

Hyperflow: A Visual Programming Language for Pen Computers

Takayuki Dan Kimura

Complete Abstract:

This paper presents the design philosophy of the Hyperflow visual programming language. It also gives an overview of its semantic model. The primary purpose of language is to provide a user interface for a pen-based multimedia computer system designed for school children. Yet it is versatile enough to be used as a system programming language. The concept of visually interactive process, vip in short, is introduced as the fundamental element of the semantics. Vips communicate with each other through exchange of signals, either discrete or continuous. Each vip communicates with the user through its own interface box by displaying on the box information about the vip and by receiving information pen-scribed on the box. There are four different communication modes: mailing, posting, channeling, and broadcasting. Mailing and posting are for discrete signals and channeling and broadcasting are for continuous signals. Simple Hyperflow programs are given for the purpose of illustration, including a Hyperflow specification for the Line-Clock device driver.

Hyperflow: A Visual Programming Language for Pen Computers

Takayuki Dan Kimura

WUCS-92-41

October, 1992

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130-4899

To appear in the Proceedings of 1992 IEEE Workshop on Visual Languages, Seattle, Washington.

This research is partially supported by the Kumon Machine Project.

Hyperflow: A Visual Programming Language for Pen Computers[1]

Takayuki Dan Kimura
Department of Computer Science
Washington University in St. Louis
(314) 935-6122 tdk@wucs1.wustl.edu

Abstract

This paper presents the design philosophy of the Hyperflow visual programming language. It also gives an overview of its semantic model. The primary purpose of the language is to provide a user interface for a pen-based multimedia computer system designed for school children. Yet it is versatile enough to be used as a system programming language. The concept of *visually interactive process*, *vip* in short, is introduced as the fundamental element of the semantics. Vips communicate with each other through exchange of signals, either *discrete* or *continuous*. Each vip communicates with the user through its own *interface box* by displaying on the box information about the vip and by receiving information pen-scribed on the box. There are four different communication modes: *mailing*, *posting*, *channeling*, and *broadcasting*. Mailing and posting are for discrete signals and channeling and broadcasting are for continuous signals. Simple Hyperflow programs are given for the purpose of illustration, including a Hyperflow specification for the Line-Clock device driver.

1. Introduction

This is a progress report on Hyperflow. Development of the Hyperflow (HF in short) visual programming language is a part of the research project in progress since January 1991 to develop a pen-based multimedia computer system. The pen computer is to be used for teaching mathematics to children from the pre-school level up to the 12th grade. The project is expected to be completed in 1995. The project also involves development of MC68030-based hardware and toolkit firmware with the pen user interface. The silicon paper technology, our term for the pen computer technology, is discussed in [6] and [7]. An overview of the project is given in [8].

HF is an extension of Show and Tell [5]. From the Show and Tell project we have learned two major lessons: that the mouse/CRT is not adequate for visual programming, and that entirely pictorial visual programming is not practical. We use the pen instead of the mouse and combine texts with icons in HF. As in Show and Tell, the syntax consists of boxes and arrows, a box representing a process and an arrow representing a data flow between processes. While data types in Show and Tell are limited to discrete types, continuous data types such as audio signals and animations are available in HF. Continuous data types are necessary not only for multimedia processing but also for processing pen-strokes.

HF is an user interface framework as well as a visual programming language. The two aspects of HF are inseparable. As an interface framework, HF is an extension of various window systems with a dataflow architecture and more independent computation power for each window. A HF

1 To appear in the Proceedings of 1992 IEEE Workshop on Visual Languages, Seattle, Washington.
This research is partially supported by the Kumon Machine Project.

window, called the interface box, communicates not only with the user but also with other windows through visually specifiable dataflow. The user controls the activities of a window by scribing a gestic command on it. Each window has a separate set of commands to be activated by the user's gestures. As in X Windows[10], buttons, menu items, scroll bars and their 'thumbs', are all windows in HF. Each HF widget can respond to a large variety of gestic commands if necessary, in contrast with the mouse-based widgets that can respond to only clicking and dragging.

In the remaining part of this section, we discuss the purposes and goals of HF and a new software engineering approach incorporated in the HF design that we propose for the construction of interactive application software. In Section 2, we present an overview of the conceptual (semantic) model of HF. Due to space limitations our presentation will be limited to the communication aspects of the model. In Section 3, we introduce some syntactic components with sample programs. A complete definition of the HF syntax would be premature because we are still experimenting with different syntactic representations of semantic concepts.

1.1 Purposes

The purpose of HF are presented in the following three tiers, listed in order of sophistication:

Level-1: Pen user interface. End-users, both school children and math instructors, interface with the pen computer through handwritten characters, numbers, and geometric shapes. The traditional mouse-based interface tools, (e.g. widgets of X Windows,) are not appropriate for pen computers. The pen interface requires a new paradigm in which gesturing and handwriting, instead of pointing and dragging, play a prominent role. By a single gesture, instead of two steps necessary in the menu-based interface, the user can select both an object and an operation. If the menu is to be eliminated from the user interface framework, what should be its replacement? HF offers a *visually interactive process* (*vip* in short) as the fundamental element of the pen user interface (replacing menus). The user communicates with a vip via gestures. The vip is described in Section 3.

Level-2: Visual shell language. The power of the UNIX operating system comes partly from its shell language. The end users of the graphic user interface are deprived of such a shell language. The philosophy of direct object manipulation and WYSIWYG makes design of a graphic shell language very challenging. Our conjecture is that a visual shell language is a natural application of visual programming languages where the control structures of procedure definition, selection, and iteration are available. HF has these capabilities.

Level-3: Visual system programming. It is commonly believed that visual programming languages are good for novice users but not for system programmers. However, recent software development tools, such as symbolic debuggers of various types [1] and Interface Builders [3], are becoming more graphics-oriented. This suggests that visual interfaces are also useful for system programmers, and a visual language for system programming should not be considered unsound, *if* an efficient compiler can be constructed. Is it possible to design a visual language in which a compiler and an operating system can be implemented? For example, how can we specify an interrupt driven device driver in visual form? HF is our proposal for such a visual system programming language.

1.2 Goals

HF has been designed as a programming language to achieve the following four goals:

Understandability. A computer system for school children needs an easy to understand programming language. In order to make HF programs easier to construct and easier to understand, three properties are incorporated: First, it is a visual language. A two dimensional arrangement of iconic modules manifests inter-relationships better than a linear textual specification. Secondly, it is dataflow based. Our experience with the Show and Tell language proved that the concept of dataflow is easier for school children to learn than the concept of control flow. Finally, it is parsimonious. A HF program specifies an ensemble of homogeneous communicating processes. There are no other constructs such as functions and procedures. We presume that the parsimony enhances the understandability.

Responsiveness. It is a common misconception that novice computer users do not demand instantaneous response from the computer. On the contrary, the less familiar the user is with computer technology, the more he expects rapid responses from the computer. The run-time support system (kernel) of HF includes real-time tasking based on preemptive scheduling. Processing the user entry of pen gestures has the highest scheduling priority.

Universality. A general goal of programming language design is to offer the most expressive power with the least variety of constructs. HF demonstrates its universality in four areas: First, it can emulate most of the traditional graphic user interfaces such as X widgets. Second, it can represent not only conventional discrete data types but also continuous data types. Thus, it is equipped to deal with multimedia applications. Third, it is capable of integrating computation, database, and communication applications into a single paradigm. This is the same goal we set and achieved successfully for the Show and Tell language. Finally, it is a system programming language as well as an end user programming language. This goal requires the capability of concurrent processing with adequate synchronization primitives, a rich set of abstraction mechanisms, and the development of an optimizing compiler.

Extensibility. As a system programming language, HF has to provide a set of tools to manage orderly growth of software complexity. There are three aspects of HF that will contribute to its extensibility: First, it is object-oriented to encourage modular programming. A *vip* is an encapsulation mechanism. It can interact with another vip only through the exchange of signals. Secondly, it is prototype based [12]. Vips can be copied and pasted individually or in groups. There is no need to artificially define a virtual class. Inheritance of attributes from a prototype to a copy can be controlled by designating each attribute in the prototype as private or public. Finally, it allows multi-lingual programming. Programming of a vip can be done visually in HF, or textually in C, in Assembly Language, or in any other language whose compiler is made available to the HF system. This makes importing programming modules from other environments easier. It also serves to improve the performance of critical vips by hand-coding them in traditional textual programming languages.

1.3 System Decomposition

The concept of *communicative organization* was introduced as a conceptual model of distributed computation in [4] and it is used as the foundation of HF semantics. A communicative organization is defined as a collection of active entities (processes), distributed over time and space, communicating with each other through exchange of passive entities (symbols) to achieve some common goal. A program specifies the social structure as well as the communication behavior of individual processes in the organization.

In HF design, we consider that the user and the computer system form a communicative organization consisting of the user process, software processes, and hardware processes (Figure 1). The user and hardware processes are *extrinsic,* and the rest are *intrinsic.* The HF system is a software tool to specify and manage the intrinsic processes. Among the intrinsic processes, some of them interact directly with hardware processes. We call them *device* processes. All processes including device processes may directly interact with the user process.

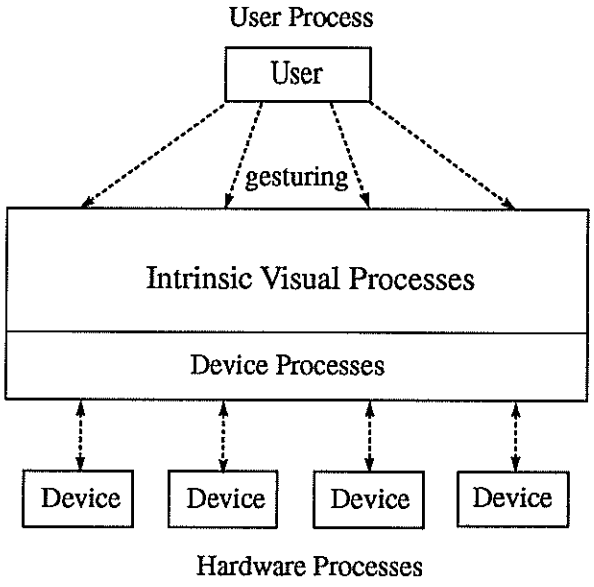

Figure 1: Communicating Processes

In the traditional development of application software for graphic workstations, the software system is decomposed into two parts, the interface and the application, e.g. the server-client model of the X Window System [10]. Each part is then decomposed into modules and sub-units. This approach presumes that the two parts are loosely coupled, and that infrequent subroutine calls and callbacks are sufficient to connect the two. However, in many applications the user interface modules require much tighter coupling, and there are some efforts to reduce the needs for call backs [9].

We adopt an alternative approach to the software system decomposition (Figure 2). In our approach, the system is first decomposed into functional modules, (i.e., vips), then each module is divided into the interface part and the computation (application) part. This approach presumes that the user interface is as ubiquitous as the computation. The unification of user interface and application has been recently urged by van Dam in his keynote speech at the 4th UIST Symposium[13]. He observed that the the user interface needs more "semantic feedback" from the application and that there should be a single environment for development of user interface and application. Our approach is consonant with his observation. HF is designed to provide just such a single environment.

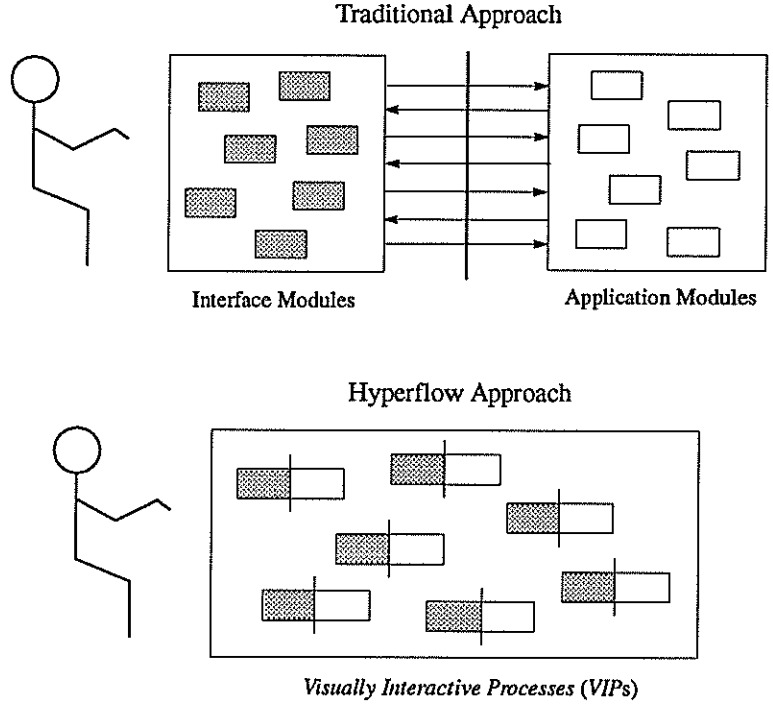

Figure 2: System Decomposition

We go one step further in the efforts to integrate the user interface with computation. Some computational processes are system processes for managing various types of resources as a part of the operating system. Device processes are typical of such system processes. Traditional user interface management systems do not interact directly with system processes. It is our hypothesis, however, that all processes, system or application, need to have an interface with the user at one time or another. For example, a graphic symbolic debugger provides system programmers with a visual interface to system processes including device drivers. Therefore, all intrinsic processes are visually interactive in HF at the user's option.

2. The Semantic Model

As stated earlier the semantic model of HF is based on the concept of communicative organization, a collection of active elements (which communicate), passive elements (which are communicated), and the environment (where communication takes place) [4]. In HF, the concept of *visually interactive process (vip)* is introduced as the active element, and the concept of *signal* as the

passive element of communication. Vip also provides the communication environment. Vip is an extension of the traditional concept of process and signal is an extension of data. Each vip has a set of commands to be activated directly by a user's gesture or by a signal from another vip.

2.1 Signals

There are two categories of signals, *discrete* and *continuous*. The traditional data types, such as numbers, text, Boolean values and bitmaps, are all discrete signal (data) types. Continuous signals are timed data streams such as audio and video[2]. Pen-strokes are also examples of continuous signals. Each signal consists of a *header* and a *body*. The header specifies how to interpret the body. We call a signal with the empty header a *datum*. Signals can be hierarchically structured; i.e., a signal may contain another signal.

2.2 Communication Modes

Humans utilize different modes of communication under different circumstances. We use telephone (*channeling*) for private and continuous communication and letters (*mailing*) for private and discrete communication. The telephone is synchronous; mailing is asynchronous. Both modes can be used for global (long distance) communication. We also use bulletin board (*posting*) and TV (*broadcasting*) for public and local communication. Posting is discrete and asynchronous, and broadcasting is continuous and synchronous. In public communication, the receiver must know the identification of the sender, while in private communication the sender must know the identification of the receiver.

In computation models, message passing which corresponds to mailing is the dominant mode of communication. Communication through a shared variable corresponds to posting. In general a memory cell can be considered a bulletin board which posts a datum received from mailing. Broadcasting and channeling have not been used in traditional software models due to the lack of capabilities for handling continuous data types.

In HF vips can communicate in all four different modes: mailing, posting, channeling, and broadcasting. Each mode is represented by a different type of arrow in the HF syntax. They are summarized in Figure 3. Note that a vip is represented by a box.

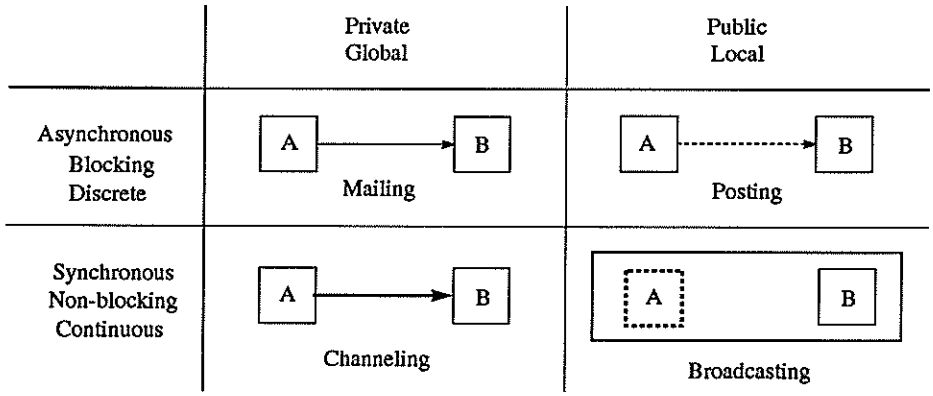

Figure 3: Communication Modes

Mailing is denoted by a regular arrow. Vip **A** sends a mail item (discrete signal) to vip **B** by executing the command *sendmail*. Vip **B** has a *mailbox*, which is a FIFO queue of discrete signals. If **B**'s mailbox is full, the *sendmail* operation is blocked. **B** has two options in its open-mailbox policy, *automatic* and *manual*. If the mailbox is set to *automatic*, the next mail will be opened automatically (as soon as possible) and its head will be interpreted as the name of command to be executed by **B**. If the mailbox is set to *manual*, the next mail will be opened when **B** execute the command *openmail*. If the mailbox is empty when *openmail* is executed, the operation will be blocked.

Posting is denoted by a dotted arrow. Vip **A** has a *bulletinboard* on which **A** can post a *note* (discrete signal) by executing the command *postnote*. Any vip, **B**, in the same environment (locality) as **A**, can read the note on **A** by executing the command *readnote*. If there is no note posted on **A**, the readnote operation will be blocked. **A** has two options in its policy for removing notes from the bulletinboard (deposting), *automatic* and *manual*. If the bulletin board is set to *automatic*, whenever the note is read by some vip, it will be removed simultaneously; i.e., only one vip can consume the note. If the bulletinboard is set to *manual*, the note remains posted until **A** excutes the command *depost* or **A** posts a new note which replaces the old note.

Channeling is denoted by a fat arrow. Vip **A** starts sending a continuous signal to vip **B** by executing the command *startsend* and stops the signal by the command *stopsend*. The signal will be sent to **B** regardless of whether **B** is ready to receive it . **B** has two options in the acceptance policy, *automatic* and *manual*. If the **B**'s channel is set to *automatic*, **B** starts receiving the signal as soon as it arrives. If the channel is set to *manual*, **B** starts receiving the signal when the command *startreceive* is executed and stops when *stopreceive* is executed. Both *startsend* and *startreceive* are non-blocking operations.

Broadcasting is not denoted by any arrow. Instead, the sender, **A**, is identified by a dotted box. Any vip, **B**, in the same environment as **A**, can listen to the signal transmitted by **A**. **A** starts transmission of a continuous signal by the command *startbroadcast* and stops it with *stopbroadcast*. **B** has two options in its listening policy, *automatic* and *manual*. In manual listening **B** starts listening to **A** by executing the command *startlisten* and stops it by *stoplisten*. In automatic listening **B** starts listening preemptively as soon as **A** starts transmission. All four operations are non-blocking.

2.3 VIP: Visually Interactive Process

A vip is a concurrent process (task) with a user interface. It is the only unit of system decomposition in HF. There are no variables, functions, procedures, or packages. Computations are carried out by a homogeneous community of vips interacting with each other through the exchange of signals. We call such a community a *vip ensemble*. The user monitors and controls activities of the ensemble by directly communicating with the individual members. Every vip is visually accessible to the user unless the user chooses to hide it.

A vip consists of two parts, the *interfacing* part and the *processing* part (Figure 4). The interfacing part is responsible for communication with the user through visual display of the vip's internal states and through accepting and interpreting pen-strokes entered by the user. The processing part is responsible for the management of communication with other vips. A vip is either *active* or

inactive. A vip is *active* when both the processing part and the interfacing part are ready to communicate. It is *passive* if only the interfacing part is ready to communicate. Note that the interface part is always active and therefore the visual representation of a vip is always *responsive*, the third level of liveness defined by [11], in other words, it can respond to a user's pen-stroke at any time preemptively.

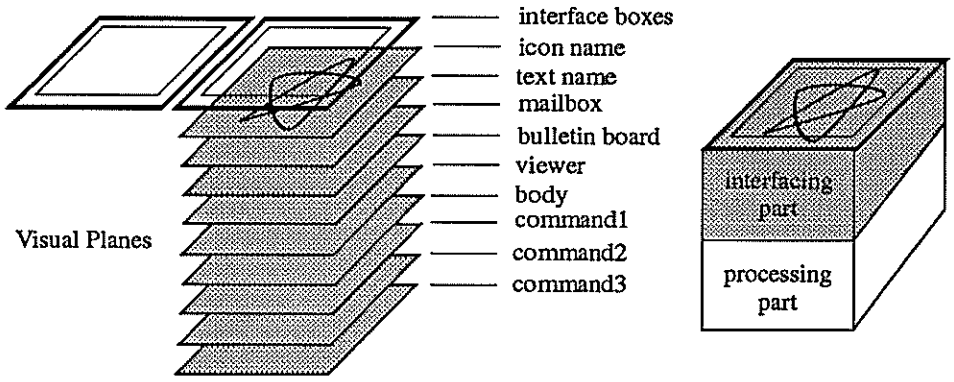

Figure 4: VIP Structure

The processing part consists of the following elements (some vips may not have all elements):

mailbox	a bounded FIFO queue of discrete signals,
bulletinboard	a discrete signal,
viewer	a slice of a continuous signal,
body	an ensemble,
commands	a sequence of named ensembles.

A vip has at most one *mailbox* and one *bulletinboard*. The *viewer* is a buffer for one slice of a continual signal, e.g., a video frame or an audio sample value. The *body* is the container of the main activities of the vip. It is *static*; i.e., whenever the vip is active so is the *body*. A *command* is a spontaneous source of activities. The vip has a set of commands, each of which has a textual name. A command is *dynamic* in the sense that the command (i.e. the named ensemble) becomes active only when its activation is requested by mail, by broadcasting, or by a user's gesture. Only one command can be active at a time. The body is active concurrently with a command. Any member of an ensemble, the body or a command, can access the mailbox, the bulletin board, and the viewer. The vip provides the local communication environment for its ensemble members. Posting and broadcasting are effective only among the ensemble members owned by the same vip.

The interfacing part consists of *interface boxes* (or simply *boxes*) and *visual planes* (Figure 4). An interface box corresponds to the (server) window in the traditional graphic user interfaces. It has a rendering on the two dimensional display screen. The user can change the size and location of the box by a simple gestic action. Through an interface box the user can inspect the content of any visual plane. Each visual plane is a two dimensional bitmap with its own coordinate system containing a visual representation of various information in the vip. This information includes the vip's iconic name, textual name, the content of the bulletin board, and the contents of the mailbox. Other displayable items are the source and object codes of the body and commands, and their documentation. The user can create an arbitrary number of visual planes for each vip.

There may be more than one interface box for a vip, each displaying a different aspect of the vip.

Two boxes belonging to the same vip are connected by a line (not an arrow). The user can create a new box and delete an old box with gestures. The user can also select, with a gesture, a visual plane and its portion to be displayed as the content of an interface box. Two boxes may display two different sections of the same visual plane. When the last box is deleted by the user, the vip itself is also deleted.

An interface box has the two parts, the *frame* and *content*. The *frame* of a box is a bitmap pattern of various width that borders the box. The *content* is the remaining internal part of the box. Different frame patterns are used for different types of contents. The user also can design the frame patterns. The content part displays a clipped portion of a visual plane. The user can select and edit a visual plane by entering an appropriate gesture command on the content part of the box.

There are system defined commands innate to all vips. Some of them are listed below:

start	make the vip active
stop	make the vip inactive
duplicate	create a copy of the vip
delete	destroy the vip
move	move the interface box to new location
resize	change the size of the interface box.

These commands can be activated either by mailing, by broadcasting, or with a gesture scribed on the interface box by the user.

The body and commands of a vip can be encoded individually either in the two dimensional syntax of HF or in any textual programming language such as C, Pascal, or assembly language. A vip may have a multi-lingual specification for its processing part. When a vip ensemble becomes active (to be executed), if it is specified by a HF program using the graphical syntax, its execution will be simulated by the HF interpreter. If the HF program is already translated into machine code by the HF compiler, it will be executed directly. Textual programs must be compiled before execution. In textual programming each vip is referenced by its textual name. Communication protocols (e.g., how to access the mailbox of a vip and how to broadcast signals to other vips) are provided by a function library constructed for each textual language.

A vip is a self-contained module in the sense that it contains all of its relevant information, including its identification, source codes, object codes, manuals, and graphic attributes. All of the information is visually accessible to the user, upon demand, through a set of interface boxes.

Figure 5 gives an example of a vip that computes the Fibonacci sequence. Figure 5.2 presents a HF program to compute 10th Fibonacci number using the vip defined in Figure 5.1. The Fibonacci vip of Figure 5.1 is visualized through twelve interface boxes. Its body contains two *memory cell* vips, x and y, which are referenced in the command program. A memory cell posts whatever data it receives by mail, with the automatic open-mail policy and manual de-posting policy. The Fibonacci vip has two commands, *init* and *next*. The *init* command initializes the memory cells in the body. The *next* command computes the next Fibonacci number and the result of computation is posted at the bulletin board. The body and the commands are encoded in different languages, Pascal and HF, to illustrate the multi-lingual programming concept.

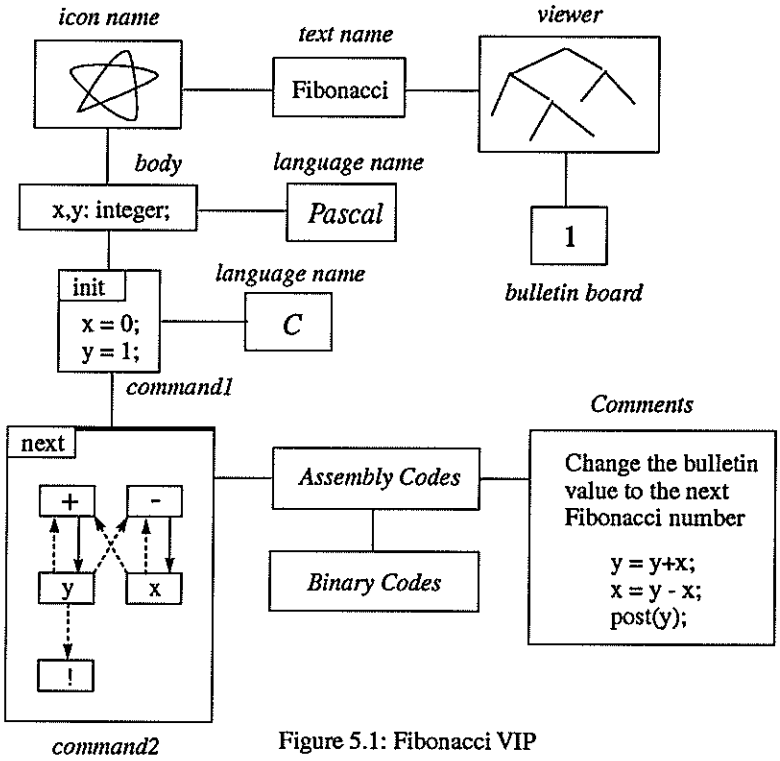

Figure 5.1: Fibonacci VIP

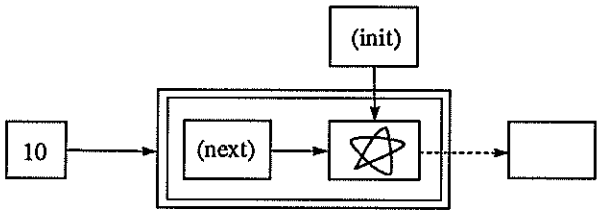

Figure 5.2: Computing Fib(10)

In the *next* command program, the character '!' in the HF syntax denotes the vip that readnotes and posts to the bulletin board. The fat horizontal line on the top edge of the command2 box denotes the vips in the box are to be activated sequentially from left to right, then top to the bottom, i.e., in this example, the execution order will be +, -, y, x, and !. The vip + reads two values from the bulletin boards of x and y, then sends the sum to y by mail. The vip - reads the values from y and then x, and send the difference to x by mail. The vip y posts the value received from + by mail. The vip x posts the value received from -. Then, the system defined vip ! posts the value (the bulletin board) of y onto the the bulletin board of the Fibonacci vip.

In Figure 5.2, the doublely framed box represents the system defined *iteration* vip which sequentially iterates the execution of its body for the number of times specified by a mail. The empty box represents another system defined *show* vip which reads a value from the bulletin board of another vip and posts it on its own bulletin board.

Every vip is visually accessible to the user. However, there are other modes of communication between the user and a vip. In general the user sends signals to vips through:

scribing	pen-strokes for gestures, texts, and shapes,
scanning	a bitmap,
keyboarding	text,
speaking	a voice, and
taping	a video.

A vip sends signals to the user through:

printing	text,
drawing	a bitmap,
voicing	speech,
sounding	sounds, and
viewing	video.

The current hardware system does not support scanning, taping, and viewing.

2.4 Prototyping and Inheritance

The Hyperflow language is an object-oriented language without classes. As in SELF [12], objects (vips) are created by duplicating other objects (prototypes). Every vip is potentially a prototype for another vip. When a vip is duplicated, its body and commands are duplicated independently, thus all the descendant vips will be recursively duplicated. Also, the user can select a set of vips (a part of an vip ensemble) as a group, then make a copy of the set in one gesture.

The cloned vip inherits all the attributes of the prototype vip. The user can modify, (i.e., delete and augment) any attributes of the clone as long as the attributes are designated as *public* in the prototype vip. The HF system keeps track of the is-a-copy-of relation, and any change in the value of a *private* attribute will be automatically propagated through the descendant clones. In HF there is no mechanism for multiple inheritance.

One technical problem associated with object-oriented programming without classes is the creation of large regular data structures, e.g., a 100 x 100 cell spread sheet. It would be tedious work to generate 10,000 copies of an object by repeated applications of a duplication operation, even with the power of group duplication. We propose to solve this problem in two ways. One way is to provide a system defined prototype vip with a unit size, e.g., a 1x1 spreadsheet, and to allow the user to change the size parameter as a public attribute. The other way is to provide the user with a special gesture command for multiple duplication with a particular layout arrangement. Our proposed gesture is illustrated in Figure 6. The latter method works only up to two dimensional structures. For higher dimensions we have to depend upon the first method.

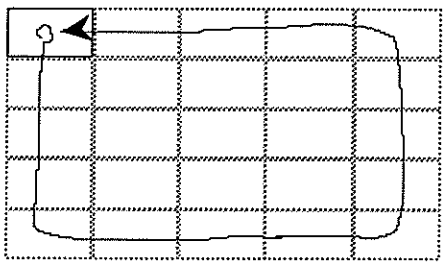

Figure 6: A Gesture for Multiple Copying

3. Hyperflow Syntax and Sample Programs

We will illustrate some of the semantic concepts introduced in Section 2 through a set of sample HF programs using the current version of the HF syntax. We expect to modify the syntax after the preliminary implementation.

3.1 Fibonacci and GCD

In this example we demonstrate the visual similarity between the Fibonacci function and the GCD function. Figure 7.1 defines the Fibonacci function in iterative form. In this figure the box for the icon and the box for the body are touched together, instead of being connected by a line. They are synonymous.

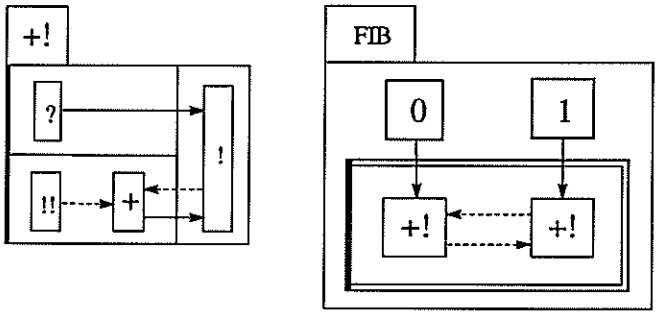

Figure 7.1: Fibonacci Function

The body of vip +! consists of three members. The vertical fat line denotes that the three members are sequentially executed from top to bottom then from left to right. First, vip ? opens mail if it is available, then sends it to the posting vip !. Otherwise, it does nothing. Second, vip !!, which is an agent of +!, *readnotes* from the vip connected to +! by a dotted arrow. If there is no note to read, it broadcast a signal to all the vips in the second member to stop immediately. Vip ! plays the role of a formal parameter as empty boxes in Show and Tell. The binding rule for HF is also the same as Show and Tell. The vip ! posts the value received by mail. The double framed box in FIB represents a vip for unbounded iteration. The vertical fat line denotes that each iteration executes the body (the content ensemble) sequentially from top to bottom.

Figure 7.2 defines the GCD function. Vip -! is similar to vip +! except that the second member contains a vip that plays the role of predicate and the - vip is used instead of the + vip. The system defined *predicate* vip, <, broadcasts the stop signal to all of the neighboring vips, if the condition is not satisfied. Otherwise, it does nothing. The notion of consistency in Show and Tell is replaced by the notion of broadcasting in HF.

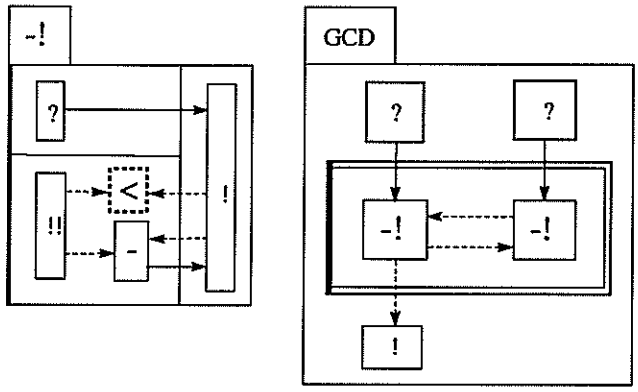

Figure 7.2: The GCD Function

Figure 7.3 illustrates how the GCD vip is used.

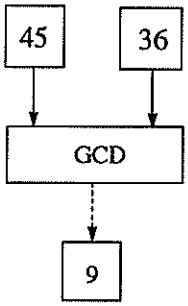

Figure 7.3: Using the GCD VIP

3.2 Line Clock Driver

In order to demonstrate how device processes can be specified in HF, we present a device driver for line clock hardware which causes a CPU interrupt every 1/60th of a second. The device has the Control Status Register (CSR) through which CPU interrupts can be enabled or disabled. When an interrupt occurs, the Interrupt Vector Address (IVA), $40, the starting address of the interrupt handling routine for this example, is transferred to the CPU (Figure 8.1).

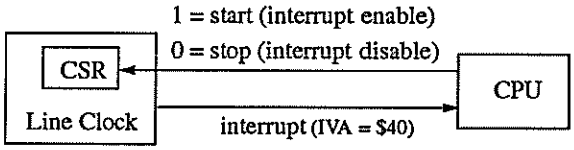

Figure 8.1: Line-Clock

The device driver in Figure 8.2 has three commands; *reset*, *interrupt* ($40), and *read*. The body is displayed on the right side of the three commands shown in sequence. The body contains two vips; CSR, which represents the hardware register, and the tic counter. The interface box for CSR has an additional L-shape pattern on the four corners to indicate that this vip is unique; i.e., there is only one in the HF system.

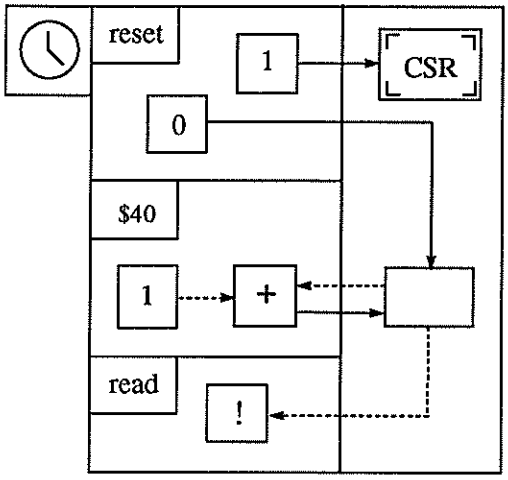

Figure 8.2: A Device Driver for Line-Clock

Figure 8.3 shows how to use the device driver. Note that the L-shaped corner patterns indicate that there is only one line-clock driver.

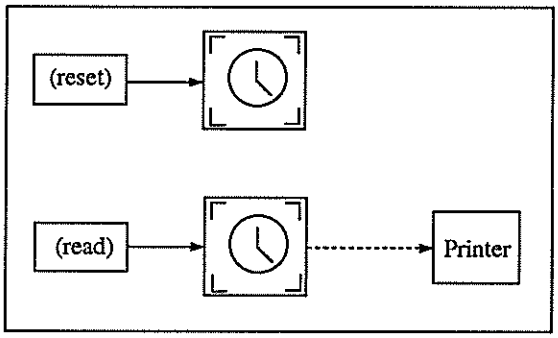

Figure 8.3: Using Device Driver

References

[1] BugByte, an advanced window based debugger for the NeXT Computer, User's Reference Manual, ONyX Systems, Inc., Fort Worth, TX 76185, 1991.

[2] Herrtwich, R. G. "Timed Data Streams in Continuous-Media Systems," ICSI Technical Report TR-90-017, Berkeley, California, May 1990.

[3] "Interface Builder", NeXTstep Concepts manual , NeXT Computer, Inc., Redwood City, CA 94063, 1990, pp. 8-1 to 8-108.

[4] Kimura, T.D. and Gillett, W.D. "Communicative Processes: A Model of Communication," *Proceedings of the 10th IMACS World Congress on Systems Simulation and Scientific Computation*, Montreal, Canada, August 1982.

[5] Kimura, T.D., Choi, J.W. and Mack, J.M. "A Visual Language for Keyboardless

Programming," Technical Report WUCS-86-6, Department of Computer Science, Washington University, St. Louis, June 1986.

[6] Kimura, T.D. "Silicon Paper and A Visual Interface for Neural Networks," *Proceedings of 1990 IEEE Workshop on Visual Languages*, Chicago, IL, October 1990, pp. 241-246.

[7] Kimura, T.D. "Pen-based User Interface," Panel session chiarman's position paper, *Proceedings of 1991 IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991, pp. 168-173.

[8] Kimura, T.D. "Learning Math with Silicon Paper," Technical Report WUCS-92-12, Department of Computer Science, Washington University, St. Louis, February 1992.

[9] Myers, B. "Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs," *Proceeding of the Forth Annual ACM Symposium on User iNterface Software and Technology*, Hilton Head, SC, November 1991.

[10] Scheifler, R.W., Gettys, J. and Newman, R. *X Window System*. Digital Press, 1988.

[11] Tanimoto, S. L. "Towards a Theory of Progressive Operators For Live Visual Programming Environments," *Proceedings of 1990 IEEE Workshop on Visual Languages*, Chicago, IL, October 1990, pp. 80-85.

[12] Ungar D., Smith, R.B. "SELF: The Power of Simplicity," *OOPSLA'87 Proceedings*. Published as *SIGPLAN Notices*, 22,12 (1987) 227-241.

[13] Van Dam, A. Keynote Speech, at the Forth Annual ACM Symposium on User Interface Software and Technology, Hilton Head, SC, November 1991.