DYNAMIC PATCHES: IMPLEMENTATION AND CONTROL
IN THE SUN-MERCURY WORKSTATION

Xavier Rodet, Gerhard Eckel
IERCAM, 31 rue Saint-Merri, 75004 Paris, France
telephone: (1) 42 77 12 33, ext. 46 27

ABSTRACT

On many workstations, patches simply consist of a fixed and linear sequence of instructions. On the contrary, the dynamic patches of the SUN-Mercury Workstation (SMW) described here have some appealing properties. First, SMW patches include control structures (while, if, etc.), loops, sub-patches, pointers, arrays and similar constructs which facilitate patching and make it close to ordinary programming. Secondly, patches can be modified very easily and rapidly without introducing "clocks" or unsystematic phenomena, even while they are running.

Let us recall briefly the outlines of this workstation under development at IRCAM, which was presented at the 81 ICMI in Champaign, Urbana [1]. Hardware is assembled from a few commercially available products: a VME SUN-III host micro-computer running Unix, a Mercury Zip 3223+ floating-point array processor, an OROS interface to Sony PCM DAC/ADC, a VME-MIDI board, and various MIDI equipment such as keyboard, synthesizer, reverberation unit, etc.

Host software is based on PREFORM, an object-oriented real-time control and graphic tool-box implemented by Le Lig [10]. Software for the Zip is based partly on standard Mercury Inc. C.S. software, partly on original developments written in Zip assembler or in micro-code.

MOON (Mercury Object Oriented Nickname) is the ensemble of software components developed specially for the SMW.

BLOCK COMPUTATION AND TIME TAGGED MESSAGES

Array processors such as the Zip are efficient only when the same algorithm can be applied iteratively on arrays or blocks of data. This is very often the case in digital signal processing, particularly for music applications where high sampling rates are required. Effectively, the update rate of parameters rarely needs to be higher than 100 Hz. Then blocks of about 500 samples at 50 KHz sampling rate render computation efficient without lowering parameter update precision. Furthermore, array processors are always provided with a library of functions highly optimized for block calculation, also called vector functions.

However, in some cases such as fast attacks, or for particular algorithms, it is necessary to induce parameter modifications, or control fast modifications, with an accuracy in time higher than the block level. We provide time tagged messages to accomplish such modifications with an accuracy as high as one sample.

ICMC Proceedings 1988 82
PREFORM OBJECTS AND ZIP OBJECTS

Functions and data in the Zip are coupled with a new object in PREFORM. For instance, one precise instruction within a given patch that adds two vectors, i.e., two signals, is represented:
- in the Zip, by an object implemented as a data structure including a pointer to the code of the appropriate vector function in the Zip micro-program memory,
- in the host, by a PREFORM object pointing at the corresponding Zip data structure.

Naturally, users only have to deal with PREFORM objects which are totally in charge of their pupils in the Zip. Note that informations contained in a Zip object cannot be duplicated in the host PREFORM object since Zip objects may exchange messages and evolve without necessarily notifying PREFORM. When such information is needed in PREFORM, it is directly accessed in the Zip object by the PREFORM object.

CLASSES AND SUB-CLASSES IN PREFORM

The super-class of the MOON system is the moon class. It essentially provides a few general fields and methods such as external access to fields, pretty print, save definition of object on file, etc.

All other classes of our system are subclasses of the class moon and inherit fields and methods of the moon class.

MODULES

In the Zip, a module object (instance of a class) consists of four data:
- a pointer to a Zip function of the library,
- a pointer to a set of handles which is a pointer to arguments of the function,
- a pointer to the next module in the patch (the next instruction of the program),
- a pointer to the first pending message (if any).

As we see, modules of a patch are arranged into a linked list according to their next pointers. Modifying a patch is simply inserting/deleting one or several modules in this list, which can be done very economically by the change of one or two of these next pointers.

In PREFORM a module object has fields for the Zip address of the data structure mentioned above, and for a list of handle objects (see below). Among other methods, it can use the data next of the Zip to point to another object, and then execute, i.e., execute the Zip function. This last method is extremely useful for debugging a patch by stepping through its sequence of modules.

To each function of the library there corresponds a module class of which module objects are instances. Let us mention some specific module classes, Read and write modules exchange data blocks with the host (boot memory and disk), DAC and ADC modules exchange samples with the PCI through the ORIG interface connected on the auxiliary ports of the Zip.

BUFFERS

Buffers are blocks of Zip data memory. They are used by Zip functions for block calculation, but they are also used for keeping other pieces of information: They have a size that is their number of words. When of size 1, they are called scalar, and when of size 2 they are pointers which point at memory addresses. Scalars are used by some Zip functions. The address pointed to by a pointer can be changed, allowing the implementation of arrays, arrays of buffers or of scalars. Buffers can be allocated dynamically.
HANDES

Handles are used by Zip functions to point to buffers and scalars on which they operate. Thus, if a module M computes a block of data, it will put it in the buffer B pointed at by its output handle H. Then, if another module N is to pursue the computation on the data provided by M, it will read it from the buffer B pointed at by its input handle I. Actually, handles do not point directly to the address of the block in memory, but to a descriptor containing this address and the size of the buffer. One may see that the descriptor allows a sort of a virtual connection between M and N, that is between modules which have to communicate data. This feature makes it possible to change the buffer pointed at by the handle without breaking any connection in the patch.

Similarly, one can change the set of handles pointed at by a module: this may be used to break one block into several pieces as needed when messages should apply in the middle of the block. We see that patches are very flexible and that, for instance, exchange of buffers can be done to efficiently implement certain algorithms (such as delay or overlap-add), or to modify their size, etc...

PATCHES AND MONITORS

A patch is composed of a monitor and an ordered list of modules. A monitor is a special type of module which is given a handle pointing to the list of the user's modules in the patch. It is in charge of executing user's modules, one after the other in the order of the list, the end of the list being indicated by a null value. Execution of modules is conditional on the value of a scalar continue attached to the monitor but accessible to other modules. As long as continue is TRUE, the list of modules is executed repetitively. When continue is found to be FALSE, control is given to the caller of the patch, i.e. the caller of the monitor, which can be the Zip executive or another module in the case where our patch is used as sub-patch of another. Thus, a patch implements a program structure which can be described in a Lisp manner:

```
(defun continue
  (module arguments)
  (module arguments)
  (module arguments))
```

The FALSE value for continue is 0. Since continue is writable from any module, loops or other control structures can easily be implemented (continue being for example a loop counter). As mentioned above, a patch can take the place of a module in another patch; actually, it is the monitor of the sub-patch which takes the place of a module. Such a program structure can be described in the following way:

```
(defun continue _1)
```

```
(define-patch patch _1
  (while continue _1
    (module arguments)
    (module arguments)
    (module arguments)
))
```
(define-patch patch_2
  (while continue_2
    (module> <arguments>)
    (module> <arguments>))
  (patch_1> <arguments>)
  (module> <arguments>))

;* this last definition is equivalent to: *
(define-patch patch_2
  (while continue_2
    (module> <arguments>)
    (module> <arguments>))
  (while continue_1
    (module> <arguments>)
    (module> <arguments>))
  (module> <arguments>))

MESSAGES

A message consists of a time-tag, a pointer to a Zip function and arguments. It is sent to a module, to be executed at a given time. The monitor, before executing a module, checks for pending messages. A time-tag consists of a block number and a sample number in the block. If the sample number is zero, the message is simply and efficiently executed before the module itself at the right block number. Otherwise, the block is cut into two pieces in order to insert the message execution in between the two parts.

A dynamic patch modification cannot be done asynchronously by the host without endangering the consistency of the module sequence. Such a modification is thus required in terms of a message sent to some module of the patch.

To accelerate response of the synthesizer (the Zip), messages can be sent directly from a module to another without passing through the host. As an example, when triggered, a grow module could start another patch.
ZIP OBJECT AND UTILITIES

This object supports standard Mercury control primitives. These primitives, written in C, are included in Le_Lisp as Lisp functions. Let us list some of the most important:

- open, close the Zip,
- load an executable micro-program in the Zip,
- call a function in the Zip,
- stop the current micro-program,
- initiate a DMA from/to the Zip,
- allocate a buffer,
- read/write floats/integers from/to the Zip memory,
- read/write control and status registers of the Zip.

The object utilities controls the Zip assembler, linker and simulator. Only implementers have to know about it. For ordinary users, a path definition and setup automatically links the necessary micro-code files and reads them into the Zip micro-program memory.

PCM AND MIDI OBJECTS

The PCM (actually the OROS interface) and the MIDI board are seen by the user as objects implementing the necessary fields and methods. The PCM object allows opening and closing, access to all interface registers, and most important, set up of the sampling frequency, since the OROS interface converts signals between any sampling rates used in the Zip and 44.1KHz standard used by the SONY PCM.

The MIDI object controls the MIDI board through "C" and system function calls to the MIDI driver(1).

CONCLUSION

As a conclusion we would like to underline some design choices of the SMW which have been found particularly interesting.

- Implementation in Le_Lisp:
  * Apart from other qualities of Lisp, use of an interpreter for controlling a machine as complex as the Zip is of great help in the debugging phase.
  * This control was easily implemented because Le_Lisp allows inclusion of C routines (Zip control primitives) in Lisp even at runtime.

- Implementation in PREFORM:
  * Object-oriented programming facilitates software development and maintenance.
  * PREFORM programs (even graphic applications) are portable: the SMW benefits from developments done in other environments or other machines such as Macintosh.
  * PREFORM provides a scheduler and simple, efficient real-time primitives.

(1) The MIDI driver was written at IRCAM by David Yadohiri. Let us thank him here.
- Zip array processor:
  * Floating-point arithmetic drastically simplifies programming of digital signal processing.
  * The Zip is provided with a large and efficient library of vector functions.
  * Block calculation: it makes implementation of sophisticated control structures and all dynamic aspects very easy and efficient since these instructions are executed once per block: it would take a highly optimised computer to achieve (if possible) the same efficiency.
  * The auxiliary port avoids using VME to transfer samples to the PCM, and being interrupt driven, avoids problems that could result from usage of I/O.

At present, all features except messages are running. However, the EMW needs some more effort before it can be let into the hands of non-computer-expert users. Also, we still need to write a few specialised or more efficient modules, and work on user interface. Among other aspects, we will concentrate on playing sounds from disk according to scores and variable tempo (as a great improvement over tapes).

REFERENCES

[1] Eckel, Gerhard, Redet, Xavier, Potard, Yves
  'A SUN-Mercury Music Workstation'
  In Proceedings of the ICMC 1987, Urbana, Champaign

  'Adding a Graphical User Interface to FORMES'
  In Proceedings of the ICMC 1986, pages 105-108

  'ZIP3232 Standard Algorithm Library'

[4] Potard, Yves, & Vandenhoede, Jan
  'Creating a musical environment around an FPS1000'
  Conference Record of the 1985 PUSE meeting, Munich 1985

  'FORMES: Composition and Scheduling of Processes'

[7] Ducote, Pierre
  'Présentation générale du système informatique MAX'
  Rapport interne, IRCAM 1987

[8] Boynton, Lee Richard
  'PREFORM'

[9] Duth, Jacques, & Potard, Yves
  'Le Lisp, an Object-Oriented Extension of Lisp for an Integrated Computer Music Environment'
  To appear in Proceedings of the ICMC 1987

ICMC Proceedings 1988
Figure 1. Patch, modules, handles and buffers.
Figure 2: A Typical view of the Sun / Mercury workstation environment.