HPorter: Using Arrows to Compose Parallel Processes

Liwen Huang[1], Paul Hudak[2], and John Peterson[3]

[1] Yale University, Dept. of Computer Science
liwen.huang@yale.edu
[2] Yale University, Dept. of Computer Science
paul.hudak@yale.edu
[3] Western State College, Computer Information Science
jpeterson@western.edu

Abstract. *HPorter* is a DSL embedded in Haskell for composing processes running on a parallel computer. Using arrows (a generalization of monads), one can "wire together" processes in a manner analogous to a signal-processing application. The processes themselves are typically existing C or C++ programs, but may also be programs written in a first-order sub-language in Haskell that supports basic arithmetic, trigonometric functions, and other related operations. In both cases, once the processes are wired together, the supporting Haskell implementation is out of the loop – imported C programs run unimpeded, the Haskell sub-language is compiled into C code, and all data paths run directly between C processes. But in addition, HPorter's event-driven reactivity permits reconfiguration of these tightly-coupled processes at any time, thus providing a degree of dynamism that is critical in many applications.

The advantages of our approach over conventional scripting languages include a higher degree of type safety, a declarative style, dynamic reconfiguration of processes, having the full power of Haskell, and portability across operating systems. We have implemented HPorter both on the QNX operating system and using conventional TCP/IP sockets, and are using it in a practical application in Yale's Humanoid Robotics Laboratory, where the processes correspond to soft-real-time tasks such as computer vision, motor control, planning, and limb kinematics.

1 Introduction

A humanoid robot has many time-critical tasks, including vision processing, motor control, limb kinematics, high-level planning, and so on. State-of-the-art applications place heavy demands on these tasks, and require parallel computers to deal with them effectively. In addition, the "modes" of a robot vary – if it is moving, it might need to focus on its kinematics, but if it is trying to pick up an object, it might need to focus on vision processing and planning. Scripting these processes efficiently and in the correct manner is thus an important task for the robotics programmer.

M. Hanus (Ed.): PADL 2007, LNCS 4354, pp. 275–289, 2007.

In this paper we describe *HPorter*,[1] a DSL embedded in Haskell for composing processes running on a parallel computer. HPorter is based on *arrows*, a generalization of monads. One way to think of the generalization afforded by arrows is that they permit functions (processes) to be composed "in parallel," rather than in the linear, sequential style dictated by monads. This makes arrows a good choice for composing parallel processes in a rigorous, robust, and type-safe manner.

Although the processes themselves could in the abstract be any arbitrary computations, including ordinary Haskell programs, our primary interest is in scripting existing processes written in C (or compiled into C), for the sake of efficiency. On the other hand, any extra processing needed to glue a couple of processes together (for example, incrementing each value in a stream, or taking the sine of each value) is something easily expressed in Haskell, and it would be inconvenient to insist that the user write a new C program for each new piece of glue code. Therefore, we also have designed a small first-order Haskell sub-language called GLUE, based on previous work on Pan and Pan# [2,14], that is easily compiled into C.

Once the C processes are wired together, the supporting Haskell implementation is completely out of the loop – the imported C programs run unimpeded, the Haskell sub-language is compiled into C code, and all data paths run directly between C processes.

But in addition, a key aspect of HPorter is that it is *reactive*, since, as mentioned earlier, there are times when the process configuration needs to change, often in drastic ways. We achieve this by using *switch* combinators borrowed from our work on FRP and Yampa [16,12]. This provides event-driven reactivity that permits dynamic reconfiguration of the otherwise tightly-coupled processes.

In contrast to existing approaches to scripting parallel processes, our approach offers the following advantages:

1. HPorter is type-safe. All input and output ports are strongly typed, thus providing a robust interface not typically found in the C world.
2. HPorter is declarative, resulting in more concise and easier to understand code. Rather than saying "how" things are wired together as in a conventional approach, HPorter describe "what" the process interconnections are in an arrow-based style.
3. HPorter is reactive, permitting reconfiguration of the processes in an event-driven manner.
4. HPorter is embedded in Haskell, thus affording the user the full expressive power of a modern functional language. Process-wiring code can be reused, recursion can replicate networks, higher-order functions can capture repeating patterns, and so on.

In our robotics application these advantages are even greater because some of the processes are actually *Dance* [6,5] programs that have been compiled into

[1] The name "HPorter" comes from the name of the QNX scripting language *Porter*, and our use of *H*askell.

C. Dance is a DSL embedded in Haskell for controlling humanoid robots, and uses principles similar to those in HPorter and Yampa – this similarity is an advantage to the user.

We originally implemented HPorter two years ago on the QNX real-time operating system running on a tightly-coupled network of four multiprocessors, each of which has four processors (thus 16 processors in all).[2] Recently, however, the hardware was upgraded to more powerful nodes (although only 8 instead of 16), and we decided to explore the use of conventional TCP/IP sockets to interconnect processes, rather than using the specialized QNX machinery. We felt that this would result in a more robust design and would allow the system to be more portable, since TCP/IP sockets are ubiquitous in Unix-based systems. In porting HPorter to this new platform, all we had to do was change the back-end interface and process-specific code – none of the arrow-based source code had to be changed. Thus we point out the final advantage of our approach:

5. HPorter is portable.

We are currently using HPorter to program a real humanoid robot in the Yale Robotics Laboratory. Our robot consists of a torso, two arms, a head, and shoulders (which move). It has twenty-one degrees of freedom, each corresponding to a separate motor, and each of those in turn requiring a separate motor controller. In addition, the robot's two eyes provide stereo vision, with two cameras for each eye – one for wide-angle viewing, and the other to simulate foveal vision. The vision processing is in fact the most demanding computational task.

The performance of HPorter is excellent. Once the processes are running, no performance degradation is apparent. Although reactive processing (for event processing and process reconfiguration) requires Haskell intervention, for our applications the response time of the reactive component is more than acceptable. Just as important, users of HPorter find the system easier to use than a conventional scripting approach.

The remainder of this paper is organized as follows. We start with a brief introduction to arrows in Section 2, following by an example of HPorter in Section 3. In Section 4 we discuss the notions of processes, ports, and connections in HPorter, as well as other implementation details. In Section 5 we discuss performance, and related work is summarized in Section 6.

2 A Brief Introduction to Arrows

We assume that the reader is familiar with Haskell. In this section we give a brief introduction to arrows; more detail can be found in [8,7].

Arrows are a generalization of monads that relax the stringent linearity imposed by monads, while retaining a disciplined style of composition. This discipline is enforced by requiring that composition be done in a "point-free"

[2] On the other hamd, hard real-time constraints are not something we address in this work, nor is it a requirement of our robotics application.

style – i.e. combinators are used to compose functions without making direct reference to the functions' values. These combinators are captured in the `Arrow` type class:

```
> class Arrow a where
>   arr   :: (b -> c) -> a b c
>   (>>>) :: a b c -> a c d -> a b d
>   first :: a b c -> a (b,d) (c,d)
```

`arr` lifts a function to a "pure" arrow computation; i.e., the output entirely depends on the input (it is analogous to `return` in the `Monad` class). `(>>>)` composes two arrow computations by connecting the output of the first to the input of the second (and is analogous to bind (`(>>=)`) in the `Monad` class). But in addition to composing arrows linearly, it is desirable to compose them in parallel – i.e. to allow "branching" and "merging" of inputs and outputs. There are several ways to do this, but by simply defining the `first` combinator in the `Arrow` class, all other combinators can be defined. `first` converts an arrow computation taking one input and one result, into an arrow computation taking two inputs and two results. The original arrow is applied to the first part of the input, and the result becomes the first part of the output. The second part of the input is fed directly to the second part of the output.

Other combinators can be defined using these three primitives. For example, the dual of `first` can be defined as:

```
>  second :: (Arrow a) => a b c -> a (d,b) (d,c)
>  second f = let swapA = arr (\(a,b) -> (b,a))
>             in swapA >>> first f >>> swapA
```

Finally, it is sometimes desirable to write arrows that "loop", such as in a signal processing application with feedback. For this purpose, an extra combinator (not derivable from the three base combinators) is needed, and is captured in the `ArrowLoop` class:

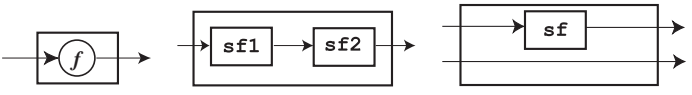
```
> class ArrowLoop a where
>    loop :: a (b,d) (c,d) -> a b c
```

We find that arrows are best viewed pictorially, especially for the application at hand: composing parallel processes. Figure 1 shows the basic combinators in this manner, including `loop`.

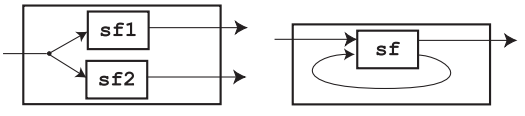
3 HPorter by Example

In this section we present some examples that highlight the three key features of HPorter: the use of arrows to wire together parallel processes, the ability to reconfigure processes dynamically, and the ability to write glue code without leaving Haskell.

```
arr    :: Arrow a => (b -> c) -> a b c
(>>>)  :: Arrow a => a b c -> a c d -> a b d
(<<<)  :: Arrow a => a c d -> a b c -> a b d
first  :: Arrow a => a b c -> a (b,d) (c,d)
second :: Arrow a => a b c -> a (d,b) (d,c)
(***)  :: Arrow a => a b c -> a b' c' -> a (b,b') (c,c')
(&&&)  :: Arrow a => a b c -> a b c' -> a b (c,c')
loop   :: Arrow a => a (b,d) (c,d) -> a b c
```



(a) `arr f` (b) `sf1 >>> sf2` (c) `first sf`



(d) `sf1 &&& sf2` (e) `loop sf`

Fig. 1. Commonly Used Arrow Combinators

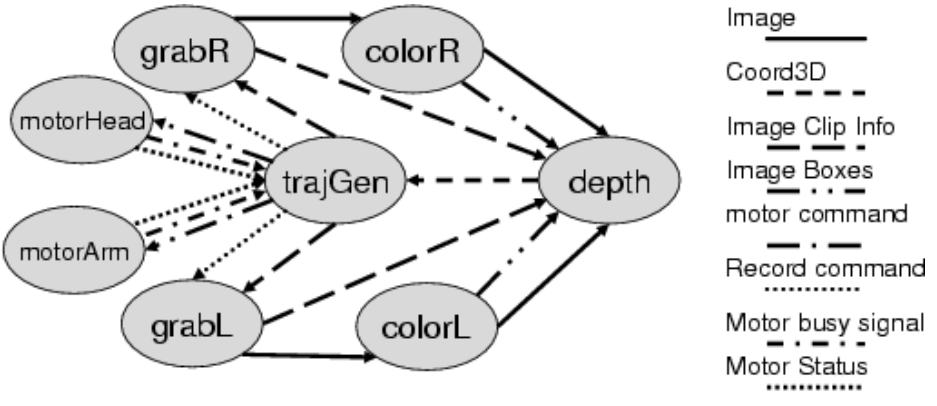


Fig. 2. Structure of a Robot System

3.1 Processes as Arrows

In HPorter, a process is represented as an arrow of type `Proc T1 T2`. In other words, a process takes as input a stream of values of type `T1`, and yields as output a stream of values of type `T2`. If a stream of values were represented as an infinite list, we would have the following correspondence:

```
Proc T1 T2 = [T1] -> [T2]
```

In fact it is easy to make this representation an instance of class `Arrow`, and an (overly) abstract semantics for HPorter can be devised. In practice, the representation is much more involved, since these processes are actually imperative C programs running as QNX processes. We defer discussion of these implementation details until a later section.

As a realistic example, suppose we want our robot to perform a vision-guided reaching task, for which we need eight processes: two video image grabbers, two color processors, a scene depth calculator, two motor controllers, and a reaching trajectory planner.[3] Our only concern here is how to wire them together: the streams of images captured from the grabbers are processed by the color filters to generate "boxes" that identify objects of interest. Then the boxes along with the images from the color filters are passed to the depth calculator to generate the 3D coordinates of the objects. These coordinates are sent to the reaching trajectory planner, which computes the arm trajectory and passes that to the motor controller to move the arm. Besides this main information flow, there is inter-process communication for auxiliary functionality, like recording requests for the image grabber. Figure 2 shows the detailed information flow graphically for the overall system – note that the graph is circular.

This information flow can be captured in HPorter as follows:

```
> vision :: Proc ((Rec,CClip),(Rec,CClip)) ((Image,Image),Coord3D)
> vision  = (grabR >>> (first colorR)) *** (grabL >>> (first colorL))
>          >>> (arr (\ (((imR,bR),cR),((imL,bL),cL))->
>          (((imR,imL),(bR,bL)),(cR,cL)))) >>> ndepth
>
> reach :: Proc () ()
> reach = loop ((motorHead *** motorArm) *** (vision >>> (arr snd))
>               >>> trajGen >>> (arr (\ ((a,b),c) -> (a,(b,c)))))
```

From this example the reader can see how cumbersome it can be to write in a point-free style – in particular, the pairing and merging of inputs and outputs becomes quite tedious. To alleviate this problem, Paterson has proposed a special syntax for arrows [13], much in the spirit of the "do" syntax for monads. Using arrow syntax, the above program can be written:

```
> reach :: Proc () ()
> reach = proc x -> do
>          rec
>            (cGR,imgR)   <- grabR     -< (cpR,rcR)
>            (cGL,imgL)   <- grabL     -< (cpL,rcL)
>            (imgCR,boxR) <- colorR    -< imgR
>            (imgCL,boxL) <- colorL    -< imgL
```

[3] The trajectory planner is actually a Dance (i.e. Haskell) program compiled into C using GHC.

```
>            ((imgDR,imgDL), depD)
>                        <- ndepth    -<
>                       (((imgCR,imgCL),(boxR,boxL)),(cGR,cGL))
>            (((cmd0,cmd1),(cpR,rcR)),(cpL,rcL))
>                        <- headarm'  -<
>                                ((bz0,hm),(bz1,am)),depD)
>            (bz0,hm)    <- motorHead -< cmd0
>            (bz1,am)    <- motorArm  -< cmd1
>          returnA -< ()
```

Unlike the "do" syntax for monads, the arrow syntax requires both an input and an output for each process. As with monad syntax, the inputs and outputs "strip off" the arrow constructor. For example, in the above, `colorR` has type `Proc Image (Image, Boxes)`, and thus `imgR` has type `Image` and `(imgCR,boxR)` has type `(Image,Boxes)`.

Although more verbose than the original point-free style, this is arguably a very natural and easy to understand way of wiring processes together. Indeed, it is isomorphic to the diagram in Figure 2. Its constrained style permits us to guarantee, eventually, that the processes run stand-alone, without the help of the Haskell subsystem.

Continuing with this example, the processes we use are generated from existing C programs in the following way. Suppose the C program for the color filter is located at `"/home/user/bin/color"`. Suppose further that the TCP/IP ports for this process have identifiers `"inputa"` and `"inputb"` for input, and `"outputc"` for output. Suppose finally that we wish to map this process to processor id 5. We can do this as follows:

```
> colorR :: Proc Image (Image, Boxes)
> colorR = makeProc progColor "-b -N 1 -s 0 -o /colorR" 5 5
```

where `progColor` is defined as:

```
> progColor = defProg { procName = "/color",
>                       progName = "/home/user/bin/color",
>                       input = image "inputa"
>                   output = lift2 (image "inputa") (box "inputb"),
>                       param = colorP}
```

The details of `image` and `box`, and of the string argument to `makeProc`, are not important. Each of the other processes can be defined in a similar way.

3.2 Reactivity

In order to add reactivity to HPorter, we adopt the ideas of *functional reactive programming* [16,12,1,3], in particular as they are embodied in *Yampa*, which also uses arrows [7].

One key idea in Yampa is a *signal function*, whose type is `SF a b`, and is analogous to HPorter's `Proc a b`. Another fundamental concept is that of an

event, which occurs at discrete points in time. This idea is captured in Yampa through an option type called `Event`:

```
> data Event a = NoEvent | Event a
```

`Event` is isomorphic to `Maybe`, but it is an abstract type whose constructors are not exposed. Yampa provides a rich set of functions for generating event sources and for operating point-wise on events.

In HPorter we treat a reactive process as a signal function that generates non-reactive processes. In other words:

```
> type (HasPort a, HasPort b) =>
>      ReactProc a b c = SF a (Proc b c)
```

Here, type `a` represents the signal type that our process reacts to. Now Yampa's facilities for reactivity – i.e. its "switching" combinators – can be used to switch to a new signal function when an event occurs. The most commonly used switching combinator is:

```
>   switch :: SF (a, (b,Event c)) -> (c -> SF a b) -> SF a b
```

For example, the expression `(sf1 &&& es) ‘switch‘ \e -> sf2` behaves as `sf1` until the first event in the event stream `es` occurs, at which point the event's value is bound to `e` and the behavior switches over to `sf2`.

With this background we can now give an example of reactivity that highlights our application domain. The robot's vision system has a variety of image processing capabilities, such as a color filter and a motion detector:

```
> color  :: Proc Image Image
> motion :: Proc Image Image
```

For input and output, suppose we also have an image grabber and a video player:

```
> grabber :: Proc () Image
> video   :: Proc Image ()
```

Now suppose we want the vision system to switch between looking for objects of a certain color (signaled by `Event 1`), objects that are moving (`Event 2`), or no objects at all (`Event 0`). This behavior can be achieved as follows:

```
> colorOrMotion :: ReactProc (Event Int) () ()
> colorOrMotion = filterSelect noFilter
>
> colorFilter  = grabber >>> color >>> video
> motionFilter = grabber >>> motion >>> video
> noFilter     = grabber >>> video
>
> filterSelect :: Proc () () -> ReactProc (Event Int) () ()
> filterSelect p = switch (proc e do
>                            returnA -< (p,e))
```

```
>                          (\a -> case a of
>                              0 -> filterSelect noFilter
>                              1 -> filterSelect colorFilter
>                              2 -> filterSelect motionFilter
```

`filterSelect` is a recursive switch function that starts with a process of type `Proc () ()`, and watches the input signal for an event. When an event happens, `filterSelect` is called recursively, but possibly with a new process, depending on the value of the integer event. It is important to understand that the switching process is not the same as a conditional – a switch may imply the reconfiguration of parallel processes.

3.3 GLUE'ing Processes Together

In this section we give an example of the third and final key feature of HPorter, namely the ability to write simple glue code without resorting to C or C++.

As mentioned in the introduction, sometimes simple glue code is needed to interconnect processes – for example, we might want to increment each value in a stream, or take the sine of each value. It would be inconvenient to insist that the user write a new C or C++ program for each new piece of glue code. Our solution is to introduce a small first-order imperative language called GLUE that allows the user to write the glue code directly within her HPorter program, but which is simple enough that it can be compiled into efficient C++ code.

In our original design we simply defined an AST data type in Haskell and wrote our glue code using values of that type. With the overloading afforded by Haskell's type classes, this was a reasonable approach, and it worked quite well. More recently, however, we have defined a simple lexical syntax for this language, which makes writing GLUE code even easier. As an example, here is a program that takes two streams of integers and adds them pairwise:

```
name glueplus
input Int a;
      Int b;
output Int c;
c := a + b
```

This program is compiled into our AST data type, where it is type-checked and compiled into C++, borrowing ideas from `Pan` and `Pan#`, which are DSLs for graphics that are embedded in Haskell. Since that compilation process is well-documented elsewhere (see [2,14]), we omit a detailed discussion in this paper.

Since GLUE is an imperative language, one might ask why we don't just write the glue code in C or C++. But in addition to the small piece of straight-line code that, in the example above, adds two numbers together, there is a plethora of additional "boilerplate code" that needs to be written as well, such as the inclusion of header files, and establishing the linkages between this process and the ones that we are scripting. Indeed, our compilation process turns the above five-line program into a ninety-five line C++ program.

4 Processes, Ports, and Connections

In this section we describe in detail how the underlying processes, ports, and connections are implemented in HPorter. All of this is hidden from the user.

Running Processes. As mentioned in Section 3.1, a process can abstractly be thought of as a stream transformer. But concretely, it is a C or C++ process running stand-alone on an individual *node* of a parallel computer with a unique TCP/IP *address*. Each process has a *pathname*, a unique *id*, and both an input and output *port*. Finally, processes are wired together via *connections* between pairs of ports.

In order to achieve this in Haskell, we need to represent all of these gory details within the abstraction for processes in HPorter. We begin with the simple notion of a *running process*, or `RProc`:

```
> type RProc     = (ID, ProgPath, Parameter, Address, Node)

> type ProgPath  = String;  type Parameter = String;
> type Address   = String;  type ID        = Int;
> type Node      = Int;     type PIDMap    = [(ID,Address)]
```

An `RProc` thus contains a unique `ID`, a program pathname, a parameter (i.e. an argument), the number of the node on which it is running, and the TCP/IP address of the node. We also introduce the concept of PID map, which maps the ID of each process to the TCP/IP address of the node on which it is running.

Ports and Connections. Next, we define the types needed for process communication. The connection of a server/client pair is built upon the notion of a *port*:

```
> type Port       = (ID, PortName)
> type PortName   = String
```

which contains the ID of the process that it is defined within and a unique local name. Then a *server port*:

```
> type ServerPort = (Port, PortNum)
> type PortNum = Int
```

is a pair of port and port number, and a *connection*:

```
> type Connection = (Port,Port)
```

is a pair of ports, whose order matters: data flows from the first to the second.

Process State and Arrow Instances. Finally, as we compose processes together (using the arrow framework), we need to generate a new ID for each composite process and a free port number for each pair of communication ports, and we need to keep track of all live socket port servers, the internal connections, and

the internal process ids. This information is contained in the `PState` data type, which is then used to define the `Proc` data type as follows:[4]

```
> data Proc a b = Proc ((PState, a) -> (PState, b))
>
> data PState = PState { nextID     :: ID,
>                       nextPort   :: PortNum,
>                       serverPort :: [ServerPort],
>                       conns      :: [Connection],
>                       procs      :: [RProc],
>                       pidMap     :: PIDMap}
> emptyPState = PState { nextID = 0, nextPort = 5000, serverPort = [],
>                       conns = [], procs = []}
```

Now we can declare `Proc` to be an instance of `Arrow` and `ArrowLoop`:

```
> instance Arrow Proc where
>    arr f = Proc (\(s, x) -> (s, f x))
>    Proc f1 >>> Proc f2 = Proc (f2 . f1)
>    first (Proc f) = Proc (\ (s, (a,c)) ->
>          let (s', b) = f (s, a) in (s', (b, c)))
>
> instance ArrowLoop Proc where
>    loop (Proc f) = Proc (\ (s, a) ->
>                      let (s', (b, c)) = f (s, (a, c)) in (s', b))
```

Running a Composite Process. At the outermost level of an HPorter program, there is one value of type `Proc () ()` that needs to be executed, just as in monadic IO there is one value of type IO () to be executed. Indeed, to execute the `Proc () ()` value in Haskell, it must be converted into a value of type `IO ()`. The function `runProc` achieves this for us:

```
> runProc :: Proc () () -> IO ()
> runProc (Proc p) =
>     let (s, output) = p (emptyPState, ())
>         obs         = procs s
>         cs          = conns s
>         sv          = serverPort s
>         adList      = pidMap s
>    in do sequence_ (map (run sv cs adList) obs)
```

(`sequence_` is a standard Haskell library functions that takes a list of monadic actions and "runs" them in sequence.)

[4] Note that if `Proc` could be defined as `Proc (a -> (PState, b))` then it would be a *Kleisli arrow*, and thus a monad. But it cannot, and thus the more general arrow class must be used.

The initial `PState`, `emptyPState`, contains no server port, no connections, no process, an initial id value and an initial port number (which is set to 5000 to avoid possible conflict with the system processes). By applying `p` to the `initPState`, we get a final `PState` named `s` that contains all of the connections, processes, PID-IP address mapping and server port number assignment for the whole program. `run` generates the appropriate QNX commands to begin execution of each process with the proper port number initialization parameters for each.

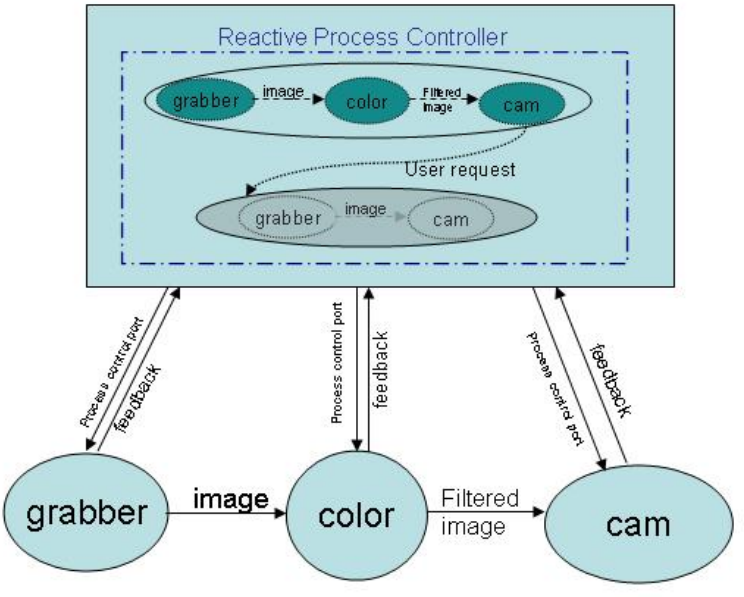


Fig. 3. Process Controller and Processes

Adding Reactivity. The presentation we have given so far has actually been oversimplified. In particular, we have not taken into account how HPorter dynamically reconfigures processes, including stopping them and restarting them if necessary. We need a new execution model to enable dynamic process re-wiring, in which we:

- Add an input port in all the source programs for process control command.
- Add an output port in all the source programs for control command feedback.
- Adjust the programs to allow process control interruption during execution.
- Add a command line option for switching between online and offline process control.

These new ports are exclusively for process control purposes, and connect only to what we call the *process controller*. They are not user-controllable and do not appear in the program or process abstractions. Through these new ports the process controller acts as a central controller for all of them.

The relationship between the process controller and each process is that of a standard client/server model, as shown pictorially in Figure 3. The controller (server) sends commands to each process (client) through a "control" port, and receives responses through a "feedback" port. The process control commands are captured in:

```
> data ProcCmd = StartServer ID PortName PortNum
>              | ConnectTo ID PortName Address PortNum
>              | Stop ID PortName
>              | Quit ID
>              | Suspend PID
>              | Continue PID
```

The command `StartServer pid pn i` asks process `pid` to start a TCP/IP socket server `pn` at port `i`. Command `ConnectTo pid1 pn addr i` asks process `pid1` to connect port `pn` to the port number `i` at address `addr`. `Stop pid pn` tells the process `pid` to close the port named `pn`, and `Quit pid` is used to kill process `pid`. The `Suspend` and `Continue` commands allow interrupting and resuming a process, for situations where a batch of commands needs to be addressed before the process can proceed safely.

Although the details are too numerous to include in this paper, reactivity works as follows: The state that is accumulated by the running system includes all of the running processes and how they are interconnected. When an event occurs that triggers a switch, a computation is performed to determine the best way to achieve the reconfiguration (some processes may need to be killed; others suspended, rewired, and restarted; and others created from scratch). The above commands are then issued to the processes to effect this reconfiguration, and the computation continues. All of this stateful computation is "hidden" within the arrow and the switching combinators.

5 Performance

We have implemented HPorter on two different networks of parallel processors running under the QNX real-time operating system, one having 8 processors, and the other having 16. The current system is running under QNX Version 6.3, and we use TCP/IP sockets for inter-process communication.

GHC Version 6.4 is used to compile any Haskell processes that are being scripted (for example the Dance program for the trajectory planner discussed in Section 3.1), as well as the GLUE code and the process controller.

We have compared our implementation of HPorter to the QNX *Porter* scripting language, and find them to be comparable in performance for our application.

- For non-reactive processes, Haskell is only needed for starting and interconnecting the processes. The extra overhead at start-up time is not noticeable, because the start-up time for most processes is much longer.
- For processes that contain GLUE code, some overhead is incurred to compile the glue code. Once compiled and interconnected, however, Haskell once

again is out of the loop. And because the glue code is usually very small, the overhead of compilation is not significant. Also, our implementation works hard to ensure that GLUE code is not recompiled every time it is invoked – thus the overhead is only incurred the first time around.
- For processes with reactivity, we have found that for our applications, where the mode switches do not happen frequently, the response time is more than adequate. In vision-based robotics, vision processing is the computationally limiting factor, and rates of 10-20 hertz are considered good. At that rate HPorter's impact on the system is negligible. For applications requiring more rapid response, we expect that pre-compilation of the glue code may be necessary. This would be straightforward using our approach, but thus far we have not needed to do so.

6 Related Work

There are many "architectural description languages," or ADLs, such as Darwin/regis [11], ACME [4], and Rapide [10], designed for specifying the architectures of a software system. HPorter shares with these language the ability to specify a software architecture, but there are several important differences:

- Most ADLs represent the architecture as a collection of components and connections, whereas we treat it as a *transition function* and cast it into an arrow framework.
- ADLs are meant primarily for the design of software systems, whereas HPorter is targeted at composing and executing a real distributed application.
- HPorter supports reactivity – i.e., the expression of dynamic, reconfigurable architectures – which is seldom found in ADLs.
- New processes can be defined and created dynamically in HPorter.
- Programs in HPorter are more concise than ADLs, which express components and their interconnections separately.

Our work is probably most similar to Ptolemy [9], which serves both as an ADL and as a language for composing real-time processes. Ptolemy is much richer than HPorter, although its notion of process interconnection is more complex than that of HPorter.

HaskellScript [15] is a scripting language embedded in Haskell that interconnects COM objects dynamically. Like HPorter, it also has strong typing. However, the focus is on uniprocessor applications, whereas HPorter allows true parallelism. Furthermore, HaskellScript uses monads to structure programs, and thus does not have the generality afforded by arrows.

7 Conclusion

In this paper we present an embedded DSL, HPorter, for composing parallel processes. HPorter has a concise and declarative syntax, via the employment of the arrow framework. The host language Haskell makes it more robust in the

sense of type safety, compared to conventional scripting techniques. Reactivity in HPorter allows system reconfiguration through the use of switching combinators derived from Yampa. We have also presented a sub-language, GLUE, for specifying the glue code that is sometimes needed when interconnecting processes. An efficient implantation of HPorter is achieved by compiling glue code into C, and by interpreting process interconnections as QNX system calls.

References

1. C. Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the first conference on Domain-Specific Languages*, pages 285–296. USENIX, Oct. 1997.
2. C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. In *SAIG*, pages 9–27, 2000.
3. C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 263–273, June 1997.
4. D. Garlan, R. Monroe, and D. Wile. ACME: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.
5. L. Huang. *Robot Dance with Functional Reactive Programming.* PhD thesis, Department of Computer Science, Yale University, December 2006.
6. L. Huang and P. Hudak. Dance: A declarative language for the control of humanoid robots. Technical Report YALEU/DCS/RR-1253, Yale University, Department of Computer Science, July 2003.
7. P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming, Oxford University.* Springer Verlag, to appear, 2003.
8. J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.
9. E. A. Lee. Overview of the ptolemy project. Technical Report Technical Memorandum UCB/ERL M03/25, Univerisity of California, Berkeley, CA, 94720, USA, July 2003.
10. D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
11. J. Magee, N. Dulay, and J. Kramer. Regis: A constructive development environment for distributed programs, 1994.
12. H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, Oct. 2002. ACM Press.
13. R. Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, Sept. 2001.
14. J. Peterson. A language for mathematical visualization. In *Proceedings of FPDE'02: Functional and Declarative Languages in Education*, October 2002.
15. S. Peyton Jones, E. Meijer, and D. Leijen. Scripting COM components from Haskell. In *Fifth International Conference on Software Reuse (ICSR'98)*, Victoria, B.C., Canada, June 1998. IEEE Computer Society Press.
16. Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proceedings of PLDI'01: Symposium on Programming Language Design and Implementation*, pages 242–252, June 2000. `http://haskell.org/frp/publication.html#frp-1st`