Interacting with Haggis: Implementing Agent Based Specifications in a Functional Style

Meurig Sage and Chris Johnson

GIST,
Department of Computer Science,
University of Glasgow, Glasgow, United Kingdom, G12 8QQ.
{meurig, johnson}@dcs.gla.ac.uk

ABSTRACT Formal specifications, of interactive systems, provide a precise and concise means of representing and reasoning about interactive systems. Executing these specifications allows designers to conduct usability testing. This, in turn, provides the feedback necessary for an iterative approach to interface design. We argue that new developments in concurrent functional languages, make them ideally suited as tools for executing specifications. To demonstrate this, we make use of Haggis, a concurrent functional graphical toolkit. We describe the development of a highly interactive game, from specification to execution, using this system. This application is appropriate as it demonstrates the real-time interaction that previous approaches to the prototyping of formal specifications could not support.

KEYWORDS formal methods, executable specifications, prototyping, functional programming.

1. INTRODUCTION

Formal specification techniques allow designers to satisfy themselves that systems are functionally correct. When applied to interactive systems, they can be used to analyse interaction problems (Shum et al, 1996). A number of different formalisms have been used for this purpose, including temporal logics (Johnson and Harrison, 1992), state-based CSP (Abowd, 1990), and LOTOS (Paterno, 1993). Specifications, based around multiagent models, which treat a system as a group of interactors or "communicating interactive processes", have received a great deal of attention (Abowd, 1992; Duke et al, 1994). However, techniques for executing these specifications, to allow proper user testing of the interfaces, are lacking (Harrison and Duke, 1995)

1.1 Why executable specifications?

It has been claimed that making specification languages executable can restrict their expressiveness (Hayes and Jones, 1989). However, (Fuchs, 1992) has shown that this is not necessarily true. (Alexander, 1987) has shown that executable specifications allow rapid prototyping, and therefore iterative development. This enables designers to fully experiment with different ideas, to ensure systems really meet their users' needs. For instance, partial implementations may be evaluated within a user's working environment.

1.2 Functional programming

In this paper, we argue that Alexander's work, should be brought up to date with new developments in functional programming. We make use of Haggis (Finne and Peyton Jones, 1995b), a Graphical User Interface

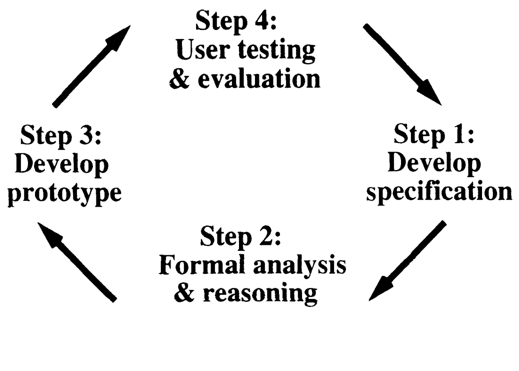
Step 4: User testing & evaluation
Step 1: Develop specification
Step 2: Formal analysis & reasoning
Step 3: Develop prototype

Figure 1: Iterative Development of Interactive Systems

development environment, for the lazy functional language Haskell (Peterson et al, 1996). To bridge the specification-execution gap, it provides a compositional, concurrent programming language. This makes it easy to transform a high level specification into Haggis code, and so to develop a prototype. This supports user testing and evaluation.

To show how this can be done, we use a highly interactive game, as an example. We specify it in LOTOS (Figure 1, step 1), and convert this into Haggis code (Figure 1, step 3). This demonstrates the power of the Haggis framework: we can represent the game at almost the same level of abstraction. We, however, still need the LOTOS specification to be able to clearly understand, and therefore reason about, the concurrent interaction (Figure 1, step 2). This is important because previous studies have shown that programmers frequently have great difficulty in exploiting the full potential of concurrent programming languages (Hoc et al, 1990). The example also serves to show that Haggis is capable of dealing with the kind of highly interactive systems that declarative languages are usually bad at.

2. FUNCTIONAL PROGRAMMING & EXECUTABLE SPECS

This paper uses 'leading edge' techniques from functional programming and formal specifications, to bring Heather Alexander's work into the 1990s.

This earlier work used eventCSP, a subset of the concurrent specification language CSP, to specify human-computer dialogues. For instance, we could specify part of an ATM (Alexander, 1990) - which accepts a card, and either ejects it, if it is invalid, or otherwise continues - as follows:

```
atm = (put-in-card ->
         (read-card -> ATM1
       [] cant-read -> eject-card ->
          take-card -> ATM))
```

Alexander argued that it is easy to understand, and reason about human-computer interaction using such a notation. Possible sequences could be built using the prefix (->) operator. Different paths of interaction could be described with the choice ([ ]) operator.

Alexander's approach was based around the specification of events, which had pre and post-conditions. These were defined using "me too". The me too system was used for executing VDM (Vienna Development Model) specifications, and was based within Lisp (Alexander and Jones, 1990). In common with her approach, we use a functional language as an execution tool. However, we exploit a number of developments in functional languages that make executing specifications more practical.

3. FUNCTIONAL PROGRAMMING & SPECIFICATION IN THE 1990s

3.1 Imperative Functional Programming

Though functional languages have been shown to be useful, they have had serious problems. One of the major difficulties has been supporting human-computer interaction. Older approaches were based around **stream based I/O**, which produce unnatural and confusing code. For instance, a simple program that copies it's standard input to its standard output (Gordon and Hammond, 1995), would be:

```
main ~(Str input: ~(Success : _)) =
  [ ReadChan stdin,
    AppendChan stdout input
  ]
```

The transfer of input from `ReadChan` to `AppendChan` is unclear, and counterintuitive.

Newer approaches based around **monadic I/O** allow a more imperative style of programming, that is familiar to most programmers (Gordon and Hammond, 1995; Peyton Jones and Wadler, 1993). The same program, with monadic I/O, is shown below.

```
main = do
         ch <- getChar
         putChar ch
         main
```

The sequencing of actions is clearer here. We can see that putChar uses the value (ch) returned by getChar.

3.2 Concurrency

Newer approaches to interactive system specification, make better use of concurrency than Alexander's work in the late 1980's. The interactor model, developed under the Amodeus project, treats a system as a number of interactors, each with a state, and the ability to communicate with users, other interactors and the underlying application. This allows for more modular and readable specifications. Systems can be built by relating tasks to interactors and so encouraging user centred design. For example, the York interactor model exploits various logics and the specification language Z. It is concerned with states, and displays and the relationships between the two (Duke et al, 1994). In contrast, the CNUCE interactor model uses LOTOS to specify interactors. It is more event based (Duke et al, 1994), making for a clearer dialogue structure.

A limitation with many of the previous approaches to interface specifications, is that they have not attempted to exploit concurrency in a specification, using a concurrent programming language. In contrast, this paper describes how concurrent specifications can be executed using concurrent functional languages. In particular, we use **Concurrent Haskell** (Peyton Jones et al, 1996), which provides for lightweight processes, and makes use of monadic IO. New child processes can be created with the forkIO function. Communication occurs in an asynchronous manner, through shared variables (MVars), that operate like semaphores. Unlike Alexander, we are able to turn our concurrent specification into concurrent code. We argue that this makes executing concurrent specifications easier.

To make the transformation from specification to execution simple, we have extended this underlying asynchronous communication, with a library of LOTOS like operators. These give programmers access to full synchronous communication through a parallel operator. We have also included separate asynchronous communication along channels. This provides, arguably a more elegant way, and certainly a more efficient way, of implementing the asynchronous communication, which is only simulated in LOTOS. Asynchronous and synchronous communication can be combined freely within this system (Sage and Johnson, 1997).

3.3 Virtual I/O device

Haggis, the graphical framework, is built on top of Concurrent Haskell. This approach offers a number of additional benefits for user interface design (Finne and Peyton Jones, 1995b). A common problem with many graphical user interface systems is the central role of an event loop. This style has well known problems (Myers, 1991). Instead Haggis treats the user interface as a virtual device, allowing the application to maintain control. The concurrent features of Haggis enable several virtual I/O devices to operate simultaneously. For instance, one process could wait for keyboard input, blocking if there wasn't any, while others did other necessary work.

3.4 Declarative structured graphics

A further problem for the development of graphical user interfaces is that conventional languages tend to be highly imperative. By this we mean that the resulting image might be completely ruined if one instruction were omitted or placed out of sequence. In contrast, all graphical output is described declaratively using a Picture type (Finne and Peyton Jones, 1995a). For instance, we could describe the simple picture in Figure 2 as follows:

```
picture = Overlay
            (ellipse (40,20))
            (ellipse (20,40))
```

The image in Figure 2 would remain the same irrespective of which ellipse was drawn first. Haggis takes care of converting pictures into calls to the window system drawing interface, with the Glyph output abstraction. This

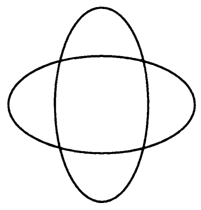

Figure 2: A simple Picture

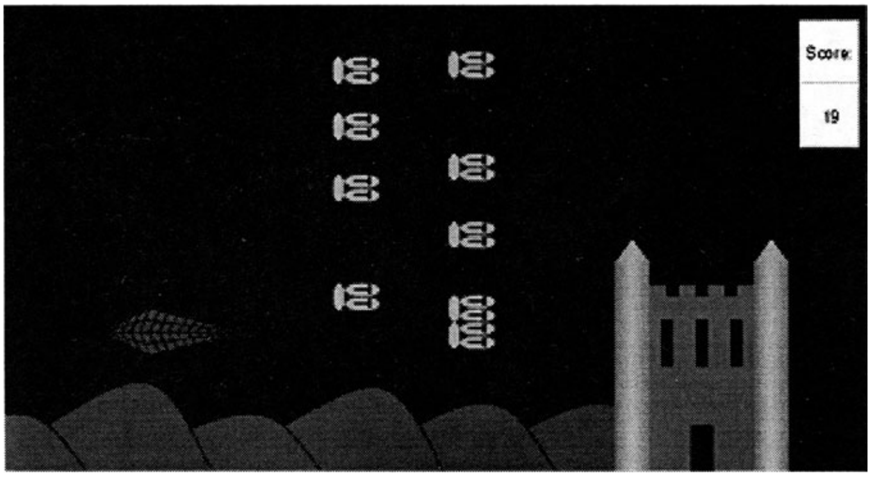

Figure 3: Game Display

declarative approach makes it easy to build, and manipulate, complicated images.

3.5 User interface, application separation

To separate the interface from its application, a user interface component is represented by two different parts. For instance, the glyph output abstraction returns a `Glyph` handle which the application can use, for instance, to update the picture displayed. It also returns a `DisplayHandle`, which is a reference to the interactive graphical surface.

```
glyph :: Picture
      -> Component
          (Glyph,
           DisplayHandle)
```

This separation helps programmers to develop more modular systems.

3.6 Compositional structure

Haggis provides layout combinators and encapsulating functions, that can combine the `Display Handles` mentioned above. This enables more complicated components to be built from simpler ones. For instance, an input device controller can be composed with a `Glyph` picture displayer, to produce an interactive widget.

```
widget =
  (gl,dh) <- glyph picture
  (kb,dh) <- catchKeyboardEv dh
```

The resulting widget would receive input and pass it on via the `kb` handle. It's image would be updatable via the `gl` handle.

4. THE EXAMPLE

In the past, these functional programming developments have been used to build very simple systems (Finne and Peyton Jones, 1995b). As a more complex example, we describe the development of a highly interactive, real-time user interface. It is a space ship game, as seen in Figure 3. Input commands occur via the keyboard. There are a number of enemy ships that move across the screen in waves. These will destroy the player's ship if they come into contact with it. The ship must also avoid hitting the hills at the bottom of the screen and the enemy base. The player's ship moves slowly forward until the enemy base becomes visible. The objective is to make it to as far as the enemy base and then shoot the base a given number of times. The current score will always be displayed on the screen. Though the graphics are fairly simple, the example does require real-time animation. The example also provides a highly concurrent system, with a number of different agents: lasers, enemies... The specification therefore concentrates on this concurrency.

5. LOTOS SPECIFICATION

In common with the CNUCE approach to interactors (Paterno, 1993), we specify our system in LOTOS. The game can be modelled as 4 different processes:

- a process for the lasers (`HandleLasers`);
- a process for enemy ships (`HandleEnemies`);
- an input process (`HandleInput`);
- a screen update process (`ScreenUpdate`).

These can then be synchronised over a number of events to allow the necessary communication. The `ScreenUpdate` process, discovers input events through the `Buffer` process.

```
process Game :=
   (ScreenUpdate
       ||
    HandleInput
       ||
    Buffer)
       ||
   (HandleEnemies
       ||
    HandleLasers)
endproc
```

In LOTOS, `a || b` means run process `a` in parallel with process `b`, synchronising over any common events. As an example of a process specification, consider the `HandleInput` process. It will continue to receive input events, and place them in the buffer until the `closeDown` event occurs. It interprets the input events before passing them on. For instance, it might change a press of the `"q"` key into the data `MoveUp`.

```
process HandleInput
            [sendChannel,closeDown,
             getKeyboardEv]
            (buffer:Buffer,
             kb:Keyboard) :=
  closeDown
 []
  getKeyboardEv!kb?inp:KeyboardEv;
  [valid inp] ->
   sendChannel!buffer!(interpret inp);
   HandleInput [sendChannel,closeDown,
                getKeyboardEv]
               (buffer,kb)
  [not (valid inp)] ->
   HandleInput [sendChannel,closeDown,
                getKeyboardEv]
               (buffer,kb)
endproc
```

In common with the interactor approach, we treat the system as a number of communicating interacting objects. We achieve a rendering of the state through presentation layer events, enacted by the `ScreenUpdate` process. Though we do not strictly adhere to the interactor model, we do model user level agents (`ScreenUpdate` and `HandleInput`) communicating with more application oriented agents (`HandleLasers` and `HandleEnemies`).

In common with Alexander (Alexander and Jones, 1990), we define the data manipulated by the processes in a functional style. This allows a more easily programmable notion of state, as data and operations can be packaged up, and combined with processes, in a modular style. This approach makes it easy to specify the behaviour of the lasers, enemies, ship and background. We therefore abandon the ACT ONE algebraic data specification language, normally used with LOTOS. ACT ONE lacks modularity and has been shown to be difficult to use and to implement. The new LOTOS standard, E-LOTOS, itself replaces ACT ONE with a functional approach because of these problems (Jeffrey and Leduc, 1996). Our approach overcomes some of the problems with a LOTOS style of specification. We now have both a behaviour (defined as a set of LOTOS processes) and internal state (manipulated by, and packaged up inside, each process), but with the behaviour still clearly visible on top.

Our approach gives us a clear, high level specification which designers may use to reason about a potential implementation. It is also possible to prove properties about the behaviour of the LOTOS specification using tools such as LITE (Eijk, 1991). This tool supports the automatic analysis and verification of concurrent specifications. Design errors may therefore be found prior to implementation, using this approach.

6. CONVERSION TO HAGGIS

We can convert LOTOS specifications into Haggis code as follows:

- events correspond to Haggis I/O actions (section 3.1); these IO actions may need to be defined. They can be input or output actions (presentation layer), usually predefined in Haggis, using the virtual I/O device concept, and structured graphics primitives (sections 3.3, 3.4.) They can also be communication actions, or can alter the state of the system (application layer);
- synchronous LOTOS communication can be implemented using our extended concurrency library (Sage and Johnson, 1997);
- data manipulation operations need not be altered, as they are already in a functional, executable form.

We can, for instance, implement the `HandleInput` process as:

```
handleInput :: Channel InputEvent
            -- an asynchronous channel
            -- (which is the buffer)

            -> Keyboard
            -- an asynchronous channel
            -- source of
            -- keyboard Events

            -> IO ()

handleInput buffer kb =
 do
   gate "closeDown"
   'choose'
   (inp <- getKeyboardEv kb
    if valid inp then
      sendChannel buffer (interpret inp)
      handleInput buffer kb
    else
      handleInput buffer kb)
```

The resulting code looks very similar to the earlier LOTOS specification. The `gate` action, provided by the LOTOS library, performs synchronisation on the given `"closeDown"` gate. The `Buffer` in the LOTOS specification, has become an asynchronous channel. The input is received via the asynchronous `kb` channel, using the predefined `getKeyboardEv` action.

We have now described what the system does. As a final step, we will show how to describe what the system looks like, and so build the full screen shown in Figure 3. Again, we are concerned with supporting full graphical interaction as easily as possible.

We build the screen with four types of operation. The `mkDC` operation creates a Display Context `dc`, which contains information about style values and the window that will be created. The `label` operation creates a label displaying the given string, which may be updated (using the `lbl` handle). The `glyph` operation creates a simple output area containing the specificed picture. The `catchKeyboardEv` operation makes the glyph interactive and able to receive keyboard events (via the `kb` channel). Finally the `realiseDH` operation renders the components on to a window using the `vbox` combinator to place Display Handles above one another, and using `hbox` to place Display Handles next to one other.

```
screen
 = do
    dc <- mkDC []
    (_,ldh) <- label "Score:" dc
    (lbl,ldh2) <- label "0" dc
    (gl,screendh) <- glyph screenImage dc
    (kb,sdh) <- catchKeyboardEv screendh
    realiseDH dc (hbox [sdh,vbox[ldh,ldh2]]
```

The resulting system can be run, and tested on users. At this stage, problems with the system can be discovered and used to reshape the initial specification. For example, early prototypes of the system did not provide on screen prompts about the current score. This was clearly necessary if users were to keep track of their interaction with the game.

7. CONCLUSIONS AND FURTHER WORK

Through our short example, we have shown that by using functional languages as an implementation tool, it is easy to convert from high-level specifications of concurrent user interfaces, into executable code. This work builds upon Alexander's work, but uses newer functional programming techniques, in particular concurrency and monadic I/O. This concurrency provides a number of benefits. For the programmer, it supports the development of highly modular, agent based systems. For the user, it supports highly interactive, real-time presentation techniques. The resulting interfaces can be subjected to rigorous user testing. It is important to emphasise, however, that high level specifications are still necessary. They make clear the behaviour of concurrent systems, and allow designers to reason about them. We could also use the specification to target areas for user testing.

We have shown, through our example, that Haggis is capable of implementing highly interactive, real-time interfaces, something most prototyping tools are incapable of. However, we need to try building systems with more complicated interaction, and therefore more interaction objects. We need to fully automate the translation between specification and executable Haggis code. This should be combined with some automatically generated graphical representation of the specification, which would make it easier for the designer to understand the interaction.

8. ACKNOWLEDGEMENTS

Thanks go to Simon Peyton Jones, and Sigbjorn Finne for providing Haggis for us to work with. Thanks also to Phil Gray and other members of the Glasgow Interactive Systems Group (GIST). This work was supported by UK EPSRC Grant No GR/K69148.

REFERENCES

Gregory D. Abowd (1990), Agents: Communicating Interactive Processes. In *Human Computer Interaction - INTERACT '90*, D. Diaper et al Eds. Elsevier Science Publishers, North-Holland, pp.143-148.

Heather Alexander (1987), Executable specifications as an aid to dialogue design. In *Human-Computer Interaction - INTERACT '87*, H. J. Bullinger and B. Shackel Eds. Elsevier Science Publishers, North-Holland, pp.739-744.

Heather Alexander (1990), Structuring dialogues using CSP,. In *Formal methods in Human computer interaction*, MD Harrison and H Thimbleby Eds, Cambridge University Press, Cambridge, pp.273-295.

Heather Alexander, Val Jones (1990), *Software design and prototyping using me too*, Prentice Hall.

David Duke, Giorgio Faconti, Michael Harrison, Fabio Paterno (1994), Unifying Views of Interactors. Technical Report WP18, ESPRIT BRA 7040 Amodeus-2, October 1994.

P.van Eijk (1991), The Lotosphere Integrated Tool Environment LITE. In *Proceedings 4th International Conference on Formal Description Techniques*, Sydney, November 1991, North-Holland, pp.473-476.

Sigbjorn Finne, Simon Peyton Jones (1995a), Pictures: A simple structured graphics model. In *Glasgow Functional Programming Workshop*, Ullapool, July 1995.

Sigbjorn Finne, Simon Peyton Jones (1995b), Composing Haggis. In *Proceedings of the fifth Eurographics workshop on Programming Paradigms in Graphics*, Maastricht, Sept 2-3 1995.

N.E. Fuchs (1992), Specifications are (preferably) executable. In *Software Engineering Journal*, September 1992, 7, (5), pp. 323-334

Andrew D. Gordon and Kevin Hammond (1995), Monadic I/O in Haskell 1.3. In Paul Hudak, editor, *Proceedings of the Haskell Workshop*, pp 50-69. La Jolla, California, June 25 1995.

M.D. Harrison and D.J. Duke (1995), The Specification of User Requirements in Interactive Systems. Technical Report WP60, ESPRIT BRA 7040 Amodeus-2, September 1995.

I.J. Hayes and C.B. Jones (1989), Specifications are not (necessarily) executable. In *Software Engineering Journal*, 1989, 4, (6), pp.330-338

J.M. Hoc, T.R.G. Green, R. Samurcay and D.J. Gilmore (eds) (1990), *Psychology of Programming*, Computers and People Series, Academic Press Ltd.

A. Jeffrey and G.Leduc (1996), E-LOTOS core language. Output of the Kansas City meeting, version 1996/09/20, (ISO-IEC/JTC1/SC21/WG7).

C.W. Johnson and M.D. Harrison (1992), Using temporal logic to support the specification of interactive control systems. In *International Journal of Man-Machine Studies*, 37, pp.357-385.

Brad A. Myers (1991) Separating application code from toolkits: Eliminating the spaghetti of callbacks. In *Proceedings of the ACM SIGCHI'91 Conference on User Interface Software Technology*. ACM Press. November 11-13 1991.

Fabio Paterno (1993), A methodology to design interactive systems based on Interactors. Technical Report WP7, ESPRIT BRA 7040 Amodeus-2. February 1993.

John Peterson et al (1996), Haskell 1.3: A non-strict, purely functional language. Technical Report YALEU/DCS/RR-1106: Department of Computing Science. Yale University. May 1996.

Simon Peyton Jones and Philip Wadler (1993). Imperative functional programming. In *ACM Conference on the Principles of Programming Languages*, pp 71-84. ACM Press, January 1993.

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne (1996), Concurrent Haskell. In *ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996.

Meurig Sage, Chris Johnson (1997), Executing LOTOS with Haggis. Technical Report, Department of Computing Science, University of Glasgow (in press).

Simon Buckingham Shum, Ann Blandford, David Duke, Jason Good, Jon May, Fabio Paterno, and Richard Young (1996), Multidisciplinary Modelling for User-Centred System Design: An Airtraffic Control Case Study. In *People and Computers XI: Proceedings of HCI '96*.