# Programming Reactive Systems in Hume: an Asynchronous Domain-specific Language for Real-time Systems

by

Meng Sun

B.S. (Beijing Institute of Technology) 2003

A Dissertation Submitted in Partial Satisfaction of

the Requirements for the Degree of

Master of Philosophy

in

Computer Science

at the

UNIVERSITY of ST ANDREWS

January 2006

I, hereby certify that this thesis, which is approximately ..... words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

Date 20/2/06 ...Signature of candidate ...

I was admitted as a research student in Sept , 2003 and as a candidate for the degree of M.Phil in Sept , 2003 the higher study for which this is a record was carried out in the University of St Andrews between 2003 and .

Date 20/2/06 ...Signature of candidate ..

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of MPhil in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date 20/2/06 ...Signature of supervisor

I, received particular assistance in the writing of this thesis in respect of matters of style, idiom, grammar, syntax or spelling, which was provided by

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

Date 20/4/06. Signature of candidate .

**Programming Reactive Systems in Hume: an Asynchronous Domain-specific Language for Real-time Systems**



by

Meng Sun

Abstract

**Programming Reactive Systems in Hume: an Asynchronous Domain-specific Language for Real-time Systems**

Meng Sun

2005

A real-time reactive system is one which continuously responds the stimuli from its environment and generates responses. A real-time system is crucial for time and space usage, because responses must be made in limited time and hardware resources for real-time systems are usually restricted. Computer control is widely used in every field of life and the number of devices which use real-time systems is increasing rapidly. There are programming languages for these devices, but they are still need to be improved.

In this dissertation, we present Hume, a language designed for real-time reactive systems. Unlike a normal programming language for real-time systems which is low-level, Hume adopts a high-level programming abstraction with the strong characteristic of guaranteeing space usage and time behavior. Hume can be used to write asynchronous, current and predictable applications with high reliability, which is preferable for real-time systems.

We also ported the Hume Abstract Machine Interpreter to a resource-bounded system and implemented a Hume program to prove that Hume has the ability to program for resource-bounded systems by providing a high guarantee of space usage. We have demonstrated Hume implementations for three types of real-time reactive systems to show Hume's abilities for programming real-time systems. We have also obvious research results under three different systems which emphasize that Hume is a safety-critical language for real-time reactive system which guarantees space usage and can react in limited time.

# Chapter 1

# Introduction

## 1.1 The Problem

Complex systems can be separated into three types [77]: *transformational systems*, *interactive systems* and *reactive systems*. Transformational systems, such as traditional computing programs, operate on the initially available inputs and stop after outputs have been generated; interactive and reactive systems do not terminate (unless a fatal error occurs) and will continuously interact with the environment. Interactive systems only operate on inputs at their own speed when the system is ready to deal with those inputs, for example in a multimedia network application. Finally, reactive systems follow a pace dictated by the environment; whether systems react and provide input or not will be determined by the environment. Reactive systems will continuously react to their environment by sending back a response.

When a reactive system is concerned about the time limitation of response to inputs, it can be defined as a real-time reactive system, or real-time system. A real-time reactive system is a system that is asked to react to stimuli from the environment within time intervals dictated by the environment [1], or any information processing activity or system which responds to externally generated input stimuli within a finite and specified time [4].

A good real-time programming language must not only meet the requirements of a general programming language, but also the requirements of real-time systems.

Real-time systems (even simple ones like a digital watch or complex ones like a satellite communication system) can be categorized as reactive systems. First of all, a real-time reactive programming language must meet the requirements of a reactive system. A good real-time language should have the following characteristics [68]:

- *Reactivity*: Real-time systems act as a kind of reactive system, continuously reacting to environment by sending back a response. A reactive program responds to input events from a host system by producing output events that are transmitted as signals to the host. To be suitable for programming real-time systems the programming language must first meet the requirement of reactivity.
- *Concurrency*: A real-time programming language must be able to deal with concurrent events. Applications of a real-time system usually need to interact with the environment. Events in the real world are often concurrent [25]. A good real-time language should be able to manipulate shared data and allow components to run as simultaneous threads.
- *Asynchronicity*: Components of some real-time systems work in discrete steps and some work continuously. A good real-time language must be concerned with asynchronicity, so that systems can respond to inputs without any delay from internal interactions.

Since real-time systems are concerned with time and many real-time systems run on resource-bounded hardware, such as real-time embedded system, they have their own characteristics beyond those of simple reactive systems:

- *Resource boundedness*: Many real-time reactive systems run in a limited space and will also have time constraints. The time is critical because a long delay for the inputs will cause severe problems in the real world. So a good real-time language should have the ability to face the challenge of resourced bounded hardware and be able to respond to the inputs in restricted time.
- *Manipulation of Data Type*: many real-time systems cannot avoid the control of some engineering activities,such as complex models of control, and distinct but

related numbers of controls, mean real-time systems must act as computing systems, for example a PDA or mobile system. Therefore, a language designed for a real-time system must provide multiple data types to define those controls and be able to deal with complex model.

- *Safety*: An application running in a real-time system always reacts to the real world, and reliability and safety are crucial in real-time systems. A real-time programming language must be safe enough and predictable to avoid any failure. Every tiny failure in a real-time system may cause an unrecoverable and severe result. Safety is more important in programming real-time systems than in programming normal systems.
- *Interrupt Handling*: The performance of embedded systems depends to a great degree on interrupt handling, which is the most important and frequent way of exchanging information. A real-time programming language should have flexible and powerful mechanisms for interrupt handling.

Of course, a real-time language is still a kind of programming language, so it should also support the basic programming language's characteristics:

- *Ease of Coding*: With and increasing complexity of tasks, programming with a language which cannot allow flexibility in large-scale tasks means that results become more and more unpredictable and uncontrollable. Correspondingly, a real-time language should have clean semantics and function library support. This makes it easier for programmers to expedite the development process, and limits the time spent in debugging and maintaining the system.
- *Portability*: a real-time language usually needs to operate at a very low level to interact with the computer hardware and control the microprocessor. This will cause languages to become more processor-specific and less portable. This in turn increases the difficulties of porting the applications between systems. A real-time system programming language must be portable enough to be of benefit for software reuse.

- *General Applicability*: Real-time systems are large and complex. Data size fluctuates because of the continuous changes of the real world and complexity is caused by the variety of different real-time systems in every field of our life. Considering this characteristic of real-time systems, a good language should allow easy reuse of previously constructed software. An application is hardly ever used only for real-time systems, and a real-time system is large and complex. If an application is hard to port to other real-time systems, it cannot be extended to the real-time world.

Real-time programming is an essential industrial activity. The choice of programming language is crucial. The programming tools currently used are often specific and low-level. Low-level programming languages are processor-specific and less portable than a high-level language and will not remain acceptable in the long-term for large safety-critical programs [10].

## 1.2 Motivation

A good programming language targeting a certain domain should meet the requirements of this domain. In the case of real-time reactive systems, a good programming language should be able to generate efficient programs not only with concurrency, asynchronicity and reactivity, but also with a high guarantee of resource bounding and correctness. There are certain programming languages that have appeared for this domain, but the programming tools are still far behind. More efficient programming tools, which can fully capture the essence of the real-time domain, are necessary.

This dissertation presents Hume, a novel domain-specific programming language targeting real-time reactive systems, providing a high-level programming abstraction whilst maintaining properties that meet the low-level requirements of real-time systems. For example, Hume provides exception handling, higher-order

functions, type polymorphism, and recursion in different levels of the language. The most important property is that the language design meets the requirement of transparent time and space costing. We will show that Hume can meet the requirements of resource bounded systems, based on our effort with the practical Hume implementation of a resource bounded system, and we will provide research results concerning bounded space usage and concrete time behaviour.

## 1.3 Contribution

The contributions of this thesis are threefold. First of all, we have ported our Hume Abstract Machine to a resource bounded system (Symbian OS) and implemented a real-time Hume program related to the Symbian OS environment with guarantees of space usage and time. Secondly, we have given Hume implementations and software solutions for three typical types of real-time reactive systems. We have extended functional programming into the Symbian OS and provided a feasible way to write program for Symbian OS using Hume. Last but not at least, we have analysed the research results of space usage and time behaviour for Hume programs under two different types of system (Linux and Symbian OS). Associated with the research results under RTLinux obtained by previous research [3], we can show that Hume is a predictable, correct, reactive and concurrent programming language with a high guarantee of bounding in time and space usage, which is suitable for programming real-time reactive systems.

## 1.4 Structure

The structure of this dissertation is as follows: Chapter 1 is the introduction, which will present the research problems and our arguments and the motivations. Chapter 2 gives related work, where we will discuss the merits and demerits of related languages for real-time reactive systems. Compared with the characteristics of

those languages, we will represent Hume for real-time reactive systems. Chapter 3 summarizes how the Hume language meets the requirements of real-time reactive systems and gives a detailed description of Hume. Chapter 4 describes our port of the Hume Abstract Machine to the Symbian OS and some simple Hume applications both related to and independent of the Symbian OS environment. Chapter 5 presents Hume implementations for three different types of real-time reactive systems, which demonstrate Hume's characteristics of reactivity, concurrency, asynchronicity and the ability to handle exceptions and interrupts. We also provide more realistic evidence to demonstrate the ability of Hume ability to program real-time reactive systems with restricted memory space and in a harsh execution environment. Chapter 6 represents our research results of space usage and time behaviours based on two types of operating system (Linux and Symbian OS), as well as the previous research results based on RTLinux to show that Hume can make it possible to write a safe and correct program for safety-critical systems (such as a real-time embedded system) with the guarantee of space usage and time. Finally, Chapter 7 summarizes our conclusions and makes suggestions for future work.

# Chapter 2

# Related Work

## 2.1 The Requirements of Real-time Languages

The canonical definition of a real-time system is by Donald Gillies, and says, "A real-time system is one in which the correctness of the computation not only depends upon the logical correctness of the computation, but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred."

As we mentioned in the chapter one, the requirements of real-time languages can be summarized as follows:

1: Predictable and determinable: All the execution of a program written by the real-time languages should be determinable and predictable in the restricted environment.

2 Resource bounded**:** All the languages should be able to write a program that can be executed in a resourced bounded systems ,so as to meet the constraints of both time and space requirements of real-time systems.

3 Concurrency: Languages must have the construction that allows the independent communication between computing units.

4 Correctness: Languages must guarantee that the constructed systems can meet their formal requirements.

5 Reactive and Asynchronization: The languages must allow input response in restricted time and will not influent other environmental and internal operations too much.

## 2.2 General-purpose Languages

Choosing a programming language for real-time systems essentially requires specifying the timing requirements of the system and ensuring that the system performance is both correct and timely. Machine-specific languages, such as assembler, were originally used for real-time systems. Whilst code written in assembler can be fast and yield good response times, it is not always easy for programmers to understand, and because it is processor-specific it is less portable than a high-level language.

In order to maximize application portability and make a smaller "semantic gap" between the programmer and the language, C [60], C++ [61], Ada [11], Modula [62] and Java [14] (languages with high-level abstractions) have been brought into use for real-time systems programming.

C is a systems programming language, aimed at efficiency, just like assembler, but also at facilitating coding in the manner of a high-level language. C is popular in embedded systems programming since its introduction in the 1980s. C++ represents an effective improvement on C by supporting modern software engineering, OOP and program, structures.

Ada is a general system development language, which supports modularization, independent compiling and co-processing, and which aims to provide dependable, maintainable and readable code [11]. In order to support OOP (Object-Oriented Programming) better, Ada has been improved as Ada95, which is currently used extensively. Specific Ada expressions are used to map the code to particular hardware.

Modula-2 is a computer programming language that is designed to be broadly

similar to Pascal [27], the important addition of the 'module' concept, and direct language support for multiprogramming. Supported by powerful data type checking and abundant low-level hardware accessing, Modula-2 can be used to implement an integrated real-time system without the assistance of assembling language. Modula-3 [63], another system developing language, related to Modula-2, improves on Modula-2 by adding co-processing, OOP, garbage collection and support for C and UNIX.

Another language that should be mentioned here is RTSJ (the Real-Time Specification for JAVA). This attempts to bring the benefits of Java to real-time programming. The RTSJ recommend a limited set of modifications to the language specifications and the Java virtual machine specifications. The RTSJ also establishes a common lexicon for real-time computing and a consensus on requirements. Because of the standard's carefully crafted resource management mechanisms, Java technology-based applications written to the Real-Time Specification for Java develop the unique ability to provide predictable performance as well as unprecedented portability across multiple RTSJ-compliant Java Virtual Machines. In contrast to SPARK Ada, without any formal guarantees, RTSJ provides specialized runtime and library support for real-time systems programming and a less controlled but much more expressible environment [7].

Great efforts have been made in these high-level languages in order to meet the specifications of real-time systems:

- Compilers are available for the chosen real-time operating system and its hardware architecture. They are also available on multiple operating systems and microprocessors. This is particularly important if the processor or the real-time operating system needs to be changed in the future.
- The language often allows direct hardware control without sacrificing the advantages of a high-level language.
- The language usually provides memory management control such as dynamic and

static memory allocation.

- In order to re-use general-purpose modules, which leads to faster development of programs and helps to reduce the time spent debugging, modular programming by using Object Oriented (OO) or structure has been applied. Using a language that supports modular programming is definitely helpful for real-time programming.

Although all these traditional languages make an effective improvement, they are "imperative" in the sense that they consist of a sequence of commands executed strictly one after the other, which limits the abstraction of these high-level programming languages. For example, C consists essentially of a carefully-specified sequence of assignments, and in Java, the ordering of method calls is crucial to the meaning of a program. Imperative languages are based upon the notion of sequencing, which they can never escape completely. The only way to raise the level of abstraction in the sequencing area for an imperative language is to introduce more keywords or standard functions, thus cluttering up the language and making the programming of complex systems even more difficult [78].

On the other hand, it is now generally accepted that modular design is the key to successful programming. To increase one's ability to modularize a problem conceptually, programming languages must provide effective methods to join models together. However, traditional programming languages are usually good at decomposing problems into pieces, whilst lacking the efficient connection to join them together.

Finally, although there are some tools being introduced to assist the understanding of time and space behaviour in some general-purpose programming languages, for example Ada and RTSJ, general-purpose programming languages do not consider resource-boundedness in their original design. This increases the difficulties in guaranteeing the space usage in resource bounds and time behaviour for interacting with the real-time system. The reliability of real-time systems is thus decreased by this unpredictability.

General-purpose languages are still at a low level of abstraction without guarantees of resource consumption [37]. The cost of development and maintenance is usually high and the software productivity and reliability are low. We can conclude that general-purpose languages are not the best choice for large, complex and reactive systems.

## 2.3 Synchronous Programming Languages

In the real world, the values of signals are different depending on the time they are observed. We want to calculate the sum of Ea and Eb at time T. After we have finished calculating Ea at time T, T has to be advanced to T'. So the precise result should be the sum of the Ea at T and the value of Eb at T'. In order to avoid complexity and confusion, the synchronous data-flow notion of computation has been proposed for language design. Synchronous languages such as SIGNAL [26, 23], LUSTRE [16], ESTEREL [33] and the visual formalism Statecharts [34] assume that the computation has zero duration and manipulates the signals as integer. All events occur instantaneously and the logical time between the appearance of consecutive events has been ignored as being zero [17].

ESTEREL is a language devoted to programming hardware reactive systems focused on producing discrete output from input signals [33]. Compilers are used to translate ESTEREL codes into finite state machines for embedded applications. ESTEREL has a good performance on hardware design and performance-critical systems with the help of these compilers which guarantee optimizations and can generate efficient code. However, ESTEREL is weak on data-processing and is not suitable for data-dominated [18] reactive systems which continuously produce output values from input ones.

SIGNAL's central concept is a *signal,* which is a time-ordered sequence of values. The SIGNAL language specifies the relationships between signals by

constraints and a SIGNAL program is a set of constraints on signals with the purpose of solving the constraints [30]. When the system is poorly defined, a solution for the constraints may not exist, or more than one solution may be found for a SIGNAL program. This causes the system to be non-deterministic and increases the difficulties for a programmer in finding a solution for a program. Hence SIGNAL programs will be difficult to understand and maintain in undefined systems (for example real-time systems).

LUSTRE is a synchronous language based on the notion of a sequence which is quite similar to SIGNAL [31]. LUSTRE programs make the improvement of defining signals by functions rather than constraints and make the system deterministic [16].

We can conclude that these languages meet the synchronous requirement of a reactive system and make the programming of reactive systems easier and more reliable. However, they allocate memory statically to guarantee resource bounds, so this decreases the flexibility in terms of expressing systems which change functionality at run time. Further, they are designed on the basis of the theory that time advances in discrete steps, so they are not able to provide adequate abstraction for continuous quantities.

## 2.4 Functional Languages

### 2.4.1 Introduction to Functional Programming

Functional programming is a method of programming that emphasizes the evaluation of expressions, rather than the execution of commands [39]. The expressions in these languages are formed by using functions to unite basic values. A functional language is a language that supports and encourages programming in a functional style [38].

For example, we want to calculat the sum of the integers from 1 to 20. An imperative language, such as C, may have a looping construct (such as *for* or *do* ...

*while)* to calculate the integers. It has to employ a variable with an initial value, a counter and a constant used to add the increment value to the variable.

```
a = 0;
for (i=1; i<=20; ++i)
 a += i;
```

However, in a functional language, for example in Haskell, the same loop would be expressed in one sentence, without any variable updates. The result can be calculated by evaluating the expression:

```
sum [1..20]
```

Here, [1..20] is an expression that represents the list of integers from 1 to 20, and sum is a function used to calculate the sum of the list of values.The definition of Sum is in the following:

```
sum xs = foldl (+) 0 xs
foldl1 f (x:xs) = foldl f x xs
```

Actually, there are many noticeable benefits of the functional language approach for real-time programming in that it is easier to write, more reliable, and easier to maintain.

Purely functional programs contain no side-effects at all in the sense that a function call can have no effect other than to compute its result. This eliminates a major source of bugs and relieves the programmer of the burden of prescribing the flow of control. The semantics of functional languages also usually yield a closer match to the problem than those for imperative languages, which improves programmer expressivity.

Most functional languages are strongly typed, avoiding a large class of easy-to-make errors at compile time. In particular, strong typing means there is simply no possibility of treating an integer as a pointer, or following a null pointer, and it really helps maintain the correctness of real-time systems.

In order to assist modular programming, functional programming languages provide higher-order functions [28].Functional programming languages not only

enable simple functions to be integrated together to make more complex ones, but whole programs can also be integrated together, which means that a complete functional program is just a function from its input to its output.

However, most early functional languages are not good for resource-bounded systems because they do not consider space usage and time behaviour.

### 2.4.2 Functional Programming for Real-time Systems

The main criterion for a functional programming language to be suitable for the real world is that the program is written primarily to perform some task, not primarily to experiment with functional programming [24]. Such languages include CAML [43], Erlang [44, 35], Haskell [41] and Standard ML [87]. The main advantages of functional language programming for real-time systems are compositionality, ease of reasoning and structured programming. Strong typing for functional languages eliminates a large number of programming errors; typical modern language designs incorporating automatic memory management decrease errors caused by poor manual memory management; higher-order functions encapsulate pattern of computations; polymorphism abstracts over data structures and recursion allows a number of algorithms, especially involving data structures, to be expressed more naturally and in a less error-prone way[8].

Typical and more accepted functional languages such as LISP [40],have programs made up of small functional components, where each component has its own purpose and is quite easy to comprehend. Each function only has a few lines and each function is also comprehended easily. LISP focuses on list processing, using recursion over lists instead of loops in programming. However, the weakness of LISP is that it does not have any concept of static types, and the compartmentalization between type and data is not ideal. This makes the implementation more costly in terms of both time and memory. We can conclude that LISP provide us a functional programming approach, however, can not meet any requirements of real-time systems.

Haskell, a pure functional programming language, provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic data types, pattern-matching, list comprehensions, a module system, a monadic I/O system, and a rich set of primitive data types, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers [41]. Strong typing can eliminates a lot of programming errors; polymorphism abstracts internal details to data structures; higher-order functions abstract over common patterns of computation; and recursion allows a large number of algorithms, especially the algorithms in data structures will be more naturally and clearly expressed. The programming errors will be decreased in a larger extent. The introduce of automatic memory management make the memory management more portable and eliminates errors caused by poor manual memory management[8].These characteristics make Haskell can meet the real-time language requirements of correctness and determinacy, but fail to meet the requirements of concurrency, asynchronicity and resource bounded. The concurrent Haskell[20] add concurrency and asynchronicity mechanism, which make Concurrent Haskell can meet the real-time language requirements of concurrency and asynchronicity. For example, a Concurrent Haskell program can implies multiple "threads" running within a single Unix process on a single processor[84].However, it still can not meet the requirements of resource bounded and does not provide any guarantee of time and space usages.

The last functional programming language to be considered here, which is popularly used in the real-time world, is Erlang. This is a concurrent functional programming language designed for manufacturing real-time systems. Erlang has explicit concurrency and incorporates asynchronous message passing. It figures out which modules are needed, and allows recursion equations, pattern matching syntax, and dynamic code substitution, allowing very tightly packed and clear programs to be written. It incorporates transparent cross-platform memory allocation and real-time garbage collection[9]. Erlang meets the real-time language requirements of

concurrency, asynchronicity, programming for resource bounded system, however it did not provide predictable mechanism to guarantee the time and space usage so as to make the system determinable and guarantee the correctness. It can not meet the requirements of determinacy and correctness.

According to the development of the real-time domain, what is demanded is more complex and comprehensive: *responsiveness*, which means a response must be made to every input; resource-boundedness, which means responses must happen in a limited amount of time, using limited hardware, *concurrency* and *networking or distribution* which have been mentioned before [59]. None of the languages covered seems to meet perfectly the requirements for programming with functions in real-time systems. There are two major reasons. First of all, all of these languages allow incomplete functions definitions, which lead to the potential for the program to contain non-terminating computations. Secondly, they disable the programmer's ability to directly control program execution[78]. For example, ML [42] and Caml are typical functional languages with properties allowing partial and imperative evaluation and calculation.

## 2.5 Hume for Real-time Systems

Having considered the advantages and disadvantages of three types of programming languages for real-time and reactive systems, we now turn our attention to Hume, which is a domain-specific programming language targeting resource-bounded computations, such as real-time embedded systems. Hume is based on a functional notation and is novel in being based on generalized concurrent bounded automata, controlled by transitions characterized by pattern matching on inputs and recursive function generation of outputs[46]. Compared with general-purpose languages such as C and Ada which provide low-level abstractions but without guaranteed space or time usage, Hume provides a higher level of abstraction but with a semantics specifically designed to support formal reasoning

about cost. Like RTSJ, which is designed for soft real-time applications, Hume has the ability to deal with exceptions using a strong exception handling mechanism. Moreover, unlike RTSJ it also supports dynamic memory allocation and has a restricted form of automatic garbage collection, similar to the *region* approach popularized by Tofte and Talpin [80].

Compared with synchronous languages, which fail to deal with continuous quantities, Hume uses an asynchronous approach [45] to provide both expressiveness and realism. Hume deals with communication by an asynchronous model and supports asynchronous execution of concurrent processes [48].

In the programming of real-time systems, weakness of memory management is one of the reasons for poor control of safety and reliability. As mentioned above, a number of languages, especially functional programming languages, cannot meet the resource bounded requirement of real-time systems. Although the three types functional languages we mention above can meet some requirements of real-time system, none of them can provide bounded time or space guarantee. In comparison with some functional languages such as Scheme and Lisp, which cannot meet the safety requirement because of weakness of types, Hume's definition does support a wide range of basic types, for issues of type coercion and type safety are fundamental to ensuring both correctness and safety. Like Haskell, the Hume expression language is purely functional and provides higher-order functions, strong polymorphism typing and recursion. Unlike Haskell, Hume provides concurrency and asynchronicity and has a strict semantics that makes it easier to construct accurate cost models. In contrast with concurrent Haskell, Hume provides a guarantee of space usage and time bounding by exploiting bounded time and space models.

In conclusions, Hume not only provides a high level of abstraction including polymorphism type inference, automatic memory management, higher-order functions, exception-handling and a good range of primitive types like other languages, but also supply a gap of other functional language by supporting rigorous

cost and space analyses.

# Chapter 3

# Hume Overview

Hume is a domain-specific functional programming language which can meets the real-time system requirements of predictable, concurrency, asynchronization, correctness and resource bounded. In the following section, we will demonstrate how Hume meets those requirements.

## 3.1 The Layered Design of Hume

Hume is a multi-layered language design, comprising a total of three layers (see figure 3.1). The *coordination layer* is a finite state language for describing interacting processes, external properties and configurations of boxes. The *expression layer* is a purely functional language with strict semantics, describing one-shot processes and input/output transitions within boxes [19]. The outermost layer is a static declaration language which provides definitions of types, streams etc. to be used in the dynamic parts of the language. Both coordination and expression languages share a single rich and polymorphic type system [47]. A multi-layer design separates Hume into different parts, each with its own specific capability. This makes Hume a combination of a high-level language that can easily abstract the state and action of the real world, and a low-level language which enables the programmer to control and interface with a real-time system.

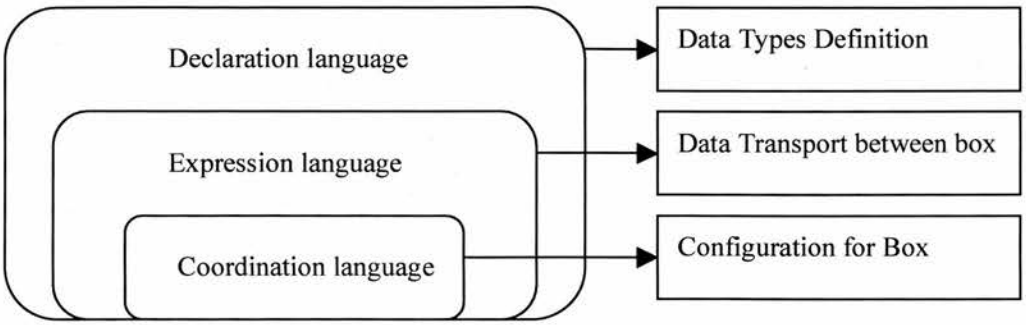


Figure 3.1 Language Layers for Hume

## 3.2 Strong Typing

The fundamental types are quite important for real-time systems. The well-defined types will allow the construction of realistic programs without requiring complex implementation in the initial stage. Hume has user defined data types of arbitrary size. Hume provides a wide range of scalar types, including fixed-precision integers, floating-point values, variable-sized word values, fixed-exponent real numbers and exact real numbers, booleans and characters. Hume also has a concise definition of the conversions between values and types. In addition to basic types, Hume provides three kinds of structure types: vectors, tuples and lists. Vectors and tuples have a fixed size, while lists have an arbitrary size [47].

Types and values will be introduced by the declaration language and are involved in coordination and expression languages.

## 3.3 Asynchronicity and Concurrency

### 3.3.1 Box and Wire

Hume is based on a generalization of standard FSA (Finite State Automata) transition notation and incorporates concurrent processing through explicit multiple communicating FSAs, which are called boxes. The operation and data transfer in a real world are abstracted in Hume by generalizing a FSA to a box with input and

output wires. A box is high-level abstraction of a finite state machine with inputs/output values and old/new states. The expression form of a box is: (oldstate,input...)->(newstate,output...).The (oldstate,input) is like a tuple pattern which will match the current state and input values, and (newstate,output) is the expression pattern that return the results of the process based on the input values and current states. The left pattern matches the inputs and current states and results will be generated by the functions and rules on the right pattern. The expression form can also be abstracted as: pattern -> expression. For example the *negate* box in the following code:

```
stream output to "std_out";
type integer = int 64;
box negate
  in  (din::integer,countin::integer)
  out (dout::integer,dold::(integer,integer,char),countout::integer)
match
 (1,c) -> (0,(1,c,"\n"),c+1)
|(0,c) -> (1,(0,c,"\n"),c+1);

wire negate (negate.dout initially 0,negate.countout initially 0)
(negate.din,output,negate.countin);
```

The *in* is the input of a box and *out* is the output of a box. When a input have seen transferred to a box, their will be matched based on the rules provided by the *match* section. The related outputs will be generated and transfer to wire.

The box will only write outputs to wires if the previous outputs have been consumed. Otherwise the box blocks on output. Both inputs and outputs must be correctly typed in fixed size. The types of inputs and outputs can be any Hume types, excluding a function, stream, port or exception [47].

We can describe a box body as a single function with input and output values. Operationally, a box can cycle repeatedly, trying to match transition patterns against the current values on the input wires, treated as a single top-level tuple value. In the

Hume context, a box is implemented as a thread without any disturbances, taking the inputs, and generating outputs are based on the function rules. More specific examples will be demonstrated in Chapter 4.

Wire is a declaration of connecting boxes to form a static network process. The main functions of wires are to provide a map between input link and output link.

### 3.3.2 Concurrency

Compared to Haskell, Hume introduces concurrency to program. Concurrency has been implemented by the special management of input and output values.

A basic execution cycle of a box can be described in five steps: Firstly, a box will check the inputs' availability and load the available input values. Secondly, inputs will be matched against rules. Thirdly, all inputs will be consumed in fixed sized buffers. Fourthly, a box will add variables to input values. Finally, the box will write outputs to the corresponding wires [49].

Communication buffers will be created for one-to-one correspondence between input and output values in the first step. Hume types used for input and output are of fixed size. This provides a guarantee that all buffers are bounded by size and prevents the occurrence of synchronization problems when no buffering is used. Available values will be loaded to the bounded size buffer and nothing is consumed (saved to the buffer) until after a match is found. If there is no input value consumed in the wires buffer, output values cannot be written to this buffer and the box will be blocked. This avoids the box generating all the outputs without enough inputs. Once a cycle has completed, a box will not start another execution step until all outputs have been written to the corresponding wire buffers. This definition introduces concurrency to the Hume program.

### 3.3.3 Asynchronicity

In order to introduce asynchronicity into Hume, two primary coordination constructions are added to the Hume design. Compared to the basic execution cycle, asynchronous coordination construction allows boxes to ignore certain inputs in the

first step and recognized outputs will be written to the wire buffers. Boxes will reorder match rules according to the fairness criteria in the final step. Only certain inputs, instead of all inputs will be involved in a given box execution cycle and not all outputs will be required. In the case of fair matching, rules may be reordered.

```
box merge
in  ( xs :: int 32, ys :: int 32)
out ( xys :: int 32)
fair
(x, *) -> x | (*, y) -> y;
```

For example the fair merge operator defined as (x,*)->(x).The * pattern indicates that the value of a corresponding position should be ignored and this pattern will match any input without being consumed. The ignored wires cannot be empty because the availability of values will be loaded at the start of box execution rather than checked during individual matching.

Hume uses notions of fairness whereby each rule will be used equally, given a stream of inputs that match all rules [5].

## 3.4 Behavioural Correctness

As we have seen earlier, Hume has a three-level structure. A multi-layer design makes Hume a readable and more easily controlled language. Strong cost mechanisms are used in the expression layer. This allows results to be produced in bounded time and space. The overall computation cost can be calculated from the costs of each box. The use of a functional expression language and simple process design yields straightforward language semantics [51]. A cost model is used for deriving both bounded time and space costs, to guarantee the correctness of programming for real-time systems.

### 3.4.1 Space Cost Model

The behavioural correctness of a Hume program can be guaranteed, because Hume provides a space cost model to predict the heap and stack usage based on the prototype Hume Abstract Machine (pHAM) [6]. The prototype Hume Abstract Machine compiler will calculate the memory requirement of a Hume program in detail, including total dynamic memory usage and other memory usage. The dynamic memory usage is obtained from the heap and stack usage of boxes and wires. The other memory usage can be constrained at compile-time based on the size of strings and the number of boxes, wires and functions which we used in the specific program. In the Hume cost model, the space usage for boxes and wires, all types expression including scalar values, function applications and type declarations of declaration level such as data structure declarations are given costs by cost rules as shown below. Figure 3.2 shows the cost rules for Hume representative expressions.[50]

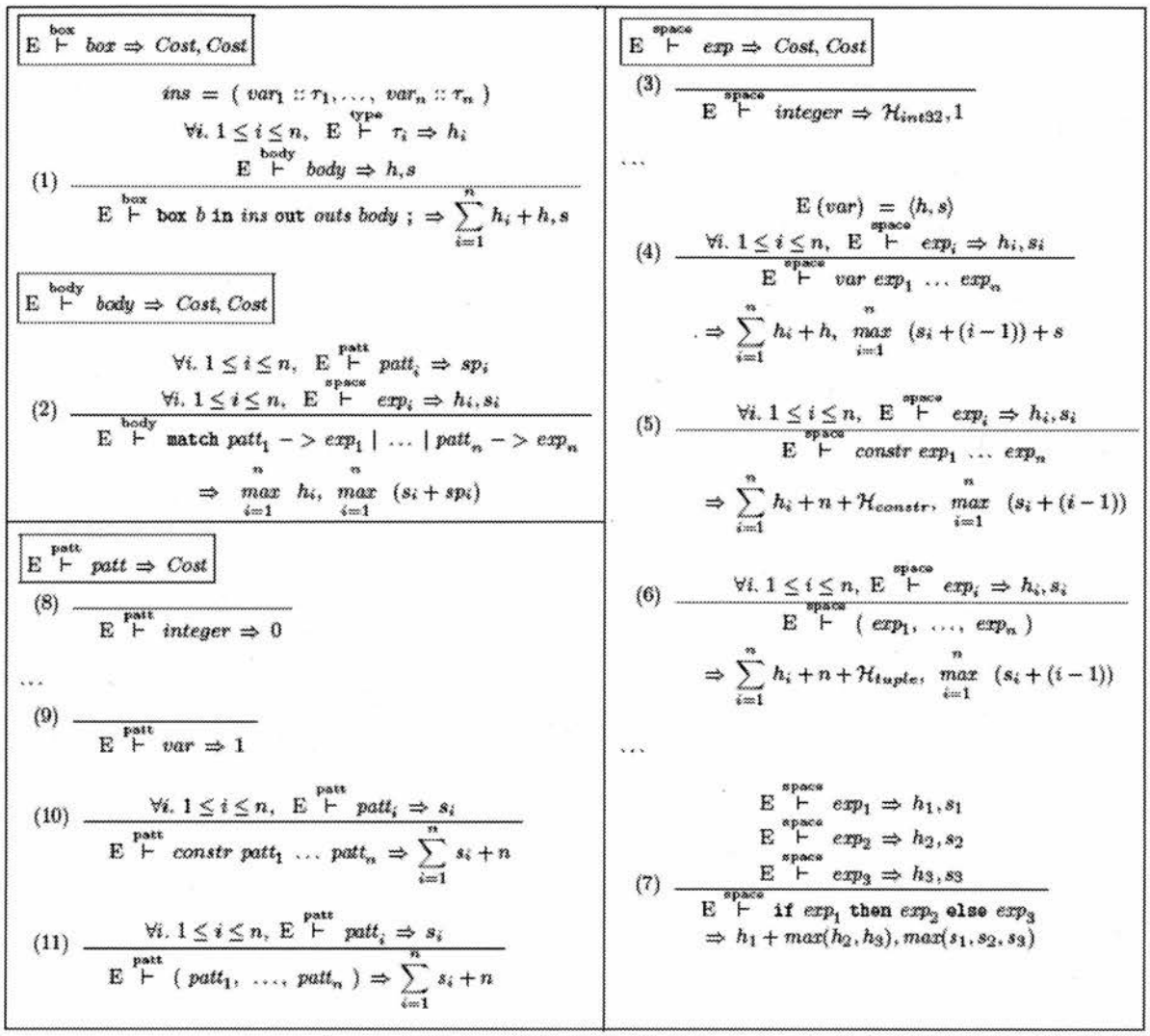

Figure3.2 Cost Model Rules[50]

## 3.4.2 Notation for Low-level Behaviours

As a language designed for a real-time system, Hume incorporates the notion of low-level behaviours for real-time systems, such as exception handling, time, scheduling and device abstraction for real-world environments.

**Exception Handling**

Hume supports exception handling for both system level and process level exceptions. Exceptions may be raised from within the expression language. Hume

provides a handler for each exception specified in the handling clause of a box. The handler will not perform any computations for boxes. For non-system exceptions, which can be raised by any expression in the body of boxes, a specific handler must be provided for those exceptions. For system exceptions, these can be handled either by explicit handlers or by general system handlers. If a handler has been defined for a system exception, the exception will be handled by this explicit handler. If there is no handler defined for a system exception, a system handler will be called when a system exception appears. Here is an example of an exception [47]:

```
exception Div0::string 10
            f n= if n==0 then
raise Div0 "f" else (n/0) as string 10;
box example
  in(c::char)
  out(v::string 29) handles Div0;
unfair n->f n
handle
```

An exception (Div0) has been defined and raised in the expression layer and handled in the body of a box. **unfair** is a system function. Exception handlers are not allowed within expressions because this reduces the cost of handling exceptions and maintains a pure expression language, as well as simplifying the expression cost calculus [51].

**Timing**

Hume can provide a hard guarantee on execution time by the hybrid static/dynamic approach to modelling time. Dynamic time requirements can be introduced by either the expression level or the coordination level. As the following code in digital watch examples shows:

```
Box button
  in(c::Char)
  out(a::BUTTON)
match
   ('a')->(Adjust)|
   ('u')->(Update)|
   ('s')->(Select)|
   ('m')->(Mode)  |
      _ -> *
handle  Timeout () -> TimedOut;
stream input from "std_in" within 1ms raise Timeout;
```

In the expression layer, internal construction is used to restrict the time. When an expression exceeds this time, a timeout exception will be raised. In the coordination layer, a timeout can be raised in both input and output wires. Timeout exceptions will be handled by an explicit handler at the box level, as with normal exceptions. In this way we can guarantee the execution time for boxes, allowing hard real-time timing constraints to be expressed; for example, we can read an input within a stated defined time.

**Scheduling**

A box is implemented as one thread by our Hume Abstract Machine (HAM) Interpreter. Threads are scheduled under the control of a built-in scheduler, which currently implements round-robin scheduling [51]. According to the following code, the thread scheduling algorithm shows that a thread is *runnable* if the required input values are all available for rules execution. For the thread i, thread[i].required[j,k] is true if input k is required to run rule j of that thread.

```
for i=1 to nThreads
    do runalbe:=false;
        for j=1 to thread[i],nRules
           do if !runalbe then runable:=true;
```

```
                do runable&=thread[i].required[j,k]=>
                    thread[i].ins[k].available
                endif
        endfor
    if runable then schedule(thread[i])
    endif
endfor
```

**Devices**

Hume provides a connection to real-world environments. Hume supports five kinds of devices with directionality, an operating system designator and a time specification, which is used to enable periodic scheduling and ensure the events are not dropped. These five devices are interrupts, memory-mapped devices, buffered stream (files), unbuffered fifo streams and ports [3]. The connections between devices and boxes can be implemented by wiring the devices to boxes using normal wiring declaration. The following example shows an operation for reading a mouse and returning the values when the mouse button has been pressed. *mouseport* has been wired to the input of the example box. When the mouse button has been clicked, a value will be returned and read as an input for the example box.

```
box example in(in::bool)out(out::bool)
unfair mouse->mouse
wire mouseport to example.in;
wire example.out to mouseuser.mouse;
```

**Recursion**

Unrestricted general recursion will cause a program to run without terminating in some situations. For example, recursing over continuous input without a clear

terminating condition may cause the program to loop or cause a memory-restricted real-time system to crash through memory exhaustion. The simplest and most straightforward way to ensure termination is to forbid recursion [56].

We can distinguish five distinct levels of Hume: HW-Hume (Hardward Hume), FSM-Hume (Finite State Machine-based Hume), HO-Hume (High-Order function Hume), PR-Hume (Primitive Recursion Hume) and Full-Hume (Full recursion Hume). HW-Hume is the most primitive level, which does not provide functions or recursive data structures. In FSM-Hume and HO-Hume, Hume prohibits recursion in function definitions. Instead, Hume allows iteration through boxes. HO-Hume adds higher-order, non-recursive functions and PR-Hume adds primitive recursive functions and Full-Hume provides the full recursion.

## 3.5 Resource Bounded

Unlike other functional languages such Haskell, concurrent Haskell, Erlang, and Lisp, Hume can meet the real-time requirement of resource bounded. Chapter 6 will demonstrate the specific research results to provide that Hume can meet the requirements of resource bounded.

## 3.6 Conclusions

Based on Leveson's guidelines: both safety and security risk analyses result in constraints (which may be regarded as negative requirements) that can conflict with functional and other performance related system requirements. We can consider high consequence systems, where the design cost is much smaller than the potential cost of system failure. [54] In general, a programming language for safety critical systems (like real-time embedded systems) must meet both strong correctness criteria and strict performance criteria [53]. The latter forces Hume to work at a low-level, whilst the former is most easily attained by working at a high level of abstraction. Hume has

been designed to support Leveson's guidelines for Safeware — software intended for safety critical applications.

As mentioned before, Hume introduces notions of low-level behaviour such as time and exceptions to meet the requirements of real-time embedded systems. The Hume language definition supports a wide range of basic types, which are fundamental to guarantee both correctness and safety. Strict semantics have been provided by the expression language, which is purely functional, deterministic and has statically bounded time and space behaviour. A relatively simple formal cost model is introduced into the expression layer to control the cost of both space and time usage. Programmers can write an application running with predictability and reliability under limited space and time with the help of the analysis tools. Asynchronicity and concurrency have been introduced into Hume implementation by boxes and wires in the coordination layer. Boxes and wires are also in charge of interaction with external, imperative states by streams and ports that are connected to the external environment. All of these properties make Hume a reactive, concurrent, asynchronous and correct programming language for real-time embedded systems. We will give more specific examples and evidences to demonstrate those properties of Hume in the following chapters.

# Chapter 4

# A Resource-bounded System in Hume

In this chapter, we will analyze Hume's language performance in a resource-bounded system. We will use the Symbian Series 60 Operating System (OS)[13] as a platform and the Nokia 7650 mobile phone as a target machine. The Nokia 7650 is a camera phone of traditional (non-smartphone) design. It has 4MB of shared memory and uses a 104 MHz ARM9 CPU. We will first demonstrate the approach that the Hume application uses to run under the Symbian series 60 OS

## 4.1 Porting Hume to Symbian OS

The motivation for porting the HAM (Hume Abstract Machine) Interpreter to the Symbian OS is to allow Hume programs to call the Symbian OS routines. We can in principle make programming for resource-bounded systems more effective and reliable by using the space cost data we obtain from the HAM Interpreter under Linux. Under Linux, we can obtain the precise space cost for the Hume applications with the help of some specific Linux commands(such as SIZE hami).The precise space cost data of Hume applications under Linux will provide us with a foundation to predict the approximate space cost for corresponding Hume applications under Symbian OS. The specific data and more details of the relation between space cost data and Hume program will be demonstrated in Chapter 6.

### 4.1.1 Implementation Method

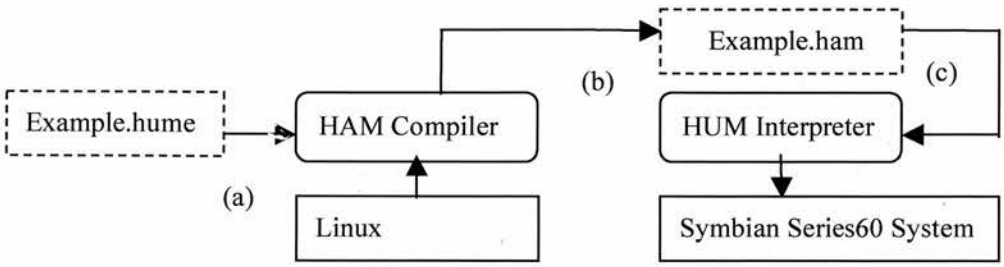


Figure 4.1 Implementation Method for Hume

As shown in Figure 4.1, the execution of the Hume implementation can be divided into two parts. In the first part, a Hume program (Example.hume) is compiled by the HAM compiler based on a normal host system (here a Linux). In the second part, an *Example.ham* file, which was generated by the HAM compiler in the first part is interpreted by the HAM Interpreter on the mobile phone. During interpretation, the Hume program can access Symbian OS APIs.

A Hume program is compiled by the prototype Hume Abstract Machine (pHAM) which is loosely based on the design of the classical G-Machine [69] or SECD-Machine [70]. The prototype Hume Abstract Machine is designed to research the bounded time and space computation of a realistic implementation and to allow formal cost models to be verified.

A Hume program will be compiled and ported to the Symbian OS by the following steps:

1. A program written with Hume code will be compiled to HAM code by pHAM.
2. The complied HAM code from step one is composed of two files. One file contains the pHAM bytecode for the compiled program and the other contains C code generated by the pHAM acting as bridge code to call foreign functions. In the case of the Symbian OS, the foreign functions are Symbian APIs. More details of the bridge code will be explained in the following part based on a specific

application.

3. A linker runs over the bytecode, which is generated in step two. The bytecode linker will produce a complete set of bytecode for the program's user part including references to Symbian functions. The linker also produces relocation code for the bytecode.
4 A completed bytecode program will be packaged as a Symbian database file. In this step, address references from one part to another are resolved as relative offsets.
5 In the final step, the bridge code and the relocation code will be compiled and linked to pHAM runtime to produce a Symbian-executable file.

Two files are loaded onto the Symbian device: one is the Symbian-executable file of the HAM Interpreter, including the bytecode database which is approximately 400 K bytes, and the other is the program file holding the pHAM bytecode, which is typically around 10K bytes(Digital Watch, Pump, Railroad). The total available memory of our target machine (Nokia 7650) is 4 M. It is obvious that the size of our HAM application is acceptable on the Symbian device.

When the program is run on the Symbian OS, a single thread will be created for the Hume abstract machine, which will read the instructions, initialized data and dynamic process structures. On system startup, an initialized heap area is created in static memory and the bytecode database file is copied into dynamic memory. A dynamic system heap (allocated from a single large byte array) is used to store this information. This system heap is responsible for allocating the heaps and stacks by the Hume abstract machine. After reading its initialized state, the Hume thread will run to completion with boxes scheduled in a number of cycles. Hume-level stream I/O, eg. the standard input channel, is run in a process whilst interfacing to the Hume FIFOs and whilst connecting to the corresponding Symbian I/O devices.

## .4.2 The Symbian Series 60 System

The Symbian OS is a 32-bit FIFO (first in, first out) operating system which supports multiple threads. The Symbian OS is designed for use on communication terminal devices (e.g. handsets) which have limited memory usage.

### 4.2.1 System Architecture

Figure 4.2 demonstrates the main features of the Symbian system architecture. The software environment of the Symbian OS is comprised of four elements: the Application, the Server, the Engine and the Kernel.

- *Engine*: An Engine is the manipulative data of the application. The Engine is not involved in interaction with the user. The traditional Symbian OS application architecture [12] can be split into two parts: the GUI (Graphical User Interface) part and the Engine part. The Engine can be an single executable code model, an independent DLL or several DLLs.
- *Application*: An Application is a UI (User Interface) software that controls the interaction with the user. Each application runs as an independent thread.
- *Server*: Server is a software with no UI A Server administers the system resources and provides a related API to a client. The API provided by the server enables a client to access the server service. The Client can be either an Application or other Servers. Each Server runs as an independent thread. The operation of a Server in the Symbian OS is similar to the operation of a driver or a kernel program in a normal operating system, for example a file access is completed in Server/Client Style.

### 4.2.2 Symbian OS Application Architecture

In the Symbian OS, graphic applications are constructed using four basic classes: the Application class, the Documentation class, the AppUi class and the AppView class. These four classes set up the graphic API for applications.

- *Application Class*: there are two main functions of the Application class. The first is *AppDllUid*, which sets the attributes for the application. Each application has

its unique UID(Manipulates globally unique identifiers). When Applications run in Symbian OS, UID is applied to the high-level framework. Framework distinguishes applications by UID. The second function is *CreateDocument*, which creates an instance of the Document class. We only need two functions: *AppDllUid* and *CreateDocument*. The Application class inherits from the CEikApplication class which defines lots of default functions.

Engine：

Shell, OPL, other applications

Application：

Dialogs, Menu, Toolbar, Icons, Resources, JavaVM, Grid, Rich Text, Edit Control, List Control, Application Framework, Jave Class Libraries.

Server：

Window Server, Process Server, Socket Server, Sound Server, Wireless Server, Database Server, File Server, Alarm Server, Comms Server

Kernel:

euser.dll, ekern.exe, supervisor server, HAL (harewar abstract layer)

Driver：

Drivers for hardware(Sound,MMC,keyboard,port,timer)

Figure 4.2 Symbian OS Architecture

- *Kernel*: The Kernel works at the highest level of the Symbian OS. The Kernel is in charge of the device hardware. It provides an API to applications when they attempt to access this hardware.

The separation between the System's kernel and User API improves the portability for different User interfaces.

- *Document Class*: the Document Class is the data model of applications. Basically it is in charge of the file management and creating instances of UI class.
- *AppUI class*: the AppUI class creates instances of the View class and assigns commands and events for applications.
- *AppView class*: the View class is a visual control. It is in charge of screen display and also provides the corresponding function to respond to the related event.

### 4.2.3 Executable File Types

There are two types of executable files in Symbian OS: EXEs and DLLs.

- *EXE File*: If an application is compiled to an EXE file, the application will start from an entry point defined to be E32main(). E32main() generates a new thread for the current application and the application runs independently in the thread.
- *DLL File*: A DLL provides multiple entry points by which the system or the current thread can simultaneously access the applications. There are two different types of the DLL. Static interface DLLs provide a API for one or more applications. A static interface DLL is compiled to an executable file with the .dll extension. When an application starts to run in Symbian system, all the files are loaded into memory. Another type of DLL is the polymorphic DLL, which is used to implement APIs defined in the other files, for example a driver for the printer, a socket protocol or an application. The extension of those files is normally not .dll but .prn, .prt or .app. Polymorphic DLLs inherit from the related classes and are not loaded until they are accessed by applications. Technically, there is no clear difference between the static interface DLL and the polymorphic DLLs. However, these two types of DLL have different functions: polymorphic DLLs are intended for some special functionalities whilst static interface DLLs are normal executables.

### 4.2.4 Code Execution

In a normal operating system, application code must be loaded into RAM in order to run. The situation is different in the Symbian OS. If code is written in ROM, it can run directly from the ROM.

### 4.2.5 Symbian OS Limitations

Some cell phones have a small amount of memory. The Symbian OS is more concerned with memory leaks. Even if there is just a small memory leak in an application, users will receive the "out of memory" alert very quickly, and will have to reboot the phone to reclaim memory. The Symbian OS is also highly concerned with error tolerance. Cell phone users are much less tolerant to errors than PC users, as they use a cell phone as a domestic application rather than as a computer. Errors have to be caught by applications and resolved automatically either by retrying the erroneous operation or by closing the application whenever possible.

The clock frequency of the CPU in today's cell phones is usually between 100MHz and 160 MHz, which is relatively slow compared to modern PCs. When an application is developed, it is not guaranteed to run as fast in a cell phone as it does in a simulator on the development system. We can observe a trend in recent PC(Personal Computer) software development where the application requires faster hardware rather than using a better algorithm. It is certainly easier to buy a fast CPU than design a faster algorithm. However, efficiency is much more important for devices like mobile phones.

## 4.3 The Porting Challenge

### 4.3.1 STDLIB

The Symbian OS has its own functions to implement I/O and other specific functionality.The Symbian OS supports STDLIB for C code, so the basic C functions such as fprintf() are available in Symbian OS. If we just want to use the basic C function, the only thing we need to do is to install STDLIB on the target machine using a SIS file. SIS is the installation file of Symbian application.

### 4.3.2 Global Data

Symbian applications are normally compiled to DLLs, for example, to an APP for a GUI application. Because of the limited memory that is available, the Symbian platform has been designed for ROM-based computing. This means no data can be written to a DLL which resides in ROM. Although a DLL can be loaded into RAM, the Symbian platform still insists that no data can be written to a DLL. The Symbian platform therefore does not provide a data segment for a DLL. All writeable data must be avoided in a DLL. This design can help memory management but it increases the difficulties of porting an existing application such as the HAM Interpreter. This problem does not happen on the Symbian simulator, which runs on a Windows PC, because a DLL runs under a simulator in Windows. In this mechanism, writeable data is allowed.

The HAM Interpreter uses global data extensively. It follows that it cannot be compiled to a DLL. One way to solve the problem is to remove all the global data from the program. This is unacceptable for the HAM Interpreter, since removing thousands of global data accesses from the program involves rewriting the whole program. We therefore use another approach to solve this problem. The HAM Interpreter is compiled to an EXE. The Symbian platform allows an EXE to use writeable data: however an EXE is invisible to the Symbian OS. This means the user cannot execute a .EXE file directly, because there is no GUI involving in .EXE files. Our solution is to use Server/Client Mechanism [29]. As shown in Figure 4.3, a GUI App is used as a client to interact with the user and to handle the active events. In the settings, the HAM Interpreter is the background server. The Client uses the *User::StartL()* function to call the Server. Once the HAM Interpreter is invoked, a new thread will be created. Then HAM Interpreter is able to interact with the user by using a text-based console.

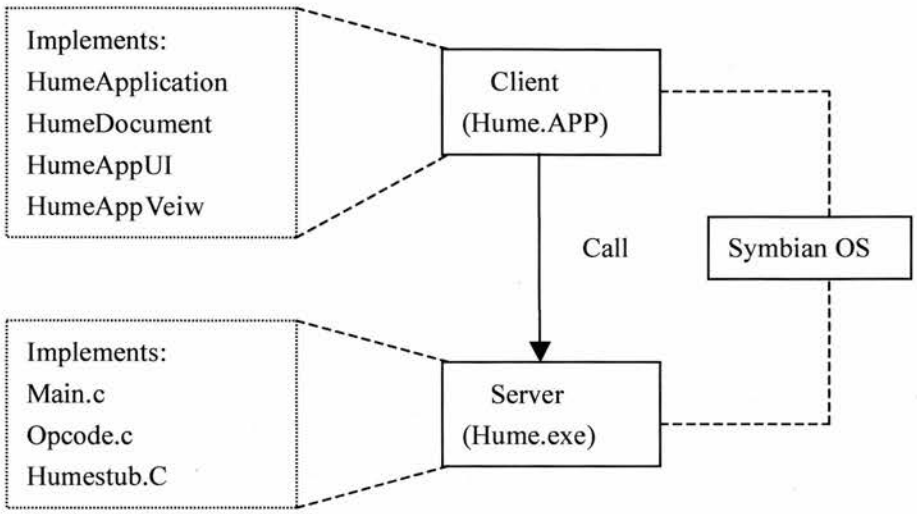


Figure 4.3 Client/Server Mechanism

### 4.3.3 Stack Usage

In the Symbian OS, memory usage is restricted and each thread can therefore only use 8K stack as standard. A static error (unresolved external symbol __chkstk) is reported if the thread takes more than 8K stack. In order to meet this limit, some memory reduction has been made in the HAM Interpreter, for instance, in the size of arrays.

### 4.3.4 The Timing Mechanism

The HAM Interpreter makes use of Linux timer features to record the execution time. These timer functions and calls are not available in the Symbian OS. All such functions are replaced by the corresponding Symbian timer functions. The Symbian OS supports either a 64Hz clock in a physical device or a 10Hz clock in a simulator. The clock interrupt has the highest interrupt priority. It can be accessed by using either *User::After()* or *RTime::After()*.

### 4.3.5 Mixed C and C++ Programming

C and C++ have different views of function names. All applications targeting the Symbian platform are compiled by the C++ compiler. The HAM Interpreter is written in C and interprets a .ham file. We must therefore ensure that we can call C++

functions from within the HAM interpreter. Since the Symbian API is encapsulated by C++ classes, we cannot access Symbian OS classes directly. The solution of this problem in the HAM Interpreter is to declare that the C++ function has a C linkage. With such a declaration, the C++ function can then be called from a function complied by the C compiler [57]. The more details concerning about mixing C/C++ programming will be demonstrated in Appendix A. On the Symbian platform, the encapsulation and inheritance between classes, makes it quite difficult to write a similar C structure. In our HAM Interpreter port, we therefore set up a normal C++ function as a bridging code between the C and Symbian classes. The Hume executable file calls Symbian routines though the bridge code.

## 4.4 Performance Comparison with Linux HAM

Our HAM Interpreter has also been ported to a Linux(1.2GHz Pentium Ⅳ processor). Like Linux HAM, our Symbian HAM have the ability to interpret Hume source code to target code and analysis time behavior focusing on the respond between wire and box. Linux HAM can precisely adjust the Hume program's memory usage by changing the numbers of instructions. Currently, our Symbian HAM can not implement this function precisely.

## 4.5 A Sound Application for a Symbian System Written in Hume

### 4.5.1 Application Function

Our sound application has two functions:

- It allows the user to play a preset sound file
- It allows the user to record a file, stop it in a limited time.

### 4.5.2 Application Architecture

The Hume Sound Record/Player program consists of three parts: sound code written in Hume, middleware written in C to act as a bridging code to interface to the

foreign functions, and C++ code to call the actual Symbian Sound API.

**Hume Code**

The implementation of Sound in Hume is fairly straightforward. The Hume program consists of four boxes: a console box which is used to respond to user. A control box interfacing with the outside world and responding to the user's command, a sound play box in charge of a sound file with default format and a record box interfacing to the record API of the Symbian OS. Hume is able to call C functions directly in the form of expressions. In the case of sound play application, the play box and the record box are defined as "*Operation*" expressions, extend calls, as shown in the following examples:

```
operation cmyfun as "cmyfunction" :: int 32 -> int 32;
operation rmyfun as "rmyfunction" :: int 32 -> int 32;
```

*cmyfun* is the name of the play box and *rmyfun* is the name of the record box. *cmyfunction* is the C function name for playing a sound and *rmyfunction* is the C function name used to record a sound. The first *int 32* is the input parameter that is to be passed to the C function and the second *int 32* is the return value of the C function. The implementations of the play box and the record box are completed by the related C functions, but do not use the match patterns rules as with normal boxes. The implementation of related C functions is part of the bridging code, described below. The initialized C function parameters correspond to the box inputs and the return value will correspond to the box outputs. There are the complete sound application design code:

```
type Int=int 32;
type String=string;
box console
   in(x::String)
   out(y::Int)
match
```

```
   (a)->(1)|                                    --a means user want to
play a sound
   (b)->(2)|                                    --b means user want to
record a sound
    _ -> * ;                                    --ignore anyother options

stream input from "std_in"                  --contact with user by console
wire console
   (input)(control.x);

box control                                     --mode is the status of
control box
 --x is the user's option from console box,c and r is used to call play
and record function
  in(mode::Int,x::Int)
  out(c::Int,r::Int)
match
  (_,1)->(1,0);  --if option is 1,then call the playsound function to
play the sound
  (_,2)->(0,1);  --if option is 2,then call the record sound function
  (_,_)->(0,0);  --ignore the other options

wire control
(cmyfun.outp,console.y) (cmyfun.inp,rmyfun.inp)

operation cmyfun as "cmyfunction" :: int 32 -> int 32; --call the c sound
play function
operation rmyfun as "rmyfunction" :: int 32 -> int 32; --call the c sound
record function

wire cmyfun (control.c)(control.mode)
wire rmyfun (control.r)(outp)

stream outp to "std_out"
```

**Bridge Code**

We showed above how to relate a box to a C function, but we did not give any implementation for the C functions. If the C function we call from Hume is a normal function which has been defined in the C library, for example *printf()* ,we can call it directly without any additional definitions. If the C function called by Hume code is

the function defined by the programmer, we must give the implementation as part of the bridging code.

In the case of the sound application, two C functions have been declared as boxes in Hume. The implementation is as follows:

```
include"CCallC.h"
int cmyfuntion(int a)
   {
        if(a==1)
            playsound();    //Call Symbian C++ Play sound functions;
        else
          { printf("a for play!b for record!");
            return 0;
             }          //Ask for user to make a choose.
       }
int rmyfuntion(int b){
        if(b==1)
            recordsound();  //Call Symbian C++ Record Sound functions;
        else return 0;
     }
```

As discussed earlier, when a Hume program has been compiled by the HAM compiler, the bridge code will be generated automatically according to the expressions in the Hume code. More details about the bridge code will demonstrated in Appendix B.

**Symbian Code**

The final part of the program is the code based on Symbian functions. The complete code has been demonstrated in Appendix C. There are three Symbian classes involved in the sound application: the *CmdaAudioPlayerUtility* class, the

*CmdaAudioRecorderUtility* class and the *CActiveScheduler* class. The *CMdaAudioPlayerUtility* class is used to play audio data (from a file or a descriptor) in any format recognised by the Media Server. *CmdAudioRecorderUtility* is used to record and play sounds. Both *CmdaAudioRecorderUtility* and *CMdaAudioPlayerUtility* are part of the media server implementation in the Symbian OS. In the Symbian OS, when multiple asynchronous services are used by a program, a waiting loop is required. A waiting loop is encapsulated by the *CActiveScheduler* class. Nearly all Symbian threads need an active scheduler, which provides a system of non-pre-emptive multi-processing within a single thread. In traditional systems, in contrast, multi-processing is often done using multiple threads [7]. Playing and recording sounds are asynchronous operations; applications will not be blocked whilst these operations complete. In traditional Symbian programs, an active scheduler will be added to a thread automatically when the UI (User Interface) or server threads have been initialized. Because of the global data problem we mentioned in chapter 4.3.2, our sound application and HAM interpreter have been complied into an executable EXE file. The active scheduler will not be added to an EXE file by default. What we can do in our Hume sound program is to create a new active schedule by *CAtiveScheduler* class and install it into the current thread; then we are able to connect to the Media Sever and respond to the active events.

### 4.5.3 The Implementation Process

The user can play a normal sound file, record a sound file and playback by inputting the command to the console. When the user presses "1", a preset file will be played by the phone. When "2" is pressed, the phone will start to record the sound and stop by invoking a stop function. When "3" is pressed, the phone will playback the sound which has been recorded into a file.

In a Hume program, once these applications have been executed, threads will be created for each of the three boxes. An event handler will respond to system events such as pressing one of the buttons. The event related sound utilities will be handled

by the Symbian OS itself though the Media Server, eg. the play mode for a sound file

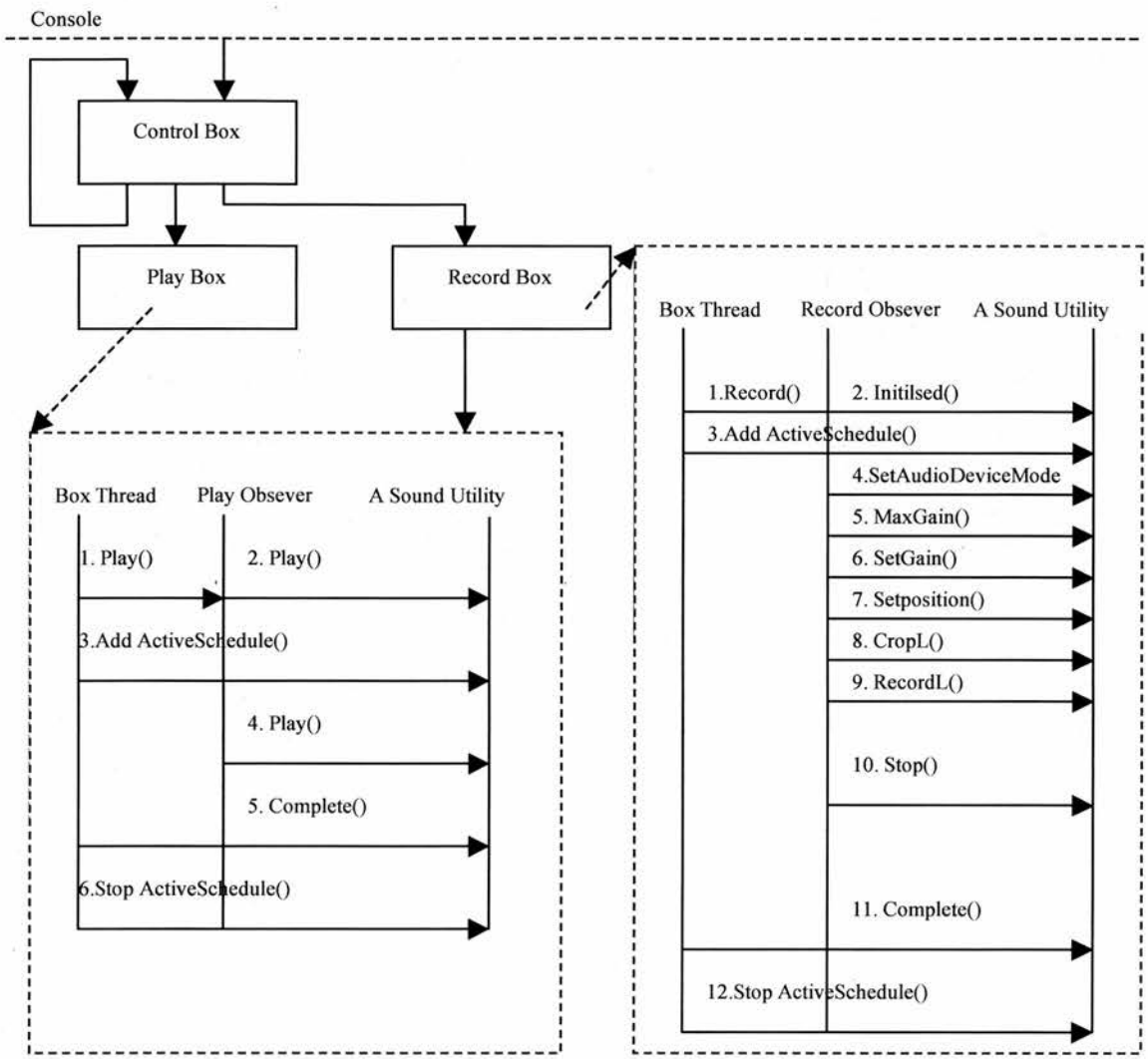


Figure 4.4 Sound Application Architecture

or the audio frequency of sound. As Figure 4.4 shows, when the user chooses to play a sound, a sound play observer object will be created and initialized. A dynamically allocated resource will be assigned to the sound play observer. The sound play observer is a class in charge of playing or stopping a sound. An active schedule will be added to play a sound thread and an event will be created for the sound events. The Media Server will be connected after the initialization of the play observer and be ready to play a sound. The *CmdaAudioPlayerUtility* class will create a new sound

object and open a preset sound file by the *OpenL()* function. An event handler will respond to all the sound actions. If a preset sound file has been opened successfully, the Media Server will start to play the sound file. When the Media Server finishes playing the preset sound file, it will provide a callback to the active schedule. This callback will be handled by the *event stop* function, which is also a private function of the *CmdaAudioPlayerUtility* class. A stop event will be invoked and the active schedule will be stopped. The Symbian OS clean up code will free the dynamically allocated resources. The procedure will go back to the play box thread and give a callback to the user that the sound file has been played.

The sound recording process is almost the same as the sound playing process. The only differences are that a sound record utility must to do more settings before it starts to record sound. It will open a preset file before it starts to record. When a stop event is invoked, the record process will be stopped and the sound will be saved to the preset file. This can then be played back by the sound play utility.

This sound application is typical of a simple Symbian OS application. The code size of our Hume program (including the Hume code and the Symbian code) is around 350 lines long. The traditional C++ Symbian code is around 800 lines long. This application shows that it is possible to provide the ability to program in a resource bounded system with concise code. The Hume sound application is related to the Symbian OS environment and to the Nokia phone's equipment. Hume can be used to write a reactive program because it can respond to the events and react to the Symbian OS environment.

Compared to the original Symbian Sound application, our Sound application would continuously record and play a sound file with less time. For the same sound file, the execution time of our sound recording is 0.5 second while the time for the original application is around 1.5 second. Our Sound application execution cycle(record and play once upon a time) is around 3 Seconds, while the original Symbian Sound application execution cycle is around 5 Seconds. Although our sound

application communicate with use by a console, our operation for recording and playing sound is much simpler and the operation time is shorter than the original Sound application.

We have also implemented several rudimentary applications in console mode on Nokia 7650. These applications are independent of the Symbian OS environment, eg. the digital watch program, the railroad program and the pump program. The Hume code for these programs is between 75 and 100 lines long, and is the same for the Symbian and non-Symbian platforms.

## 4.6 Related Work

Many languages are available for the Symbian platform [58]. The Symbian OS is written in C++, which is also the primary language for programming on the Symbian OS. C++ offers the greatest access to the Symbian OS APIs, because C++ usage is heavily concentrated in GUIs, multimedia toolkits and games (the major success areas for OO (Object-Oriented)design. However, there is some evidence that C++ programs have higher life-cycle costs than equivalents in C, or Ada [75] and none of them have fully automatic dynamic-memory management.

Java has been ported to the Symbian OS successfully through an adaptation of the JVM (Java Virtual Machine) to mobile phone technology. Java is famous for the "write once, run anywhere" philosophy, although it is not always achieved seamlessly in practice. Compared with C++, the design of Java has the advantage of possessing automatic dynamic-memory management. Compared to other languages, eg. Python [73], however, Java's class-visibility and implicit-scoping rules are quite complex. Although the interface mechanism avoids the worst problems of multiple inheritance, the class structuring mechanisms are also still fairly complex. Finally, considering raw execution speed, Java cannot normally compete with C++ for programming Symbian OS.

Python is another language that has been recently ported to the Symbian OS [74]. A Python Interpreter has been ported to the Symbian OS platform. When the Python interpreter runs, the user can interactively enter Python programs or code fragments in console mode. Python is very clear and elegant in design with good modularity features suitable for programming large complex projects with many cooperating developers. The polymorphic design of runtime types decreases Python's raw execution speed. While this limitation may not be very significant in modern fast processors, such as a normal PC, for a resource-bounded system such as a mobile phone, this limitation should not be ignored.

No other functional languages have yet been ported to the Symbian OS. Hume is the first functional language to be implemented on the Symbian OS. However, three other functional languages are available for the similarly resource-bounded Palm OS: Haskell [71], LispMe [72] and Caml Light [73].

Both the Symbian OS and the Palm OS are resource-bounded systems with similar hardware environments. Palm OS supports 8MB memory whilst Symbian's RAM is usually 8 or 16MB. The processor speed of Palm OS is in the order of 16 MHz or 33MHz and the recent Palm-based PDAs have switched to ARM processors with speeds of up to 400MHz. The Symbian OS's processor speed is currently normally between 100MHz and 160MHz.

The standards-compliant compiler for Haskell (*nhc98*) has been ported to the Palm OS, allowing Haskell to run with relatively little recoding required. Basically, it understands the Palm memory model and makes use of it. Several applications have been implemented both in GUI mode and in console mode. All of the applications are independent of the Palm OS environment and are not real-time reactive applications. Crucially for real-time systems, software execution times are not considered in this port, nor is there an effective strategy to predict the memory usage.

LispMe is another functional language that has been ported to Palm OS, and which provides a traditional loop style interface for developing Scheme programs on the

Palm itself. The SECD machine is used to make the LispMe interface to the Palm OS API and retain a maximal heap usage within Palm OS limited resources [72]. Finally, there also exists a Caml Light implementation for the Palm; however, progress on this now appears to have ceased [79].

Our Hume port is quite similar to the Haskell and LispMe ports in general structure. All the ports are based on compiling on a regular computer. The bytecode and executable files are then loaded to the target machine and installed. The bytecode will then be interpreted to interface to the APIs of the target machine. In the LispMe and Haskell cases, the virtual machines are the SECD and *nhc98* respectively. In our case, the HAM Interpreter has been ported to the Symbian OS. Compared to the Haskell case, not only do we provide applications independent of the Symbian OS environment, but we also provide a reactive application related to the Symbian environment, as exemplified by the sound application described here. Like the LispMe compiler, the HAM Interpreter provides a similar mechanism for restricting the heap and stack usage.

## 4.7 Conclusions and Improvements

Since no functional language has ever been implemented on the Symbian OS, our Hume port provides an effective way to extend functional language programming to the Symbian OS. Based on our sound application example, Hume can be used to write reactive, responsive and asynchronous programs based on the Symbian OS environment. Moreover, Hume can be used to write Symbian applications with relatively small code sizes. Time performance and memory resource limitations have also been considered in the Hume program. More details of these will be given in Chapter 6.

However, because of limited research time, our port is still at a very early stage. Our interface to the Symbian OS is quite rudimentary, because we have only implemented a few simple applications. The encapsulation and inheritance structures

required by the Symbian OS classes increases the difficulty for Hume to interface to the Symbian OS APIs. We hope to find more effective and flexible approaches to interface more GUI functions and build more functional and realistic programs in the longer term.

# Chapter 5

# Reactive Systems in Hume

In modern society, computer technology is widely used in every field of our world. Computers are involved more and more in our lives, so many kinds of equipment and systems concentrate on providing sufficient intelligence to interact with users and to cope with real world behaviours. Real-time systems provide this kind of intelligence more than other systems. In this section, we will give Hume solutions for three types of real-time reactive systems.

## 5.1 Types of Real-time Reactive Systems

It has been estimated that 99% of the worldwide production of microprocessors is used in embedded systems[1]. The real-time systems can be categorized into four types: general embedded, command control, process control and production control[1].

### 5.1.1 A General Embedded System

As the above examples show, the computer is directly interfaced to the real world. Computers sample the physical facilities at regular intervals, record state changes and provide information for an operator to make decisions for the system. That is what a typical embedded system does (Figure 5.1). A general embedded system covers a wide range of real world applications, from the complex system for a

mobile phone or satellite, to a very simple system such as a digital watch. They have the common basic modules that control a system's operations. These modules contain the algorithms necessary for controlling the devices, recording responses, displaying the state changes and interacting with the operator.

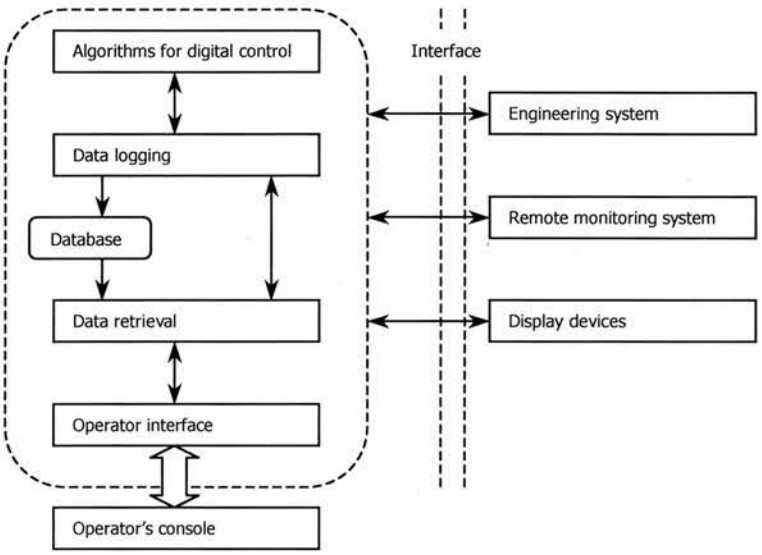


Figure 5.1 A General Embedded System[83]

### 5.1.2 A Command & Control System

A command & control system is a common type of real-time system, which is used in a wide range of applications and simulations, for example, railroad controllers, diagnostic instrumentations and airplane control systems. All of those systems contain a complex set of policies, which are used to gather information, administrate procedures to support decisions and provide an appropriate approach. This is illustrated in Figure 5.2.

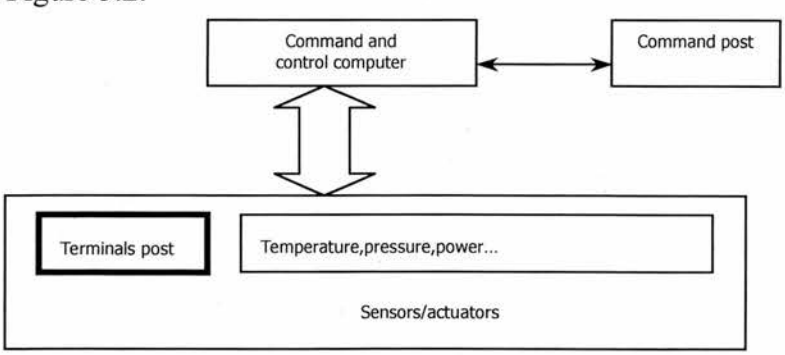


Figure 5.2 A Command and Control System[81]

### 5.1.3 A Process Control System

Figure 5.3 is an example of a process control system, which illustrates the role of a real-time computer embedded in a complete process control environment. A computer is used to control an even flow of liquid in the pipe by controlling a valve and detecting the increase and decrease in flow. The computer must be sensitive enough to respond to the changes of the valve angle, and this response must occur in a limited time to prevent the valve overloading. It should be noted that quite a complex calculation of the new valve angle is essential for the actual response. The computer communicates with equipment by sensors and an actuator (a valve is an actuator, a temperature or pressure transducer is a sensor). According to the control of the sensors and actuators, the computer can make sure that the system operates correctly at the proper time.

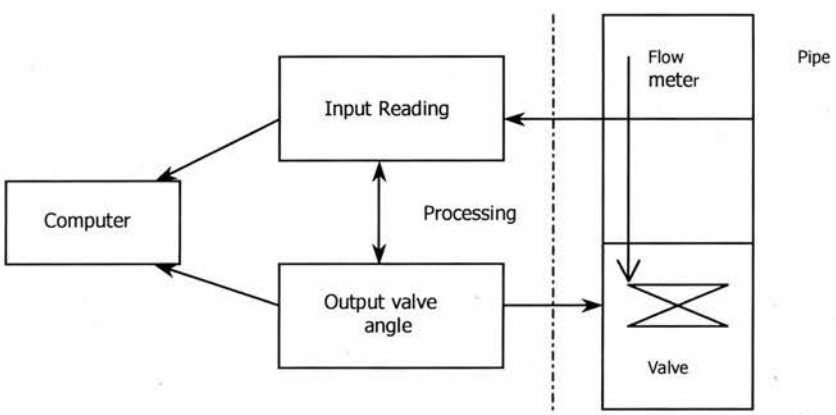


Figure 5.3 A Process Control

### 5.1.4 A Production Control System

Figure 5.4 represents the role of the production control computer in the manufacturing process. The participation of the computer in the entire manufacturing process from product design to fabrication keeps costs low and increases productivity.

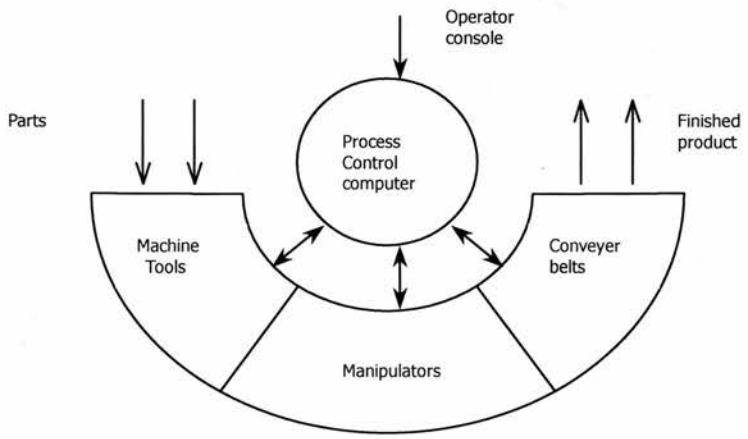


Figure 5.4 A Production Control System[55]

### 5.1.5 Summary

In order to estimate the Hume performance for programming real-time systems, all aspects of real-time system must be considered. As the real-time systems can be categorized into the above four types, we will give our implementation and estimation of Hume program based on the four types.

## 5.2 The Hume Solution for a General Embedded System

We have chosen a Digital Watch application [67] as a representative example of a general embedded system, as it represents the basic module for a general embedded system, has a wide range of products in our real world and is also a typical type of real-time system. Although the digital watch application is not complicated, it represents the requirements for a general embedded system. A digital watch records the responses of user, contains the algorithms necessary for controlling the devices, displays the time changes to the user and interacts with the user.

We will now demonstrate Hume's performance in a real-time system.

### 5.2.1 Application Specification for a Digital Watch

The digital watch, which has been demonstrated by Figure 5.5, has the following features: four adjureliable buttons used to interact with users; one screen used to

display time, date and alarm; and a bell used for the alarm clock.

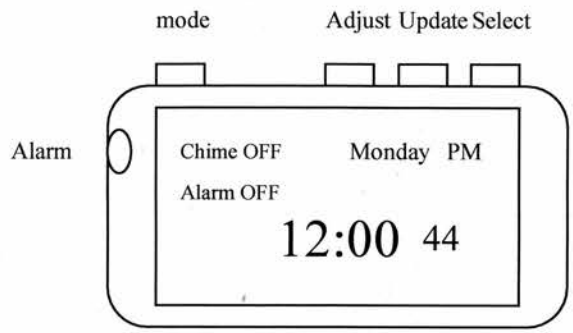


Figure 5.5 Hume Version of Digital Watch

- Time display: There are two formats to display time: 24 hours and AM/PM.
- The Alarm: The Alarm can be set to three ringing types: at a specific time, on each hour or both.
- Buttons: The user can control the digital watch with these four buttons. Each button has different functions in different adjusting states. The states can be separated into four modes:

**Time Display State**

- Button Mode: when pressed, the watch will switch to time display state if the current state is not time display
- Button Adjust: in this state, this button is used to change the format of time display (24 hours or AM/PM)
- Button Update: invalid in this state
- Button Select: invalid in this state

**Time Update State**

- Button Mode: switch to time display state.
- Button Update: when pressed, the watch is switched to time update state, the second digit is the default digit which will flash and is to be adjusted
- Button Select: press it to switch the flashing digit between hour, second, minute and day.

- Button Adjust: used to increase the digit that is currently flashing. The number increases once by one press.

**Alarm Display State**

- Button Mode: when pressed, it will switch to alarm display state if the current state is time display
- Button Adjust: in this state, this button is used to change the alarm setting (daily, hourly or both)
- Button Update: invalid in this state
- Button Select: invalid in this state

**Alarm Update State**

- Button Mode: switch to alarm display state.
- Button Update: when pressed, the watch is switched to alarm update state; the minute digit is the default digit which will flash and is to be adjusted
- Button Select: press it to switch the flashing digit between hour and minute
- Button Adjust: used to increate the current digit that is flashing. The number increases once by one press.

### 5.2.2 Finite State Machine

The Mode button is used to switch between time display state and alarm display state, while the Update button is used to switch from time or alarm display state to time or alarm update state.

As shown in Figures 5.6, 5.7 and 5.8, the operation process of the digital watch can be simulated as two levels of a Finite State Machine (FSM). The first level of FSM is demonstrated in Figure 5.6(A).

The top level has four states: time display, time update, alarm display, and alarm update. The time display state is the default state. After one of the four states has been chosen, the digital watch will switch to the second level FSM (Figs 5.6 (B), (C); 5.7 (D), (E)). The FSM defined in Figures 5.6 (C) 5.7 (E) shows the updating process of

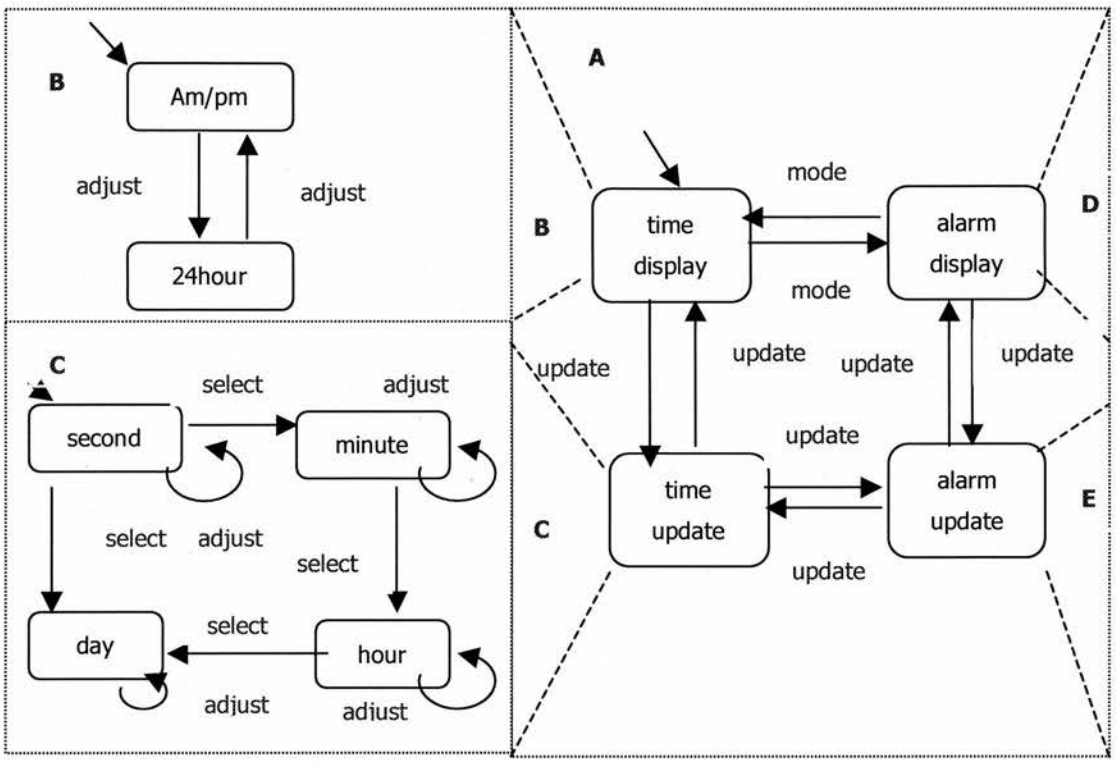


Figure 5.6 FSM for Time

alarm and time respectively. They are in charge of updating the time or alarm display and changing the numbers. As Figure 5.6 shows, the digital watch was in time update mode. When select button has been the flashing digits will be changed among hour, minute and second. The adjust button will be used to advance the number of the flashing digit. For example, if second digit is flashing and current time is 3 o'clock 5.5 minutes and 20 seconds, when we press adjust button once, the current time will change to 3 o'clock 45 minutes and 20 seconds. Figure 5.7(E) shows the same process of update a alarm, the difference is only hour and minute digit are available for updating. Figure 5.6(B) shows that the time display format is switched between 24 hours and AM/PM by pressing the adjust button in the time display state. Figure 5.7(D) illustrates that a daily alarm or hourly chime can be enabled or disabled in the alarm display state.

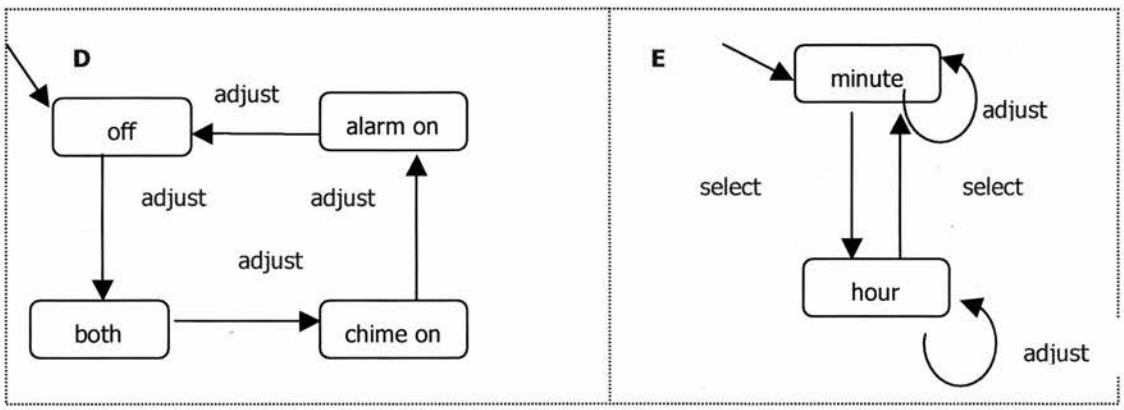


Figure 5.7 FSM for Alarm

Figure 5.8 demonstrates the timeout event of the digital watch. The digital watch will wait for the input from outside (one press of one of the four buttons) when the state is not time display state. If there is no reaction from any button for a certain time, a timeout event will be triggered. The digital watch will then switch to the time display state.

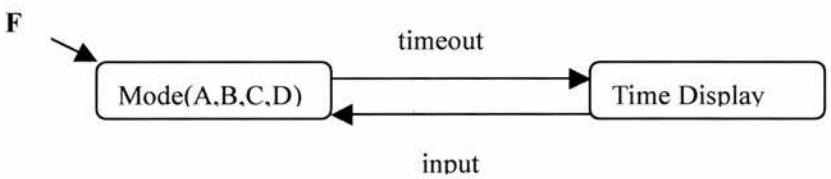


Figure 5.8 FSM for Timeout

### 5.2.3 Hume Implementation for the Digital Watch Problem

In the Hume implementation, boxes and wires are used to simulate the behaviour of the digital watch. The whole behaviour is composed of four boxes: button box, watch box, watch display box and Fanout box.

The four boxes have been wired together. Each box has input digits and output digits. The output values from one box are transferred as the input value of another box by wires. The values will be changed depending on the operation in each box and

passed to another box. With this configuration, the whole program is built as a complete loop. It will run continuously if there is no special operation to interrupt it. As mentioned in Chapter 3, recursion in the function can be replaced by iterative boxes in Hume, so it is not necessary to write any loop into the program. However, the box and wire construction replaces recursion and keeps the program running continuously. The program terminates by special operations. In the case of the digital watch, it will end if the battery runs out or if timeout events have been triggered.

**Button Box**

The Button box is a simulation of four button presses (Fig. 5.9 (A)). Its inputs are from the user's inputs and its outputs go to the inputs of the watch box. The Watch box will be used to adjust the internal state of the digital watch. The timeout event shown in Figure 5.8 is triggered by the wires and handled by the Button box.

**Watch Box**

The Watch box simulates the internal operation of the digital watch (Fig. 5.9 (B)). It generates the next state of the digital watch based on the previous state and the inputs from the button box. The digital watch box has seven input values and generates six output values in every adjust step. *State'* and *State* represent the current watch state and the state for next step. *Button* is used to obtain the value generated by the button box, which represents the button pressed by user. *Digit'* and *Digit* represent the current flashing digit (hour, minute or second) which will be adjusted, and the flashing digit after one operation. *Format'* and *Format* represent the time display format (24 hours or AM/PM). *Chime'* and *Chime* represent the alarm type (hourly or daily). *Time'* and *Time* represent the time display. The *Alarm'* and *Alarm* represents the alarm display. The watch box generates output values depending on rules, represented in the above FSMs. Every digit has its own input and output data type.

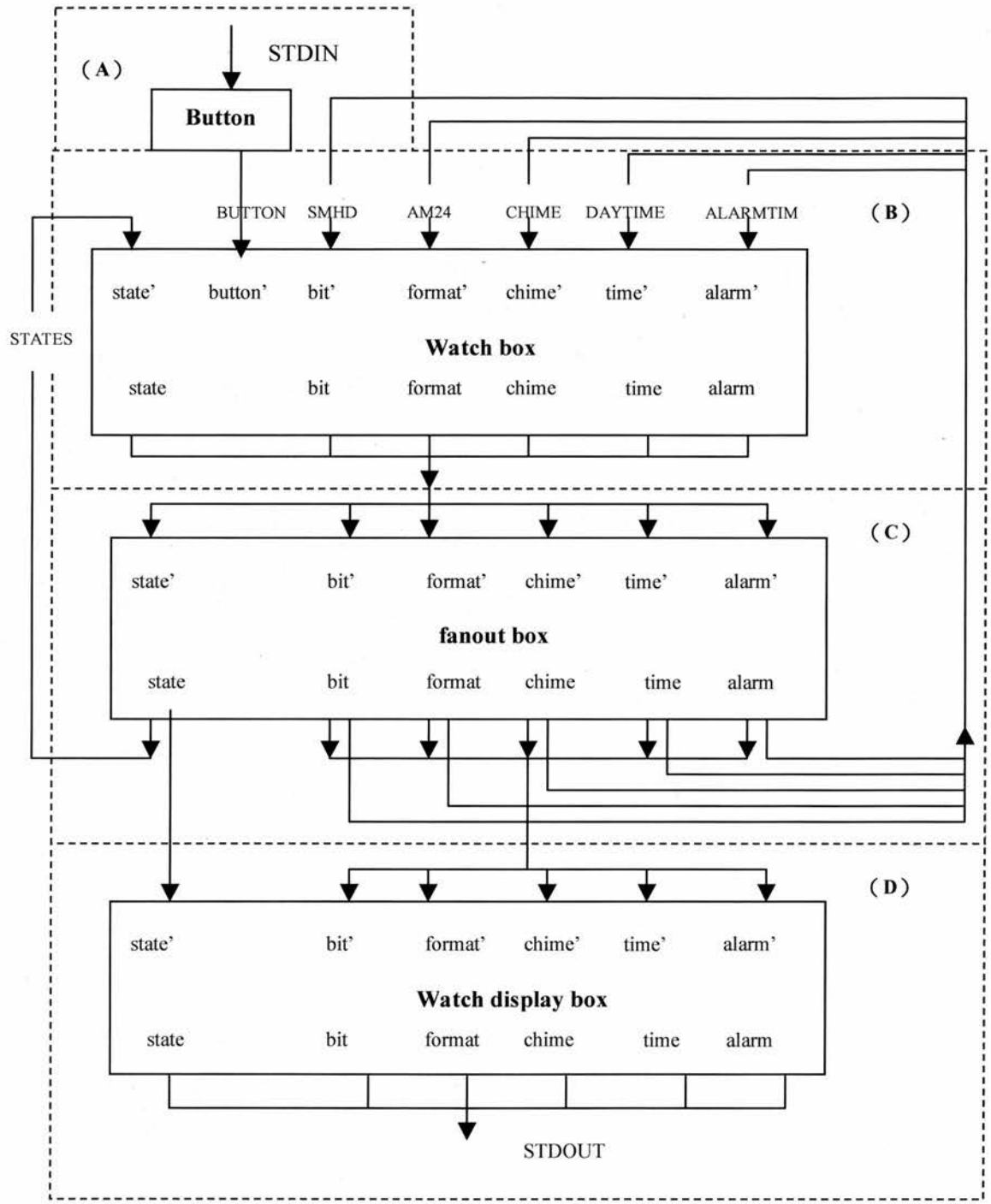


Figure 5.9 Boxes and Wires for a Digital Watch

```
STATE=AlarmDisplay|TimeDisplay|TimeUpdate|AlarmUpdate|TimeDisplayOut;
SMHD=Second|Minute|Hour|Day;
AM24=AMPM|Hour24;
BUTTON=Mode|Update|Adjust|Select|TimedOut;
CHIME=Daily|Hourly|Both;
```

```
DAY=Mon|Tue|Wed|Thu|Fri|Sat|Sun;
DAYTIME=(Int,Int,Int,DAY);
ALARMTIME=(Int,Int);
```

As an example, the following code shows a one-step operation of the watch box. The complete digital watch program will be demonstrated on Appendix D. If it receives the inputs from the previous step, the *Button'* digit gets the input: "Mode", which means that user pressed the Mode button. The digital watch will check the value of the *State'* digit. The current value of the *State'* digit is "Time Display", which means the watch is in the time display state. As shown in Figure 5.6, the digital watch will change to the alarm display state. The watch box will then generate the output of Alarm Display in the *State* digit while retaining the other values.

```
match
(TimeDisplay, Mode, smhd, am, chime, time', alarmtime)->
(AlarmDisplay, smhd, am, chime, time', alarmtime)|
(AlarmDisplay, Mode, smhd, am, chime, time', alarmtime)->
(TimeDisplay, smhd, am, chime, time', alarmtime)|
```

**Fanout Box**

The Fanout box (Fig. 5.9(C)) is used to separate the values in two ways. A band of output values is the inputs of the watch box, which will be the prerequisite of the next step, and another band of output values becomes the inputs of the watch display box.

**Watch Display Box**

The Watch display box (Fig. 5.9(D)) simulates the watch display screen. It shows the user the current watch state.

## 5.3 The Hume Solution for a Command & Control System

A Railroad controller [67] is a specific example of a command & control system. It contains a complex set of policies for controlling signals for two trains, devices for gathering all the information for the movement of trains, signal controls and

administrative procedures, which enable decisions to be supported. We demonstrate Hume's ability to program a command control system.

### 5.3.1 Application Specification for a Railroad

There are two trains (train A and train B) setting out on the two circular railroad tracks, which is demonstrated by Figure 5.10. They move counter-clockwise. Each track has 20 cells. We assume that the speed of a train is one cell per second and the length of a train is one cell. There is a bridge which connects the two tracks, allowing only one train to pass at one time. In order to avoid two trains crashing on the bridge, there are two signal controllers (signal A for train A and signal B for train B). When the signal light turns red, it means that there is a train on the bridge and the arriving train needs to wait until the signal turns green. If the operation for control is not designed well, a crash will ensue if both signals turn green and the two trains arrive at the same time.

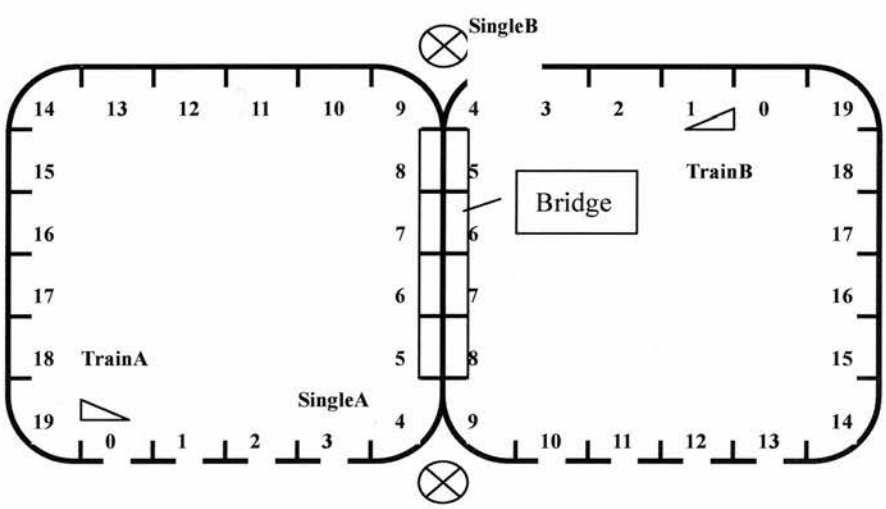


Figure 5.10 Hume Version for RailRoad

### 5.3.2 The Hume Implementation for the Railroad Problem

In the Hume version, the behaviour of the railroad controller is implemented by seven boxes. As in the example of the digital watch, boxes are the abstraction of real time behaviour. In the case of the railroad controller, there is one Speed box used to

generate the speed of the trains, one Controller box for the signal of each train, two Train boxes (the TrainA box for train A and the TrainB box for train B) which simulate the real time behaviour of trains (such as movements and locations), two Fanout boxes used to separate the values and a Log box used to record the information of every movement and interact with the operator.

**Speed Box**

The Speed box (Fig.5.11(A)) has two input digits used to interact with the outside world. The operator of the railroad station can define the speed for the two trains. According to the input from outside, the Speed box generates outputs and passes them to the train boxes via the wires between them.

**Controller Box**

The Controller box (Fig. 5.11 (B)) is the abstraction of the railroad controller of signals. It has five inputs and three outputs. There are two data types in the definition of this box, which is shown in Table 5.1 . *CROSS* defines the *State'* and *State* digits. ARRLEA defines the MA', MA, MB', and MB digits.

*State'* defines the current state of the signals and *State* defines the state of the signals for the next step. *A'* and *A* represent the digitization of the state of signal A (1 represents red, 0 represents green). *B'* and *B* digits represent the digitization of the state of signal B (1 represents red, 0 represents green). *mA'* and *mA* represent the current state of train A and *mB'* and *mB* represent the current state of train B.

| Digital | Value | Comments |
|---|---|---|
| Signal | ArBr | Signal A is red, Signal B is red |
| | AgBg | Signal A is green, Signal B is green |
| | ArBg | Signal A is red, Signal B is green |
| | AgBr | Signal A is green, Signal B is red |
| Cross | Arr | Train arrived on the bridge |
| | Lea | Train left the bridge |

| | | |
|---|---|---|
| | NotArr | Train has not arrived on the bridge |
| | NotLea | Train has not left the bridge |

Table 5.1 Data Structure of Railroad

Having considered the input of the current state, the controller box generates outputs for the next movement circle based on the rules defined by the FSM in Figure 5.15. As shown in the example the Controller box gets the value of *AgBr* from the *State'* digit. This means that signal A is green and signal B is red. The value of the digit is *Lea* and *mB'* is *NotArr*. This means that train A is leaving the bridge and train B has not yet arrived. As shown in the following code, signal A will be reset to green and signal B will stay green. The Controller box generates *(AgBg,0,0)* to the output.The complete code of railroad will be demonstrated in Appendix E

```
box control
  in(state::CROSS,A'::Int,B'::Int,mA::ARRLEA,mB::ARRLEA)
  out(state'::CROSS,A::Int,B::Int)
match
 (AgBr,0,1,Lea,NotArr)->(Agbg,0,0)| -- Agbr-Agbg

wire control
(fanout.state1 initially Agbg,fanout2.sinA1 initially 0,fanout2.sinB1
initially 0,fanout.motivationA1 initially Notarr,fanout.motivationB1
initially Notarr)
(fanout.state,fanout2.sinA,fanout2.sinB);
```

The operation process of signal control is demonstrated by Figure 5.12.There are three possibilities for the two signal controllers.

- **Case one**: signal A is green and signal B is green. In this situation, there is no train on the bridge. If train A moves onto the bridge. Signal A is green and this sets the signal B to red to stop train B. After train A leaves the bridge, the signal B will be reset to green. If two trains arrive at the same time, the controller will let train A move onto the bridge and set the signal B to red simultaneously. Train B will wait until train A leaves the bridge.

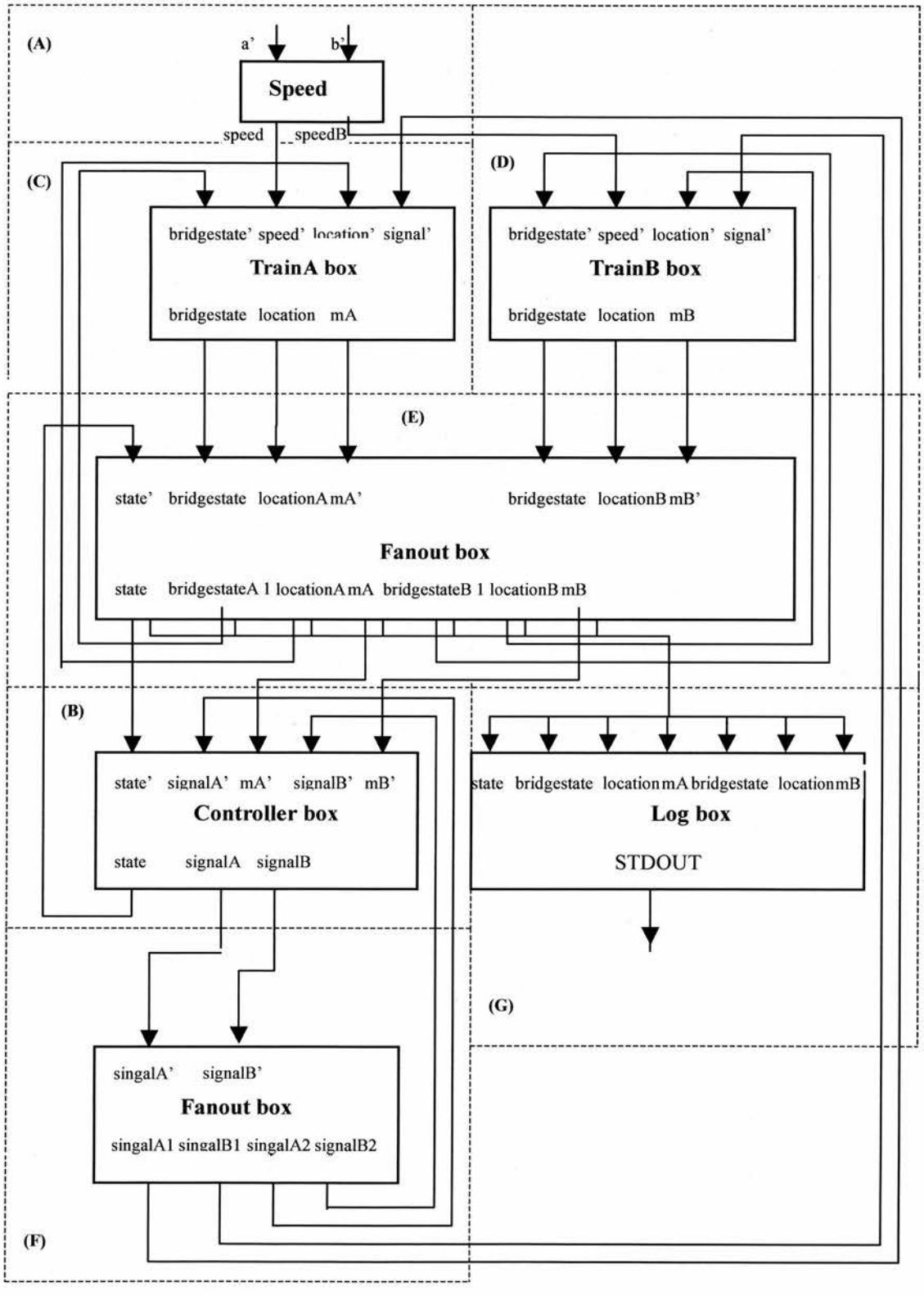


Figure 5.11 Boxes and Wires for a Railroad Controller

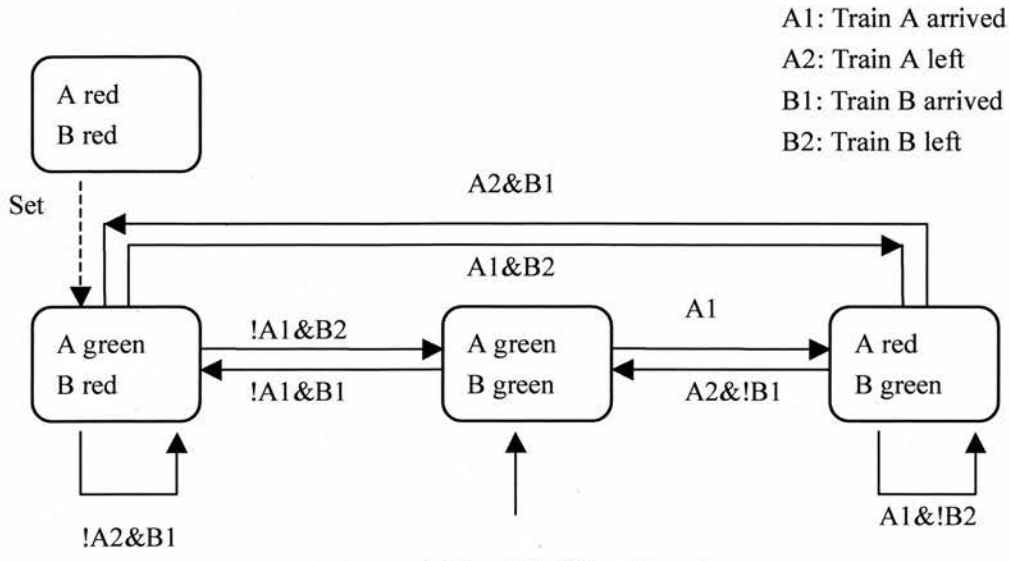


Figure 5.12 FSM for Signals

- **Case two**: signal B is red and signal A is green. In this situation, train A is on the bridge or leaving the bridge. If train B arrives and train A is on the bridge, train B will wait until signal B turns to green. If train A leaves the bridge and train B arrives, signal B will turn to green. Train B will move onto the bridge and set the signal A to red. The state of signals will go to case three. If train A leaves and train B doesn't arrive, both the signal A and signal B will turn green. The state of the signals will revert to case one.
- **Case three**: Signal A is red and signal B is green. In this situation, train B is on the bridge and about to leave the bridge. If train A arrives and train B is still on the bridge, train A will wait until signal A turns to green. When train B leaves the bridge and train A arrives, signal A will turn to green and train A will move onto the bridge and set the signal B to red. The state of the signals will revert to case two. If train B leaves and train A does not arrive, both the signal A and signal B will turns green. The state of the signals will revert to case one.
- **Case four**: Signal A is red and signal B is red. This case covers the emergency situation where the operator sets all signals to red. Once the problems causing the emergency have been resolved, the state will return to its initial setting. signal

A will be set to green and the system will proceed as in case two, with the controller tsking charge of the signal settings once more. This case does not form part of the normal operation of the system: unless there is an emergency, the system will simply move between the other three cases as described above.

Train A Box

The Train A box (Fig. 5.11(C)) is the abstract of the movement of train A. It has four inputs and three outputs. There is one data type (BRIDGE) defining the value of the *bridgestate'* and *bridgestate* digits. *bridgestate'* represents the current state of train A (on the bridge or off the bridge), while *bridgestate* shows a one second later state for train A. The value of the *location'* digit is a number with integer type which shows the current location of train A. It will generate an output which represents train A's location after one second.

Trains have two states: off the bridge and on the bridge. The states switch from one to the other depending on the evaluation of the train's location. Figure 5.13 shows the FSM for train A's location states.

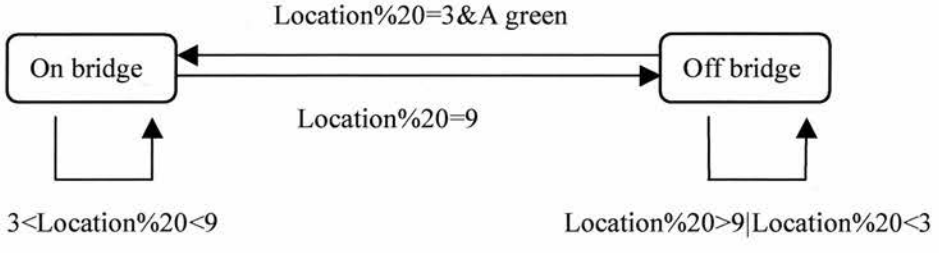


Figure 5.13 FSM for Location

- **Case one**: Train A is off the bridge. In this case, the current state of train A is off the bridge. Location 3 is the location of the bridge's entrance and Location 9 is the location of the bridge's exit. If train A will arrive on the bridge in the next second (location+speed>=3) and the signal A is red, train A will stop and wait at the entrance (location 3) until signal A turns to green. If train A arrives on the bridge in the next second and the signal A is green, train A will move on to the

bridge and passes the value to tell the controller box that train A is on the bridge. The controller box will set signal B to red. If train A doesn't arrive on the bridge in the next second, it will keep running on the track at the speed set by the operator.

- **Case two**: Train A is on the bridge. If train A's next location is location 9, train A will leave the bridge. It will generate a relative output value to tell the controller box that train A is leaving the bridge. The controller box will set the signal B to green. If train A does not arrive at the bridge exit in the next second, it will keep running onto the bridge. The signal B will be set to red.

```
(Offbridge,location,1,speed)->
                if ((location+speed)>=3&&(location+speed)<=9)
                then (Offbridge,3,Arr)
                else (Offbridge,((location+speed) mod 20),Notarr)|
(Offbridge,location,0,speed)->
                if ((location+speed)>=3&&(location+speed)<=9)
                then (Onbridge,location+speed,Arr)
                else (Offbridge,((location+speed) mod 20),Notarr)|
(Onbridge,location,0,speed)->
                if (location+speed)>=9
                then (Offbridge,((location+speed) mod 20),Leave)
                else (Onbridge,((location+speed) mod 20),Notleave);
```

**Train B Box**

The above codes show the algorithm for a train's location. The Train B box is the abstract of train B (Fig. 5.11(D)). It implements the same function as the Train A box.

**Fanout Box**

The main function of the Fanout boxes (Figure 5.11(E), (F)) is to divide a group of values into two groups. The values will not be changed in the Fanout boxes.

**Log Box**

Figure 5.11(G) exhibits the structure of the Log box. The Log box communicates with the operator. It records all the information of the trains and signal

controller and reports them to the operator.

## 5.4 The Hume Solution for a Process control system

As mentioned before, a process control system is an important type of real-time reactive system. To evaluate a programming language for suitability for real-time reactive systems, its suitability for process control systems cannot be ignored. We will show our Hume implementation for a steam boiler problem because a steam boiler is a representative example of a process control system. This requires a real-time computer embedded in a complete process control environment that must respond to control the pump in a finite period. A complex computation is necessary to calculate the water quantity in the steam boiler. The steam boiler thus represents typical requirements for a process control system.

### 5.4.1 Application Specification for a Steam Boiler

Our Hume version for a steam boiler control(Figure 5.14) is based on the specification problem demonstrated by Jean-Raymond Abrial [66]. This demonstrates a new approach to solve the problem of controlling the pump. The system is comprised of seven units, of which the main ones are:

- *A steam boiler*: Its total capacity is C Litres. It has a valve for evacuation. M represents the quantity of water. The minimum quantity of water (in Litres) is M1, whilst the maximum is M2. The steam boiler will be in danger after five seconds, if the current water quantity is higher than M2 or lower than M1. N defines the normal quantity of water. The minimum normal quantity of water (in litres) is N1, while the maximum is N2 (M1<N1<N2<M2).W is the maximum quantity of steam(litres/sec).U1 and U2 defines the increase and decrease of the quantity of steam.

- *A pump*: Its capacity is P. It pumps water into the steam boiler. The reaction time is five seconds. It can be stopped instantly by an operator.

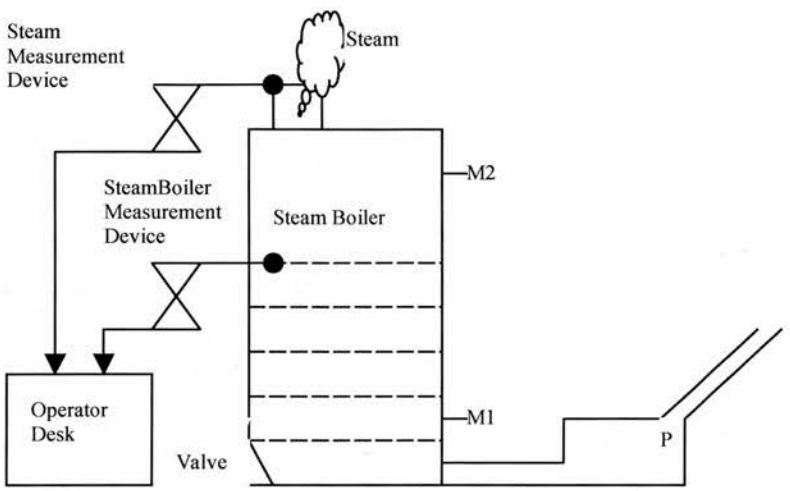


Figure 5.14 Hume Version Steam Boiler

- *A steam measurement device*: It is used to measure the quantity V of steam (in litres/sec)
- *A water level measurement device*: This device is used to measure the quantity q (in litres) of water in the steam boiler.
- A operator desk: This device is used to contact with operator.

### 5.4.2 The Hume implementation for the Steam Boiler Problem

Hume resolves the steam boiler problem by the approach demonstrated by Figure 5.15.

**Water level box**

The water level box simulates the water quantity in the steam boiler and the quantity of steam. If the operation mode is initialization mode, the quantity of steam will be set to zero. In the other modes, the water level box will simulate the water quantity and steam quantity

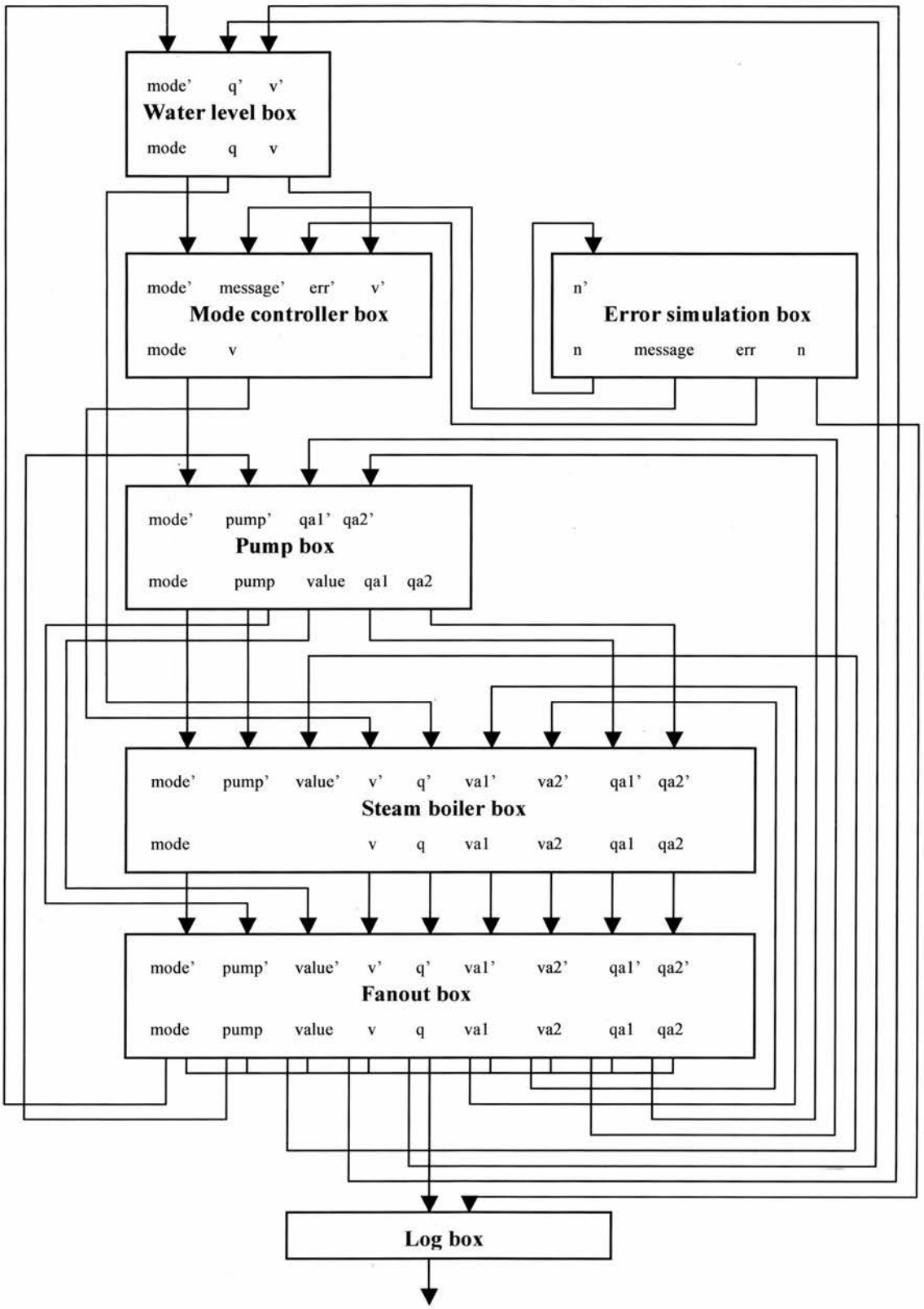


Figure 5.15 Boxes and Wires for a Steam Boiler

based on relative data. Table 5.2 summarizes the input and output value of the boxes

| digit | Value | Comment |
|---|---|---|
| mode | Initialization \| Normal \| Emergency\| Degraded \| Rescue \| Adjust \| Ready | Operation modes of the program |
| value | On \| Off | State of value |
| pump | 0 \| 1 | Disable or enable pump |
| message | Trans \| Steamwaiting \| Phyready | The message from the operator desk |
| f | (Int, Int, Int) | Errors caused by the physical equipment |
| v | Int | Current quantity of steam |
| q | Int | Current quantity of water in the steam boiler |
| va1 | Int | Intended minimum quantity of steam |
| va2 | Int | Intended maximum quantity of steam |
| qa1 | Int | Intended minimum quantity of water in the steam boiler |
| qa2 | Int | Intended maximum quantity of water in the steam boiler |

Table 5.2 Input and Output Values .

**Error Simulation Box**

This box simulates the potential physical errors caused by the physical equipment. There are three kinds of physical errors: the errors caused by the breakdown of physical units, the errors caused by failure of water level measurement, and the errors caused by failure of steam measurement. These three types of error were digitized to (0,1,0),(1,0,0),(0,0,1).The error simulation box also simulates the operator desk and generates the messages. The operator desk will send the relative messages when all physical units are ready, when steam is ready or when there is a transmission failure.

**Pump Box**

The pump box simulates the action of the pump. The pump is enabled or disabled according to the intended water quantity in the steam boiler and the current mode. If the predicted maximum water quantity is lower than the minimum normal water quantity (qa1>N2), the pump is enabled. If the predicted water quantity is in the range of the safe level (qa1>N1&&qa1<N2&&qa2>N2), the pump is enabled. If the intended minimum quantity is higher than the maximum normal quantity, the pump is disabled.

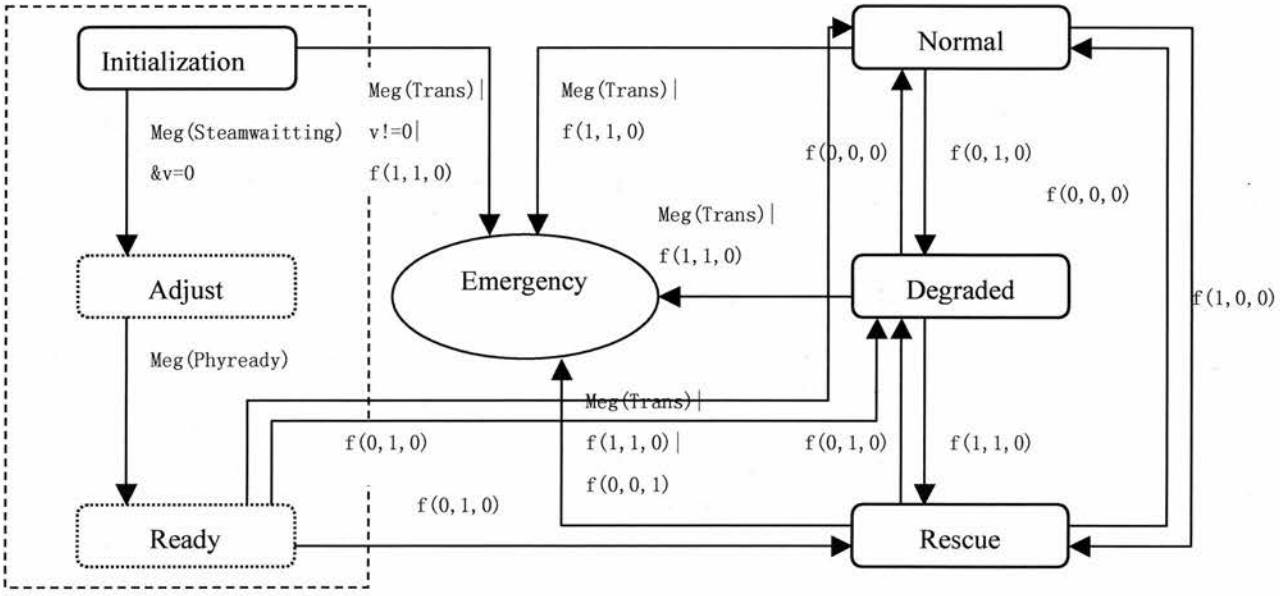


Figure 5.16 FSM for Operation Model

**Mode Controller Box**

Figure 5.16 exhibits the five operation modes of the steam boiler. These are the initialization mode(includes Adjust and Ready steps), the normal mode, the degraded mode, the rescue mode and the emergency mode.

- *Initialization mode*: In this mode, the whole system is checked and ready to start. The steam boiler controller checks the message from the operator and the error information from the physical units and water level measurement (f=(1,1,0)). If there is a transmission error ( r=trans), the system goes to the emergency mode. If

the steam boiler is waiting (message=Steamwaiting) and the quantity of steam is zero (v=0), the system switches to the adjust mode. If the quantity of steam is not zero, the system goes to the emergency mode. When the system is in the adjust mode, it checks the message sent by the operator. If physical units are ready (message=Phyready (Physical equipments are ready)), the steam boiler switches to the ready mode. In the ready mode, the system does the final check for errors. If there are any problems with the physical units (f = (1, 0, 0)), the system switches to the rescue mode. If there is a problem with the measurement of the water level (f = (0, 1, 0)), the system switches to the degraded mode. If everything is ok, the system goes to the normal mode and is ready to work. The valve is enabled if the quantity of water in the steam boiler is higher than N2 and the pump is enabled if the quantity of water in the steam boiler is lower than N1.

- *Normal mode:* The normal mode is the standard operating mode in which the quantity of water is maintained between N1 and N2 with all physical units operating correctly. The system switches to the degraded mode and asks for physical repairs, if any physical unit breaks down (f = (0, 1, 0)). If there is a failure in the water level measurement (f = (1 ,0 ,0)), the system switches to the rescue mode to try to maintain a satisfactory water level. If the quantity of water in the steam boiler is close to one of the limit values, the system goes to the emergency mode.
- *Rescue mode*: In this mode, the system intends to maintain the quantity of water in the steam boiler within a safety range (N1<q<N2), despite the failure of the water level measurement. The quantity will be estimated based on the data from the steam measuring unit and the current quantity of water in the steam boiler. This estimate will be implemented by the water level box. If the steam measuring unit cannot work correctly or the predicted quantity of water is in danger (qa1<M1 or qa2>M2), the system goes to emergency mode. If the water level measuring unit has been fixed (f = (0, 0, 0)) and all physical units work correctly,

the system will go to the normal mode. If any other physical units cannot work, the system goes to degraded mode.

- *Degraded mode*: The degraded mode is a state in which some physical units cannot work correctly, except for the water level measuring unit. The system predicts the quantity of water in the steam boiler and maintains it within a safe range. After the broken physical units have been repaired, the system switches to the normal mode. If the water level measuring unit is defective in this mode, the system switches to the rescue mode.
- *Emergency mode*: The emergency mode stops the whole system. In this situation, the whole system is in danger of unrecoverable failure. This mode cannot be connected to outside world directly through the operator. When the system reaches this mode, the physical environment will respond and take proper action to stop the program.

**Steam Boiler Box**

```
(Initi,p,value,v,q,val,va2,qa1,qa2)->
                    (Initi,v,q-value*A*T,v,v,q-value*A*T,q-value*A*T)|
(Normal,p,value,v,q,val,va2,qa1,qa2)->
                    (Normal,v,q-v*T+P*p-value*A*T,v,v,
                     q-v*T+P*p-value*A*T,q-v*T+P*p-value)|
(Rescue,p,value,v,q,val,va2,qa1,qa2)->
                  (Rescue,val-U2*T,qa1-va2*T-U1*T*T-P*p,val-U2*T,
                   va2+U1*T,qa1-va2*T-U1*T*T-P*p,qa2-val*T+U2*T*T-P*p)|
```

The above codes show the function of the steam boiler box,which is to simulate the change of quantity of water in the steam boiler. The completed code has been demonstrated on Appendix F. When the steam boiler box empties, the water in the steam boiler is in the initialization mode. The steam boiler box increases or decreases the quantity of water in the normal mode and estimates the quantity of water in the rescue and degraded modes.

## 5.5 Comparison

There are many digital watch and railroad implementations written in different languages. We will compare our Hume version with a *Charts implementation [65] for a digital watch application in detail and with that for a railroad application in outline.

### 5.5.1 *Charts

*Charts is a heterogeneous HCFSM (Hierarchical Concurrent Finite State Machine) formalism which aims to specify and simulate reactive systems with control logic. The use of a hierarchy allows an FSM to be nested into another FSM (for example a set of sub-states) and concurrency allows multiple active states, which will be refined as a FSM. A FSM will operate and communicate with other FSMs simultaneously. FSMs have been used to analyse sequential control logic for reactive systems. As in the FSM examples for a digital watch and a railroad, FSMs are effective in describing the control behaviour of the systems.

*Charts provides a good formalism to simulate reactive systems logic controls by introducing FSMs. concurrency and hierarchical properties into its semantics. With these properties, *Charts provides a good method to describe both control logic and computational tasks, specify hierarchical concurrent composite behaviours and allows the choice of different semantics. *Charts provides five concurrency models for reactive systems: discrete events (DE), synchronous/reactive (SR), synchronous dataflow (SDF), dynamic dataflow (DDF) and communicating sequential process (CSP). The DE model provides a way to model distributed or parallel systems and their communication infrastructure, eg. digital hardware. The SR model is good for describing concurrent behaviours of control-intensive systems. The SDF and DDF models are suitable for describe signal processing and computation-intensive systems. The CSP model is useful for managing resources at the system level [65].

### 5.5.2 Comparison for a Digital Watch

***Charts Implementation of a Digital Watch**

In a *Charts implementation, the behaviour of the digital watch is simulated by DE and SDF. The topmost level DE (Discrete Event) is to model the environment of the watch (including the human user), the second level is SDF (Synchronous Dataflow), which includes a status keeper and an alarm keeper to trace the current time and alarm settings. They are basically composed of counters with various maximum counts.

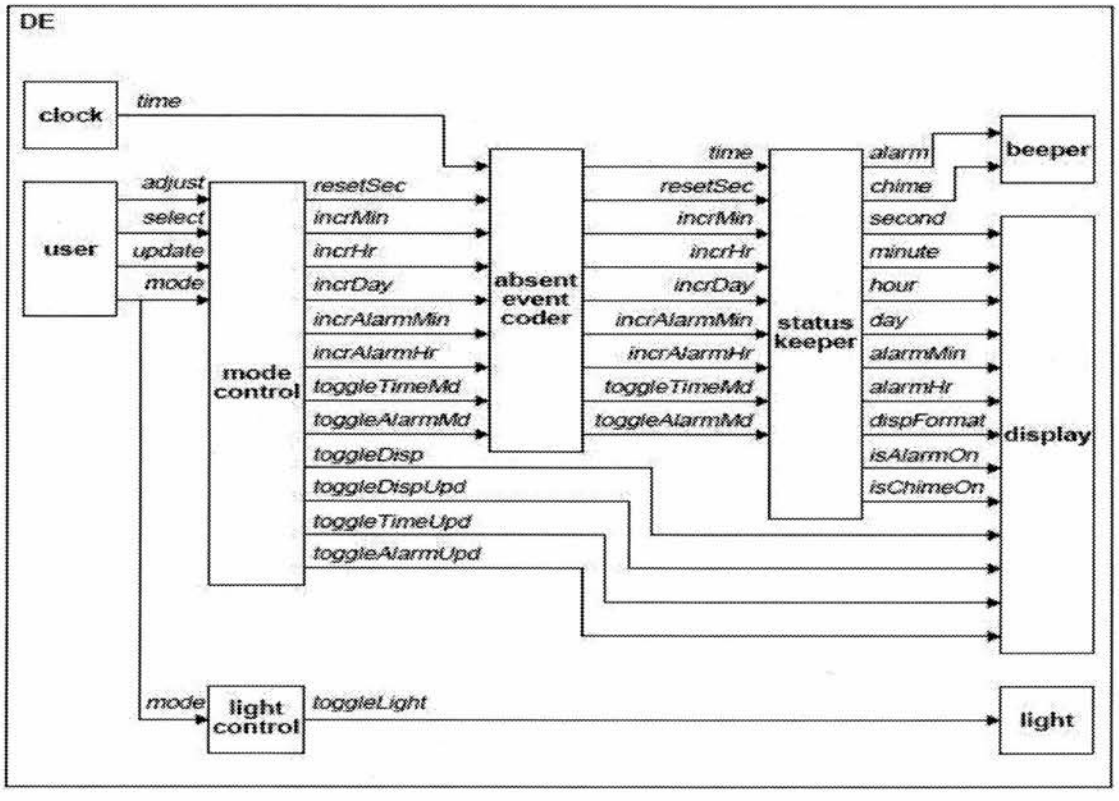


Figure 5.17 The Topmost Level DE [67]

**Finite State Machine**

In *Charts, FSMs are used in terms of *nodes* denoting the states (e.g. the four button states) and *arcs* denoting transitions (states for display models). FSMs effectively convey the control behaviour of the system.

Like *Charts, Hume introduces FSA (Finite State Automata) into its design. Hume generalizes FSA transitions, and provides a way of constraining the types of inputs, choosing matched patterns in left-hand expressions and generating outputs by

the control structures in right-hand expressions [2]. In the example of the digital watch, FSMs for time and alarm will be embedded in the watch box in order to constrain the inputs and generate the output based on pattern rules. The introduction of FSA confirms the Hume constraints as predictable and decidable program properties.

**Hierarchies**

The *Charts implementation uses DE and SDF to describe the digital watch problem: DE simulates the environment of the digital watch, such as the human user performance and the display screen. The SDF is in charge of the mode status control. The hierarchy semantics specifies the interaction between the states and the refining FSM in those states.

Like *Charts, Hume also introduces hierarchy. The digital problem is described in three layers. The declaration layer defines the status and functions of the digital watch; the coordination layer links the functions into concurrent process and simulates the environment as box; and the declaration layer defines the data structure and functions for the coordination layer.

**Reaction to the Enviroment**

The *Charts implementation uses the DE model to respond to the environment. It simulates the user's actions with the **user** (Figure 5.17), maintains current time and alarm settings with a status keeper and is in charge of control actions with light control and mode control.

In the Hume solution, the environment performances are encapsulated into boxes in the coordination layer. For example, the button box simulates the action of pressing the buttons by the user; the digital watch box is in charge of the internal action of the digital watch; and the log box is in charge of the abstract objects on screen which will be used to interact with the user.

**Response to Input**

In the *Charts implementation, the whole digital watch system is combined into

a network of blocks connected by directed arcs. The events with a time stamp, which points out the action time, are used to communicate between blocks. The whole process is controlled simultaneously by the global time. In the digital watch example, every action of pressing the button is simulated to an event and will be given a time stamp. If the user presses the mode button once and the adjust button twice, there will be three events triggered with time stamps. Three events are sent to the mode control block. The mode control block will put these three events into a time queue, arrange them according to their time stamps and send them to the relative blocks as inputs. It is obvious that the *Charts implementation simulates an adjusting action (including several button presses) as a one-shot process. However, it causes time delay because each event does not immediately initiate a response between the blocks. Events are sent to a queue, arranged by the DE simulator and then sent to the relevant blocks.

Hume, alternatively, can simulate the process precisely. Boxes are waiting for inputs. When the user presses a button, the input is sent to the digital watch box immediately, and the digital watch box responds to the input and generates the output based on the current status. Users respond immediately to every button press ('event' in the *Charts implementation), not just to one adjusting action that includes several events.

**Exception Handling**

Both *Charts and Hume have the ability to handle an exception occurring in real time. In the case of the digital watch, the normal exception is the timeout exception. In a *Charts implementation, the timeout event is handled by a counter in the SDF level. The counter for timeout is reset every time whether there is a state or not. The design of a timeout counter can handle a timeout event, but continuous counting would burden the memory usage of the hardware. It also decreases the safety of the system, because the more the hardware is used, the greater the risk to the system.

In our Hume implementation, exceptions are raised in the expression layer and handled by the surrounding box [3]. In the case of the digital watch, the exception of

timeout is raised in the wire between the user box and the digital watch box and handled by the digital watch box. It is not necessary to set up any time counter to count the timeout event. We raise a timeout exception as an input to the digital watch box, if there is no input passed by the wires. The digital watch box handles it and gives a proper response.

**Resource Boundedness**

Since *Charts is designed for reactive systems, there is no consideration of memory usage and weak consideration of resource bounded and time behaviour in a *Charts implementation.

In a Hume implementation, we can measure the space cost of stack and heap requirements for the boxes and wires using the Hume Abstract Machine (pHAM) [46]. The Hume implementation makes it possible for a real-time program to run on resource-bounded hardware with high safety and reliability.

### 5.5.3 Comparison with a Railroad

In this section, we give a comparison between *Charts and Hume implementations for the problem of a process control system. We will not demonstrate the comparisons that are similar with the digital watch example and have already been described, but will identify those comparisons that are more specific to the railroad system problem.

**Concurrency**

In *Charts, an SR model has been used as a topmost level to describe concurrent problems for a railroad. SR models, comprised of communicating blocks, are used to describe events. Execution of the system occurs at a sequence of discrete instants, which are defined as ticks. Ticks are independent and initiated by the environment. In each tick, each block generates output from input by its internal function. Blocks communicate with each other in that the inputs of one block will be the outputs of another block [65]. The SR model contains a clock generating the ticks, the railroad controller, the two signals, the two trains and two speed blocks. The trains and

controller blocks will communicate in the ticks.

The *Charts implementation provides an effective way to resolve the concurrent problems. Like *Charts, Hume has the ability to deal with concurrent events. The coordination layer is a finite state language, which is used to describe interacting processes. Boxes are the basic units of coordination, which are similar with blocks in *Charts. The difference between a box and a block is that a box is an abstract notion of a process with inputs and outputs based on a functional expression [78]. Boxes are wired into a (currently static) process network. In the case of the railroad problem, Hume provides two train boxes, a control box and two speed boxes, like the blocks in *Charts. All these boxes are wired into a process network and communicate with each other by wires.

## 5.6 Conclusions

In this chapter we have provided Hume implementations for different types of reactive systems. We have given a comparison with *Charts for the first and second applications and demonstrated a Hume solution for a process control system.

*Charts provides a good formalism to resolve the problem of reactive systems. It separates the reactive system into different portions and provides the appropriate modelling techniques for these portions. It provides diverse models that can coexist and interact by hierarchical combination [65]. Concurrency has been considered because *Charts allows interaction between different models. FSMs have been used to describe and analyse sequential control logic.

As with *Charts, hierarchy, concurrency and FSM have been introduced into the Hume design, making Hume a good programming language for reactive systems. *Charts provides a good method for programming reactive systems. Hume has this good formalism and improves on it to meet the requirements for real-time systems, such as the requirements for transparent time and space costing and for safety and

correctness.

# Chapter 6

# Space and Time Behaviour

## 6.1 Space Costing Under Linux and Symbian OS

The main characteristic of the Hume language design for real-time systems is that an efficient cost model [76] has been produced for Hume programs. A simple space cost model has been defined for FSM-Hume to predict upper bound stack and space usage with the support of the prototype Hume Abstract Machine. Because the heap and stack requirements for the boxes and wires only demonstrate the dynamically variable memory requirements, we provide all other memory costs at compile-time based on the number of functions, boxes, wires and the sizes of static strings. We can get the complete static information about system memory requirements by using the FSM-Hume analysis to predict the stack and heap requirements for an FSM-Hume program.

In the following, we will illustrate cost modelling and the use of real-time features in Hume, accompanied by four representative real-time applications, including the digital watch program and the railroad program demonstrated in Chapter 5. We also choose two other simple real-time applications (a pump application and a vehicle application). These are different types of typical real-time applications with real-time requirements. The pump application simulates a pump control system, which is used to drain water from a mine shaft. The vehicle application is the simulation of a basic

| | box | Actual heap | Predicted heap | Excess P-A((P-A)/A) | Actual stack | Predicted stack | Excess P-A((P-A)/A) |
|---|---|---|---|---|---|---|---|
| Digital Watch | button | 5 | 5 | 0 (0%) | 6 | 6 | 0(0%) |
| | display | 254 | 266 | 12(4.7%) | 24 | 32 | 8(33.0%) |
| | Fanout | 49 | 53 | 4 (8.2%) | 28 | 30 | 2(7.1%) |
| | watch | 52 | 66 | 14(26.9%) | 29 | 33 | 4(13.8%) |
| | Total | 498 | 552 | 54(10.8%) | 84 | 103 | 15(17.8%) |
| Rail Road | control | 25 | 25 | 0(0%) | 15 | 15 | 0(0%) |
| | log | 286 | 308 | 22(7.7%) | 26 | 28 | 2(7.7%) |
| | Fanout | 45 | 45 | 0(0%) | 42 | 42 | 0(0%) |
| | speed | 13 | 15 | 2(15.4%) | 18 | 21 | 3(16.7%) |
| | trainA | 40 | 40 | 0(0%) | 14 | 15 | 1(7.1%) |
| | trainB | 40 | 40 | 0(0%) | 14 | 15 | 1(7.1%) |
| | Total | 555 | 582 | 27(4.8%) | 133 | 140 | 7(5.3%) |
| Vehicle[50] | control | 57 | 60 | 3(5.3%) | 36 | 42 | 6(16.7%) |
| | env | 49099 | 49580 | 571(1.2%) | 128 | 139 | 11(8.5%) |
| | vehicle | 49164 | 49648 | 484(1%) | 132 | 142 | 10(7.6%) |
| | Total | 98511 | 99488 | 977(0.9%) | 287 | 324 | 27(9.4%) |
| Pump[50] | Total | 425 | 483 | 68 (16%) | 166 | 162 | 4(2.4%) |

Table 6.1 Heap and Stack Usage

autonomous vehicle trying to follow a white line by continuously analyzing a camera image, including one row of bits from a two dimensional bit-map scene. The specific implementations have been demonstrated in a previous paper [50].

Table 6.1 demonstrates the four Hume applications' actual and predictable heap and stack usage (calculated in four byte words), which have been calculated by the HAM Interpreter. The results show that the cost for major boxes of Hume programs can be predicted with high precision. Overall, the predictions are very accurate for the

majority of boxes and wires. The total predicted heap usage for each application is precise to 0.9% in the best case (the vehicle application), 16% in the worst case (the Pump application) and less than 11% in the other two cases (4.8% in the rail road application and 10.8% in the digital watch application). The total predictable stack usage for each application is accurate to 2.4% in the best case (the pump application), 17.8% in the worst case (the digital watch application) and less than 10% in the the other two cases (5.3% in the railroad application and 9.4% in the pump application). The serious discrepancy for the heap and stack usage of the watch box for the digital watch application is due to the fact that it is relatively complex, with many alternative choices (the worst case will be assumed even if it does not actually). These boxes are asynchronous and cannot be active unless certain inputs are available. Because the unavailable inputs could be present in other dynamic contexts, the HAM Interpreter will reserve space for these unavailable inputs for correctness reasons. For example, in the case of the digital watch application, the display box makes extensive use of strings to display the information of each execution cycle. The heap usage of the display box will be slightly overestimated, because the sizes of output strings are different dynamically in different cases. A conservative estimate will be used in this situation. The space predictions for the four application boxes are completely satisfied especially, in the case of the vehicle application, with an overestimate of 968 words for an actual 98511-word heap usage and an overestimate of 27 words for and actual 287-word stack usage. Based on the precise analysis results for the four Hume applications shown in figure 6.1, we can conclude that the dynamic space usage for Hume applications would be precisely estimated by our HAM Interpreter.

We have also run these four applications on a 1.2GHz Pentium Ⅳ processor under Linux to show that the HAM can meet constant space requirements. Table 6.2 demonstrates the memory requirements for the Hume Abstract Machine running four Hume applications on a Linux. The Hume code size is given by the *text* field; *data* represents the immutable data, eg. literal strings; *bss* demonstrates the mutable data

area (include the abstract machine data); and *dec* and *hex* represent the totals as decimal and hexadecimal numbers respectively.

```
a;Hume Abstract Machine (0 Instructions; 0 Labels; 0 Strings; 0 Boxes)
  text    data     bss     dec     hex filename
  59187     536    1068   60791    ed68 hami
```

```
b;Digital watch (4097 Instructions; 93 Labels; 137 Strings; 29 Boxes)
  text    data     bss     dec     hex filename
  60134     540   19340   80014   1388e hami
```

```
c;Pump (2628 Instructions; 170 Labels; 121 Strings; 42 Boxes)
  text    data     bss     dec     hex filename
  60134     540   14412   75086   1254e hami
```

```
d:Rail Road (1630 Instructions; 78 Labels; 100 Strings; 43 Boxes)
  text    data     bss     dec     hex filename
  60134     540    9164   69838   110ce hami
```

```
e;Vehicle (3875 Instructions; 100 Labels; 81 Strings; 12 Boxes)
  text    data     bss     dec     hex filename
  60134     540   18252   78926   1344e hami
```

Table 6.2 Space Usage on a Linux

Table 6.2(a) shows the memory requirement for HAM without any applications. Table 6.2 (b, c, d, e) represents specific memory requirements for the Hume Abstract Machine running for the four applications. Our measurements show that the total memory usage for HAM is less than 62 KB and the total memory usage for HAM running with these four application on Linux is less than 80 KB (80KB for the digital watch, 75KB for the pump, 70 KB for the railroad, and 79 KB for the vehicle). The total memory usage includes heap and stack overheads, Linux code and data, Hume runtime system code and data, and the abstract machine instructions.

In order to test whether the HAM can meet the space requirements for real-time embedded systems, we ran each of these four applications on the Symbian OS of a

Nokia 7650 mobile phone (8M RAM)for a period of 10 minutes, using the calculated memory settings. Based on the information we received from the calcuated memory settings on the Linux (the minimal numbers of instructions, labels, strings and boxes for specific programs),we can measure the minimum dynamic memory usage of these four applications. The total dynamic memory usage for each of the four applications is quite similar. All the dynamic memory usages are around 437.8KB on the Symbian OS, which has a 8M memory for applications. This includes the code for the HAM Interpreter and Hume applications, the runtime stack and the Symbian libraries.

## 6.2 Time Behaviour Under Linux and Symbian OS

We also ran each of these four applications continuously under a Linux (1.2GHz Pentium Ⅳ processor) and a Symbian OS (104 MHz Arm9 processor) for a period of 1000 cycles for the purpose of verifying that a Hume program can meet real-time requirements.

Table 6.3 shows the response time to inputs under both the Linux and the Symbian OS. Clock timings under an RTLinux are accurate to the nanosecond level. The Symbian OS supports the 64Hz clock in an actual device, so the clock timing is accurate only to this level. We cannot get a more accurate number when the response time is less than 0.15ms, since this is the granularity of the real-time clock on the Nokia 7650. Any interval less than 0.15ms will be set to 0 automatically by the clock timer.

| | Linux | | | Symbian OS |
|---|---|---|---|---|
| | box | Execution Time | Maximum Delay | Maximum Delay |
| Digital | button | 0.4ms | 0.009ms | <0.15ms |

| | | | | |
|---|---|---|---|---|
| Watch | display | 2.2ms | 0.072ms | <0.15ms |
| | watch | 2.2ms | 0.092ms | 0.467ms |
| Railroad | Control | 0.102ms | 0.032ms | 0.484ms |
| | Log | 0.106ms | 0.024ms | <0.15ms |
| | Train A | 0.220ms | 0.0678ms | <0.15ms |
| | Train B | 0.210ms | 0.0675ms | <0.15ms |
| Vehicle[50] | control | 0.12ms | 0.042ms | <0.15ms |
| | environ | 1.5ms | 0.063ms | <0.15ms |
| | vehicle | 0.8ms | 0.167ms | 0.867ms |
| Pump[50] | Airflow | 0.3ms | 0.140ms | <0.15ms |
| | carbonmonoxide | 0.08ms | 0.138ms | <0.15ms |
| | environ | 0.8ms | 0.006ms | <0.15ms |
| | methane | 3.5ms | 0.237ms | 0.156ms |

Table 6.3 Time Behaviour

As Table 6.3 shows, in the case of the digital watch, the maximum execution times were 0.43ms for the button box, 2.2ms for the display box and 2.2 ms for the watch box. The maximum delay between results produced by the button box and consumed on the watch box is 0.092ms under the Linux, whilst this delay is less than 0.467ms under the Symbian OS. The maximum delay on any other wires is less than 0.1ms.

In the case of the vehicle application, the execution time for the boxes is less than 1.5ms. Under the Linux, the worst-case real-time response is 0.042ms to the control inputs, is 0.042ms and the worst-case real-time response to the environment inputs and 0.063ms to the environment inputs. Under the Symbian OS, the worst-case real-time response time is 0.867to the vehicle box inputs and less than 0.15ms for other wires.

In the case of the railroad application, the execution times for the train A and

train B boxes under the Linux are less than 0.22ms, the worst-case real-time response to the control input being 0.067ms. The corresponding delays under the Symbian OS are 0.484 in the worst case (the control box) and less than 0.15ms for the train A and train B wires.

In the case of the pump application, we can see that the operation of the methane alarm box can be executed in 3.5ms, with a 0.237ms real-time delay in response to the inputs generated by the environment box. The maximum delay in servicing any other wires is less than 0.14ms. The corresponding maximum delay under the Symbian OS is around 0.156ms.

## 6.3 Space and Time Analysis under RTLinux

HAM has also been ported to an RTLinux by other researchers and there are also some convincing research results [46, 50]. The total dynamic memory usage of Pump application under RT-Linux was less than 412 KB and the worst-case real-time responses to the control and vehicle inputs were 0.0224ms and 0.0312ms respectively [50].

More detail research results would be obtained from the related paper [46].

## 6.4 Conclusions

Compared to RTLinux and Linux porting, we can conclude that our Symbian HAM primary achieve the aim to precisely predict the time usage like RTLinux HAM and Linux HAM. The related results have been demonstrated in the above chapters. Because of the difference between Linux and Symbian system(eg,Symbian do not have **Size** command like Linux) and our porting is still in a very elementary stage.Our Symbian HAM can not predict the memory usage as precise as RTLinux HAM and Linux HAM. However, we believe that the accuracy will be improved with the process of porting.

Based on our space and time analysis for Hume applications under Linux and Symbian OS and the previous research results under RTLinux, we can conclude that space usage for Hume programs can be predicted with a high degree of precision and also that these programs can be robustly run in real-time systems on limited memory, without fear of space leaks. The research results of response time prove that Hume programs can respond to the inputs and outputs of the environment in a restricted time.

# Chapter 7

# Conclusions and Future Work

## 7.1 Conclusions

This dissertation has presented a novel domain-specific language that is designed for resource bounded domains such as real-time reactive systems. Based on our implementations of representative types of reactive systems, we can conclude that Hume is suitable for writing reactive, concurrent, asynchronous and correct programs for real-time reactive systems.

In order to estimate Hume ability to programming resource boundedness, we successfully port Hume to Symbian OS in Chapter 4. Our port of Hume to the Symbian OS extends Hume into a new application domain. We implement a real-time application (The Sound application). The short and concise Hume application demonstrate the ability of Hume to write real-time programs for the resource bounded system. Our port of Hume to Symbian OS and the port of Hume to the RTLinux prove the portability and applicability of Hume. According to the comparison with the Haskell applications on Palm OS and the C++ applications on Symbian, we can conclude that Hume can write short applications and is a easy-to-use programming language.

In chapter 5, we implement Hume applications targeting for three typical real-time system domains. These implementations prove Hume's ability to program for different real-time domains. With our comparison against *Chart solutions for

real-time reactive systems, we can conclude that not only does Hume has *Chart's good formalisms such as hierarchy, concurrency and FSM, but also it improves upon it, meeting the requirements for transparent time and space costing and for safety and correctness. These characteristics make Hume a good programming language for reactive systems.

We also demonstrated our precise research results in Chapter 6. Our research results under Linux and the Symbian OS, combined with the previous research results under RTLinux[46,50], show that Hume combines a high level of programming abstraction with the capability of meeting low-level requirements such as the required bounded space and time behaviour of real-time reactive systems. Our results show that a functional programming language can, in principle, be highly practical in the real-time domain, with the help of good design.

Considering the current research with Hume, we can conclude that Hume is a good programming language for real-time reactive systems. Firstly, Hume supports basic programming language characteristics such as straightforward coding, portability and general applicability. Secondly, Hume supports real-time programming language characteristics such as concurrency, reactivity, asynchronicity, interrupt handling and manipulation of data types. Lastly and most importantly, Hume makes programming for real-time systems more safe by combining a high level of programming abstraction with the capability to meet low-level requirements such as the required bounded space and time behaviour of real-time reactive systems.

## 7.2 Future Work

### 7.2.1 Porting Hume to Symbian OS

Our port of Hume to the Symbian OS is still at a very early stage, with more testing required. With our research results, we can conclude that Hume programs can respond to the environment in a limited time and the program can run in a resource bounded system. This is just a first step that extends Hume to program for the

Symbian OS. Compared to RTLinux HAM, our Symbian HAM can not report the programming memory usage as concise as RTLinux because of the difference between RTLinux and Symbian, for example, RTLinux has the **Size** command to show the memory usage for us. I believe this problem can be sloved with more deeply research.

Our interface to the Symbian OS is fairly rudimentary because we have only implemented a few simple applications. More work can be done on interfacing GUI functions of the Symbian OS and interfacing other mobile phone equipment such as builtin cameras and bluetooth. The high encapsulation of the Symbian OS interface and inheritance between classes increase the difficulties for Hume in interfacing with the GUI functions, and complicates the implementation process of Hume applications, which will decrease the efficiency of Hume. We would like to build applications at a high level, rather than interface directly with the operating system. We would like to find an easier and more straightforward way to constrain the space usage and time behaviour of Hume applications under the Symbian OS. We believe that Hume's properties will be more obvious after these difficulties have been overcome.

### 7.2.2 Programming for Reactive Systems

We have implemented Hume applications for three real-time aspects, more applications can be done on programming more real-time domains. After doing more and more implementations targeting for different real-time domains, we believe that Hume's ability in programming real-time system will be more specific. After we do more practical programming in Hume, more special requirements of specific real-time domains will be found and we can make Hume's properties be more and more suitable for programming real-time systems.

# Reference


[1] A.Burns and A.Wellings. Real-time Systems and Programming Languages ( Ada95, real-time Java and real-time POSIX), Third Edition. Addison Wesley, pages: 3-8, 2001.

[2] G.Michaelson, K. Hammond and J. Serot. The Finite State-ness of FSM-Hume. Chapter 1, The Symposium on Trends in Functional Programming, Edinburgh, Sept. 11-12, 2003.

[3] K.Hammond and G.Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In Proceeding Programming and Component Engineering(GPCE'03), Erfurt, Germany, Sept.2003. Springer-Verlag, LNCS, 2003.

[4] S. J.Young. Real-time Languages: Design and Development. Chichester: Ellis Horwood, page:8,1982.

[5] K.R.Apt and E.R.Olderog. Verification of Sequential and Concurrent Programs. 2nd Edition, Springer Verlag, pages:89-92,1996.

[6] K.Hammond and G.Michaelson. Predictable Space Behaviour in FSM-Hume. In Proc. Implementation of Functional Languages,Madrid,Spain, Springer-Verlag Lecture Notes in Computer Science, pages:1-16, 2003.

[7] Nokia Corporation. Symbian OS: Active Objects And The Active Scheduler. Version 1.2, page:10, 2003.

[8] K. Hammond. Is It Time for Real-Time Functional Programming. Trends in Functional Programming Volume 4, Chapter1,pages:5-10, 2005.

[9] J.Armstrong. The development of Erlang. In ACM SIGPLAN International Conference on Functional Programming, June 1997; SIGPLAN Notices 32(8):196--203, August 1997. Available at : http://www.erlang.se SIGPLAN ACM Functional Programming. Accessed date: 28th,July,2005

[10] G.Berry. Real-Time Programming: General Purpose or Special-Purpose Languages. In 11th IFIP World Congress, San Francisco, Californie, pages:11-17, 1989.

[11] J.Barnes. High Integrity Ada, The SPARK Approach. Addison-Wesley, pages:18-20, June 1997.

[12] D.Marshall and C.Turfus. Developer Platform 2.0 for Series 60: Application Framework，Handbook 1.0., Nokia corporation, June 2002

[13] D. Mery, Symbian OS Version 6.x functional description, Symbian Ltd. Avaible at http://www.symbian.com/technology/symbos-v6x-det.html, Accessed date: 12nd, Dec,2005

[14] J. Gosling, W. Joy, and G. Steele: The Java Language Specification, 2nd Edition, Addison-Wesley, Reading, Massachusetts, Pages:35-36,1996

[15] G. Muller, C. Consel, R. Marlet, L. P. Barreto,F. Merillon, and L. Reveill`ere. Towards Robust OSes for Appliances: A New Approach Based on Domain-Specific Languages. In ACM SIGOPS European Work-shop 2000 (EW'2000), Oct. 2000.

[16] P. Caspi, D. Pilaud, N. Halbwachs and J. Place. Lustre: a Declarative Language for Programming Synchronous Systems.In Proc. 14th ACM Symposium on Principles of Programming Languages (POPL '87), M□unchen, Germany, 1987.

[17] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. In Proc. of IEEE, volume 79(9), pages: 1270-1282, 1991.

[18] K. Masselos, K. Danckaert, F. Catthoor and H. De Man. A Specification Refinement Methodology for Power Efficient Partitioning of Data-Dominated Algorithms Within Performance Constraints. Journal of VLSI Signal Processing Systems 26(3), pages:291-317, Nov. 2000.

[19] K. Hammond. Hume: a Bounded Time Concurrent Language. In Proc. IEEE Conf. on Electronics and Control Systems (ICECS '2K), pages: 407-411,Kaslik, Lebanon, Dec. 2000.

[20] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In Proc. POPL'96 .ACM Symp. on Principles of Programming Languages, pages:295.308, Jan. 1996.

[21] A.W. and MacQueen, D. B. Standard ML of New Jersey. In 3rd International Symposium on Programming Language Implementation and Logic Programming. M. Wirsing, Ed., New York, NY, USA, Volume 528 of Lecture Notes in Computer Science. Springer-Verlag, pages:1-13,1991.

[22] J. Ostroff. A Logic for Real-Time Discrete Event Processes. In the IEEE Control Magazine 10(4), pages: 95-102,1990.

[23] A. Benveniste and P.Le. Guernic. Synchronous Programming with Events and Relations: the Signal Language and its Semantics. Science of Computer Programming16(2), Elsevier, pages: 103-149,1991.

[24] T. Brus, M. C. J. D. van Eekelen, M. van Leer, M. J. Plasmeijer, and H. P. Barendregt. CLEAN - a Language for Fnctional Gaph Rewriting. In Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA'87), No. 274 in LNCS, pages: 364-384. Springer-Verlag, 1987.

[25] J. R. Burch. Automatic Symbolic Verification of Real-Time Concurrent Systems. PhD thesis, Carnegie Mellon University, Aug. 1992.

[26] T. Gautier, P. L. Guernic, and L. Besnard. SIGNAL: A Declarative Language For Synchronous Programming of Real-Time Systems. In G. Kahn, editor, Functional Programming Languages and Computer Architecture, volume 274 of Lect Notes in Computer Science. Springer-Verlag, pages: 257-268,1987.

[27] Wirth, N., and K. Jensen. Pascal: User Manual and Report, 3rd Edition. Springer-Verlag, Pages:25-26,1988.

[28] L. Turbak. Higher-Order Functions. Wellesley College, Feb. 2005. Available at: http://homepages.inf.ed.ac.uk/wadler/realworld/. Accessed date: 6th July, 2005.

[29] D. Mery. Client Server Overview, Symbian OS V6.1 Edition For C++, Symbian Ltd. Availabe at:http://www.symbian.com/developer/techlib/ v8.1adocs/doc_source/guide/Base-subsystem-guide/N1007A/InterProcess Communication/ Client ServerOverview.guide.html. Accessed date:22nd, July,2005.

[30] P. L. Guernic, T. Gautier, M. L. Borgne, and C. de Marie. Programming Real-time Applications with SIGNAL. In Proc. the IEEE 79(9), pages:1321-1335, Sept. 1991.

[31] N. H. P. Caspi, D. Pilaud and J. Place. Lustre: a Declarative Language for Programming Synchronous Systems. In Proc. 14th ACM Symposium on Principles of Programming Languages (POPL '87), Munchen, Germany, 1987.

[32] F. Boussinot and R. de Simone. The Esterel Language. In Proc. the IEEE, 79(9), pages:1293–1304, Sept. 1991.

[33] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics. Journal of Science Of Computer Programming, Vol. 19, Num. 2, pages: 87-152, 1992.

[34] D. Harel. Statecharts: A Visual Formalism for Complex Systems. Science of

Computer Programming 8(3), pages:231–274, June 1987.

[35] L. Brown. Erlang: An Open Source Language for Robust Distributed Applications. School of Computer Science, Australian Defense Force Academy, Canberra, Australia, 2002

[36] S. Thompson. Haskell: The Craft of Functional Programming, Addison -Wesley, pages:20-27,1999.

[37] G.. Berry. Real-time Programming: General Purpose or Special-Purpose Languages. In G. Ritter, editor, Information Processing 89, Elsevier Science Publishers P.V, pages:11-17, 1989.

[38] A. Sabry. What is a Purely Functional Language. Journal of Functional Programming 8(1), Cambridge University Press, pages:1-22,Jan. 1998.

[39] G. Hutton. Requently Asked Questions for comp.lang.functional. University of Nottingham, Nov. 2002. Available at: http://www.cs.nott.ac.uk/~gmh //faq.html. Accessed date: 15th,Dec.2005

[40] S SE. Keene. Object-Oriented Programming in Common Lisp. Addison-Wesley, 1989.

[41] S.P. Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language. Tech. report , Yale University, 1999.

[42] J. H. Reppy. Concurrent Programming with Events-The Concurrent ML Manual. AT&T Bell Lab., version 0.9.8 edition, Feb 1993.

[43] X. Leroy, D. R'emy, J. Vouillon, and D. Doligez. Objective Caml. User's manual, 1998. Available at http://pauillac.inria.fr/ocaml/. Accessed date:28th,July,2005

[44] J.Armstrong, M.Williams, R.Virding and C. Wikström. Concurrent Programming in Erlang, 2nd Edition, Prentice Hall, pages: 17-23,1996.

[45] G. Michaelson, K. Hammond and J. Serot. The Finite-Stateness of FSM-Hume. In TFP '03— Symposium on Trends in Functional Programming, Edinburgh, Scotland, Sept. 2003.

[46] K. Hammond and J. Serot. An Abstract Machine for Resource-Bounded Computations in Hume. In IFL '03— Implementation of Functional Languages, Edinburgh, Scotland, Sept. 2003.

[47] K. Hammond and G. Michaelson. The Hume Report 0.0, University of St Andrews and Heriot-Watt University, pages:4-7, July 2000.

[48] G. Michaelson and K. Hammond. The Dynamic Properties of Hume: a Functionally Based Concurrent Language with Bounded Time and Space Properties. In Proc. IFL 2000 — Implementation of Functional Languages, Aachen, Germany, Springer-Verlag LNCS, Sept. 2000.

[49] K.Hammond. An Abstract Machine Implementation for Hume. unpublished paper, University of St Andrews, 2001.

[50] G. Michaelson, K. Hammond and J. Serot. FSM-Hume: Programming

Resource-Limited Systems Using Bounded Automata. In Proc. the 2004 ACM symposium on Applied computing, March 14-17, 2004.

[51] K. Hammond and G. Michaelson. Hume: a Functionally-Inspired Language for Safety-Critical Systems. In 2nd Scottish Functional Programming Workshop, St Andrews, Scotland, July 2000.

[52] J. Hughes.Why Functional Programming Matters. Programming Methodology Group Memo PMG-40, CTH Gothenburg, 1984.

[53] M. Jaffe. Software Requirements Analysis For Real-Time Process-Control Systems. IEEE Transactions on Software Engineering 17(3), pages:241-258, 1991.

[54] Leveson, N., G.: Safeware, System Safety and Computers, Addison-Wesley, pages:485-486,1996

[55] A. Burns and A. Wellings. Real-time Systems and Programming Languages ( Ada95, real-time Java and real-time POSIX), Third Edition. Addison-Wesley, pages: 5, 2001.

[56] G. Michaelson. Constraint on Recursion in the Hume Expression Language. IFL 2000 — Intl. Workshop on Implementation of Functional Languages, Aachen, Germany, Sept. 2000.

[57] S. Clamage, Mixing C and C++ Code in the Same Program, Sun Microsystems, Sun ONE Studio Solaris Tools Development Engineering, Available at: http://developers.sun.com/prodtech/cc/articles/mixing.html

Accessed Date:25th July, 2005

[58] D, Mery, Programming languages. Symbian OS Guide. Symbian Develop Library.Availabe at:http://www.symbian.com/developer/techlib/v70sdocs/doc_source/DevGuides/ProgLanguages.html. Accessed date: 2nd July,2005.

[59] S. J. Young. Real Time languages, design and development, Ellis Horwood, page:44-45,1982

[60] B. Kernighan and D. Ritchie. The C Programming Language, Second Edition. Prentice Hall,pages:17-20,1988.

[61] B. Stroustrup. The C++ Programming Language, Third Edition and Special Edition. Addison-Wesley, pages:33-35,Oct. 1997.

[62] J. Adams, M. Gabrini, J. Philippe and L. Barry. An Introduction to Computer Science with Modula-2 Lexington. MA D.C. Heath & Co, 1988.

[63] E.A. Heinz. Modula-3*: An Efficiently Compliable Extension of Modula-3 for Problem-oriented Explicitly Parallel Programming. In Proc. the Joint Symposium on Parallel Processing 1993, Tokyo, Japan, pages: 269-276, May 17-19, 1993.

[64] A. Coombes and J. McDermid. Specifying Temporal Requirements for Distributed Real-time Systems in Z. Software Engineering Journal 8(5), pages:273-283, Sept. 1993.

[65] B. Lee. Specification and Design of Reactive Systems. PhD thesis, Dept

of Computer Science, University of California , Berkeley, pages:10-13, Spring, 2000.

[66] J. R. Abrial. Steam-Boiler Control Specification Problem. Dagsthul Meeting. June,1995.

[67] B. Lee. Specification and Design of Reactive Systems. PhD Thesis, Dept of Computer Science, University of California, Berkeley, page:75, Spring, 2000.

[68] W. Taha, P. Hudak and Z. Wan. Directions in Functional Programming for Real(-Time) Applications. Lecture notes in Computer Sciences, ISSU 2211,. Springer-Verlag, pages:185-203, 2001.

[69] L. Augustsson. Compliling Lazy Functional Languages. Part 2, PhD Thesis, Dept of Computer Science, Chalemers University of Echnology, Sweden, 1987.

[70] P. Landin. The Mechanical Evaluation of Expressions. The Computer Journal, 6(4), pages:308-320, Jan. 1964.

[71] A. Sloane, A. Tarnawskey and D.Verity. Haskell on Handhelds: A Port to the

Palm Platform. In Proc. IFL Prelim'03, 2003.

[72] F. Bayer. LispMe: An Implementation of Scheme on Palm Pilot. Lispme Home Page, 2000. Avalaible at: http://www.lispme.de/lispme/. Accessed date:July 3rd,2005

[73] G. Rossum and F. Drake. An Introduction to Python. Network. Theory Ltd, page:124, Sept. 2003.

[74] G. Rossum. Python Reference Manual. Corporation for National Research Initiatives, USA, pages:7-15,1997.

[75] L. Hatton. Does OO Sync with How We Think. IEEE Software 15(3), pages: 46-54, 1998.

[76] K.Hammond and G. Michaelson. Predictalbe Space Behaviour in FSM-Hume. In Proc. 2002 Intl. Workshop on Implementation of Functional languages(IFL '02).Madrid, Spain, No.2670. Springer-Verlag Lecture Notes in Computer Science, Sept. 2002.

[77] D. Harel and A. Pnuoli. On the Development of Reactive Systems. Logics and Models of Concurrent Systems, NATO ASI series 13, pages: 477-498, 1985.

[78] K. Hammond. Hume: a Bounded Time Concurrent Language. Proc. IEEE m Conf. on Electronics and Control Systems (ICECS '2K), Kaslik, Lebanon, pages: 407-411, Dec. 2000.

[79] F. Rouaix. Caml Light for PalmOS. An Implementation of the Caml Light Runtime Environment for the Palm, 1998. Available at: http://cristal.inria.fr/~rouaix/pilot/cl.html. Accessed date: July 15th,2005

[80] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney.

Region-based Memory Arrangement in Cyclone. In ACM Conf. on Berlin, June 2002. ACM Press.

[81] A. Burns and A. Wellings. Real-time Systems and Programming Languages ( Ada95, real-time Java and real-time POSIX), 3rd Edition. Addison -Wesley, page:6, 2001.

[82] A. Burns and A. Wellings. Real-time Systems and Programming Languages ( Ada95, real-time Java and real-time POSIX), 3rd Edition. Addison-Wesley, page: 7, 2001.

[83] A. Burns and A. Wellings. Real-time Systems and Programming Languages ( Ada95, real-time Java and real-time POSIX), 3rd Edition. Addison-Wesley, page:12, 2001.

[84] S.P. Jones. Concurrent and Parallel Haskell, Concurrent Haskell, Chapter 7.18 Available at:http://threading.2038bug.com/threads/http,,,www. haskell.org,ghc,docs,latest,set,concurrent-and-parallel.html. Accessed date: 23rd, Dec. 2005

# Appendix A

As the following code shows, the addition of external "C" directive to the C++ function is particularly useful because it closes the relationship between C and C++ implementations. The "C" in the extern "C" is not a language but a linkage convention. A standard header file including the macro definition of _cplusplus is enclosed in both C and C++ files as well.

**C++ file**

```
extern "C" int playsound()
                {…}
extern "C" int recordsound()
                 {…}
```

**.h standard source file**

```
#ifdef __cplusplus
#include <e32base.h>
#include <e32std.h>
extern "C" {
#endif
//functions that you going to call in C++
int playsound();
int recordsound();
#ifdef __cplusplus
}
#endif
```

This approach is only applicable for normal C++ functions but not for C++ classes. If we want to call C++ classes from C code, we also need to configure a structure in C code, which is similar to the C++ classes.

# Appendix B

The following is an example of the bridge code generated for the sound application:

```
CCALL1(ccall_cmyfun,cmyfunction,int,*,mkTup0)

CCALL1(ccall_rmyfun,rmyfunction,int,*,mkTup0)

CCALL ccalls[] = {"cmyfun", ccall_cmyfun,"rmyfun",ccall_rmyfun};

unsigned nccalls = sizeof(ccalls)/sizeof(CCALL);
```

All the C functions called in Hume code are stored as members of an array with type *CCALL*. In the initialise for the *ccalls[]* array, the first argument is the literal name of the C function and the second argument points to a related box structure. The main implementation of calling C function is performed by *CCALL1* macro instantiations.

```
#define CCALL1(hf,f,a1,p1,res)

void hf(BOX *b) {

HEAP *arg = &(b->heap[b->stack[--b->sp]]);

b->stack[inc_sp(b)] = res(b,f(p1(a1 *)(arg+1)));

}
```

In the case of the sound application, *ccall_cmyfun* is the box's address for the play sound box and *cmyfunction* is the name of the C function called by the play sound box. The third and fourth arguments are the input and output. Further details about the implementation of the *CCALL1* function can be found in the source code for the HAM. We allow different C procedures to take different numbers of arguments of different types and sizes. The maximum number of arguments we can take is four. *CCALL1* means the C function has one parameter, while the *CCALL4* function can take charge of the C function with four arguments. We also can call a C function with a vector argument such as a file or a picture. This is implemented by the *CCALLV* function.

```
#define CCALL1V(hf,f,a1,p1,res,n);

void hf(BOX *b)
```

```
{   HEAP *arg = &(b->heap[b->stack[--b->sp]]);
    b->stack[inc_sp(b)] = res(b,f(p1(a1 *)(arg+1)),n);}
```

# Appendix C

```
/**************Class CPlayerAdapter definition*********************/
#include <MdaAudioSamplePlayer.h>
class CPlayerAdapter : public CBase, public MAudioAdapter,public
MMdaAudioPlayerCallback
{ public:
/*!
  @function NewL
  @discussion Create a CPlayerAdapter object using two phase
construction
  and return a pointer to the created object
  @param aFileName the audio file
  @result pointer to new object
*/
static CPlayerAdapter* NewL(const TDesC& aFileName);
/*!
  @function NewLC
  @discussion Create a CPlayerAdapter object using two phase
construction,and return a pointer to the created object
  @param aFileName the audio file
  @result pointer to new object
*/
static CPlayerAdapter* NewLC(const TDesC& aFileName);
/*!
  @function ~CPlayerAdapter
  @discussion Destroy the object and release all memory objects
*/
~CPlayerAdapter();
```

```
public: // from MAudioAdapter
/*!
  @function PlayL
  @discussion Begin playback of the audio sample.
*/
void PlayL();
/*!
  @function StopL
  @discussion Stop playback of the audio sample.
 Note that this implementation of the virtual function does not leave.
*/
void StopL
public: // from MMdaAudioPlayerCallback
/*!
  @function MapcInitComplete
  @discussion Handle the event when initialisation of the audio player
utility is complete.
  @param aError The status of the audio sample after initialisation
  @param aDuration The duration of the sample
*/
void MapcInitComplete(TInt aError, const TTimeIntervalMicroSeconds&
aDuration);
/*!
  @function MapcInitComplete
  @discussion Handle the event when when the audio player utility
completes asynchronous playing.
  @param aError The status of playback*/
void MapcPlayComplete(TInt aError);
```

```
private:
/*!
  @function CPlayerAdapter
   @discussion Perform the first phase of two phase construction
   @param aAppUi the Ui to use
*/
CPlayerAdapter();
/*!
  @function ConstructL
  @discussion Perform the second phase construction of a CPlayerAdapter
object
*/
 void ConstructL(const TDesC& aFileName);
private:
/** The audio player utility object. */
CMdaAudioPlayerUtility* iMdaAudioPlayerUtility;

/********************class CRecordAdapter definition***************/
#include <e32std.h>
#include <MdaAudioSampleEditor.h>
#include <mda\\client\\utility.h> // for MMdaObjectStateChangeObserver
#include "audioadapter.h"
/*!
  @class CRecorderAdapter
   @discussion An instance of class CRecorderAdapter is an adapter object
for the CMdaAudioRecorderUtility class.
*/
class CRecorderAdapter : public CBase, public MAudioAdapter, public
```

```
MMdaObjectStateChangeObserver
    {
public:
/*!
  @function NewL
  @discussion Create a CRecorderAdapter object using two phase
construction, and return a pointer to the created object
  @result pointer to new object
*/
static CRecorderAdapter* NewL();
/*!
  @function NewLC
  @discussion Create a CRecorderAdapter object using two phase
construction,and return a pointer to the created object
  @result pointer to new object
*/
static CRecorderAdapter* NewLC()
/*!
  @function ~CRecorderAdapter
  @discussion Destroy the object and release all memory objects
*/
~CRecorderAdapter();
public: // from MAudioAdapter
/*!
  @function PlayL
  @discussion Begin playback of the audio sample.
*/
void PlayL();
```

```
/*!
  @function StopL
  @discussion Stop playback or recording of the audio sample.Note that
this implementation of the virtual function does not leave.
*/
void StopL();
/*!
  @function RecordL
  @discussion Record using the audio utility.
*/
void RecordL();
public: // from MMdaObjectStateChangeObserver
/*
  @function MoscoStateChangeEvent
  @discussion Handle the change of state of an audio recorder utility.
  @param aObject The audio sample object that has changed state
  @param aPreviousState The state before the change
  @aCurrentState The state after the change
  @aErrorCode if not KErrNone, that error that caused the state change
*/
void MoscoStateChangeEvent(CBase* aObject, TInt aPreviousState, TInt
aCurrentState, TInt aErrorCode);
private:
/*!
  @function CRecorderAdapter
  @discussion Perform the first phase of two phase construction
  @param aAppUi the Ui to use
*/
```

```
CRecorderAdapter();
/*!
  @function ConstructL
  @discussion Perform the second phase construction of a
CRecorderAdapter object
*/
void ConstructL();
private:
/** The audio recorder utility object. */
CMdaAudioRecorderUtility* iMdaAudioRecorderUtility;

/**************class PlayAdapter implementation*********************/
CPlayerAdapter::CPlayerAdapter() :
{ }
CPlayerAdapter* CPlayerAdapter::NewL(const TDesC& aFileName)
{
    CPlayerAdapter* self = NewLC(aFileName);
    CleanupStack::Pop();  // self
    return self;
}
CPlayerAdapter* CPlayerAdapter::NewLC(const TDesC& aFileName)
{
    CPlayerAdapter* self = new (ELeave) CPlayerAdapter();
    CleanupStack::PushL(self);
    self->ConstructL(aFileName);
    return self;
}
void CPlayerAdapter::ConstructL(const TDesC& aFileName)
```

```
{ // Create an audio player utility instance for playing sample data from
a file
  // causes MMdaAudioPlayerCallback::MapcInitComplete to be called
    iMdaAudioPlayerUtility =
  CMdaAudioPlayerUtility::NewFilePlayerL(aFileName, *this);
 }
CPlayerAdapter::~CPlayerAdapter()
{
    delete iMdaAudioPlayerUtility;
    iMdaAudioPlayerUtility = NULL;
}
// Note that this implementation of the virtual function does not leave.
void CPlayerAdapter::PlayL()
{
    iMdaAudioPlayerUtility->Play();
}
// Note that this implementation of the virtual function does not leave.
void CPlayerAdapter::StopL()
{
    iMdaAudioPlayerUtility->Stop();
    iState = EReadyToPlay;
}
// from MMdaAudioPlayerCallback
void    CPlayerAdapter::MapcInitComplete(TInt    aError,    const
TTimeIntervalMicroSeconds& /*aDuration*/)
{
    iState = aError ? ENotReady : EReadyToPlay;
}
```

```
void CPlayerAdapter::MapcPlayComplete(TInt aError)
{
    iState = aError ? ENotReady : EReadyToPlay;
     CActiveScheduler::stop();
}
/****************** class CRecorderAdapter implementation**********/
CRecorderAdapter::CRecorderAdapter()
{}
CRecorderAdapter* CRecorderAdapter::NewL()
{
    CRecorderAdapter* self = NewLC();
    CleanupStack::Pop(); // self
    return self;
}
CRecorderAdapter* CRecorderAdapter::NewLC()
{
    CRecorderAdapter* self = new (ELeave) CRecorderAdapter();
    CleanupStack::PushL(self);
    self->ConstructL();
    return self;
 }
void CRecorderAdapter::ConstructL()
{
    iMdaAudioRecorderUtility = CMdaAudioRecorderUtility::NewL(*this);
// Open an existing sample file for playback or recording
//causesMMdaObjectStateChangeObserver::MoscoStateChangeEvent      be
called
    iMdaAudioRecorderUtility->OpenFileL(KRecorderFile);
```

```
}
CRecorderAdapter::~CRecorderAdapter()
{
    delete iMdaAudioRecorderUtility;
    iMdaAudioRecorderUtility = NULL;
}
void CRecorderAdapter::StopL()
{
    iMdaAudioRecorderUtility->Stop();
     CActiveScheduler::stop();
}
void CRecorderAdapter::RecordL()
{// Record from the device microphone
iMdaAudioRecorderUtility->SetAudioDeviceMode(CMdaAudioRecorderUtilit
y::ELocal);
// Set maximum gain for recording
iMdaAudioRecorderUtility->SetGain(iMdaAudioRecorderUtility->MaxGain(
));// Delete current audio sample from beginning of file
iMdaAudioRecorderUtility->SetPosition(TTimeIntervalMicroSeconds(0));
    iMdaAudioRecorderUtility->CropL();
iMdaAudioRecorderUtility->RecordL();
     for (i=0;i<1000;i++)
     printf(".");
     StopL()    //Describe the record process
// from MMdaObjectStateChangeObserver
void CRecorderAdapter::MoscoStateChangeEvent(CBase* /*aObject*/, TInt
/*aPreviousState*/, TInt /*aCurrentState*/, TInt /*aErrorCode*/)
{ }
```

```
/****************record and play function**************************/
#include <eikmenup.h>
#include "sound.pan"
#include "sound.hrh"
#include "playeradapter.h"
#include "soundappui.h"
void soundp;
void soundr;
_LIT(aFilename, "C:\\System\\Apps\\Sound\\record.wav");
//Location record file
extern "C" int soundplay();
{
 CTrapCleanup* cleanup=CTrapCleanup::New();  //New a stack
              TRAPD(error,soundp()); //call sound play function
              if (error != KErrNone)
           {
              printf("get error %d\r\n", error);
            }
             delete cleanup; //destroy the stack
             return 0;
}
extern "C" int soundrecord();
{
        CTrapCleanup* cleanup=CTrapCleanup::New();
              TRAPD(error,soundr());
              if (error != KErrNone){
              printf("get error %d\r\n", error);
            }
```

```
            delete cleanup;
            return 0;
}
void soundp
{
  CActiveScheduler* activeScheduler = CActiveScheduler::Current();
//To find the current active scheduler
       if( activeScheduler == NULL )
     {
              activeScheduler = new (ELeave) CActiveScheduler();
              CleanupStack::PushL(activeScheduler) ;
              CActiveScheduler::Install(activeScheduler);
     }
//If there is no active scheduler, create a new one and install it
    CPlayAdapter *soundplay = CPlayAdapter::NewL(aFilename);
//Create a new sound play object
     CleanupStack::PushL(soundplay);
     CActiveScheduler::Start();
//Start the active scheduler
      soundplay.playL();
//call PlayL function to play a sound
    CleanupStack::PopAndDestroy(2);
}
void soundr{
   CActiveScheduler* activeScheduler = CActiveScheduler::Current();
          if( activeScheduler == NULL )
     {
              activeScheduler = new (ELeave) CActiveScheduler();
```

```
            CleanupStack::PushL(activeScheduler) ;
            CActiveScheduler::Install(activeScheduler);
    }
   CRecorderAdapter *soundrecord = CRecorderAdapter::NewL();
// Create a recorder object
    CleanupStack::PushL(soundrecord);
    CActiveScheduler::Start();
     soundrecord.recordL();
// Call sound record function to record a sound
 CleanupStack::PopAndDestroy(2);}
```

# Appendix D

```
--*****************************************************************
--*   digitalwatch.hume                                           *
--*   Written by: Meng Sun                                        *
--*               ms94@st-andrews.ac.uk                           *
--*                  Mar 17th 2004                                *
--* The purpose fo this script is                                 *
--*    -to show how a digital watch works                         *
--*****************************************************************

data
STATE=AlarmDisplay|TimeDisplay|TimeUpdate|AlarmUpdate|TimeDisplayOut
--four states of digital watch
data SMHD=Second|Minute|Hour|Day;              --detals of time
data AM24=AMPM|Hour24;                         --time display mode
data BUTTON=Mode|Update|Adjust|Select|TimedOut;
--four buttons of digital watch used to change the statue
data CHIME=Daily|Hourly|Both;                   --chime mode
data DAY=Mon|Tue|Wed|Thu|Fri|Sat|Sun;
type Int=int 32;
type DAYTIME=(Int,Int,Int,DAY);  --second minute hour (ms)time tuple
type Char=char;
type ALARMTIME=(Int,Int);

box button            --button box used to connect input with button state
in(c::Char)
out(a::BUTTON)
match
('a')->(Adjust)|
('u')->(Update)|
('s')->(Select)|
('m')->(Mode) |
   _ -> *
handle
Timeout () ->TimedOut;
stream input from "std_in" within 5s raise Timeout;
wire button
(input)(watch.button);
```

```
box watch      --watch box used to control the statue change of watch
in (watch_state' :: STATE, button :: BUTTON, select_smhd :: SMHD, am ::
AM24, watch_chime :: CHIME, time' :: DAYTIME, alarmtime::ALARMTIME)
out(watch_state :: STATE, select_smhd1 :: SMHD, am1 :: AM24, watch_chime1
:: CHIME, time1 :: DAYTIME,alarmtime1::ALARMTIME)
match
(s, TimedOut, smhd, am, chime, time', alarmtime)->
(TimeDisplayOut, smhd, am, chime, time', alarmtime)|
(TimeDisplayOut,Mode,smhd,am,chime,time',alarmtime)->
(TimeDisplay,smhd,am,chime,time',alarmtime)|
(TimeDisplay, Mode, smhd, am, chime, time', alarmtime)->
(AlarmDisplay, smhd, am, chime, time', alarmtime)|
(AlarmDisplay, Mode, smhd, am, chime, time', alarmtime)->
(TimeDisplay, smhd, am, chime, time', alarmtime)|
     --mode button to switch between TimeDisplay and AlarmDisplay

(TimeDisplay, Adjust, smhd, AMPM, chime, time', alarmtime)->
(TimeDisplay, smhd, Hour24, chime, time', alarmtime)|
(TimeDisplay, Adjust, smhd, Hour24, chime, time', alarmtime)->
(TimeDisplay, smhd, AMPM, chime, time', alarmtime)|
     --in TimeDisplay use Adjust button to change to timedisplay
mode(AM/PM or 24Hours)

(AlarmDisplay, Adjust, smhd, am, Daily, time', alarmtime)->
(AlarmDisplay, smhd, am, Hourly, time', alarmtime)|
(AlarmDisplay, Adjust, smhd, am, Hourly, time', alarmtime)->
(AlarmDisplay, smhd, am, Both, time', alarmtime)|
(AlarmDisplay, Adjust, smhd, am, Both, time', alarmtime)->
(AlarmDisplay, smhd, am, Daily, time', alarmtime)|
--in AlarmDisplay use Adjust button to choose to chime by daily,hourly
or Both

(TimeDisplay, Update, smhd, am, chime, time', alarmtime)->
(TimeUpdate, Second, am, chime, time', alarmtime)|
(TimeUpdate, Mode, smhd, am, chime, time', alarmtime)->
(TimeDisplay, smhd, am, chime, time', alarmtime)|
--if now is TimeDisplay,update botton used to select updating time or
not,Mode, button is used to return to TimeDisplay
```

```
(TimeUpdate, Select, Second, am, chime, time', alarmtime)->
(TimeUpdate, Minute, am, chime, time', alarmtime)|
(TimeUpdate, Select, Minute, am, chime, time', alarmtime)->
(TimeUpdate, Hour, am, chime, time', alarmtime)|
(TimeUpdate, Select, Hour, am, chime, time', alarmtime)->
(TimeUpdate, Day, am, chime, time', alarmtime)|
(TimeUpdate, Select, Day, am, chime, time', alarmtime)->
(TimeUpdate, Second, am, chime, time', alarmtime)|
             --use Select button to select which bit should be adjust

(TimeUpdate, Adjust, Second,am, chime,(sec, min, hour,day),alarmtime)->
(TimeUpdate, Second, am, chime, (0, min, hour,day), alarmtime)|
(TimeUpdate, Adjust, Minute,am,chime,(sec, min, hour,day), alarmtime)->
if min<59 then (TimeUpdate, Minute, am, chime, (sec, min+1,hour,day),
alarmtime)
 else (TimeUpdate, Minute, am, chime, (sec, 0, hour,day), alarmtime)|
(TimeUpdate, Adjust, Hour, am, chime, (sec, min, hour,day), alarmtime)->
if hour<24 then (TimeUpdate, Hour, am, chime, (sec, min, hour+1,day),
alarmtime)
else (TimeUpdate, Hour, am, chime, (sec, min, 1,day), alarmtime)|
--Adjust to advance the number

(TimeUpdate, Adjust, Day, am, chime,(sec, min,hour,Mon),alarmtime)->
(TimeUpdate, Day, am, chime,(sec, min, hour, Tue),alarmtime)|
(TimeUpdate, Adjust, Day, am, chime,(sec, min, hour,Tue),alarmtime)->
(TimeUpdate, Day, am, chime,(sec, min, hour, Wed),alarmtime)|
(TimeUpdate, Adjust, Day, am, chime,(sec, min, hour,Wed),alarmtime)->
(TimeUpdate, Day, am, chime,(sec, min, hour, Thu),alarmtime)|
(TimeUpdate, Adjust, Day, am, chime,(sec, min, hour,Thu),alarmtime)->
(TimeUpdate, Day, am, chime,(sec, min, hour, Fri),alarmtime)|
(TimeUpdate, Adjust, Day, am, chime,(sec, min, hour,Fri),alarmtime)->
(TimeUpdate, Day, am, chime,(sec, min, hour, Sat),alarmtime)|
(TimeUpdate, Adjust, Day, am, chime,(sec, min, hour,Sat),alarmtime)->
(TimeUpdate, Day, am, chime,(sec, min, hour, Sun),alarmtime)|
(TimeUpdate, Adjust, Day, am, chime,(sec, min, hour,Sun),alarmtime)->
(TimeUpdate, Day, am, chime,(sec, min, hour, Mon),alarmtime)|
--advance the day

--the following is alarmupdate,the same to timeupdate
(AlarmDisplay, Update, smhd, am, chime, time', alarmtime) ->
(AlarmUpdate, Minute, am, chime, time', alarmtime)|
```

```
(AlarmUpdate, Mode, smhd, am, chime, time', alarmtime)->
(AlarmDisplay, smhd, am, chime, time', alarmtime)|

(AlarmUpdate, Select, Hour, am, chime, time', alarmtime)->
(AlarmUpdate, Minute, am, chime, time', alarmtime)|
(AlarmUpdate, Select, Minute, am, chime, time', alarmtime)->
(AlarmUpdate, Hour, am, chime, time', alarmtime)|


(AlarmUpdate, Adjust, Minute, am, chime, time', (hour,min))->
if min<59 then (AlarmUpdate, Minute, am, chime, time',(hour,min+1))
else  (AlarmUpdate, Minute, am, chime, time', (hour,0))|
(AlarmUpdate, Adjust, Hour, am, chime, time', (hour,min))->
if hour<24 then (AlarmUpdate, Hour, am, chime, time', (hour+1,min))
else (AlarmUpdate, Hour, am, chime, time',(1,min))|
(s,_,h,ampm,chime,t,alarmtime) -> (s,h,ampm,chime,t,alarmtime);

wire watch
(fanout.stateA initially TimeDisplay, button.a, fanout.smhd1A initially
Second, fanout.pmam1A initially AMPM, fanout.chime1A initially Both ,
fanout.time1A initially (54,23,2,Mon), fanout.alarmtime1A initially
(0,0))
(fanout.state, fanout.smhd, fanout.pmam, fanout.chime, fanout.time',
fanout.alarmtime);

box fanout                  --fanout the one input value to two way
in (state :: STATE, smhd :: SMHD, pmam :: AM24, chime :: CHIME, time'
    :: DAYTIME, alarmtime :: ALARMTIME)
out(stateA :: STATE, smhd1A :: SMHD, pmam1A :: AM24, chime1A :: CHIME,
   time1A :: DAYTIME, alarmtime1A :: ALARMTIME, stateB :: STATE, smhd1B
   :: SMHD, pmam1B :: AM24, chime1B :: CHIME, time1B :: DAYTIME,
   alarmtime1B :: ALARMTIME)
match
(x, y, z, e, h, g)->(x, y, z, e, h, g, x, y, z, e, h, g);

wire fanout
(watch.watch_state, watch.select_smhd1, watch.am1, watch.watch_chime1,
watch.time1, watch.alarmtime1)
(watch.watch_state', watch.select_smhd, watch.am, watch.watch_chime,
watch.time', watch.alarmtime, display.state, display.smhd,
display.pmam, display.chime, display.time', display.alarmtime);
```

```
--definite some keywords which is easy to understand in display screen
watchsmhd Second="SECOND";
watchsmhd Minute="MINUTE";
watchsmhd Hour="HOUR";
watchsmhd Day="DAY";

watchchime Daily="DAILY";
watchchime Hourly="HOURLY";
watchchime Both="BOTH";

watchday Mon="Monday";
watchday Tue="Tuesday";
watchday Wed="Wednesday";
watchday Thu="Thursday";
watchday Fri="Friday";
watchday Sat="Saturday";
watchday Sun="Sunday";
h=12;

box display                                    --display the state of watch
in (state :: STATE, smhd :: SMHD, pmam :: AM24, chime :: CHIME, time'
    :: DAYTIME, alarmtime :: ALARMTIME)
out(output':: string)

match
(TimeDisplayOut,smhd,pmam,chime,(sec,min,hour,day),alarmtime)->
"timeout:please input Mode\n"|

(TimeDisplay, smhd, AMPM, chime, (sec, min, hour,day), alarmti11)->
if hour>12 then "Time Display as:AMPM"++"\n"++"Alarm
chime:"++watchchime chime++"\n"++"Time(Hour/Min/Sec):"++(hour mod 12)
as string++":"++min as string++":"++sec as string++" PM"++"\n"++"Today
is:"++watchday day++"\n"
else "Time Display as:AMPM"++"\n"++"Alarm chime:"++watchchime
chime++"\n"++"Time(Hour/Min/Sec):"++hour as string++":"++min as
string++":"++sec as string++" AM"++"\n"++"Today is:"++watchday
day++"\n"|

(TimeDisplay, smhd, Hour24, chime, (sec, min, hour,day), alarmtime)->
"Time Display as:24Hours"++"\n"++"Alarm chime:"++watchchime
chime++"\n"++"Time(Hour/Min/Sec):"++hour as string++":"++min as
string++":"++sec as string++"\n"++"Today is:"++watchday day++"\n"|
```

```
(AlarmDisplay, smhd, AMPM, chime, time', (hour,min))->
if hour>12 then "Alarm Display as AMPM"++"\n"++"Alarm
chime:"++watchchime chime++"\n"++"Time(Hour/Min):"++(hour mod 12 ) as
string++":"++min as string++"PM"++"\n"
else "Alarm Display as AMPM"++"\n"++"Alarm chime:"++watchchime
chime++"\n"++"Time(Hour/Min):"++hour as string++":"++min as string++"
AM"++"\n"|

(AlarmDisplay, smhd, Hour24, chime, time', (hour,min))->
"Alarm Display as 24Hours"++"\n"++"Alarm chime:"++watchchime
chime++"\n"++"Time(Hour/Min):"++hour as string++":"++min as
string++"\n"|

(TimeUpdate, smhd, AMPM, chime, (sec, min, hour,day), alarmtime)->
if hour>12 then "Watch is updating time as:AMPM"++"\n"++watchsmhd
smhd++"was flashing."++ "\n"++"Time(Hour/Min/Sec):"++(hour-12) as
string++":"++min as string++":"++sec as string++" PM"++"\n"++"Today
is:"++watchday day++"\n"
else "Watch is updating time as:AMPM"++"\n"++watchsmhd smhd++"was
flashing."++ "\n"++"Time(Hour/Min/Sec):"++hour as string++":"++min as
string++":"++sec as string++" AM"++"\n"++"Today is:"++watchday
day++"\n"|

(TimeUpdate, smhd, Hour24, chime, (sec, min, hour,day), alarmtime)->
"Watch is updating time as:24Hours"++"\n"++watchsmhd smhd++"was
flashing."++ "\n"++"Time(Hour/Min/Sec):"++hour as string++":"++min as
string++":"++sec as string++"\n"++"Today is:"++watchday day++"\n"|

(AlarmUpdate, smhd, AMPM, chime, time', (hour,min))->
if hour>12 then "Watch is updating alarm as:AMPM"++"\n"++watchsmhd
smhd++"was flashing."++"\n"++"Time(Hour/Min):"++(hour-12) as
string++":"++min as string++"PM"++"\n"
else "Watch is updating alarm as:AMPM"++"\n"++watchsmhd smhd++"was
flashing."++"\n"++"Time(Hour/Min):"++hour as string++":"++min as
string++" AM"++"\n"|

(AlarmUpdate, smhd, Hour24, chime, time', (hour,min))->
"Watch is updating alarm as:24Hours"++"\n"++watchsmhd smhd++"was
flashing."++"\n"++"Time(Hour/Min):"++hour as string++":"++min as
string++"\n";
```

```
stream output to "std_out";

wire display
(fanout.stateB, fanout.smhd1B, fanout.pmam1B, fanout.chime1B,
fanout.time1B, fanout.alarmtime1B)
(output);
```

# Appendix E

```
--*****************************************************************
--*   realroad.hume   Version 0.0                                  *
--*   Written by: Meng Sun                                         *
--*               ms94@st-andrews.ac.uk                            *
--*                 4th Mar 2004                                   *
--*                                                                *
--* The purpose of this script is                                  *
--*    -simulation of a railroad crash                             *
--*****************************************************************

data BRIDGE=Onbridge|Offbridge;
data ARRLEA=Arr|Leave|Notarr|Notleave;
--Arrive/Leave/Notarrival/Notleave=1/2/3/4data
STATESTRAIN=Agbr|Agbg|Arbg;
data CROSS=Agbg|Arbg|Agbr;
type Int=int 32;
type Float=float 32;
DIS=2;
SPEED=1;SPEE=2;

box speed
in(a', a''::Int)
out(ao, a, b::Int)
match
 (*,a) -> (a,a,a+1)
|(a,*) -> (a,a,a+1);
stream input from "std_in";

wire speed
(input,speed.ao)(speed.a'',trainA.speed,trainB.speed);

box control
in(state::CROSS,sinA::Int,sinB::Int,motivationA::ARRLEA,motivationB:
:ARRLEA)
out(state'::CROSS,signalA::Int,signalB::Int)

match
```

```
(Agbr,0,1,Leave,Notarr)->(Agbg,0,0)|    -- Agbr-Agbg
(Agbg,0,0,Notarr,Arr)  ->(Arbg,1,0)|    -- Agbg-Arbg
(Arbg,1,0,Arr,Leave)   ->(Agbr,0,1)|    -- Arbg-Agbr
(Agbr,0,1,Leave,Arr)   ->(Arbg,1,0)|    -- Agbr-Arbg
(Arbg,1,0,Notarr,Leave)->(Agbg,0,0)|    -- Arbg-Agbg
(Agbg,0,0,Arr,_)       ->(Agbr,0,1)|
(Agbg,0,0,Notleave,_)  ->(Agbr,0,1)|
(Agbg,0,0,_,Notleave)  ->(Agbr,0,1)|    -- Agbg-Agbr
(st,s1,s2,_,_)->(st,s1,s2);

wire control
(fanout.state1 initially Agbg,fanout2.sinA1 initially 0,fanout2.sinB1
initially 0,fanout.motivationA1 initially Notarr,fanout.motivationB1
initially Notarr)
(fanout.state,fanout2.sinA,fanout2.sinB);

box fanout2
in (sinA::Int,sinB::Int)
out(sinA1::Int,sinB1::Int,sinA2::Int,sinB2::Int)

match
(x1,x2)->(x1,x2,x1,x2);

wire fanout2
(control.signalA,control.signalB)
(control.sinA,control.sinB,trainA.signal,trainB.signal);

box trainA
in (bridgestate::BRIDGE,location::Int,signal::Int,speed::Int)
out (bridgestate'::BRIDGE,location'::Float,motivation::ARRLEA)

match
(Offbridge,location,1,speed)->
if ((location+speed)>=3&&(location+speed)<=9) then (Offbridge,3,Arr)
else (Offbridge,((location+speed) mod 20),Notarr)|
(Offbridge,location,0,speed)->
if((location+speed)>=3&&(location+speed)<=9) then (Onbridge,3,Arr)
else (Offbridge,((location+speed) mod 20),Notarr)|
(Onbridge,location,1,speed)->(Offbridge,3,Arr)|
(Onbridge,location,0,speed)->
if (location+speed)>=9 then (Offbridge,((location+speed) mod 20),Leave)
else (Onbridge,((location+speed) mod 20),Notleave);
```

```
wire trainA
(fanout.bridgeA1 initially Offbridge,fanout.locationA1 initially
0,fanout2.sinA2,speed.a)
(fanout.bridgeA,fanout.locationA,fanout.motivationA);

replicate trainA as trainB;

wire trainB
(fanout.bridgeB1 initially Offbridge,fanout.locationB1 initially
0,fanout2.sinB2,speed.b)
(fanout.bridgeB,fanout.locationB,fanout.motivationB);

box fanout
in(state::CROSS,bridgeA::BRIDGE,bridgeB::BRIDGE,locationA::Int,locat
ionB::Int,motivationA::ARRLEA,motivationB::ARRLEA)
out(state1::CROSS,bridgeA1::BRIDGE,bridgeB1::BRIDGE,locationA1::Int,
locationB1::Int,motivationA1::ARRLEA,motivationB1::ARRLEA,state2::CR
OSS,bridgeA2::BRIDGE,bridgeB2::BRIDGE,locationA2::Int,locationB2::In
t,motivationA2::ARRLEA,motivationB2::ARRLEA)

match
(x1,x2,x3,x4,x5,x6,x7)->(x1,x2,x3,x4,x5,x6,x7,x1,x2,x3,x4,x5,x6,x7);

wire fanout
(control.state',trainA.bridgestate',trainB.bridgestate',trainA.locat
ion',trainB.location',trainA.motivation,trainB.motivation)
(control.state,trainA.bridgestate,trainB.bridgestate,trainA.location
,trainB.location,control.motivationA,control.motivationB,log.state,l
og.bridgeA,log.bridgeB,log.locationA,log.locationB,log.motivationA,l
og.motivationB);

box log
in(state::CROSS,bridgeA::BRIDGE,bridgeB::BRIDGE,locationA::Int,locat
ionB::Int,motivationA::ARRLEA,motivationB::ARRLEA)
out(log::string)

fair
(x1,x2,x3,x4,x5,x6,x7)->"The traffic state is: "++x1 as
string++"\n"++"Train A:"++x2 as string++"**Location"++x4 as
string++"**"++x6 as string++"\n"++"Train B:"++x3 as
string++"**Location"++x5 as string++"**"++x7 as string++"\n";
```

```
stream output to "std_out";

wire log
(fanout.state2,fanout.bridgeA2,fanout.bridgeB2,fanout.locationA2,fan
out.locationB2,fanout.motivationA2,fanout.motivationB2)(output);
```

# Appendix F

```
--*****************************************************************
--*   steamboiler.hume                                             *
--*   Written by: Meng Sun                                          *
--*               ms94@st-andrews.ac.uk                             *
--*               April 20 2004                                       *
--*****************************************************************
type MSG=Transmission_fail
        |Steam_waitting
        |Phy_ready;
type
FAIL=(Int,Int,Int);--(Level_fail,Physical_unit_fail,Steam_outcome)
type PHY=R|NR;                              --ready or not ready
type MODE=Initi|Normal|Emergency|Degraded|Rescue|Adjust|Ready;
type STATUE=On|Off;
type TF=Nofail|Fail;
type PUMP=Pump1|Pump2|Pump3|Pump4;
type READY=Ready|Stop;
type PUMPCASE=Case1|Case2|Case3|Case4|Case5|Case6;
data Int=int 32;
C=500;
N1=20;
N2=480;
M1=10;
M2=490;
U1=5;
U2=5;
V=20;
T=5;
A=20;

box steamboiler
in (mode:: MODE,trans::MSG,f::FAIL,v::Int)
out(mode'::MODE,v::Int)
--v:the quanity of steam out,n: the level of water
--f::FAIL(Int,Int,Int) 0:repaired,1:fail
match
(_,Transmissoin_fail,f,v)->(Emergency,*)|
(_,_,(1,1,_),_)->(Emergency,*)|
```

```
(Emergency,_,_,_)-> (Emergency,*) |
(Initi,t,f,v)->if v!=0 then (Emergency,*)|
(Ready,t,f,v)->if v!=0 then (Emergency,*)|
(Adjust,t,f,v)->if v!=0 then (Emergency,*)|
(Initi,Steam_waitting,f,0)->(Adjust,0)|
(Adjust,Phy_ready,f,0)->(Ready,0)|
(Adjust,Phy_ready,f,0)->(Ready,0)|
(Ready,_,(0,1,0),v)->(Degreded,v)|
(Ready,_,(0,0,0),v)->(Normal,v)|
(Normal,t,(1,0,0),v)-> (Rescue,v)|
(Normal,t,(0,1,0),v)->(Degreded,v)|
(Degraded,t,(0,0,0),v)->(Normal,v)|
(Degraded,t,(1,1,0),v)->(Rescue,v)|
(Rescue,t,(0,1,0),v)->(Degreded,v)|
(Rescue,t,(0,0,0),v)->(Normal,v)|
(Rescue,t,(_,_,1),v)->(Emergency,*)|
(m,_,_,_,v)->(m,v);                    --any other situations
wire steamboiler
(simulation_waterlevel.mode',simulation.phy,simulation.f,simulation_
waterlevel.v')(pump.mode,water_level.v)

box pump
in(mode::MODE,p::Int,qa1::Int,qa2::Int)
out(mode'::MODE,p'::Int,p1,value::Int,qa1'::Int,qa2'::Int)
match
(Normal,p,qa1,qa2)->if(qa2>=M2||qa1<=M1) then (Emergency,*,*,*,*,*)|
(Degraded,p,qa1,qa2)->if(qa2>=M2||qa1<=M1) then(Emergency,*,*,*,*,*)|
(Rescue,p,qa1,qa2)->if (qa2>=M2||qa1<=M1) then (Emergency,*,*,*,*,*)|
(Adjust,p,qa1,qa2)->if qa1>=N1&&qa2<=N2 then (Adjust,p,p,0,qa1,qa2)
                else if qa2>=N2 then (Adjust,p,p,1,qa1,qa2)|
(mode,0,qa1,qa2)->if qa2<N1 then (mode,1,1,0,qa1,qa2)
             else if qa1<N1&&qa2>N1&&qa2<N2 then (mode,1,1,0,qa1,qa2)|
(mode,1,qa1,qa2)->if qa1>N2 then (mode,0,0,0,qa1,qa2)
             else if qa1>N1&&qa1<N2&&qa2>N2 then (mode,1,1,0,qa1,qa2)|
(mode,p,qa1,qa2)->(mode,p,p,0,qa1,qa2);
wire pump
(steamboiler.mode',fanout.p1,fanout.qa11,fanout,qa21)
(water_level.s,fanout.p,water_level.pump,fanout.value,water_level.qa
1,water_level.qa2)

box water_level
in(s::MODE,pump::Int,value::Int,v::Int,q::Int,va1::Int,va2::Int,qa1:
```

```
   :Int,qa2::Int)
out(s'::MODE,v'::Int,q'::Int,va1'::Int,va2'::Int,qa1'::Int,qa2::Int)
match
(Initi,p,value,v,q,va1,va2,qa1,qa2)->
(Normal,v,q-v*A,v,v,q-v*A,q-v*A)|
(Adjust,p,value,v,q,va1,va2,qa1,qa2)->
(Normal,v,q-v*t+P*p-v*A,v,v,q-v*t+P*p-v*A,q-v*t+P*p-v*A)|
(Normal,p,value,v,q,va1,va2,qa1,qa2)->
(Normal,v,q-v*t+P*p-v*A,v,v,q-v*t+P*p-v*A,q-v*t+P*p-v*A)|
(Rescue,p,value,v,q,va1,va2,qa1,qa2)->
(Rescue,va1-U2*t,qa1-va2*t-0.5*U1*t*t-P*p,va1-U2*t,va2+U1*t,qa1-va2*
t-0.5*U1*t*t-P*p,qa2-va1*t+0.5*U2*t*t-P*p)|
(Degraded,p,value,v,q,va1,va2,qa1,qa2)->
(Degraded,v,q-v*t,v,v,q-v*t,qa2-v*t+P)|
(mode,p,value,v,q,va1,va2,qa1,qa2)->
(mode,v,q,va1,va2,qa1,qa2);
wire water_level
(pump.mode',pump.p1,fanout.value1,steamboiler.v',simulation_waterlev
el.q',fanout.va11,fanout.va21,pump.qa1',pump.qa2')
(fanout.mode,fanout.v,fanout.q,fanout.va1,fanout.va2,fanout.qa1,fano
ut.qa2)

box fanout
in(mode::MODE,pump::Int,value::Int,v::Int,q::Int,va1::Int,va2::Int,q
a1::Int,qa2::Int)
out(mode1::MODE,pump1::Int,value1::Int,v1::Int,q1::Int,va11::Int,va2
1::Int,qa11::Int,qa21::Int,mode2::MODE,pump2::Int,value2::Int,v2::In
t,q2::Int,va12::Int,va22::Int,qa12::Int,qa22::Int)
match
(x1,x2,x3,x4,x5,x6,x7,x8,x9)->
(x1,x2,x3,x4,x5,x6,x7,x8,x9,x1,x2,x3,x4,x5,x6,x7,x8,x9);
wire fanout
(water_level.s',pump.p',pump.value,water_level.v',water_level.q',wat
er_level,va1',water_level.va2',water_level.qa1',water_level.qa2')
(simulation_level.mode,pump.p,water_level.value,simulation_waterleve
l.v,simulation_waterlevel.q,water_level.va1,water_level.va2,pump.pa1
,pump.pa2,log.mode2,log.pump2,log.value2,log.v2,log.q2,log.va12,log.
va22,log.qa12,log.qa22)

box simulation_waterlevel
in(mode::MODE,q:Int,v::Int)
out(mode'::MODE,q'::Int,v'::Int)
```

```
(m,q,5)->(m,q-5,1)|
(Initi,q,v)->(Initi,q,0)|
(Adjust,q,v)->(Adjust,q,0)|
(Ready,q,v)->(Ready,q,0)|
(m,q,v)->(m,q-5,v+1);                --
wire simulaten_waterlevel
(fanout.mode1,fanout.q1,fanout.v1)
(steamboiler.mode,water_level.q,steamboiler.v)

box log
in(mode::MODE,pump::Int,value::Int,v::Int,q::Int,va1::Int,va2::Int,q
a1::Int,qa2::Int)
out(output::string)
match
(Emergency,_,_,_,_,_,_,_,_)->
"Fatal error! program stop!"
(Initial,_,_,_,_,_,_,_,_)->
"Initializing...Please input s if steam is waiting)"++"\n"|
(Adjust,_,value,_,_,_,_,_,_)->
"Waitting Physical unit ready...Please input r,if program is
ready)"++"\n"++"value is: "++value as string++"\n"++"Pump: "++p as
string++"\n"|
(Ready,_,_,_,_,_,_,_,_)->
"Program is ready now,input the working fail:0 is normal,1 physical,2
waterlevel,3 steam"++"\n"|
(Normal,p,_,v,q,va1,va2,qa1,qa2)->
"Mode is normal"++"\n"++"WaterLevel is:  "++q as string++"steam is:
"++v as string++"\n"++"Pump:"++p as string++"\n"|
(Degraded,_,value,v,_,_,_,qa1,qa2)->
"Mode is Degraded,physical unit error!"++"\n"++"The caculated water
level is Min: "++qa1 as string++"   Max: "++qa2 as string++"\n"++"stream
is:"++v as string++"\n|"
(Rescue,p_,_,_va1,va2,qa1,qa2))->
"Mode is Rescue,water_level error!"++"\n"++"The caculated water level
is Min: "++qa1 as string++"   Max: "++qa2 as string++"\n"++"stream
is:"++va1 as string++":"++va1 as string++"\n|"
stream output to "std_out";
wire log
(fanout.mode2,fanout.pump2,fanout.value2,fanout.v2,fanout.q2,fanout.
va12,fanout.va22,fanout.qa12,fanout.qa22)
(output)
```