4º SBRC
RECIFE - 24 A 26 DE MARÇO 86

Applicative Specification of Computer Networks and Protocols

Silvio Lemos Meira
Departamento de Informática
Universidade Federal de Pernambuco
50000 Recife, PE, BRAZIL.

Resumo

Este trabalho demonstra o uso de linguagens de programação puramente funcionais, de alta ordem, na especificação de redes de computadores e seus protocolos. Como exemplo desta aplicação, nós especificamos um "Cambridge Ring" e um de seus protocolos, e indicamos como tal especificação pode ser usada para provar propriedades da rede, e também como um simulador da mesma.

Abstract

We consider the specification of computer networks and protocols using higher-order, purely functional programming languages. As an example of such application, we specify a basic Cambridge Ring and one of its protocols, and indicate how that specification could be used to prove properties of the system, and also how to modify it to be a simulator of the network.

1. Applicative Languages.

For some time now, we are experiencing what is commonly know as the "software crisis". This has to do with the ever increasing cost of software, together with the ever decreasing cost of hardware. Applicative programming languages have been heralded as offering a solution to the problems that cause the software crisis to exist, but that is not the whole truth.

The software crisis is not a problem caused by lacking of good theories and practices of running software projects only. It exists basically because the programming languages used in those projects are not appropriate for doing so. They are (normally) not modular, they have side-effects and other sorts of dirty tricks, and they do not posses the necessary abstraction mechanisms to think about the problem being solved, instead of the implementation details of the solution. These are just a few problems we can name here and now.

The current aim of the class of languages we use in this paper is not to solve the software crisis once and for all, but at least a great part of it related to the project's programming language *per se*.

In this paper, we will use a "style" of programming that resembles the languages KRC and MIRANDA [Tur82k, Tur84m], developed by Professor David Turner of the University of Kent at Canterbury, UK. MIRANDA is the latest in a family of purely functional languages developed at Kent, and it incorporates such concepts as modules, algebraic types, abstract data types and separate compilation. We do not make use of the MIRANDA itself in this paper, using the "style" of programming encouraged by the language, the point being not to clutter the presentation with language details unnecessary to our objective in this paper.

The point of using this style of programming here is to try and show that functional languages are general programming languages, and not just another theoretical toy. They have been used in situations as far apart as interpreters, compilers, editors and programming proving tools. Network and protocol specification is just another application, where we can put to good use the executable specification concept [Tur84m], so that we can go from a prototype to a runnable simulator in a period of time far smaller than it could be achieved by using other means.

The language used here does not intend to be as abstract an specification tool as the ones normally used for such job, but on the other hand it offers the capability of the user being able to run her "specification" at various levels of definedness.

A word of warning is that, from a purely functional point of view, the world must be reated as if all processes were functions. We skip the question of whether the universe is functional or not, which is not proper in this forum. However, we must say that some readers might find that their nórmal view of communicating processes might be in just a little conflict with the views presented in this work. Furthermore, we consider our view as an exercise only, and not as the final –or only– truth about the matter.

2. The Cambridge Ring.

The Cambridge Ring is a local area network developed at the University of Cambridge Computer Laboratory, in the late 70's, and it has been widely used both as research instrument and in practical applications since then.

Hosts are connected to the ring via a station and repeater setup, where the repeater generates the ring signals and allows the stations to alter the data flowing through the ring. At the physical level* the ring works on the empty slot principle. A slot is a group of 38 bits, with the format shown below:

```
|1|e|m| dest | sour | data | data |resp|p |
```

where the first bit, always set to 1, is the start bit, the next is the full/empty bit, and the third is the monitor bit. After that, there are 8 bits in each of the dest(ination) and sour(ce) fields, and also in the two data fields. The resp(onse) field of the packet has two bits, and the last bit, p(arity), carries the parity information for the packet.

There is a fixed number of slots circulating around the ring. Stations that want to make a transmission wait for an empty slot to arrive, mark it full and set the address and data fields. The slot circulates to its destination (if it exists) which accepts the data or not, and marks the response bits accordingly. The slot then circulates to the sender, which marks the slot as empty and reads the response bits.

The ring stations have a selection register, which tunes the station to particular sources or none. A station may listen to all stations by writing 255 in its selection register, or listen to no stations by writing 0.

Using the two response bits in the slot, the four possible responses are:

ignored:	no station with the destinadion address is active;
busy:	the host attached to the destination station hasn't yet read the last slot from the station;
unselected:	the destination station did not receive from the source;
accepted:	the destination stattion received the packet.

* This terminology is used by the Cambridge group, and it is not necessarily the same as ISO's OSI level 1, i.e., the OSI physical level.

The monitor and parity bits are used to detect error conditions: the first allows the monitor, which is a special station, to detect when a transmitter fails to empty a returning slot. The second can be used to determine the links of the ring on which transmission error are occurring.

At the higher level, some of protocols used are the Basic Block Protocol (BBP), Byte Stream Protocol (BSP) and Single Shot Protocol (SSP), which are described in the ring literature (see [NeH82]).

3. Specifying a Ring.

Our first approach to the specification of a Cambridge Ring is to write, for a generic station, a definition that specifies the behaviour of that station.

A Cambridge Ring node can be described via its attributes and behaviour, and a node may comprise several levels, each of which corresponding to virtual or physical levels, in OSI nomenclature. To start with the simplest possibility, let us write the specification of a "bare" ring with three stations, at physical level only. In a real situation, of course, one of the three would be the monitor, but we do not worry about that at this stage.

The simplest specification we could possibly write is a set of mutually recursive definitions, using a function definition for each station. Noting that [] is the empty list, and that : is the same as the **cons** constructor of LISP, that works out as

```
sta1 []    = []
sta1 (a:x) = sta2 x
sta2 []    = []
sta2 (a:x) = sta3 x
sta3 []    = []
sta3 (a:x) = sta1 x
```

At this level of specification, the list represents the traffic in the channel, each of the list elements is a slot and every station "receives" a slot every time the slot is passing by.

Still using the same idea, a better specification is to use a single definition of type

```
ring :: [station] -> [slot] -> null
```

where :: means "is of type", [x] is "list of elements of type x" and null is the null type. Now, a single definition can be made to care for all the stations in the ring, written as

```
ring (a:x) []    = []
ring (a:x) (b:y) = ring (x ++ [a])  y
```

where the stations are simulated by the first parameter (a list of stations), from which we take the head and always append (++) it to the tail. This specification copes even with a single station ring!

In this basic specification, though, all we have described is the "idea" of the ring, i.e., that stations process information in a "ring" fashion. Now we need to materialize such concepts as slots, the station states, addressing modes, etc.

If in the preceding specification a station is given by its name only, we now give its specification as a tuple of objects, which includes its name, as

```
station == (mad, lad, status)
```

where == makes **station** a "type synonym" of the tuple to the right of the sign, which can be used as records in Pascal, **mad** and **lad** are the station's own address and the listening address, respectively, and **status** is the current (unspecified) status of the station. At the same time, consider a ring slot defined by

```
slot    == (start,      $$ 1 bit, always set to 1
            empty,      $$ 1 bit, true if the slot is empty
            monit,      $$ 1 bit, the monitor station mark bit
            dest ,      $$ 8 bits, the destination station
            sour ,      $$ 8 bits, the transmitter station
            data ,      $$ 16 bits, data
            resp ,      $$ 2 bits, response, see Section 2
            par         $$ 1 bit, parity
           )
```

where everything after **$$** is a comment and we omit the full definition of all the data types involved. For our purposes now, it is enough to know their intended meanings.

Next, we retype definition of **ring** to

```
ring :: [station] -> slot -> null
```

where we now use the one slot per ring configuration, as it can be seen. But note that, as **[station]** is only a list of attributes of ring stations, it is not easy to see how the stations could interact over the ring. In order to to just that, first we add to **station** a new item, which will be called **traffic**, and here we assume its type to be

```
traffic :: [slot]
```

without loss of generality, to avoid having to specify packet assembling and desassembling. As of now, we are also abstracting the existence of higher levels of hardware and protocols, to simplify the development.

Before we go on to the next set of equations, it is necessary to say that so far we have been writing unconditional equations, or conditional equations using pattern matching (see, for example, **stal**). We can also write "guarded equations", where we state, as if we had an "if", the conditions under which the preceding expression is to be evaluated. This can be better examplified in the definition of the factorial function

```
factorial n = n * factorial (n-1), n ~= 0
            = 1
```

where * and ~ mean times and not, respectively, which has the same meaning as

```
factorial 0 = 1
factorial n = n * factorial (n-1).
```

Having said so, our specification is still yelding the object of the null type as final result, and we can rewrite the definition of ring to be

```
$$ st:lsst is a pattern matcher for the list of stations,
$$ there is only one slot and
$$ the attributes of slot and st are accessed in the
$$ usual Pascal dot notation
ring (st:lsst) slot = ring newstlist slot,
                        slot.empty & st.traffic = [] \/
                        ~slot.empty & ~matchadd slot st
                    = ring newstlist (hd st.traffic),
                        slot.empty & st.traffic ~= []
                    = ring newstlist (busy slot),
                        ~slot.empty & matchadd slot st & st.busy
                    = ring newstlist (empty slot),
                        ~slot.empty & (matchdadd slot st \/
                                       slot.sad = st.name)
                    = ring newstlist (deaf slot),
                        ~slot.empty & matchadd slot st
                      WHERE
                        newstlist = lsst ++ [st]
```

with \/ and & as or and and, respectively.

From the first line, the function matchadd is defined by

```
matchadd slot station
   = TRUE, (station.lad = 255 \/ station.lad = slot.dest) &
           (slot.dest = 255 \/ slot.dest = station.mad)
     = FALSE,
```

and the "slot filling" function busy is defined by

```
busy slot = (slst, slem, slmo, slde, slso, slda, BUSY,
             parity [slst, slem, slmo, slde, slso, slda,BUSY])
              WHERE
                slst = slot.start
                slem = slot.empty
                slmo = slot.monit
                slde = slot.dest
                slso = slot.sour
                slda = slot.data
```

where BUSY is the code for the busy answer in the packet, and parity a function that computes the parity of the packet. The remaining functions, empty and deaf, are defined in the same way.

Now we have a basic "working" specification. Just follow the lines in its definition and you could see what it does. The ring is "simulated" by appending the current station to the list of stations, such that the slot transmitted by the current station will be processed by the "next station", in a circular fashion until we (i.e. the slot) come back to the current station.

From this specification, for example, we can already prove that no station can hold on to the slot, indefinitely, for any reason. This follows trivially from the fourth equation: a station, on receiving a full slot, or removing one of its own, which has circulated all the way around and was not removed, sends an empty slot. That means that stations can't immediately re-use the ring, and "hogging" is avoided by this very simple mechanism.

Note that a n-slot ring follows trivially follows from the definitions above, since we can redefine ring to have type

```
ring :: [station] -> count -> [slot] -> null
```

where count is a counter of the number of slots processed, and all the stations would be redefined accordingly. For example, the first equation of ring, for a ring with three slots, would now be

```
    ring (st:lsst) 3 (sl:lssl) = ring (lsst ++ [st]) 0 (sl:lssl)
```

without any other conditions (the condition on the equation above is count = 3). This equation would come before the already defined equations for ring. The action of the right hand side is simply to put the current station at the end of the list of stations, thus handing control over to the next station down the ring. The fourth equation on our previous definition of ring can now be rewritten as

```
    ring (st:lsst) n (sl:lssl)
        = ring (st:lsst) (n+1) (lssl ++ [empty sl]), ...
```

with the same guard as before. Note the incrementing of n, the slot counter.

From this basic specification, we can go on to more and more complete descriptions of the Cambridge Ring, until we have a full working simulator. To our current version, we need adding the monitor station, which controls the flow of transmission and processes the response bits in the slots.

Adding a monitor station needs changing the definition of station, with the addition, to its specification, of a further field, which we call monitor. If we have monitor = TRUE, the station is a monitor, and it takes charge of maintaining the ring operational, from the point of view of flow of information. To see why we need a monitor station, and with the attributes we are going to give it, imagine the following situation: station a has sent a message to station b, which is not operational. Just after that, station a becomes not operational. Then, we now have a circulating slot whose sending and destination addresses do not correspond to any station in the ring. That means that if we do not have a priviledged station that has the power to remove such slots, either that slot is lost (if the ring has more than one), or the ring is deadlocked. The situation is the same in any case, since in a multi-slot ring, lots of lost slots will cause a deadlock, given time.

Let us remind the reader that the slot has a monitor bit and an empty bit, as well as two response bits. When inserting its message into a slot, a station marks the slot as full and clears the monitor and response bits. When that full slot gets to the monitor station -if at all- it marks the slot as "monitored", by setting the monitor bit. Then, if the slot comes back to the monitor station marked both as full and monitored, this is so because no station -including the sending station- could remove it from the ring. In this case, the monitor station will mark the slot as empty and not monitored. That will avoid deadlocks, as far as there is a monitor station operational.

Back to our specification, adding a monitor station means changing the definition of station to

```
station == (mad, lad, status, traffic, monitor)
```

and the specification of ring to (for a one-slot ring)

```
ring (st:lsst) slot = ring (lsst ++ [st]) (empty slot),
                       st.monitor & ~slot.empty & slot.monitor
                    = ring (lsst ++ [st]) (monitor slot),
                       st.monitor & ~slot.empty & ~slot.monitor
```

besides the equations already given.

As the specification is written now, we could use it as a simulator, and all stations would use the ring in a daisy-chain, because there is no way to specify random arrival of messages to be transmitted at each station. This is not a deep functional programming problem, and one possible solution is shown in [Mei86r]. By using a distribution for each station, we could simulate the need for transmission in time.

A further problem is the specification and simulation of the several levels that make the network up. As we do not have side-effects in functional languages, we cannot "pass" a message from one level to another in the usual way. In this case, we can make good use of the higher order function concept of languages such as KRC and MIRANDA and use it, together with tuples, to specify the various levels that interact over the network. Let us call ring the specification of the physical level itself, and bbp the the specification of the basic block protocol that runs on top of the ring packets. The "bbp station" also has a status, traffic, address control, etc., but we are not to incur in the details here. We just intend to show a possible way of specifying and/or simulating the two levels above, in a way that could be surely extended to the specification of a n-level system.

First of all, we redefine ring to be of type

```
ring :: stbbpslot -> stbbpslot
```

with stbbpslot being of type

```
stbbpslot == (stationlist, bbplist, slot)
```

with ring having its usual functions plus passing and receiving information to/from the next level up. The rewriting needed is not difficult and is left to the interested reader.

Then we need to write a driver function, which will referee the cooperation of the two levels. We call this function cambridge and define it by

```
cambridge [f, g] stbbpslot = cambridge [g, f] (f stbbpslot)
```

which, when called as

```
cambridge [ring, bbp] stbbpslot
```

applies ring and then bbp, in a circular way, to the triple stbbpslot, *ad aeternum*.

To use this specification as a simulator, all that is needed is to set some conditions under which the simulation terminates, and to process the information gathered during the simulation to produce the statistics of the run.

4. Conclusions.

We have shown how to use purely applicative programming languages to construct specifications and simulators of networks and protocols. Although the research that originated this paper is still in a very early stage, we are led to believe that functional programming languages such as KRC and MIRANDA, who originated the style of programming used in this paper, are very useful tools in this area.

To assert this in its fullness, we need building a larger specification, coupled with its simulator. An ACK/NACK protocol specification has been written already, and used in practice to simulate that protocol over a channel with the characteristic of an HF radio link. The full specification of a Cambridge Ring and its protocols is now on its way.

5. Acknowledgements.

Thanks are due to Paulo Cunha (DI - UFPE), who made me promise I would write this paper. This research is supported by the Conselho Nacional de Desenvolvimento Científico e Tecnológico, CNPq, Brazil, grants 300.133-85 and 406.434-85.

6. References.

[Mei86r] S. L. Meira, "Mathematical Software in Applicative Languages". RT-DI-05/86, Dep. Informática, UFPE, Recife, PE, Brazil. Em Preparo. (*In Portuguese*)

[NeH82] R. M. Needham and A. J. Herbert, "The Cambridge Distributed Computer System". Addison-Wesley (1982).

[Tur82k] D. A. Turner, "Recursion Equations as a Programming Language", *in* Functional Programming and its Applications, an Advanced Course, C.U.P, Cambridge, UK (1982).

[Tur85m] D. A. Turner, "Functional Programs as Executable Specifications", Phil. Trans. Royal Soc. of London **312**, (1522) 363-388.