

DSL for Network Protocols

Hsueh-Li Lu

2 May 2005

Existing Methods

- Natural Languages
- Time Charts
- The C Programming Language

Natural Languages

- Imprecise
 - Verbs
 - P sends message m to q
 - P forwards message m to q
 - P delivers message m to q
 - Nouns
 - P sends message m to q
 - P sends file f to q
- Ambiguity can be exploited.

Time Charts

- Unambiguous but not scalable – exponential explosion in the number of charts.

The C Programming Language

- Well-defined semantics that doesn't match the domain
 - Concurrent assumption
 - Transmission assumption
- Hard for reasoning

The Abstract Protocol Notation

- Invented by Professor Mohamed Gouda
- A domain specific language
 - Formal
 - Scalable
 - Relatively easy for reasoning and proving

A Network

- A set of Processes
- A set of channels (FIFO)

A Process

- Local variables
 - Integer, Boolean, Array
- Actions: Guard -> Statements
 - ready -> send msg to q
 - receive req from q -> send rep to q
 - timeout #ch.p.q
 - Number of messages in the channel from p to q

The AP Notation's Drawbacks

- No abstract data type / procedure
 - Revealing too many implementation details
 - Extensive use of
 - Integers as Process ID's
 - Arrays as sets and maps
 - Loop as operations on sets and maps

Broadcasting to All Neighbors

- $h := \text{NEXT}(N, f);$ { N is the set of neighbors
}
do $h \neq f \rightarrow$
 if $\text{up}[h] \rightarrow$ send msg to $p[h]$
 [] $\sim\text{up}[h] \rightarrow$ skip
 fi; $h := \text{NEXT}(N, h)$
od;
if $\text{up}[f] \rightarrow$ send msg to $p[f]$
[] $\sim\text{up}[f] \rightarrow$ skip
fi

Finding an Up Neighbor

- `x := random;`
`y := NEXT (N, x);`
`do ~up[y] && y != x ->`
 `y := NEXT (N, y)`
`od;`
`if up[y] -> send data to p[y]`
`[] ~up[y] -> skip`
`fi`

Updating Mapping

- `do m < n ->`
 - `if (m in N && up[m]) ->`
 - `vp[m] := true`
 - `[] ~(m in N && up[m]) ->`
 - `vp[m] := false`
 - `if;`
 - `m := m + 1`
- `od`

A Vending Machine Protocol

- Customer
 - C is either ready or not ready.
 - When ready, it sends money and selection to V and becomes not ready.
 - When receiving item, it becomes ready.
- Vending Machine
 - V is either ready or not ready.
 - When receiving money, it becomes ready.
 - When receiving selection and ready, it sends item to C and becomes not ready.

Embedded in Haskell

- `data PID – ADT`
 - `instance Eq PID`
- `type Statement a = Network -> (Network, a)`
- `skip :: Statement ()`
- `(:=) :: Var -> Value -> Statement ()`
- `new :: Statement Var`
- `random :: Statement Value`
- `type Guard = Statement Bool`

Embedded in Haskell (cont.)

- `type Message = (Value, PID)`
- `type Channel = [Message]`
- `data Value = VI Integer`
 - | `VB Bool`
 - | `VT (Value, Value)`
 - | `VS [Value]`
 - | `VP PID`
- `send :: Message -> Statement ()`
- `retrieve :: Statement Message`

Embedded in Haskell (cont.)

- Network states:
 - Process state variables
 - Channels
- `simulate ::`
`Statement () -> [(Guard, Statement ())] ->`
`Network`
- `data Network =`
 `Network [(Index, Value)] [(PID, Channel)] Integer PID`
`type Variable = Index`
`type Index = Integer`

Difficulties with Haskell

- Name binding
 - PID's for processes
 - Local variable names
 - Receiving and retrieving messages

LISP

- Term rewrite by macro expansion
- Processes identified by symbols
- Processes implemented as objects
- Guards, Statements implemented as Closures
- Data structures implemented as lists

Customer

- (process c
 (variable (readyc true))
 (when readyc
 (send 'money v)
 (send 'selection v)
 (readyc := false))
 (when (receive 'item 'v)
 (readyc := true)))

- (setf c (let ((readyc nil)
 (channel (make-queue)))
 (list #'(lambda (msg) (enqueue msg channel))
 (cons #'(lambda () readyc)
 #'(lambda ()
 (send 'money v)
 (send 'selection v)
 (setf readyc nil))))
 (cons #'(lambda ()
 (receive 'item v channel))
 #'(lambda ()
 (retrieve channel)
 (setf readyc t))))))))))

Vending Machine

- (process v
 (variable (readyv false))
 (when (receive 'money 'c)
 (readyv := true))
 (when (receive 'selection 'c)
 (if
 (when readyv (send 'item c))
 (when (not readyv) (skip)))
 (readyv := false))))

- (setf v (let ((readyv nil)
 (channelv (make-queue)))
 (list #'(lambda (msg) (enqueue msg channelv))
 (cons #'(lambda () (receive 'money c channelv))
 #'(lambda ()
 (retrieve channelv)
 (setf readyv t))))
 (cons #'(lambda () (receive 'selection c channelv))
 #'(lambda ()
 (retrieve channelv)
 (choose (list (cons #'(lambda () readyv)
 #'(lambda () (send 'item c)))
 (cons #'(lambda () (not (readyv)))
 #'(lambda () (skip))))))
 (setf readyv nil))))))

Simulation

- (defun simulate (processes)
 (let ((all-axns nil))
 (dolist (p processes)
 (setf all-axns (append all-axns
 (cdr p))))
 (do () (nil)
 (funcall (choose all-axns))))))

Parameterized Actions

- `(foreach (x list 0 1 2 3 4)`
 `(when (aref a x)) (send 'mag 'q))`

=>

```
(when (aref a 0) (send 'mag 'q))  
(when (aref a 1) (send 'mag 'q))  
(when (aref a 2) (send 'mag 'q))  
(when (aref a 3) (send 'mag 'q))  
(when (aref a 4) (send 'mag 'q))
```

Current Status

- Basic AP Notation
 - send, receive, if, assignment
- Parameterized Actions
- Nondeterministic assignment (partial)
 - any
 - random
- Abstract data types

LISP macro

- Macro expansion points
 - (`{macro name}` `{abstract syntax tree}`)
- Not composable
 - Macro process can't be composed by macro guard and macro statements
 - (`defmacro process ...`) can't be defined by (`defmacro when ...`), (`defmacro receive ...`), (`defmacro if...`), ...

Future work

- Process arrays
- Messages with fields
- Timeout guards
- Dynamic Parameterized Actions

Related work

- **The Austin Protocol Compiler**
 - <http://www.cs.utexas.edu/~mcguire/software/apc>
- **Prolac**
 - <http://pdos.csail.mit.edu/prolac/>
- **Computer-Aided Protocol Engineering**
 - <http://www.bayfronttechnologies.com/capeprod.htm>