UML Modelling Case Study: A GSM BTS

Sorin Manolache, sorma@ida.liu.se
Linköping University, IDA/ESLAB

Motivation

- Investigate the versatility of UML as an implementation independent specification language for complex hardware/software real-time systems
- Possibly perform architecture exploration on the model

Therefore...
Specification Requirements

- executable specification
- (explicit) specification of time constraints and execution time
- specification of concurrency
- modelling of resources (hw: processors, comm. links, memory, etc.; sw: shared data, message queues, locks, etc.)
- specifying the binding
- scheduling policies and scheduling units

Application Characteristics

Application implies:
- data oriented processing
- control oriented processing
- protocol processing
- analog part
- concurrency
The GSM BTS (1)

Radio interface -> Base Station Controller (BSC) -> Network Switching Subsystem (NSS)

The GSM BTS (2)

Radio Sub-assembly -> Measurements Processing Subassembly -> Baseband Subassembly

Base station Control Functions

Radio interface -> Abis interface
Model Architecture

FU – functional unit
BC – baseband controller
TRX – transmitter-receiver
BuG – burst generator
BlG – block generator

Protocol processing
Control oriented
Data oriented
Specified at a high level of abstraction

Main Packages of the Model

- FunctionalUnits
- BasebandCtrl
- TRX
- PIs
- Abis
- Radio
- Globals
- Shell
The *FunctionalUnits* Package

- operates on the data path
- the DSP algorithms are specified in C++
- no control functions (very primitive statechart)
- simple interface (in command, out notification) ⇒ event-driven
- instrumented for timed simulation
- examples of units:
  - CRC encoders, CRC decoders
  - burst interleavers, burst deinterleavers
  - convolutional encoders, convolutional decoders

---

![FunctionalUnits Class Diagram](image-url)
**FunctionalUnits Behaviour**

```
for (k = 0; k < NORMAL_BLOCK; k++) {
    val = block->getBitAt(k);
    b = ((4 * currBlock + (k % 8)) % 8);
    j = 2 * ((49 * k) % 57) + (k % 8) / 4;
    outBuf[b]->setBitAt(j, val);
}
```

**The BasebandCtrl Package**

- control oriented
- instrumented for timed simulation (execution deadlines of managed functional units)
- example of baseband controllers:
  - SpeechDownLkCtrl
  - DataUpLkCtrl
  - CtrlDownLkCtrl
**BasebandCtrl Class Diagram**

```
ConvEnc 0,1  DownLk 0,1  CRCEnc
        |       |  Interl
        v       v
          Data
```

**BasebandCtrl Statechart**

```
active
- evBursts/del->GEN(evDelCmd(params->bursts));
  evDelCompl/report(result);
- evConvCompl/else
  evCRCDecCompl/else
- params->status==0
```

```
idle
  watchdog
  tm(deadline)
```

```
deinterl
  convDec
  CRCDec
  errConv
  errCRC
```
The **TRX Package**

- control oriented
- controls signalling, uplink and downlink transmission
- allocated and deallocated channels
- "database" (keeps track of allocated channels and their types)
- time-triggered

---

**The TRX Statechart**

```
inactive
```

```
<table>
<thead>
<tr>
<th>signalling</th>
<th>uplink</th>
<th>downlink</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>access</strong></td>
<td><strong>dispatch</strong></td>
<td><strong>idle</strong></td>
</tr>
<tr>
<td>[TSN()==3]</td>
<td>[isAlloc()]</td>
<td>[else]</td>
</tr>
<tr>
<td>[else]</td>
<td>[else]</td>
<td>[evBurstTick]</td>
</tr>
<tr>
<td>evAlloc/alloc(params);</td>
<td>evBurstTick</td>
<td>[evAbisBlock/extract(params);]</td>
</tr>
</tbody>
</table>
```
The PIs Package

• protocol processing on the datalink layer
• aspects:
  – sliding window and transmit–wait–acknowledge concepts
  – segmenting and reassembling ⇒ packets may arrive out of order
  – channel loss
### Sender Statechart

- **idle**
  - evSendMsg
  - evAck/reset()
- **dummy**
- **toTrans**
  - [canTrans]
  - [else]
  - [timerRuns/rst()]
  - [else]
- **trans**
  - tm(trans)
  - [windowFull]
  - [windowFull]
  - [else]
- **cannotTrans**
  - evAck/reset()
  - evError
- **running**
  - evStart
  - tm(error())
  - evReset

### Layer 2 Message Transmission

**Sender**
- evSendMsg(msg)
- segment()
- tm(trans)
- sendMsg(pkg)

**L2Channel**
- evConvey(pkg)
- tm(line)

**L2ChHelper**
- evGetPkg(pkg)
- buffer(pkg)
- enqueue(ack)
- tm(trans)

**Receiver**
- evAck(ack)
- evConvey(ack)
- tm(line)
The **Globals Package**

- contains various utility classes and functions
- examples:
  - classes: BitAddressableBuffer, Queue, BTSConfig (singleton), GlobalClock (singleton)
  - functions: getTSN(), getFrame()
- *GlobalClock* implements the subscriber pattern

The **Shell Package**

- for interfacing the designer with the model
- provides a means to feed "coarse-grained" events from the outside (MS or BSC)
**Channel Allocation**

<table>
<thead>
<tr>
<th>MS</th>
<th>BSC</th>
<th>Shell</th>
<th>ToBSCLink</th>
<th>FromBSCLink</th>
<th>TRX</th>
</tr>
</thead>
<tbody>
<tr>
<td>evAccess(reson)</td>
<td>RACHAccess(reson)</td>
<td>evSendMsg(msg)</td>
<td>evSendMsg(msg)</td>
<td>evAlloc(tsn, mode)</td>
<td></td>
</tr>
<tr>
<td>evGrant(trx, tsn, mode)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Rhapsody’s Simulation Engine**

- the objects of the model are clustered in one or several OS threads (under the user’s control)
- an event queue and a timeout pool are associated to every thread
- the time is advanced to the value of next due timeout when all threads are idle, that is, when
  - the objects wait for timeouts to be due, or
  - the event queue is empty
- we observed that the result of the timed simulation depends on how the objects are grouped in threads ⇒ probably because of the OS the simulator relies upon
Simulation

- implied a small modification of the tool simulation engine in order to have access to its clock
- a list of events to be fed to the model can be specified in a file ⇒ possibility of specifying test scenarios
- for telecom applications automatic generation of large set of events with various probability distributions would be helpful ⇒ an idea would be to define an test scenario specification language
- impossibility at the moment to feed events to the system at explicit moment in time ⇒ we adopted two approaches

Feeding Test Vectors to the Model (1)

- “wrap” the model in an “eventGenerator”

```
switch (time) {
  ...
  case 48000:
    sprintf(buf, “Car passes on track 2 at %d”, t);
    cout << buf << endl;
    insertMessage(buf);
    itsSensor[5]->GEN(evCar());
    nextStep = 1000;
    break;
  case 49000:
    sprintf(buf, “Car passes on track 2 at %d”, t);
    ...
}
Feeding Test Vectors to the Model (2)

- GUI that allows feeding of events ⇒ dynamic scenario generation
- applicable when
  - coarse-grained events (e.g. `evAllocate`, `evGrant`)
  - not necessarily interested when an input event arrives but rather in the response time

Conclusions

- an industrial-scale application has been modelled in UML
- it comprised heterogeneous aspects: protocol processing, DSP, control oriented, real-time requirements
- UML leverages the quality of the specification and the ease of understanding by the visualization of the various relationships it provides
- the timeout transition is currently the only means for dealing with real-time aspects
- manual instrumentation is required for correctness checking
- software or hardware generation from UML is equivalent to synthesis from statecharts or from the language of the imported/generated code
- limited possibility for architecture exploration