

Time for Timber

Per Lindgren
Johan Nordlander
Linus Svensson
Joakim Eriksson

Luleå University of Technology
Department of Computer Science and Electrical Engineering
Division of EISLAB

Time for Timber

**Per Lindgren, Johan Nordlander,
Linus Svensson and Joakim Eriksson**

Luleå University of Technology
Department of Computer Science and Electrical Engineering
Division of EISLAB
Luleå, Sweden

Luleå, December 2004

Time for Timber

Abstract - Embedded systems are often operating under hard real-time constraints, while at the same time being constrained by severe restrictions on power consumption. For such systems, robustness and reliability can be a question of life and death, which calls for rigorous system design and methodologies for validation. In this paper we advocate a design methodology for low-power, real-time systems, based on Timber; a pure reactive system model that allows for formal reasoning about various system properties. We outline how system specifications in Timber can be "compiled" into efficient standalone executables for general light-weight microcontroller based target platforms. Methods for resource analysis and implications to system dimensioning and validation are further discussed.

1 Background

Embedded systems are often operating under hard real-time constraints, while at the same time being constrained by severe restrictions on power consumption. The increasing market of mobile devices makes out for a wide range of battery powered, real-time systems. Other examples can be found in industrial measurement and control systems. For such systems, robustness and reliability can be a question of life and death, which calls for rigorous system design and methodologies for validation. In this paper we set out on a search for the "holy grail" of real-time embedded system design. On our journey, we will come across and challenge daemons of real-time embedded system design. Luckily, to our advantage, we come equipped with a design methodology that combines theoretical computer science with hands on applicability of today's commercially deployed tools.

1.1 Challenges of Real-Time Embedded System Design

Scavenging the field of real-time embedded system design, we come across a number of identifiable key problems;

- Capturing the timely behaviour of a reactive system
- Ensuring state integrity while avoiding unintended blocking
- Correctly managing dynamic memory
- Performing system dimensioning and verification

The following subsections further elaborate on the mentioned problems.

1.2 Capturing the Specification

Real-time system design is usually identified with the concept of using a Real-Time Operating Systems (RTOS) for concurrency and process management. A quick survey gives at hand a number of RTOS approaches, ranging from minimalistic OS cores such as TinyOS [1] and Contiki [2], to fully fledged systems such as VxWorks, RT-Linux and QNX [3, 4, 5].

Traditional C/C++ coding for RTOS lacks the ability to natively capture the timely behaviour of a system. Whereas the real-time behaviour of the system is often specified in terms of reaction deadlines, the common RTOS programming model requires the reactions to be translated into processes or threads, and the deadlines to be translated into priorities.

Even if we can derive appropriate thread priorities, the problem of mapping a concept of time constrained reactions onto traditional language constructs remains.

To remedy this fundamental problem, a number of language-based approaches have been suggested. Real-Time Specification for Java (RTSJ) is an example of a concurrent language with a real-time extension [6]. Although RTSJ introduces the notion of deadlines, the current priority-based scheduler implemented in the Real-Time Java Virtual Machine (RTJVM) requires the programmer to assign priorities to each thread. Deadlines are currently intended mainly for on-line schedulability analysis and detection of missed deadlines and time budget (cost) over-runs.

In the realm of experimental, real-time programming languages we find designs that do provide general forms of timing constraints that, at least in principle, could open up for deadline-based scheduling. Examples are Real-time Euclid [7], RTC++ [8], and CRL [9]. Hooman and van Rosmalen describe a generic language extension in the same spirit, exemplified with an unnamed language design that even comes with a formal definition [10]. However, a general remark regarding these approaches is that timing constraints apply to very fine grain program units (statement blocks in RTC++, individual statements in CRL and the Hooman/van Rosmalen design); which, by the presence of general threads and blocking constructs, do not correspond very well to the actual schedulable units as they appear at run-time. However, even if the mentioned languages allow timely properties to be defined in the program specification, there is no direct correspondence to events managed by the underlying operating system. Hence, the basic problem of formal system analyses remains.

1.3 State Integrity

Concurrent programming in itself constitutes a complex and delicate matter. One major obstacle is ensuring state integrity in the presence of concurrent access. The traditional approach is to enforce state protection by the use of mutexes, semaphores and monitors; constructs that, in effect, reduce the parallelism of the system by silently blocking threads that risk violating state integrity.

Outmost care must be taken in how these blocking constructs are applied; otherwise the system might suffer from deadlock or other similar situations where the system is unable to respond unless some specific external event first occurs. What the programmer might hope for are guidelines or so-called programming patterns that enforce a sound blocking structure. To strictly obey these rules is a tedious and error-prone task, as neither programming languages nor operating system interfaces offer much help in detecting possible problems with unexpected blocking. This pinpoints a fundamental problem of concurrent programming in traditional languages: blocking is both a transparent code property and the corner-stone on which event-driven programs are built, all at the same time!

1.4 Memory Management

Another problem facing the user of modern heap based languages is the task of correctly managing dynamic memory. Although not a problem specific to embedded systems, the scarce memory resources of embedded systems make memory leakage an urgent and intolerable problem.

One commonly suggested solution is to lessen the burden on the programmer by applying some form of automatic garbage collection. However, if the system is supposed to operate under hard real-time constraints, special requirements are imposed on the garbage collection (GC) mechanism [11]. Firstly, the latency of memory allocation must still be

predictable, to allow for accurate execution-time analysis. Secondly, the garbage collector itself must be scheduled so as to never violate any deadlines in the system.

In the general case, events can occur at any time, which suggests that the GC should be interruptible. That is, in the case the GC is running when an event occurs, there is a risk its deadline will be missed if handling it is suspended until the GC has finished. Thus, it must be possible to interrupt the GC and handle the event immediately. However, this kind of real-time GC has not yet reached wide-spread acceptance, and traditional RTOS design typically excludes automatic memory management altogether in favour of the traditional "malloc/free" memory management interface

An illustrative example of this dilemma is found in the Real-Time Specification for Java. Java threads operating under real-time conditions are restricted to use only non-GC memory (which might be either static or scoped). This is to guarantee that memory allocations by real-time threads will never invoke the garbage collector, which would risk violating real-time properties. The effect is that even though RT Java is a garbage collected language, the programmer is forced to resort to manual memory management for all real-time threads.

1.5 System Dimensioning and Verification

When designing and dimensioning hard real-time systems it is crucial to ensure that reaction deadlines will be met by the implementation at all times. However, as mentioned in Section 1.2, it is hard to capture reaction deadlines into traditional programming language constructs. This precludes formal verification to be directly based on the system description. Hence, over the years, a number of indirect approaches have been developed.

One simple but obviously less satisfactory approach is to validate the system through simulation. Only in cases where the system (and its environment) has a totally predictable behaviour this is an acceptable solution. A prominent example thereof is a statically (off-line) scheduled system handling periodic inputs. However, such systems are quite inflexible and cumbersome to design and update. Another drawback is that these systems are active (opposed to reactive) in turn leading to higher power consumption than a reactive counterpart.

Dynamically scheduled systems offer many advantages, e.g. they are responsive to sporadic events (events that can occur at any time with a least interval), they are more flexible, hence easier to design and update and they can efficiently be put to sleep when awaiting further events. System verification may be done through formal schedulability analysis (given reaction deadlines, reaction execution times, and a timely event model).

Rate-monotonic (RM) scheduling is a popular and well understood simplification of the general case. It indicates the use of priority based scheduling together with the assumption that reaction deadlines equal the inter-arrival times of periodic events. Priorities are computed online (inverse proportional to deadline). Although applicable in many cases, RM suffers from suboptimal CPU utilization (down to about 70% for a large number of simultaneously executing periodic processes). However, a more fundamental problem concerns establishing reliable execution times for the event reactions – a requirement for formal schedulability analysis. Encoding event reactions in the form of RTOS and middleware services complicates such analyses, as formal descriptions of all system components might be hard to obtain.

2 Time for Timber

In order to meet the challenges of real-time embedded system design we advocate a design methodology based on Timber; a pure reactive system model that allows for formal reasoning about various system properties. Timber (the name is derived from the words time, embedded and reactive) is a system description language that captures the sought real-time system behaviour as events and their reactions in the form of reactive objects. The rigorous type system and formal semantics ensure the description to be concise and well defined. Timber descriptions preserve all native parallelism of the system, without burdening the designer with any need of explicit coding of the parallel behaviour. Under the Timber execution model, an event causes a reaction that in turn can generate further events. A central property of Timber semantics is that each reaction will terminate (run to end) independent of further events, hence a system described in Timber will be responsive at all times (under the limitation of available CPU resources) and free of deadlocks. This provides for meaningful reasoning about lower and upper time bounds for the system reactions. These time bounds make out the foundation for further system analyses.

A system description in Timber can be viewed as a platform independent model, which for a given target system can be translated to an efficient executable (consisting of a real-time scheduler, memory manager and code that implements the system's reactions). Thus, the generated software is a complete implementation of the initial system description and does not add any components of uncertainty that external code such as operating systems and device drivers may invoke. This leads to the conclusion that properties derived from system analyses will hold also for the target system. Given appropriate tools for analyses, we may derive a minimal, custom-made target system that meet the resource demands of the initial specification.

2.1 Capturing the Specification in Timber

Here we will just briefly overview the constructs of Timber that are most relevant to the rest of the paper. For an in depth description, we refer to the draft language report [12], the formal semantics definition [13] and previous work on reactive objects [14, 15] and functional languages [16].

2.1.1 Objects in Timber

On the top level, a Timber program is a set of bindings of names to expressions. In the below example, the *template* keyword indicates the definition an object generator (class) with the identifier *counter*, the instantiation code *val := i* and a public interface consisting of a record with labels *inc* and *read*.

An object consists of state variables and action (asynchronous)/request (synchronous) methods used to update/inspect the state (Figure 1).

```
counter(i) = template
    val := i
    return {
        inc(j) = action val := val + j
        read  = request return val
    }
```

Figure 1. Example object: counter.

```
test (env) = action
```

```

c <- counter(1)
c.inc(5)
n <- c.read
env.putStr(show(n))

```

Figure 2. Example use of object counter.

Figure 2 defines a client method *test*, that given *env* (an i/o interface to the environment) will perform the following operations in sequential order;

- `c <- counter(1)`
An instance *c* of *counter* will be created, and the initialization code of *c* will be executed with the parameter *i* equal to 1.
- `c.inc(5)`
An asynchronous message *inc(5)* will be posted to *c*.
- `n <- c.read`
Leads to posting the synchronous message *c.read* and awaiting its result. The pending event *c.inc(5)* will now be executed by *c*, which makes a destructive update to its local state $val := 1 + 5$. The *c.read* request thus returns the value 6, which the client can access under the name *n*.
- `env.putStr(show n)`
The function *show* is invoked with the parameter 6, resulting in the string primitive "6". The asynchronous message *env.putStr("6")* will now be posted. The client method *test* has run-to-end.
- The event *env.putStr("6")* is now handled, the effect of this action is to output the string "6" to the environment.

The execution model of Timber ensures mutual exclusion between the methods of an object instance, thus relieving the programmer from the burden of explicitly enforcing state integrity. In the above example, the action *inc(j)* and the request *read* will execute under mutual exclusion. However, the execution model also allows object instances to be fully parallel, unless involved in a chain of synchronous requests. To ensure that blocking will not be indefinite, each method must run-to-end, and deadlocks must be detected and turned into exceptions. Fortunately the latter comes for free as an inherent feature of the Timber run-time system. The former criterion, a run-to-end semantics, is actually guaranteed by the language, as long as methods do not voluntarily step into infinite loops. Infinite loops are however always degenerate in Timber, and in particular not part of the structure of an infinite reactive process. Instead, every Timber object is identical to a process, and objects will exist as long as there are external references to them. Due to the run-to-end semantics of methods, we may thus say that Timber processes are infinite and reactive by construction.

2.1.2 Deadlock Avoidance

The problem of deadlocks is closely related to blocking constructs, as illustrated in the example below. Two objects *o1* and *o2* are instantiated as state variables in *test*. Notice, that we may use *o2* as a parameter in the creation of *o1*, even if *o2* is not yet created. This is possible since the state initiation does not require the evaluation of parameters. So far so good, now let us execute the start method of *test*. This leads to a synchronous message *o1.req*, which in turn leads to the synchronous message *o2.req*, which in turn leads to the

synchronous message `o1.req`. Timber semantics implies mutual exclusion between methods of the same object, i.e. we cannot execute two invocations of `o1.req`. The program has entered a deadlock. However, the run-time semantics of Timber allows detecting circular events, and the programmer can be presented the circular event chain in the presence of deadlock. This is opposed to the situation when a deadlock just causes a system to be unresponsive. In such case it is hard to tell if the system just awaits some specific input, or if it suffers a deadlock (and if so, what caused the deadlock).

```
object o =
  template
    obj := o
    return {
      req = request
      r <- obj.req
      return r
    }

test =
  template
    o1 <- object o2
    o2 <- object o1
    return {
      start = action
      r <- o1.req
    }
```

Figure 3. Deadlock example.

2.1.3 Timber and Time; the After Construct

Periodic processes are expressed using self-addressed messages with time offsets. This can be exemplified as follows.

```
test(env) =
  template
    c <- counter(0)
    let
      poll = action
      r <- env.inport.readData
      case r of
        Nothing -> done
        Just d -> c.inc(d)
      after (50*milliseconds) poll

    printer = action
    r <- c.read
    env.putStr(show r)
    after (5*seconds) printer

  return {
    start = action
    poll
    printer
  }
```

Figure 4. Polling example.

Invoking *test env* causes the following operations to be carried out;

- The scheduler will set the *baseline* to the current value of the system timer, we assume *0ms*.
- The creation of a counter *c* with an initial state *val=0*.
- The *test.start* action posts messages to *poll* and *printer*.
- The scheduler will now process the posted events, starting with *poll*. The corresponding action polls the inport by a request *env.inport.readData*. If data is available an asynchronous message to the counter is posted *c.inc(d)*. The *after* construct posts a message *poll*, to be scheduled no earlier than *baseline+50*milliseconds=50ms*.
- The next event from the event queue is processed, in this case *printer* which post a request to *c.read*, awaits and stores the result in *r*. The pending event *c.inc(d)*, will be carried out first, before the *c.read* is handled. The result *r* is passed to the function *show*, resulting in a primitive string. This string is posted as the parameter to the asynchronous event *env.putStr(show r)*. The *after* construct posts a message *printer (env c)*, to be scheduled no earlier than *baseline+5*seconds=5000ms*. This ends the chain of immediately schedulable events caused by invoking *test env*.
- When the system timer reaches *5ms* the previously posted message *poll* is released, i.e. the *poll* action is scheduled for execution. The *baseline* is updated by adding *5ms*. The operations of 3 are carried out periodically, every *5ms*.
- After 5 seconds the *printer* event will be released (since the system timer has reached *5000ms*). The operations of 5 are carried out, periodically, every *5th* second.

The example above actually shows how two describe a parallel system with two periodic processes (defined by the actions *poll* and *printer*) that are using a shared resource (the counter *c*). Notice that no explicit coding for parallelism is required, neither is there any need to worry about multiple timers, nor any explicit coding for ensuring state integrity. Since *poll* and *printer* are encapsulated in the same object they execute under mutual exclusion. If true parallelism is needed we can implement *poll* and *printer* into separate objects. In this way the object-oriented and parallel execution models go hand in hand.

Another important feature of Timber is that all references to time are referring to the *baseline*. The *baseline* is managed by the scheduler and is free of jitter caused for example by the actual time for posting events (which indeed might vary due to the actual schedule). Hence, the timely behaviour will be free of “drift” (besides that of hardware related fluctuations).

Because of the run-to-end requirement of Timber methods, we cannot pause the execution by a *wait(time)* statement. Instead we can use the *after* construct to establish such behaviour, e.g. after invoking *start*, the *outport* should take the value 1, 2 and 3, after 0, 10 ms and 20 ms respectively.

```

start = action
    outport.set 1
    after (10*milliseconds) set2
    after (20*milliseconds) set3
set2 = action
    outport.set 2
set3 = action
    outport.set 3

```

Figure 5. Wait encoding in Timber.

Where, both asynchronous events *set2* and *set3* are posted by the *start* method. Or alternatively;

```

start = action
    outport.set 1
    after (10*milliseconds) set2
set2 = action
    outport.set 2
    after (10*milliseconds) set3
set3 = action
    outport.set 3

```

Figure 6. Wait encoding in Timber, alternative implementation.

Where the asynchronous event *set3* is posted by the *set2* method. This coding of intermediate methods (states) can be avoided by anonymous (unnamed) methods.

```

start = action
    outport.set 1
    after (10*milliseconds)
        action
            outport.set 2
    after (20*milliseconds)
        action
            outport.set 3

```

Figure 7. Wait encoding using anonymous methods.

Or alternatively;

```

start = action
    outport.set 1
    after (10*milliseconds)
        action
            outport.set 2
            after (10*milliseconds)
                action
                    outport.set 3

```

Figure 8. Wait encoding using anonymous methods, alternative implementation.

This encoding resembles the sequential programming style commonly adopted among programmers. The difference however, that a system implemented in Timber will remain responsive.

2.1.4 Timber and Time; the Before Construct

For each method Timber also allows a deadline (upper time limit) to be defined. In the example below, this implies that a correct (i.e. schedulable system) will finish the invocation of *poll* within $5 * \text{milliseconds}$ from the release of *poll*, (i.e. before the absolute deadline of $\text{baseline} + 5 * \text{milliseconds}$). Events produced directly or indirectly by tick will inherit the absolute deadline of *poll*. In the below example, the eligible execution window for *poll* is set to $1/10^{\text{th}}$ of the period, in effect limiting the jitter of the polling.

```
poll = before (5*milliseconds) action
  r <- env.inport.readData
  case r of
    Nothing -> done
    Just d  -> c.inc(d)
  after (50*milliseconds) poll
```

Figure 9. Before and after constructs.

When some asynchronous messages posted by the method invocation are less time critical we can relax their deadlines. In the below example, reading of data and resetting the data available signal are still tightly bound by the deadline, but the actual “processing” *c.inc(data)* may be postponed until $100 * \text{milliseconds}$, i.e., even passed the period time. Naturally, relaxing deadlines increases the schedulability of a resource constrained system.

```
poll = before (5*milliseconds) action
  r <- env.inport.readData
  case r of
    Nothing -> done
    Just d  -> before (100*milliseconds) inc(d)
  after (50*milliseconds) poll
```

Figure 10. Extending the deadline.

Increasing the deadline of synchronous events, would indeed increase the deadline of the method itself, hence considered superfluous. The semantics of Timber does not allow the deadline to be decreased, and this for good reasons. The deadline of a method (action or request) is defined by the before construct in the method definition, e.g.;

```
poll = before (5*milliseconds) action
```

This deadline is a constant relative to the baseline, and the absolute deadline will be known to the scheduler when an event to *poll* is posted. However, defining a deadline by the *before* keyword inside a method, may be variable and unknown to the scheduler at the time an event to *poll* is posted e.g.;

```
t <- getNewDeadline;
before t c.inc(d)
```

Figure 11. Dynamic deadline extension.

Decreasing the deadline dynamically would imply dynamically decreasing the deadlines of all previous statements in the method. At the time of evaluating this new tighter deadline, it may already been missed, hence decreasing deadlines is prohibited by the Timber semantics.

2.1.5 Message Tags

We have already seen examples, creating periodic behaviour, where the baseline is increased by the `after` construct. However, the `after` construct can be used for other purposes as well, e.g. to implement timeout. The example below assumes a template `set`, implementing the actions `set.include index`, `set.exclude index`, and the request `set.member index`. Furthermore it assumes `env.send data ack index`, to send `data`, and use `ack index` as a callback when the send operation has been acknowledged. This specific implementation manages 16 outstanding send events.

```
tick_and_timeout(env) =
  template
    s <- set
    index := 0
    let
      send(data) = action
        index := (index + 1) mod 16;
        s.include(index)
        env.send(pack(data,ack(index)))
        after (1*seconds) timeout(index)

      ack(i)      = action
        s.exclude(i)

      timeout(i) = action
        if (s.member(i)) then
          "timeout!!!"
        else
          "OK"

    return {
      sendTimeout(data)= send(data)
    }
```

Figure 12. Creating a Timeout.

In the above example, each `send` is given a unique identifier by annotating it with `index`. In `timeout(index)`, we explicitly check if this specific timeout has already been `ack`ed, by checking `s.member(index)`. However, Timber allows this to be managed in a more elegant manner, by in effect cancelling the timeout message whenever the corresponding `ack` is received. Each message is given a unique message tag, implementing the method `cancel`.

```
tag <- after t timeout
...
tag.cancel
```

Figure 13. Message tags.

We can utilize this to implement timeout with message tags;

```
tick_and_timeout(env) =
  template
  let
    send(data) = action
      fix tag <- env.send(pack(data,ack(tag)))
      after (1*seconds) timeout(tag)
    ack(t)      = action
      t.cancel
    timeout(t) = action
      env.putStr("timeout" ++ show(t))
  return {
    sendTimeout = send
  }
```

Figure 14. Creating a Timeout using message tags.

Notice, the recursive use of *tag*; it is created (*tag <-*) and used in the same statement as a parameter to *env.send*. The call by value semantics of Timber normally requires parameters to be evaluated first, since *tag* is not yet created this statement would be illegal. However, the keyword *fix*, allows overriding this semantic rule. In this case it is safe, since *tag* is used only as a parameter to the asynchronous message *ack*. At the time of posting *ack(tag)*, the value of *tag* will be known, hence the statement is sound. The above implementation can handle any number of outstanding *send/ack* messages (limited only by the amount of memory available for the message queue).

2.1.6 Interfacing the Environment

The examples so far have been limited to handling events internal to the Timber system. In the following example, we show how the Timber program can be interfaced to the physical environment, through the use of interrupts and hardware registers. The below example consists of the *sonar* class, the *alarm* class (not depicted) and the *main* class. *Sonar* periodically sends a pulse and listens to the echo to determine if an alarm should be alerted. The pulse is generated by writing to a hardware register (port). The echo and alarm reset are received as interrupts, *sonarIRQ* and *buttonIRQ* bound to the methods *s.sonar* and *a.off* respectively.

```

sonar (port,alarm) =
  template
    t := baseline
    let
      ping = before (50*microseconds) action
        port.write(beepOn)
        t := baseline
        after (2*milliseconds) stop
        after (1*seconds) ping
      stop = action
        port.write(beepOff)
      echo = before (5*milliseconds) action
        let distance = k*(baseline - t)
        if (distance < limit) then
          alarm.on
    return {
      sonar = echo
      start = ping
    }

main regs =
  template
    s <- sonar ((regs!0xac00) a)
    a <- alarm (regs!0xa3f0)
    return [
      (resetIRQ, s.start),
      (sonarIRQ, s.sonar),
      (buttonIRQ, a.off)
    ]

```

Figure 15. Sonar example.

On this level, the interface to the software, as seen from the hardware, is the array of interrupt handlers it provides in the form of a list of pairs. Consequently, the interface to the hardware as seen by the software, takes the shape of an array of device registers that can be read or written.

In this section we have demonstrated how sought real-time behaviour of can be captured by Timber constructs. Recapitulation and conclusions;

- *Objects and parallelism*: The parallel and object oriented models go hand in hand. An object instance executes in parallel with the rest of the system, while the state encapsulated in the object is protected by forcing the methods of the object instance to execute under mutual exclusion. This implicit coding of parallelism and state integrity coincides with the intuition of a reactive object. In a correct Timber program, all methods are non-blocking, hence the system will be responsive to incoming events at all times.
- *Events, methods and time*: The semantics of Timber conceptually ties events and methods in a way that makes it possible to unify the timely requirements for a reaction to an event, with the run-time demands on the execution of a method. The baseline states the absolute time for the release of an event, i.e. the point in time when the corresponding method becomes eligible for scheduling. Before and after constructs define points in time relative to the method's baseline, free of jitter due to the actual scheduling of events. This allows the deadline of a reaction to an event, to be accurately described by the before construct of the corresponding

method. Moreover, the `after` construct, can be used to precisely define future points in time. This is useful for example in generating periodic-, timeout-, and wait-constructs.

2.2 From Specification to Implementation

A system description in Timber can be viewed as a formal model of the system implementation. The designer may also choose to model the system's environment in Timber. This is especially useful in modelling embedded applications, where the behaviour of the environment often allows precise and detailed modelling. Similar approaches are widely applied in the design, debugging, validation and verification of digital systems. Hardware description languages (HDLs) such as VHDL, Verilog, Superlog and SystemC are commonly used to describe system specifications and their environment in the form of implementations and test-benches. Using environment models, stimuli for functional validation can be automatically generated and the simulation results analyzed. A Timber model of the *inport* used in the polling example of Figure 4, is shown below. It realistically mimics a buffer with internal state being either empty (*Nothing*) or having a value (*Just d*).

```
inport =
  template
    state := Nothing
    return {
      readData = request
        let r = state
        state := Nothing
        return r
      setData(d) = action
        state := Just d
    }
```

Figure 16. Modelling the environment.

In the realm of digital design, formal methods are getting increasing attention, this largely since faults are considered to be so costly that it pays off in the end to use a rigorous, formal, methodology for design verification. In the following we will argue that system and environment descriptions in Timber opens up for such possibilities in the realm of embedded real-time system design. Hence, we advocate Timber, not as some theoretical tool merely useful to weird computer scientists, but indeed as an alternative for hands on implementation of real-time embedded systems.

The semantics of Timber does not state in which way the execution mechanisms of methods, posting of events and management of memory should be devised. It only defines the expected results, and states the timing properties for a correct implementation. Hence there is no “single” format for the implementation of Timber. However, with the basis of today's available platforms for embedded applications, a practical approach to a “Timber compiler” is underway.

From a system specification in Timber an executable is generated consisting of;

- real-time scheduler
- dynamic memory manager
- the implementation for all methods (actions/requests) of the specification

The real-time scheduler uses a hardware abstraction layer (HAL) to interface the environment. Fundamental hardware related issues, such a definition of word-length, interrupt vectors, system timer, the context switch mechanism, memory layout, and power/clock modes, are managed by the HAL. Once a Timber executable is downloaded, a startup procedure is carried out. The memory manager is initiated and the object structure together with the corresponding state variables are created and bound to the environment (i.e., interrupts and registers). In the example below, the sonar and alarm objects *s* and *a* are instantiated, and the interrupts *resetIRQ*, *sonarIRQ* and *buttonIRQ* tied to *s.start*, *s.sonar* and *a.off* respectively. As there are no events eligible for scheduling, the system will be idle. Hardware specific low-power mode is now entered, allowing the system to deep sleep, woken only by external events (interrupts).

```
main regs =
  template
    s <- sonar ((regs!0xac00) a)
    a <- alarm (regs!0xa3f0)
    return [
      (resetIRQ, s.start),
      (sonarIRQ, s.sonar),
      (buttonIRQ, a.off)
    ]
```

Figure 17. Interfacing the environment.

Whenever an interrupt occurs, the corresponding method becomes eligible for execution. E.g. *resetIRQ* causes the execution of *s.start*, which in turn posts messages *stop* and *ping*. The method *stop* calculates the distance to the object causing the echo, and posts an event *a.on* (starting the alarm) if the distance is below a predefined limit. The current Timber real-time scheduler maintains separate memory stacks and message queues for each object instance, *s* and *a* in this example. When all methods eligible for scheduling has run-to-end the system is idle. As previously mentioned this can be utilized to put the system into suitable hardware specific low-power mode. In the case all message queues are empty, the system can be put to deep sleep, woken only by external events (interrupts). In the case we have outstanding timer events (not yet eligible for scheduling) the system can be put to sleep, woken by external- or internal timer interrupts. Of course, the actual set of low-power modes varies with the hardware at hand.

Dynamic (heap) memory is allocated during run-time, e.g. when posting messages and when creating objects. It is imperative that memory that is no longer reachable will be recovered at some point in time. As explained above, a Timber system is either executing scheduled methods or being idle. Hence, when the system is in idle state, we can safely perform garbage collection without violating any timing constraints. However, in the general case we cannot guarantee that garbage collection will terminate before a method becomes schedulable. This calls for the use of interruptible garbage collection schemes. Observe that in the idle state all method invocations have run-to-end. That implies that all memory stacks are empty. This fact can be used to simplify garbage collection procedures. We refer the reader to [17] for a comprehensive overview of garbage collection algorithms based on their relation to hard real-time systems and Timber in particular. Furthermore, [17] describes an experimental implementation of an interruptible reference counting garbage collector for Timber.

The current Timber compiler takes a Timber program as input and generates a platform independent implementation in the form of C code. The C code defines object instantiation

procedures and object methods. The generated C code is then compiled and linked with the hardware specific HAL, the real-time scheduler and the garbage collector. The result is an executable that runs on a bare system. Thus, the generated software is a complete implementation of the initial system description and does not add any components of uncertainty that external code such as “external” operating systems and “third party” device drivers may invoke. This lead us to the conclusion that properties derived from system analyses on the specification will hold also for the target system implementation. We foresee implications to many areas; low-power system design, validation, verification and robustness; system dimensioning and context awareness; fault detection and system maintenance etc.

In this section we have demonstrated how a Timber specification can be turned into a system implementation. Recapitulation and conclusions;

- *Timber Specification*: A description in Timber can be viewed as a formal model for the system implementation. The designer may also choose to model the system’s environment in Timber. Using environment models, stimuli for functional validation can be automatically generated and the simulation results analyzed.
- *Timber Implementation*: A system description in Timber can for a given target system be translated to an efficient executable (consisting of a real-time scheduler, memory manager and code that implements the system’s reactions). Thus, the generated software is a complete implementation of the initial system description and does not add any components of uncertainty. This leads to the conclusion that properties derived from system analyses will hold also for the target system. Given appropriate tools for analyses, we may e.g. derive the least complex target system that meet the resource demands of the initial specification.

2.3 Tools of the Trade

In previous sections we have presented Timber as a way to capture reactive systems with hard real-time constraints. We have shown how a Timber specification can be turned into an efficient executable for a target platform. We have concluded that properties derived from the specification will hold also for the implementation. In the following section we give an example showing how theoretical program analysis has direct implication to deriving worst case execution time (WCET) for the schedulable units of a Timber program.

It is common that embedded systems are designed as hard *real-time systems* and the only way to guarantee that a system has enough capacity to ensure functionality even under extreme conditions is through analysis. One approach is to apply schedulability analysis on the system and use that information as a basis for safe system dimensioning. Schedulability analysis requires knowledge about the *worst-case execution time* (WCET) for the schedulable units as well as a timely model of the environment. An important part of WCET analysis is the problem of establishing loop bounds, recursion depth and execution paths automatically.

In [18, 18, 20] Liu and Gómez introduce a *language-based* approach to solve the problem with loop bounds, recursion depth and execution paths automatically. They use transformations to create *time-bound functions* which together with *partially known input structures* can accurately estimate the WCET for a program. This is conveniently done at the source-language level. However, the language used by Liu and Gómez is a purely functional language, lacking support for e.g. state variables, parallelism and interrupt handling; features typically used in real-time systems. Timber, on the other hand, captures state vari-

ables and parallelism by the concept of reactive objects. The schedulable units of a Timber program are the reactions caused by events, and the reactive behaviour of a program is a direct result from the reactive behaviour of the programs individual schedulable units. In [21] we have extended the language based approach [18, 19] to a sufficient subset of Timber to create a number of time-bound functions that will give the execution time for each schedulable unit.

Architectures for Embedded Internet Systems (EIS) are being developed at EISLAB, Luleå University of Technology [20]. This work includes an experimental platform based on the Renesas M16 microcontroller [21]. The experiments conducted in [21] show that the theoretical WCET (derived through program analysis and cycles/instruction taken from [24]), give a safe and accurate estimation of actual WCET (derived through worst-case simulations using [25]). Language based WCET analysis is an important step towards full system schedulability analysis and system dimensioning.

3 Conclusions

Embedded systems are often operating under hard real-time constraints, while at the same time being constrained by severe restrictions on power consumption. For such systems, robustness and reliability can be a question of life and death which calls for rigorous system design and methodologies for dimensioning and validation. In this paper we have discussed a number of key problems associated with traditional design of low-power embedded real-time systems; specification, state integrity/blocking, memory management and system dimensioning/verification.

We have presented an alternative design methodology for low-power, real-time systems based on Timber; a pure reactive system model that allows for formal reasoning about various system properties. Timber allows the timely behaviour to be captured into reactive objects that represent a unification of the object-oriented and parallel programming paradigms. Timber offers dynamic (heap based) memory management, such revealing the programmer from the heavy burden of correct manual memory management.

We have outlined how a Timber specification can be turned into an efficient executable for a target platform and that properties derived from the specification will hold also for the implementation. A method for WCET resource analysis has been reviewed and the implication to system dimensioning and validation has been further discussed.

We conclude that Timber holds the potential to challenge the daemons of traditional design methodology. Ongoing research further explores compiler design, memory management and the potential of formal methods.

References

- [1] TinyOS official homepage, <http://www.tinyos.net/>, 2004.
- [2] A. Dunkels, B. Grönvall, T. Voigt, “Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors”, *Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004*, Tampa, Florida, USA, November 2004.
- [3] VxWorks documentation, <http://www.windriver.com/>, 2004
- [4] RTLinux official homepage, <http://www.fsmlabs.com/>, 2004
- [5] QNX official homepage, <http://www.qnx.com/>, 2004
- [6] Java Community Press (JCP), Java Specification Request (JSP), “<http://www.jcp.org/jsr/detail/1.jsp>”, 2004
- [7] E. Kligerman, A. D. Stoyenko, “Real-Time Euclid: A Language for Reliable Real-time Systems”, *IEEE Transactions on Software Engineering*, SE-12(9), 1986.
- [8] Y. Ishikawa et al, “Object-Oriented Real-Time Language Design: Constructs for Timing Constraints”, *SIGPLAN Notices*, 25(10):289–298, Oct 1990.
- [9] A. D. Stoyenko, T. J. Marlowe, M. F. Younis, “A Language for Complex Real-Time Systems”, *The Computer Journal*, 38(4), 1995.
- [10] J. Hooman, O. van Roosmalen, “An Approach to Platform Independent Real-Time Programming”, *Real-Time Systems, Journal of Time-Critical Computing Systems*, 19(1):61–112, 2000.
- [11] T. Ritzau, “Memory Efficient Hard Real-Time Garbage Collection”, Ph.D. Thesis, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 2003.
- [12] A. P. Black, M. Carlsson, M. P. Jones, R. Kieburtz, J. Nordlander, “Timber: A Programming Language for Real-Time Embedded Systems”, Technical Report, CSE-02-002, 2002.
- [13] M. Carlsson, J. Nordlander, D. Kieburtz, “The semantic layers of Timber”, In *Atsushi Otori, editor, Programming Languages and Systems*, First Asian Symposium, APLAS 2003, Beijing, China, volume 2895 of Lecture Notes in Computer Science. Springer, November 2003.
- [14] J. Nordlander, “Reactive Objects and Functional Programming”, Ph.D. Thesis, Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden, ISBN 91-7197-823-2, May 1999.
- [15] J. Nordlander, M. P. Jones, M. Carlsson, R. B. Kieburtz, A. Black, “Reactive Objects”, *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2002.
- [16] The Haskell Home Page, <http://www.haskell.org/>, Last visited 31 October 2004.
- [17] J. Mattsson, supervisors P. Lindgren and J. Nordlander, “Garbage collection with hard real-time requirements”, Master Thesis SSN 1402-1617 / ISRN LTU-EX--04/262--SE / NR 2004:262, <http://publ.luth.se/1402-1617/2004/262/index.html>, 2004.
- [18] Y. A. Liu, G. Gómez, “Automatic Accurate Time-Bound Analysis for High-Level Languages”, *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp. 31–40, June 1998.
- [19] Y. A. Liu, G. Gómez, “Automatic Accurate Cost-Bound Analysis for High-Level Languages”, *IEEE Transactions on Computers*, Volume 50, Issue 12, pp. 1295–1390, December 2001.
- [20] G. Gómez, Y. A. Liu, “Automatic Time-Bound Analysis for a Higher-Order Language”, *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 75–86, 2002.

- [21] L. Svensson, J. Eriksson, P.Lindgren, J. Nordlander, “Language-Based WCET Analysis of Reactive Programs”, *Submitted to the 17th Euromicro Conference on Real-Time Systems*, Palma de Mallorca, Spain, June 2005.
- [22] Å. Östmark, “Embedded Internet System Architectures”, Licentiate Thesis, Department of Computer Science and Electrical Engineering, Luleå University of Technology, Luleå, Sweden, ISSN 1402-1757, July 2004.
- [23] Mitsubishi Electric Corporation, “M16C/62M Group (Low voltage version) Data Sheet REV.B1”, 2001.
- [24] Sales Strategic Planning Div. Renesas Technology Corp., “Renesas 16-Bit Single-Chip Microcomputer Software Manual M16C/60, M16C/20, M16C/Tiny Series, rev.4.00”, 21 January 2004.
- [25] Renesas, Simulator debugger, M3T-PD30SIM, version 5.20, release 1, 2003.