

# Timber

timber-lang.org

Johan Nordlander, Björn von Sydow, Per Lindgren,  
Magnus Carlsson, and Andy Gill

Timber is a general programming language specifically aimed at the construction of complex **event-driven** systems. It allows programs to be conveniently structured in terms of **objects** and **reactions**, and the real-time behavior of reactions can furthermore be precisely controlled via platform-independent **timing constraints**. This property makes Timber particularly suited to both the specification and the implementation of real-time embedded systems.

## 1 History

Timber is deeply rooted in the *functional programming* tradition, although it also draws heavily on *object-oriented* concepts, and has the notion of concurrent execution built into its core.

Timber is a direct descendant of *O'Haskell*, which was developed at Chalmers University of Technology in 1999 [2]. O'Haskell extended the lazy functional programming language Haskell [1] with object-oriented concepts such as methods, classes and subtyping, while retaining the purely functional execution model of its ancestor. It also introduced the characteristic notion of concurrent reactive objects that Timber subsequently has adopted.

The Timber language attained its main shape during the *Timber project*, which was run at the Oregon Graduate Institute between 2000 and 2003 as part of the DARPA PCES program (Program Composition for Embedded Systems) [4]. Here the semantics of time-constrained reactions was developed, and it was also decided to abandon the lazy execution model of Haskell and O'Haskell in favor of a more standard (i.e., strict) parameter passing mechanism; primarily for the purpose of facilitating more predictable execution times. A prototype Timber interpreter was developed at this time, based on a similar implementation of the O'Haskell language.

After 2003, work on the Timber language regrettably had to continue at a slower pace, primarily concentrating on completing the implementation of the Timber compiler. Since 2007, however, the project has regained momentum, and the language is now being actively developed and maintained by groups and individuals at Luleå University of Technology, Chalmers University of Technology, University of Kansas and Portland State University.

Version 1.0 of the Timber compiler was publicly released in the summer of 2008.

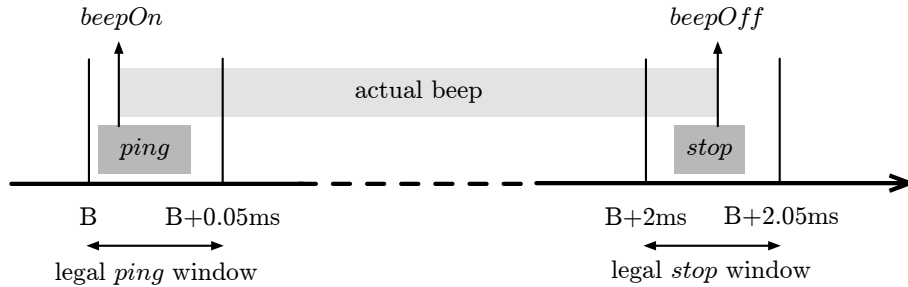


Figure 1: Sonar beep timing windows. The pattern repeats at  $B+3s$ ,  $B+6s$ , ...

## 2 Description

The overall purpose of a Timber program is to *react to events* sent to it from its execution environment. This process is potentially infinite, and the order of external events is also not generally known in advance.

To capture this intuition, Timber defines its primary run-time structure to be a set of interconnected *reactive objects*, that each encapsulate a piece of the global program state. A reactive object is a passive entity defined by a set of methods, whose relative execution order is left to be determined by clocks and external events. Between invocations, a Timber object just maintains its state, ready to react when a method call occurs. Like real-world objects, Timber objects evolve in parallel, although the methods belonging to a particular object are always run under mutually exclusion. Concurrency as well as state protection is thus implicit in Timber, and does not require any direct mentioning of threads or other concurrency constructs.

Objects are created by instantiating a **class**. Methods are either *asynchronous* or *synchronous*, as denoted by the keywords **action** and **request**, respectively. The most radical property of Timber is that it is free from any indefinitely blocking constructs; because of this, a Timber object is always fully responsive when not actively executing code. This process structure should be contrasted to the common infinite event-loop pattern in other languages, where blocking system calls are used to partition an otherwise linear thread of execution into event-handling fragments.

Each reaction in Timber is furthermore associated with programmable *timing window*, delimited by upper and lower constraints called the *baseline* and the *deadline* of a reaction. The semantics of these constraints is formally defined like the rest of the language, and serve the purpose of codifying the legal behavior of a Timber system in platform-independent manner. The timing window is inherited by each method call by default, but can also be manually set by the programmer; using the constructs **after** for moving a window forward with an offset, and **before** for setting a timing window width. Both values are measured relative to the fixed reference-points of baselines, never from the point in time a method call is actually made. Posting a message to arrive at some specific time into the future is thus easy in Timber, and defining a periodic process amounts to the special case of repeating such a pattern recursively.

The following Timber example shows a simple implementation of a *sonar*

*driver* that is coupled to an alarm. The specifications assumed state that a sonar beep should be 2 ms long, with a maximum jitter of 50  $\mu$ s, and that the required accuracy of the measurements dictate that time-stamps associated with beeps must also be accurate down to the 50  $\mu$ s range. Figure 1 illustrates the timing windows constraining the involved methods *ping* and *stop*, and thus, indirectly, the actual beep produced.

Furthermore, the sonar is supposed to sound every 3 seconds, and the deadline for reacting to off-limit measurements is 5 ms. These specifications look as follows when translated into Timber code:

```

sonar port alarm = class
  t := Genesis
  ping = before (USec 50) action
    port.write beepOn
    t := baseline
    after (MSec 2) stop
    after (Sec 3) ping
  stop = before (USec 50) action
    port.write beepOff
  echo = before (MSec 5) action
    distance = k * (baseline - t)
    if distance < limit then
      alarm
  result {interrupt = echo; start = ping}

```

The sequence *sonar port alarm* here means the definition of function *sonar* taking *port* (the port controlling the beeping hardware) and *alarm* (the method to call in emergency) as parameters. The shown **class** construct initializes state variable *t* to *Genesis* (the earliest point in time a system can express), and furthermore defines the asynchronous actions *ping*, *stop* and *echo*. The resulting interface is a record containing the methods names *interrupt* and *start*, which are just exported aliases for the local methods *echo* and *ping*, respectively.

The actual baseline of the ongoing reaction is available as an identifier *baseline* within each method. In *ping* this value is sampled as state variable *t*, and when *echo* is invoked, this *t* is compared to the baseline of that timing window. Notice also how *ping* ensures two future events will be triggered: one call to *stop* 2 ms from the current baseline, and a recursive invocation of *ping* itself after 3 s to keep the periodic sonar activity alive.

As an example of how a sonar object might be instantiated to run on a bare-metal embedded system, here follows an example of the *root* declaration for such a system.

```

root regs = class
  a = new alarm (regs!com_port_addr)
  s = new sonar (regs!sonar_port_addr) a
  result [s.start, s.interrupt, a.ack]

```

On this level, the interface to the software (as seen from the hardware) is the list of interrupt handlers it provides (enclosed in square brackets). Consequently (from the software perspective), the hardware takes the shape of an array of device registers that can be read or written (with ! being the indexing operator).

The parametric class *root* above is thus of a type that can be instantiated and used directly by the run-time system of a typical bare-metal platform, although it should be noted that the exact shape of a root definition can vary between Timber platforms – under the POSIX environment, for example, the root class returns just a startup procedure, while taking a full-featured operating system interface as an argument.

The semantics of Timber allows a natural implementation in terms of an EDF scheduler, where the asynchronous methods of a program correspond to tasks, objects take the role of shared resources, and all methods (synchronous as well as asynchronous) simply denote code sequences that require exclusive access to the owning object. Timber also relegates management of its garbage-collected heap to idle time, thus facilitating direct application of known schedulability and execution time analysis methods. The implementation technique used furthermore allows both objects and messages to be created in large numbers without incurring any run-time penalty, at the same time as the number of threads behind the scenes (i.e., the number of execution contexts and stacks) can be limited to the maximum preemption depth a system is expected to need.

Additional features of Timber not covered here include a strong static type system supporting subtyping, parametric polymorphism with overloading, and automatic type inference; a rich set of heap-allocated immutable datatypes; first-class citizenship to all values (meaning that methods as well as classes can be sent as arguments and stored inside data structures); and a referentially transparent evaluation semantics in the purely functional tradition.

### 3 Conclusions and Future

Timber actually has a longer history than what is indicated by its publicly available implementation, and it has already begun to demonstrate its ability to seamlessly integrate the different perspectives of functional programming, object-oriented modeling & design, predictable real-time scheduling, and efficient run-time execution.

Future plans for the language include more extensive libraries and documentation, a growing number of ports to popular embedded platforms, and experimentation with more resource-aware implementation techniques such as SRP. Another goal is to offer tight integration between Timber and existing timing analysis tools such as SymTA/S [3]. In a slightly different direction, the potential for Timber objects to be used as plug-in models inside simulation environments such as Simulink will also be explored.

For further information, visit [timber-lang.org](http://timber-lang.org)

### References

- [1] The Haskell Home Page. <http://haskell.org>
- [2] J. Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden, May 1999.
- [3] Symtavision SymTA/S. <http://symtavision.com/>
- [4] Project Timber. <http://www.cse.ogi.edu/PacSoft/projects/Timber/>