The semantic layers of Timber

Magnus Carlsson[1], Johan Nordlander[2], and Dick Kieburtz[1]

[1] Oregon Health & Science University, {magnus,dick}@cse.ogi.edu
[2] Luleå University of Technology, nordland@sm.luth.se

Abstract. We present a three-layered semantics of *Timber*, a language designed for programming real-time systems in a reactive, object-oriented style. The innermost layer amounts to a traditional deterministic, pure, functional language, around which we formulate a middle layer of concurrent objects, in terms of a monadic transition semantics. The outermost layer, where the language is married to deadline-driven scheduling theory, is where we define message ordering and CPU allocation to actions. Our main contributions are a formalized notion of a *time-constrained reaction*, and a demonstration of how scheduling theory, process calculii, and the lambda calculus can be jointly applied to obtain a direct and succinct semantics of a complex, real-world programming language with well-defined real-time behavior.

1 Introduction

Timber is a new programming language being developed jointly at the Oregon Health & Science University, Chalmers University of Technology, and Luleå University of Technology. The scope of Timber is wide, ranging from low-level device interfaces, over time-constrained embedded systems or interactive applications in general, to very high-level symbolic manipulation and modeling applications. In the context of this text, the distinguishing attributes of the Timber language can be summarized as follows:

- It is based on a programming model of *reactive objects*, that abandons the traditional active request for input in favor of a purely event-driven program structure [15].
- It fully integrates *concurrency* into its semantic foundation, making each object an encapsulated process whose state integrity is automatically preserved.
- It enables *real-time constraints* to be directly expressed in the source code, and opens up the possibility of doing off-line deadline-based schedulability analysis on real code.
- It is a full-fledged pure functional language that achieves referential transparency in the presence of mutable data and non-determinism by means of a monad.
- It upholds *strong static type safety* throughout, even for its message-based, tag-free interprocess communication mechanism.

In this paper we provide a formal semantic definition of Timber. The combination of object-based concurrency, asynchronous reactivity, and purely functional declarativeness is challenging in itself, but we believe it is the existence of a real-time dimension that makes the question of a formal semantics for Timber particularly interesting. The core of our approach is the definition of a *layered* semantics, that separates the semantic concerns in such a way that each layer is meaningful and can be fully understood by referring to just the layer before it. The semantic layers of Timber can be summarized as follows:

- *The functional layer*. This layer amounts to a traditional pure functional language, similar to Haskell [17] or the pure subset of ML [10]. By pure, we mean that computations in this layer do not cause any effects related to state or input/output. We will not discuss this layer very much in this paper, but instead make sure that the other layers do not interfere with it.
- *The reactive layer* is a system of concurrent *objects* with internal state that *react* to messages passed from the environment (input), and in turn send synchronous or asynchronous *messages* to each other, and back to the environment (output). This layer constitutes a form of process calculus that embeds a purely functional sub-language. Although it abstracts away from the flow of real time, the calculus does associate with each message precise, but uninterpreted time constraints.
- *The scheduling layer*. The reactive layer leaves us with a set of messages and objects with time constraints. Some of the messages may compete for the same objects, which in turn are competing for computing resources. The scheduling layer makes precise what the constraints are for the resulting scheduling problem.

The semantic definition has been used to implement an interpreter for Timber, by which we have been able to conduct serious practical evaluations of the language (spanning such diverse areas as GUI toolkit implementation [13] and embedded robot control [7]). A compiler aimed at stand-alone embedded systems is also being developed in tandem with the language specification. We consider the main contributions of this paper to be: (1) a formal definition of the *time-constrained reactions* that give Timber its distinct flavor, and (2) a demonstration of how scheduling theory, process calculii, and the lambda calculus can be jointly applied to obtain a direct and succinct semantics of a complex, real-world programming language with well-defined real-time behavior.

The rest of the paper is organized as follows: Section 2 introduces Timber with an example of an embedded controller. Section 3 constitutes the core of the paper, as it defines the semantics of Timber in three separate subsections corresponding to each semantic layer. Related work is then discussed in Section 4, before we conclude in Section 5.

2 An embedded Timber system

Although Timber is a general-purpose programming language, it has been designed to target *embedded systems* in particular. The software of such a system,

when written in Timber, consists of a number of *objects* which mostly sit and wait for incoming *events* from the physical environment of the embedded system. On the lowest level, events are picked up by peripheral devices, and are transformed into interrupts injected into the CPU. Each interrupt is converted into an *asynchronous message* that has one particular object in the Timber program as its destination. Each message is associated with a *time constraint* in the form of a *baseline* and a *deadline*, both of which are absolute times. The baseline constrains the starting point of a reaction and also functions as a reference point to which time expressions might subsequently relate; it is normally set to the absolute time of the interrupt. The deadline is the time by which the system must have *reacted* to the event for the system to behave correctly. In the tradition of declarative programming , this parameter thus acts as a specification for a correct Timber implementation, ultimately derived from the time constants of the physical environment with which the system interacts.

Once an interrupt has injected a message into a Timber program, the system is no longer in an idle state. The message has *excited* the system, and the destination object starts reacting to the message so that a response can be delivered. As a result, the object may alter its state, and send secondary messages to other objects in the system, some of which may be *synchronous*. Synchronous communication means that the object will rendezvous with the destination object, facilitating two-way communication. Some of the secondary messages generated during the excited state can have baselines and deadlines that are larger than the original, interrupt-triggered message. For example, a car alarm system may react immediately to an event from a motion sensor by turning on the alarm, but also schedule a secondary, delayed reaction that turns off the alarm after a minute.[3] However, eventually the chain reaction caused by an external event will cling out, and the system goes back to an idle state.

Because a Timber system is concurrent, multiple reactions caused by independent external events may be in effect at the same time. It is the job of the presented semantics to specify how the interactions within such a system are supposed to behave.

2.1 Templates, actions and requests

The syntax of Timber is strongly influenced by Haskell [17]. In fact, it stems directly from the object-oriented Haskell extension *O'Haskell*, and can be seen as a successor of that language [12].

To describe Timber more concretely, we present a complete Timber program that implements the car alarm refereed to above in Figure 1. This program is supposed to run on a naked machine, with no other software than the Timber program itself. On such a machine, the environment provides merely two operations, as the definition of the record type `Environment` indicates: one for writing a byte to an I/O port, and one for reading. These methods are synchronous, as indicated by the monadic type constructor `Request`. On the other

[3] A delay perceived as much longer than a minute, though!

hand, the interface a naked program presents to its environment will merely consist of handlers for (a subset of) the interrupts supported by the hardware. This is expressed as a list of asynchronous methods (actions) paired with interrupt numbers, as the type `Program` prescribes.

```
record Environment where
   write      :: Port -> Byte -> Request ()
   read       :: Port -> Request Byte

record Program where
   irqvector  :: [(Irq,Action)]

alarm :: Environment -> Template Program
alarm env =
  template
     trigged := True
  in let
     moved = before (100*milliseconds) action
        if trigged then
           env.write siren 1
           trigged := False
           after (1*minutes) turnoff
           after (10*minutes) enable
     turnoff = action
        env.write siren 0
     enable = action
        trigged := True
  in record
     irqvector = [(motionsensor,moved)]
```

Fig. 1. The car alarm program.

A Timber program is a function from the program environment to a *template* that returns an interface to an object. At boot time, the template `alarm` will be executed, thereby creating one instance of an alarm handling object, whose interface is the desired interrupt vector.

We assume that there is a particular I/O port `siren`, whose least significant bit controls the car siren, and that the interrupt number `motionsensor` is associated with the event generated by a motion sensor on the car.

The definition of `alarm` reveals a template for an object with one internal state variable `trigged`, initialized to `True`. The purpose of `trigged`, as we will see, is to ensure that once the alarm goes off, there will be a grace period until it is possible to trigger the alarm again.

The returned interrupt vector associates the action `moved` with the motion sensor interrupt. When `moved` is invoked, and if the alarm is trigged, it will turn on the siren, and set `trigger` to false. It will also invoke two local asynchronous methods, each one with a lower time bound on the actual message reception. The first message, which is scheduled to be handled one minute after the motion sensor event, simply turns off the siren. The second message, which will arrive after ten minutes, re-enables the alarm so that it may go off again.

By means of the keyword **before**, action `moved` is declared to have a deadline of one tenth of a second. This means that a correct implementation of the program is required to turn on the alarm within 100 milliseconds after the motion sensor event. This deadline also carries over to the secondary actions, which for example means that the alarm will shut off no later than one minute and 0.1 seconds after the motion sensor event (and no earlier than one minute after the event, by the lower time bound).

The handling of delays in our example captures a characterizing aspect of the reactive semantics of Timber. Instead of temporarily halting execution at some traditional "sleep" statement, our method `moved` will terminate as quickly as the CPU allows, leaving the alarm object free to react to any pending or subsequent calls to `moved`, even while the 1 and 10 minute delay phases are active. A degraded motion sensor generating bursts of interrupts will thus be handled gracefully by our program; something that would require considerably more work in a language where events are queued and event-handling can be arbitrarily delayed.

```
alarm env = do
  self <- new True
  let moved = act self ⟨0,100*milliseconds⟩
                (do trigged <- get
                    if trigged then do
                       env.write sirenport 1
                       set False
                       aft (1*minutes) turnoff
                       aft (10*minutes) enable
                    else return () )
      turnoff = act self ⟨0,0⟩
                (env.write sirenport 0)
      enable = act self ⟨0,0⟩
                (set True)
  return ( record irqs = [(motionsensor,moved)] )
```

Fig. 2. The car alarm program, desugared.

2.2 The O monad

```
return:: a -> O s a
(>>=) :: O s a -> (a -> O s b) -> O s b
handle:: O s a -> (Error -> O s a) -> O s a
raise :: Error -> O s a
bef   :: Time -> O s a -> O s a
aft   :: Time -> O s a -> O s a
set   :: s -> O s ()
get   :: O s s
new   :: s -> O s' (Ref s)
act   :: Ref s -> (Time,Time) -> O s a -> O s' Msg
abort :: Msg -> O s ()
req   :: Ref s -> O s a -> O s' a
```

Fig. 3. The constants in the O monad.

The Timber constructs **template**, **action**, and **request** are nothing but syntactic sugar for monadic computations that take place in the `O` monad, as described in [14, 12]. The type `O` s a stands for a computation that can be executed inside an object whose local state has type s, resulting in a value of type a. In the full Timber language, *polymorphic subtyping* is applied to give constants that are independent of the local state a more flexible type (see for example the types `Action` and `Request` mentioned above) [13]. However, we will ignore this issue in the following presentation, as it has no operational implications.

A desugared version of the alarm program is given in Figure 2. Strictly speaking, the program is not a literate result of the desugaring rules—for readability, we introduced the do-notation and performed some cosmetic simplifications.

The desugared program refers to a number of primitive operations in the O monad, whose type signature are given in Figure 3. An informal meaning of these constants can be obtained by comparing the desugared program in Figure 2 with the original. Their precise formal meaning will be the subject of the next section. The actual desugaring rules are given in Appendix A.

3 The semantic layers

3.1 The functional layer

We will not specify very much of the functional layer in this presentation; instead we will view it as a strength of our layered approach that so much of the first layer can be left unspecified. One of the benefits with a monadic marriage of effects and evaluation is that it is independent of the evaluation semantics—this property has even been proposed as the definition of what *purely functional* means in the presence of effects [19].

$$
\begin{aligned}
\texttt{return}\ e_1\ \texttt{>>=}\ e_2 &\mapsto e_2\ e_1\\
\texttt{raise}\ e_1\ \texttt{>>=}\ e_2 &\mapsto \texttt{raise}\ e_1\\
\texttt{return}\ e_1\ \texttt{`handle`}\ e_2 &\mapsto \texttt{return}\ e_1\\
\texttt{raise}\ e_1\ \texttt{`handle`}\ e_2 &\mapsto e_2\ e_1\\
\texttt{bef}\ d'\ (\texttt{act}\ n\ \langle b, d\rangle\ e) &\mapsto \texttt{act}\ n\ \langle b, d'\rangle\ e\\
\texttt{aft}\ b'\ (\texttt{act}\ n\ \langle b, d\rangle\ e) &\mapsto \texttt{act}\ n\ \langle b', d\rangle\ e
\end{aligned}
$$

Fig. 4. Functional layer: reduction rules.

That said, it is also the case that a lazy semantics in the style of Haskell adds extra difficulties to the time and space analysis of programs, something which is of increased importance in the construction of real-time and embedded software. For this reason, we are actually shifting to a strict semantics for Timber, which is a clear breach with the tradition of its predecessors. Still, strictness should not be confused with impurity, and to underline this distinction, we will assume a very general semantics that is open to both both lazy and strict interpretation in this paper. We hope to be able to report on the specifics of strictness in Timber in later work, especially concerning the treatment of recursive values and imprecise exceptions [4, 11, 8].

We assume that there is a language $\mathcal{E}$ of expressions that includes the pure lambda calculus, and whose semantics is given in terms of a small-step evaluation relation $\mapsto$. Let e range over such expressions. Moreover, we assume that the expressions of $\mathcal{E}$ can be assigned types by means of judgments of the form $\Gamma \vdash e : \tau$, where τ ranges over the types of $\mathcal{E}$, and Γ stands for typing assumptions.

A concrete example of what $\mathcal{E}$ and $\mapsto$ might look like can be found in [12]. There a semantics is described that allows concurrent reduction of all redeces in an expression, even under a lambda. We will neither assume nor preclude such a general semantics here, but for the purpose of proving type soundness of the reaction layer, we require at least the following property:

Theorem 1 (Subject reduction). *If $\Gamma \vdash e : \tau$ and $e \mapsto e'$, then $\Gamma \vdash e' : \tau$.*

In order to connect this language with the reactive semantics of the next section, we extend $\mathcal{E}$ with the constants of Figure 3, and the extra evaluation rules of Figure 4.

The first four constants listed in Figure 3 constitute a standard exception-handling monad, for which meaning is given by the first four rules of Figure 4. Likewise, the constants `bef` and `aft` merely serve to modify the time constraint argument of the `act` constant, so their full meaning is also defined in the functional layer.

The remaining constants, however, are treated as *uninterpreted constructors* by the functional semantics—it is giving meaning to those constants that is the purpose of the reactive semantic layer. Similarly, concrete realization of the type constructors `O`, `Ref` and `Msg` is postponed until the reactive layer is defined; as far as the functional layer is concerned, `O`, `Ref` and `Msg` are just opaque, abstract types.

3.2 The reactive layer

To give semantics to the reactive layer, we regard a running Timber program as a system of concurrent processes, defined by the grammar in Figure 5. A primitive process is either

- an *empty message*, tagged with the name m. This is the remnant of a message that is either delivered or aborted;
- a *message* tagged with m, carrying the code e, to be executed by an object with name n, obeying the time constraint c. In the case of a synchronous message, it also carries a *continuation* K, which amounts to a suspended requesting object (see below);
- an *idle object* with identity n, which maintains its state s awaiting activation by messages;
- an *active object* with identity n, state s, executing the code e with the time constraint c. In case the object is servicing a synchronous request, the continuation K amounts to the suspended requesting object.

$P ::= \langle\rangle_m$		Empty message
	$\langle n, e, K\rangle^c_m$	Pending message
	$(\!\lvert s \rvert\!)_n$	Idle object
	$(\!\lvert s, e, K \rvert\!)^c_n$	Active object
	$P \parallel P$	Parallel composition
	$\nu n.\, P$	Restriction

Fig. 5. The process grammar.

Finally, processes can be composed in parallel, and the scope of names used for object identities and message tags can be restricted. We will assume that parallel composition has precedence over the restriction operator, which therefore always extend as far to the right as possible.

Continuations are requesting objects that are waiting for a request (synchronous message) to finish. Since a requesting object can in turn be servicing requests from other objects, continuations are defined recursively, according to the following grammar that denotes a *requesting object* or an *empty continuation*:

$$K ::= (\!\lvert s, \mathcal{M}, K \rvert\!)_n \mid 0$$

A requesting object with identity n contains the state s, and a *reaction context* $\mathcal{M}$, which is an expression with a hole, waiting to be filled by the result of the

request. Just as for normal objects, there is a continuation K as well, in case the requesting object is servicing a request from another object. For asynchronous messages, and objects that are not servicing requests, the continuation is empty.

Reaction contexts are used in two ways in the reactive layer. As we have already seen, they are used in an unsaturated way to denote expressions with holes in requesting objects. They are also used in the following section to pinpoint where in an expression the reaction rules operate. Reaction contexts are given by the grammar

$$\mathcal{M} ::= \mathcal{M} \texttt{ >>= } e \mid \mathcal{M} \texttt{ `handle` } e \mid [\,] \ .$$

The structural congruence relation In order for the relation to bring relevant process terms together for the reaction rules given in the next section, we assume that processes can be rearranged by the *structural congruence* $\equiv$,

Associativity	$P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3$
Symmetry	$P_1 \parallel P_2 \equiv P_2 \parallel P_1$
Scope extension	$P_1 \parallel \nu n.P_2 \equiv \nu n.P_1 \parallel P_2$ if $n \notin fv(P_1)$
Scope commutativity	$\nu n.\nu m.P \equiv \nu m.\nu n.P$
Renaming	$\nu n.P \equiv \nu m.[^m/_n]P$ if $m \notin fv(P_1)$
Inert message	$P \parallel \nu m.\langle\rangle_m \equiv P$
Inert object	$P \parallel \nu n.(\!|s|\!)_n \equiv P$

Fig. 6. The structural congruence.

induced by the elements in Figure 6. These allow for the process terms to be rearranged and renamed, so that *e.g.* a message and its destination object can be juxtaposed for the relevant rule to apply. The last two elements of the equivalence allow us to garbage collect inert objects or messages that cannot possibly interact with any other process in the system. More specifically, an idle object whose identity is unknown to the rest of the system cannot possibly receive any message. Similarly, an empty message whose tag is "forgotten" can be eliminated.

The reactive relation The reactive layer of our semantics consists of a *reaction relation* $\longrightarrow$, that defines how objects interact with messages and alter internal state. The reaction relation is characterized by the axioms shown in Figure 7, which together with the structural rules in Figure 8 define $\longrightarrow$ for general process terms. Most of the axioms use a reduction context to pinpoint the next relevant O monad constant inside an active object.

Rules Set and Get allows for the local state of an object to be written or read. The next three rules introduce a fresh name on the right-hand side by using restriction, and we tacitly assume here that the names m, n' do not occur free in the process term on the left-hand side.

Rule New defines creation a new, idle object whose identity is returned to the creator, and with an initial state as specified by the argument to the constant `new`.

$$
\begin{array}{lrl}
\textsc{Set} & (\!|s, \mathcal{M}[\texttt{set}\ e], K|\!)^c_n & \longrightarrow (\!|e, \mathcal{M}[\texttt{return}\ ()], K|\!)^c_n \\
\textsc{Get} & (\!|s, \mathcal{M}[\texttt{get}], K|\!)^c_n & \longrightarrow (\!|s, \mathcal{M}[\texttt{return}\ s], K|\!)^c_n \\
\textsc{New} & (\!|s, \mathcal{M}[\texttt{new}\ e], K|\!)^c_n & \longrightarrow \nu n'.\, (\!|s, \mathcal{M}[\texttt{return}\ n'], K|\!)^c_n \parallel (\!|e|\!)_{n'} \\
\textsc{Act} & (\!|s, \mathcal{M}[\texttt{act}\ n'\ d\ e], K|\!)^c_n & \longrightarrow \nu m.\, (\!|s, \mathcal{M}[\texttt{return}\ m], K|\!)^c_n \parallel \langle n', e, 0\rangle^{c+d}_m \\
\textsc{Req} & (\!|s, \mathcal{M}[\texttt{req}\ n'\ e], K|\!)^c_n & \longrightarrow \nu m.\, \langle n', e, (\!|s, \mathcal{M}, K|\!)_n\rangle^c_m \\
\textsc{Run} & (\!|s|\!)_n \parallel \langle n, e, K\rangle^c_m & \longrightarrow (\!|s, e, K|\!)^c_n \parallel \langle\rangle_m \\
\textsc{Done} & (\!|s, r\ e, 0|\!)^c_n & \longrightarrow (\!|s|\!)_n \quad \textit{where } r \in \{\texttt{raise}, \texttt{return}\} \\
\textsc{Rep} & (\!|s, r\ e, (\!|s', \mathcal{M}, K|\!)_m|\!)^c_n & \longrightarrow (\!|s|\!)_n \parallel (\!|s', \mathcal{M}[r\ e], K|\!)^c_m \\
 & & \quad \textit{where } r \in \{\texttt{raise}, \texttt{return}\} \\
\textsc{Abort} & (\!|s, \mathcal{M}[\texttt{abort}\ m], K|\!)^c_n \parallel P & \longrightarrow (\!|s, \mathcal{M}[\texttt{return}\ ()], K|\!)^c_n \parallel \langle\rangle_m \\
 & & \quad \textit{where } P \in \{\langle n', e, 0\rangle^{c'}_m, \langle\rangle_m\}
\end{array}
$$

Fig. 7. Reactive layer: axioms.

In rule Act, an object is sending an asynchronous message with code e to a destination object n'. The time constraint, or *timeline*, of a message is computed by adding the *relative* time constraint d to the constraint of the sending object. The addition of time constraints is described in more detail in the section 3.2. Note that, as a consequence of the substitution-based evaluation semantics we have assumed, messages contain the actual code to be run by destination objects, instead of just method names. This should not be confused with breaking the abstraction barrier of objects, since object interfaces normally expose only action and request values, not the object identifiers themselves. Without knowledge of an object's identity, it is impossible to send arbitrary code to it.

$$
\frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \longrightarrow Q}\ \textsc{Equiv}
$$

$$
\frac{P \longrightarrow P'}{P \parallel Q \longrightarrow P' \parallel Q}\ \textsc{Par}
$$

$$
\frac{P \longrightarrow P'}{\nu n.\, P \longrightarrow \nu n.\, P'}\ \textsc{Res}
$$

Fig. 8. Reactive layer: structural rules.

Rule Req forms a synchronous message, by suspending the requesting object and embedding it as a continuation within the message. Here the unsaturated context $\mathcal{M}$ of the caller is saved, so that it can eventually be filled with the result of the request. The time constraint of a synchronous message is the same as that of the requesting object—the intuition here is that a request is analogous to a function call, and that servicing such a call can be neither more nor less urgent than the computation of the caller.

In rule Run, an idle object is juxtaposed to a message with matching destination. The "payload" of the message (the code, continuation and time constraint) is extracted from the message, which is left empty. This rule forms the essence of what constitutes a reaction in Timber.

When an active object eventually reaches a terminating state (as represented by a code expression of the form `return` e or `raise` e), the action taken depends on its continuation. Rule Done specifies that an object executing an asynchronous message just silently enters the idle state, where it will be ready to accept new messages. An object occupied by a request, on the other hand, also needs to reply to the requesting object. This is handled in the rule Rep, in which the waiting caller context $\mathcal{M}$ is applied to the reply (which may be an exception), and the continuation is released as an active object again. At the same time, the servicing object turns idle, ready for new messages.

The Abort rule shows how a pending message can be turned into an empty one before delivery, thus effectively removing it from the system. If the destination object has already started executing the message, or if the message was previously aborted, the rule will match against an empty message instead, leaving it unaffected.

Finally, there is a rule Eval that connects the functional and reactive layers, by promoting evaluation to reaction:

$$\frac{s \mapsto s' \quad e \mapsto e'}{(\!|s, \mathcal{M}[e], K|\!)^c_n \longrightarrow (\!|s', \mathcal{M}[e'], K|\!)^c_n} \text{ Eval}$$

Note that we allow for the concurrent evaluation both the state and code components of an object here, although with a strict functional layer, the state component will of course already be a value.

Timeline arithmetic The time constraint, or *timeline*, of an asynchronous message is obtained by adding the time constraint $\langle b, d\rangle$ of the sending object to the *relative* timeline $\langle \beta, \delta\rangle$ supplied to the constructor `act`. This operation is defined as follows:

$$\begin{aligned} \langle b, d\rangle + \langle \beta, \delta\rangle &= \langle max(b, b+\beta), max(d, b+\beta+\delta)\rangle \quad \text{if } \delta > 0 \\ &= \langle max(b, b+\beta), max(d, d+\beta)\rangle \qquad \text{if } \delta \leq 0 \end{aligned}$$

The maximum functions used here serve to ensure that the timeline of a message cannot be tighter than the timeline of the sending object; *i.e.*, both the baseline and the deadline of a message must be at least as large as those of the sender. This prevents the introduction of paradoxical situations where servicing a secondary reaction would be more urgent than executing the code that caused it.

As can be seen, a special case occurs when the relative deadline of a message is zero or below; then the deadline of the sender is kept but interpreted relative to the new baseline. This is how the two delayed messages sent in the `moved` method in Figure 2 are assigned their deadlines of 1 minute + 100 milliseconds, and 10 minutes + 100 milliseconds, respectively.

Note that this exception is added purely for the convenience of the programmer, who would otherwise always have to specify an explicit deadline whenever a new baseline is given. Note also that a relative deadline of zero amounts to a timeline that cannot be satisfied by any kind of finite computing resource, so an exception for this value does not really limit the expressiveness for the programmer.

Deadlock A Timber program that only uses asynchronous communication is guaranteed to be free from deadlock; however, since the sender of a synchronous message is unresponsive while it waits for a reply, the potential of deadlock arises. On the other hand, unlike many other languages and interprocess communication mechanisms, Timber allows for really cheap detection of deadlock. What is required is that each object keeps a pointer to the servicing object as long as it is blocked in a request, and that that the `req` operation starts by checking that setting up such a pointer will not result in cycle. If this is indeed the case, `req` results in an exception.

Preferably, our reaction semantics should formalize this behavior, as it is of utmost importance for the correctness of systems in which deadlock can occur. Unfortunately, our recursive definition of continuations actually denotes a linked structure that points in the other direction; from a server to its current client, if any. Duplicating this structure with links going in the forward direction makes the reaction axioms look extremely clumsy, and we have not found the formal definition of deadlock detection valuable enough to outweigh its cost. Instead we supplement the reaction axioms of Figure 7 with an informal rule that reads

$$(\!|s, \mathcal{M}[\texttt{req}\ n'\ e], K|\!)^c_n \longrightarrow (\!|s, \mathcal{M}[\texttt{raise Deadlock}], K|\!)^c_n$$

if the sending object n is found by following the forward pointers starting at n'. It is our hope that a neater way of specifying this behavior can eventually be developed.

Properties of the reactive layer Analogous to the subject reduction property that we assume for the functional layer, we establish what we may call a *subject reaction* property for our process calculus; *i.e.*, the property that all reaction rules preserve well typedness of processes. Well-typedness is defined by the straightforward typing judgments of Appendix B, and the subject reaction theorem looks as follows:

Theorem 2 (Subject Reaction). *If* $\Gamma \vdash P$ well-typed *and* $P \longrightarrow Q$*, then* $\Gamma \vdash Q$ well-typed

An attractive property of our layered semantics is that reduction and reaction live in independent worlds. There are no side effects in the functional layer, its only role is to enable rules in the deterministic layer, and further reduction cannot retract any choices already enabled. This can be captured in a diamond property that we call *functional soundness*, which says that the $\mapsto$ and $\longrightarrow$ relations commute.

Let $\underset{\text{EVAL}}{\longrightarrow}$ be the relation formed by the structural rules in Figure 8 and rule EVAL, but none of the axioms in Figure 7. Let $\underset{\neg\text{EVAL}}{\longrightarrow}$ be the reaction relation formed by including the structural rules and the axioms, but not rule EVAL.

Theorem 3 (Functional soundness). *If* $P \underset{\neg\text{EVAL}}{\longrightarrow} Q$ *and* $P \underset{\text{EVAL}}{\longrightarrow} P'$*, then there is a* Q' *such that* $P' \underset{\neg\text{EVAL}}{\longrightarrow} Q'$ *and* $Q \underset{\text{EVAL}}{\longrightarrow} Q'$*.*

The reader is referred to [12] for more elaboration and proofs of the above properties.

3.3 The scheduling layer

The reactive layer leaves us with a system that is highly non-deterministic—it says nothing about in which order objects should run messages, or in which order concurrently active objects may progress. The scheduling layer puts some extra constraints on the system, by consulting the hitherto uninterpreted time constraints attached to messages and objects. The requirements on the scheduler are formulated in terms of a *real-time trace*.

Definition 1 (Real-time trace). *A* real-time transition *is a transition* $P \longrightarrow Q$ *associated with a value* t *of some totally ordered time domain, written* $P \xrightarrow{t} Q$*. A* real-time trace $\mathcal{T}$ *is a possibly infinite sequence of real-time transitions* $P_i \xrightarrow{t_i} P_{i+1}$ *such that* $t_i < t_{i+1}$*. We call each* P_i *in a real-time trace a* trace state*.*

A real-time trace thus represents the execution of a Timber program on some specific machine, with machine-specific real-time behavior embedded in the t_i. Our first goal is to define which real-time behaviors are acceptable with respect to the time-constraints of a program.

Definition 2 (Transition timeline). *Let the timeline of a reaction axiom be the pair* c *as it occurs in the reaction rules of Figure 7, and let the timeline of a structural reaction rule be the timeline of its single reaction premise. We now define the timeline of a transition* $P \xrightarrow{t} Q$ *as the timeline of the rule used to derive* $P \longrightarrow Q$*.*

This definition leads to the notion of *timeliness*:

Definition 3 (Timeliness). *A real-time transition* $P \xrightarrow{t} Q$ *with timeline* $\langle b, d\rangle$ *is* timely *iff* $b \leq t \leq d$*. A real-time trace* $\mathcal{T}$ *is timely iff every transition in* $\mathcal{T}$ *is timely.*

Our second goal is to constrain the dispatching of messages to occur in priority order. First we need to formalize the notion of a dispatch:

Definition 4 (Dispatch). *A* dispatch *from* $\mathcal{T}$ *is a real-time transition* $P \xrightarrow{t} P'$ *derived from rule* RUN*, such that* P *is the final trace state of* $\mathcal{T}$*. An* (n, Q)-dispatch *from* $\mathcal{T}$ *is a dispatch from* $\mathcal{T}$ *where* n *is the name of the activated object, and* Q *is the dispatched message.*

The intention is that whenever there are several possible dispatches from some $\mathcal{T}$, the semantics should prescribe which one is chosen. To this end we will need to define an ordering relation between messages.

First, we let timelines be sorted primarily according to their deadlines.

Definition 5 (Timeline ordering). $\langle b, d\rangle < \langle b, d\rangle$ *iff* $d < d'$, *or* $d = d'$ *and* $b < b'$.

Second, we want to induce an ordering on messages whose timelines are identical. We will assume, without loss of generality, that ν-bound names are chosen to be unique in a trace state, and that the renaming congruence rule is never applied in an EQUIV transition. This makes it possible to uniquely determine the time when a bound name appears in a trace.

Definition 6 (Binding times). *The* binding time $bt(n, \mathcal{T})$ *of a name* n *in a real-time trace* $\mathcal{T}$ *is the largest time* t_i *associated with a real-time transition* $P_i \xrightarrow{t_i} P_{i+1}$ *in* $\mathcal{T}$, *such that* P_{i+1} *contains a* ν*-binder for* n *but* P_i *does not.*

From the previous two definitions we can construct a partial order on pending messages.

Definition 7 (Message ordering). $\langle n, e, K\rangle^c_m <_{\mathcal{T}} \langle n, e', K'\rangle^{c'}_{m'}$ *iff* $c < c'$, *or* $c = c'$ *and* $bt(m, \mathcal{T}) < bt(m', \mathcal{T})$

Note that this definition only relates messages aimed for some particular object, and that these messages are actually totally ordered.

From the ordering of messages follows the notion of a *minimal* dispatch.

Definition 8 (Minimal dispatch). *An* (n, Q)*-dispatch from* $\mathcal{T}$ *is* minimal *iff, for all possible* (n, Q')*-dispatches from* $\mathcal{T}$, $Q <_{\mathcal{T}} Q'$. *A dispatch from* $\mathcal{T}$ *is minimal if it is a minimal* (n, Q)*-dispatch from* $\mathcal{T}$ *for some* n *and* Q.

Finally, the timeliness and minimality constraints can be applied to real-time traces in order to identify the *valid* ones.

Definition 9 (Valid trace). *A real-time trace* $\mathcal{T}$ *is* valid *iff* $\mathcal{T}$ *is timely and every dispatch from a prefix* $\mathcal{T}'$ *of* $\mathcal{T}$ *is minimal.*

The real-time semantics of a Timber program is thus the set of valid traces it generates.

Some notes regarding these definitions:

1. The notion of minimality makes message dispatching fully deterministic. What the semantics prescribes is actually a *priority queue* of messages for each object, that resorts to FIFO order when priorities (timelines) are identical. This is also how our Timber compiler implements the semantics. Our reasons for defining message queuing here and not in the reactive layer are twofold: First, the complexity that arises from maintaining explicit queues in the process calculus is daunting. Second, leaving out ordering concerns is more in line with the process calculus tradition, and allows for easier comparison between Timber and other languages based on similar formalisms, like Concurrent Haskell.

2. All scheduling flexibility is captured in the selection of which object to run—because message order is fixed, the reaction axioms of Figure 7 do not offer any flexibility in choosing the next transition for a particular object. On the other hand, whenever there are several objects capable of making a timely transition, the semantics allows any one of them to be chosen. This opens up for pre-emptive scheduling, and coincides with our intuition that objects execute in parallel, but are internally sequential.
3. It follows from the monotonicity of real time that if a dispatch meets the baseline constraint of its timeliness requirement, all transitions involving the same object up to its next idle state will also meet the baseline constraint. Likewise, if an object becomes idle by means of a DONE or REP transition that meets its deadline, all transitions involving this object since it was last idle must also have met this deadline.
4. Meeting the baseline constraint of a dispatch is always feasible; it just amounts to refraining from making a certain transition. This can easily be implemented by putting messages on hold in a system-wide timer queue until their baselines have passed. On the other hand, meeting a deadline constraint is always going to be a fight against time and finite resources. Statically determining whether the execution of a Timber program will give rise to a valid trace in this respect is in general infeasible; however, we note that scheduling theory and feasibility analysis is available for attacking this problem, at least for a restricted set of Timber programs.
5. It can be argued that a Timber implementation should be able to continue execution, even if the deadline constraint of the timeliness requirement cannot be met for some part of its trace. Indeed, this is also what our Timber compiler currently does. However, it is not clear what the best way of achieving deadline fault tolerance would be, so we take the conservative route and let the Timber semantics specify only the desired program behavior at this stage.

On a uni-processor system, the scheduling problems generated by our semantics bear an attractively close resemblance to the problems addressed by deadline-based scheduling theory [21]. In fact, the well-known optimality result for fully preemptive Earliest-Deadline-First scheduling [3] can be directly recast to the Timber setting as follows:

Theorem 4 (Optimality of EDF). *For a given real-time trace, let the execution time attributed to a transition only depend on the reaction axiom from which the transition is derived. Moreover, let a re-ordering of the trace be the result of repeatedly applying the equivalence* $P \parallel Q \longrightarrow P' \parallel Q \longrightarrow P' \parallel Q' \equiv P \parallel Q \longrightarrow P \parallel Q' \longrightarrow P' \parallel Q'$ *(structurally lifted to transitions).*

Then, if there exists a re-ordering of transitions that results in a timely trace, re-ordering the transitions according to the principle of EDF will also result in a timely trace.

It is our intention to study this correspondence in considerable more detail, especially how the presence of baseline constraints affects existing feasibility

theory. However, it should also be noted that the scheduling layer semantics does not *prescribe* EDF scheduling. In particular, static schedules produced by off-line simulations of a program is an interesting alternative we are also looking into [9].

4 Related work

The actions in Timber resemble the *tasks* in the E machine [5], where the programmer can specify that a reaction to an event should be delivered precisely at a certain point in time. Consequently, the output of a task will be queued until it is time to react, and the E machine becomes a deterministic system, in sharp contrast to Timber. Similarly, the language H [22] is a Haskell-like language where functions over *timestamped messages* are interconnected through ports. For an input message, a timestamp indicates the time of an event, and for output, it specifies exactly when the message should be output from the system.

In the *synchronous* programming school (Esterel, Signal, Lustre), programs are usually conducted by one or more periodic clocks, and computations are assumed to terminate within a clock period [2]. In contrast, Timber does not make any assumptions about periodic behavior, even though it shares the concept of reactivity with the synchronous languages.

The UDP Calculus [20, 25] provides a formalism for expressing detailed behavior of distributed systems that communicate via the UDP protocol. Part of the structure of our process calculus can be found in the UDP Calculus as well; for example the representation of hosts (objects) as process terms tagged with unique names, and the modelling of messages as free-floating terms in their own right. The UDP Calculus has a basic notion of time constraints in the shape of terms annotated with timers, although at present this facility is only meant to model message propagation delays and timeout options. Moreover, the focus on actual UDP implementations in existing operating systems makes the UDP Calculus more narrow in scope than Timber, but also significantly more complex: the number of transition axioms in the UDP Calculus is 78, for example, where we get away with about 10.

The language Hume [18] has similar design motives as Timber: it has asynchronous, concurrent processes, is based on a pure functional core, and targets embedded, time-critical systems. However, Hume is an example of languages that identify the concept of *real time* with bounds on resources, not with means for specifying time constraints for the delivery of reactions to events. Apart from Hume, this group of languages also include Real-Time FRP and Embedded ML [24, 6].

On a historical note, Embedded Gofer [23] is an extension to a Haskell precursor Haskell aimed at supporting embedded systems. It has an incremental garbage collector, and direct access to interrupts and I/O ports, but lacks any internal notion of time. The same can also be said for Erlang [1]. Although it has been successfully applied in large real-time telecom switching systems, it

only provides best-effort, fixed-priority scheduling, and lacks a static type-safety property.

The technique of separating a semantics into a purely functional layer and an effectful process calculus layer has been used in the definition of Concurrent Haskell [16] and O'Haskell [14]. Although it is well known that a functional language can be encoded in process calculii, such an encoding would obscure the semantic stratification we wish to emphasize.

5 Conclusions and future work

We have given a semantics for Timber, stratified into independent layers for functional evaluation, object and message reactions, and time-sensitive scheduling. The language is implemented in terms of an full-featured interpreter, and we are currently developing a compiler that generates C code for targeting embedded systems. In the near future, we plan to apply and implement deadline-based scheduling analysis techniques for Timber, and nail down the specifics of shifting to a strict semantics in the functional layer.

6 Acknowledgments

The design of the Timber language has been influenced by contributions by many people within the Timber group at OHSU; Iavor Diatchki and Mark Jones in particular have been active in the development of Timber's time-constrained reactions. We would also like to thank members of the PacSoft research group at OHSU for valuable feedback, notably Thomas Hallgren.

References

1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.
2. A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.
3. M. Dertouzos. Control Robotics: the Procedural Control of Physical Processes. *Information Processing*, 74, 1974.
4. L. Erkök and J. Launchbury. Recursive monadic bindings. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00*, pages 174–185. ACM Press, September 2000.
5. T. A. Henzinger and C. M. Kirsch. The embedded machine: Predictable, portable real-time code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.
6. J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *International Conference on Functional Programming*, pages 70–81, 1999.

7. M. P. Jones, M. Carlsson, and J. Nordlander. Composed, and in control: Programming the Timber robot. `http://www.cse.ogi.edu/~mpj/timbot/ComposedAndInControl.pdf`, 2002.
8. S. L. P. Jones, A. Reid, F. Henderson, C. A. R. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–36, 1999.
9. R. Kieburtz. Real-time reactive programming for embedded controllers. `ftp://cse.ogi.edu/pub/pacsoft/papers/timed.ps`, 2001.
10. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
11. E. Moggi and A. Sabry. An abstract monadic semantics for value recursion. In *Workshop on Fixed Points in Computer Science*, 2003.
12. J. Nordlander. *Reactive Objects and Functional Programming*. Phd thesis, Department of Computer Science, Chalmers University of Technology, Gothenburg, 1999.
13. J. Nordlander. Polymorphic subtyping in O'Haskell. *Science of Computer Programming*, 43(2-3), 2002.
14. J. Nordlander and M. Carlsson. Reactive Objects in a Functional Language – An escape from the evil "I". In *Proceedings of the Haskell Workshop*, Amsterdam, Holland, 1997.
15. J. Nordlander, M. Jones, M. Carlsson, D. Kieburtz, and A. Black. Reactive objects. In *The Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, 2002.
16. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *ACM Principles of Programming Languages*, pages 295–308, St Petersburg, FL, Jan. 1996. ACM Press.
17. S. Peyton Jones et al. Report on the programming language Haskell 98, a non-strict, purely functional language. `http://haskell.org`, February 1999.
18. A. Rebón Portillo, K. Hammond, H.-W. Loidl, and P. Vasconcelos. Granularity analysis using automatic size and time cost inference. In *Proceedings of IFL '02—Implementation of Functional Languages*. Springer Verlag, September 2002.
19. A. Sabry. What is a Purely Functional Language? *Journal of Functional Programming*, 8(1):1–22, 1998.
20. A. Serjantov, P. Sewell, and K. Wansbrough. The UDP Calculus: Rigorous Semantics for Real Networking. In *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, Sendai, Japan, Oct 2001.
21. J. Stankovich, editor. *Deadline Scheduling for Real-time Systems, EDF and Related Algorithms*. Kluwer, 1998.
22. S. Truvé. A new H for real-time programming. `http://www.cs.chalmers.se/~truve/NewH.ps`.
23. M. Wallace and C. Runciman. Lambdas in the liftshaft - functional programming and an embedded architecture. In *Functional Programming Languages and Computer Architecture*, pages 249–258, 1995.
24. Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *International Conference on Functional Programming (ICFP '01)*, Florence, Italy, September 2001.
25. K. Wansbrough, M. Norrish, P. Sewell, and A. Serjantov. Timing UDP: mechanized semantics for sockets, threads and failures. In *11th European Symposium on Programming, ESOP 2002*, Grenoble, France, April 2002.

A Desugaring

$$
\begin{aligned}
[\![\textbf{do } c]\!]_v &= [\![c]\!]_v \\
[\![\textbf{do } p \texttt{ <- } e\texttt{;}\ cs]\!]_v &= [\![e]\!]_v \texttt{ >>= \ } p \texttt{ -> } [\![\textbf{do } cs]\!]_v \\
[\![\textbf{do } c\texttt{;}\ cs]\!]_v &= [\![c]\!]_v \texttt{ >> } [\![\textbf{do } cs]\!]_v \\
[\![\textbf{do let } ds\texttt{;}\ cs]\!]_v &= [\![\textbf{let } ds \textbf{ in do } cs]\!]_v \\
[\![\textbf{template } as \textbf{ in } e]\!]_v &= \left[\!\!\left[\begin{array}{l} \textbf{do } \texttt{self <- new } es \\ \quad \texttt{return } e \end{array} \right]\!\!\right]_{v'} \\
\text{where } & \begin{cases} v' = [\ x \mid x \texttt{ := } e \leftarrow as\ ] \\ es = [\ e \mid x \texttt{ := } e \leftarrow as\ ] \end{cases} \\
[\![\textbf{action } cs]\!]_v &= \texttt{act self } \langle 0,0 \rangle\ [\![\textbf{do } cs]\!]_v \\
[\![\textbf{request } cs]\!]_v &= \texttt{req self } [\![\textbf{do } cs]\!]_v \\
[\![\textbf{before}]\!]_v &= \texttt{bef} \\
[\![\textbf{after}]\!]_v &= \texttt{aft}
\end{aligned}
\qquad
\begin{aligned}
[\![p \texttt{ := } e]\!]_v &= [\![\texttt{set } ((\texttt{\textbackslash } p \texttt{ -> } v)\ e)]\!]_v \\
&\quad \text{if } fv(p) \subseteq v \\
[\![e]\!]_v &= \texttt{get >>= \textbackslash } v \texttt{ -> } [\![e]\!]_v \\
&\quad \text{if } fv(e) \cap v \neq \emptyset \\
[\![e]\!]_v &= [\![e]\!]_v \\
&\quad \text{otherwise}
\end{aligned}
$$

B Well-typed processes

Typing judgments for processes and continuations are given below. $\tau \rightsquigarrow$ is the type of a continuation waiting for a reply of type τ. Reduction contexts are given function types, treating their hole as a normal, abstracted variable.

$$\frac{\Gamma \vdash K : \tau \rightsquigarrow \qquad \Gamma \vdash n : \texttt{Ref } \rho \qquad \Gamma \vdash e : \texttt{O } \rho\ \tau \qquad \Gamma \vdash c : \texttt{(Time,Time)} \qquad \Gamma \vdash m : \texttt{Msg}}{\Gamma \vdash \langle n, e, K \rangle^c_m \text{ well-typed}}\ \textsc{Pending}$$

$$\frac{\Gamma \vdash K : \tau \rightsquigarrow \qquad \Gamma \vdash s : \rho \qquad \Gamma \vdash e : \texttt{O } \rho\ \tau \qquad \Gamma \vdash c : \texttt{(Time,Time)} \qquad \Gamma \vdash n : \texttt{Ref } \rho}{\Gamma \vdash (\!| s, e, K |\!)^c_n \text{ well-typed}}\ \textsc{Active}$$

$$\frac{\Gamma \vdash m : \texttt{Msg}}{\Gamma \vdash \langle\rangle_m \text{ well-typed}}\ \textsc{Empty} \qquad \frac{\Gamma \vdash s : \rho \qquad \Gamma \vdash n : \texttt{Ref } \rho}{\Gamma \vdash (\!| s |\!)_n \text{ well-typed}}\ \textsc{Idle}$$

$$\frac{\Gamma \vdash P \text{ well-typed} \qquad \Gamma \vdash P' \text{ well-typed}}{\Gamma \vdash P \parallel P' \text{ well-typed}}\ \textsc{Parallel}$$

$$\frac{\Gamma, n : \tau \vdash P \text{ well-typed}}{\Gamma \vdash \nu n.P \text{ well-typed}}\ \textsc{Restriction}$$

$$\frac{\Gamma \vdash s : \rho \qquad \Gamma \vdash \mathcal{M} : \tau \to \texttt{O } \rho\ \sigma \qquad \Gamma \vdash K : \sigma \rightsquigarrow \qquad \Gamma \vdash n : \texttt{Ref } \rho}{\Gamma \vdash (\!| s, \mathcal{M}, K |\!)_n : \tau \rightsquigarrow}\ \textsc{Cont}$$

$$\frac{}{\Gamma \vdash 0 : \tau \rightsquigarrow}\ \textsc{EmptyCont} \qquad \frac{\Gamma, [\,] : \tau \vdash \mathcal{M}[\,] : \texttt{O } \sigma\ \rho}{\Gamma \vdash \mathcal{M} : \tau \to \texttt{O } \sigma\ \rho}\ \textsc{Context}$$