

# The UniForM Concurrency Toolkit and its Extensions to Concurrent Haskell

Einar W. Karlsen

Bremen Institute for Safe Systems (BISS)  
FB3 Mathematik und Informatik  
Universität Bremen, Germany  
ewk@informatik.uni-bremen.de

**Abstract.** The UniForM Concurrency Toolkit is a comprehensive library of abstract data types for shared memory and message passing communication that extends Concurrent Haskell with a concept of dynamic types, thread identity, thread local state and selective communication as found in CML. Notable features of the toolkit are its support for reentrant monitors, interactors providing iterative choice and the uniform representation of internal channel events as well as external tool events of the environment in the form of first class synchronous event values.

## 1 Introduction

The *UniForM Concurrency Toolkit*<sup>1</sup> has primarily been designed to support the development of reactive systems using *Concurrent Haskell* [PJGF96].

The toolkit provides, partially backed up by a class system, a number of archetypical shared memory abstractions such as semaphores, locks and shared variables. The most important contribution, however, is the concept of reentrant monitors, whose main advantage over `MVar`'s is that recursion (and thus abstraction) is adequately supported.

The message passing model is very similar to the one of *Concurrent ML* [Rep92] by representing events in terms of first class composable event values that can be combined using guarded choice and the event-action combinator. Internal channel events, as well as external tool events of the environment, such as OS, GUI or DBMS events, can however be given a uniform representation by interposing adaptors between the event source and the internal set of listening agents.

The remainder of this paper is organized as follows. The next section defines the concept of threads, thread identity and thread local state. The third section presents shared memory concepts such as critical regions and reentrant monitors. The fourth section presents the operations of relevance to synchronous events. The fifth section gives a brief overview of external events and their related operators. The last sections compare the approach to CML with respect to performance and discusses the results.

---

<sup>1</sup> This work has been supported by the German Ministry of Research (BMBF) in the Project UniForM [KBPO<sup>+</sup>96] ("Universal Formal Methods WorkBench")

## 2 Threads

Concurrent Haskell provides a bare-bone concept of autonomous and unnamed *threads*. The toolkit extends this basis with concepts of *thread identities* and *thread local state*. The following base computations are provided over threads (Fig. 1):

- `spawn c` creates a new thread executing as defined by the computation `c`. The identity of the new thread is returned as the result.
- `getThreadID` returns the identity of the current thread.
- `delay d` suspends the execution of the current thread for the number of time units given by `d`.
- `stop` terminates the execution of the current thread.

```

data ThreadID    deriving (Eq,Ord)

spawn            :: IO () -> IO ThreadID
getThreadID     :: IO ThreadID
delay           :: Duration -> IO ()
stop            :: IO ()

type Env        = FiniteMap String Dyn

class Typeable a where
    toDyn       :: a -> Dyn
    fromDyn    :: Dyn -> Maybe a

retrieve       :: Typeable a => String -> IO (Maybe a)
define         :: Typeable a => String -> a -> IO ()
remove        :: Typeable a => String -> IO ()

```

Fig. 1. Threads

Some imperative thread packages provide a concept of a per thread local state that is exclusively owned by a single thread of control [PKB<sup>+</sup>91]. Provision for a per thread local state component has been made in terms of an *environment* [Dea89] that manages a collection of name-value pairs, i.e. in terms of a mapping from names to values of type *dynamic* [ACPP91]. The value in such an association is required to be an instance of class `Typeable`, meaning that there exists an *injection* function (`toDyn`) into type `Dyn` for the kind of value at hand, and a *projection* function from type `Dyn` (`fromDyn`). Dynamic values in this form are also provided by Hugs [JP97].

The application interface to the thread local state hides the environment component and operates with individual name-value pairs (Fig. 1):

- `define n v` associates the value `v` with the name `n`,
- `retrieve n` returns the value associated with `n` (if any),
- `remove n` removes the association identified by `n`.

The thread local state is basically meant as a substitute to Gofers [Jon94] concept of state transformers [LHJ95]. It can therefore be used to extend the IO monad with new hidden state components. We have exploited this feature in implementing the actor model [Agh86], in which case the thread local state is used to hold the identity of the current actor.

### 3 Shared Memory Abstractions

The base mechanism for synchronization and communication in Concurrent Haskell is `MVar`'s - a hybrid between a semaphore and a variable inspired by `Id` [BNA91]. To recall, the following operations are of relevance to `MVar`'s:

- `newMVar v` creates a new `MVar`.
- `takeMVar mv` returns the value associated with `mv` or blocks if `mv` is empty.
- `putMVar mv v` assigns to `mv`, if it is empty, the value `v`. Otherwise, the operation is undefined.

The toolkit provides a number of shared memory abstractions such as binary and counting *semaphores*, *mutex locks*, protected *variables* and reentrant *monitors*. Some main stream operations, to be discussed next, are defined through the class `Lock` and `Variable` respectively (Fig. 2).

```
class Variable v where
  updVar    :: v a -> (a -> IO (a,b)) -> IO b
  setVar    :: v a -> a -> IO ()
  getVar    :: v a -> IO a
  changeVar :: v a -> (a -> IO a) -> IO ()
  withVar   :: v a -> (a -> IO b) -> IO b
  setVar v x = changeVar v (\_ -> return x)
  getVar v   = withVar v return
  ...

class Lock l where
  release :: l -> IO ()
  acquire :: l -> IO ()
```

Fig. 2. Shared Memory Classes

### 3.1 Critical Regions

Solving problems directly in terms of `MVar`'s is a dangerous task: one `putMVar` too many and the program aborts, and one `takeMVar` too many and the thread blocks waiting for the `MVar` to get filled. The risk of getting deadlocks caused by dangling take's or put's are, in the context of if-then-else branches and abnormal flow of control, very high, but can be reduced effectively by introducing *critical regions* of code. The major advantage of using critical regions is that it is ensured that every `takeMVar` is always followed by a `putMVar`.

The `updVar v f` computation of class `Variable` provides such a critical region. The *transaction* `f` is a computation that given the current value of the variable `v`, will return a new value of the variable paired with some visible result returned to the callee. Derived commands with default definitions are then, in turn, provided for accessing (`getVar`), assigning (`setVar`), changing (`changeVar`) and querying (`withVar`) the value associated with the variable.

The following Haskell declaration defines the instantiation for `MVar`'s:

```
instance Variable MVar where
  updVar mv f = do
    v <- takeMVar mv
    ans <- try (f v)
    case ans of
      (Right (v',r)) -> do { putMVar mv v'; return r }
      (Left e) -> do { putMVar mv v; raise e }
```

First the value is retrieved, and the transaction `f` is applied to it. The `MVar` is then assigned the new value `v'`, and the visible result `r` is returned. In case of errors, the `MVar` is left unchanged, and the error is propagated.

### 3.2 Reentrant Monitors

A major problem with `MVar`'s is that deadlock will occur if we try to acquire the variable while being inside it's critical region. In other words, recursion is strictly forbidden and abstraction is consequently at risk. For a functional programmer, this is not an ideal situation.

For more complicated applications it may therefore be worthwhile to use a *reentrant monitor* [Ber96]. With monitors, it is ensured that if a thread has already acquired the monitor, then it will be allowed to re-enter the same monitor again (and again) as many times as it would like.

The reentrant monitor has been implemented in terms of a *reentrant mutex lock* (`Mutex`) combined with a `MVar`. The mutex lock provide commands by which an application may **acquire** and **release** the lock through operations of class `Lock` (see Fig. 2). The `MVar` is in turn used to hold the current value associated with the variable.

The following Haskell declaration defines the instantiation of class `Variable` for reentrant variables (i.e. so-called `RVar`'s):

```

data RVar a = RVar Mutex (MVar a)

instance Variable RVar where
  updVar (RVar mx mv) f = do
    acquire mx
    ans <- do {v <- getVar mv; try (f v)}
    case ans of
      Right (v',r) -> do {
        setVar mv v'; release mx; return r}
      Left e -> do {release mx; raise e}

```

The definition of `updVar` is very similar in structure to the one for `MVar`'s. The implementation of `Mutex` locks uses thread identities to test whether the holder of the mutex (if any) is the same as the thread currently trying to acquire the lock. If not, the thread is forced to wait until the monitor is free.

### 3.3 Example

Suppose that we would like to use a monitor to store a `Font`, `Colour` and `Size` attribute, with primitive computations for setting and retrieving the attributes one by one. The computation `setFC` is a composite transaction that updates the font and colour attribute. The definition ensures that there is no interference by other threads :

```

type Widget = RVar WST
data WST     = WST {col::Colour, font::Font, size::Size}

setCol w c = changeVar w (\wst -> return (wst{col=c}))
setFont w f = changeVar w (\wst -> return (wst{font=f}))
setFC w c f = withVar w (\_ -> do {setCol w c; setFont w f})

```

Using `MVar`'s, deadlock would occur by the call to `setCol`. Using reentrant monitors we do not have this problem, and consequently, there is no harm done to abstraction! Reuse is no longer risky business.

## 4 Synchronous Events

The concurrency toolkit extends Concurrent Haskell with a message passing model similar to the one of CML [Rep92], where concurrently executing agents communicate over *typed channels*. Communication is expressed by letting agents synchronize on first class, composable *event values*.

A concurrent system is expressed using two kind of domains. Values of type `IO a` represent reactive computations that are executed for their effect, whereas values of type `EV a` represent events that will return a value of type `a`, or fail with an error, whenever the event occur. The following computations, base events and event combinators are provided (Fig. 3):

- `channel` creates a new channel of type `Channel a`.
- `receive ch` denotes the event for reading a value over the channel `ch`.
- `send ch v` denotes the event for sending value `v` over channel `ch`.
- `inaction` denotes the empty set of events corresponding to the *null process* of process algebras.
- `e1 +> e2` denotes the *guarded choice* operator.
- `e >>>= c` is the *event-action* combinator that combines an event `e` with some reactive behaviour given in terms of a continuation function `c`.
- `sync e` is the operation that synchronizes on the event `e`. Execution of `sync e` will suspend until one of the communications denoted by `e` occurs.
- `event c` denotes the event computed by `c`. `c` is a computation that will, when executed during a call to `sync`, yield an event value as result. This event is used for synchronization.
- `tryEV e` wraps an error handler around the event `e`.

Communication is by handshake between two threads, which means that a *sender* and a *receiver* must perform a *rendezvous* over one and the same channel in order to communicate. It should be noted that the event-action combinator `>>>=` is to events, what the sequential composition operator `>>=` is to computations. The implementation of generalized selective communication in terms of `MVar`'s is documented in [Nor97].

```

data Channel a
data InterActor

channel    :: IO (Channel a)
receive   :: Channel a -> EV a
send      :: Channel a -> a -> EV ()

sync      :: EV a -> IO a

(>>>=)    :: EV a -> (a -> IO b) -> EV b
(>>>)     :: EV a -> IO b -> EV b
inaction  :: EV a
(+>)     :: EV a -> EV a -> EV a
event     :: IO (EV a) -> EV a
tryEV     :: EV a -> EV (Either IOError a)

interactor :: EV () -> IO InterActor
become     :: EV () -> IO ()
self       :: IO (Maybe InterActor)

```

**Fig. 3.** Synchronous Events

This basis is sufficient to define a number of message passing abstractions such as *timeouts*, *futures*, *remote procedure calls* [FGPS96], *message queues* and *broadcasting groups*. We shall demonstrate the primitives by defining some derived events to be used later on.

The `choose` combinator turns a list of events into a single event using guarded choice:

```
choose :: [EV a] -> EV a
choose = foldr (+>) inaction
```

A *precondition* can be wrapped around an event using the `whenEV` and `unlessEV` combinators:

```
whenEV, unlessEV :: Bool -> EV a -> EV a
whenEV p e        = if p then e else inaction
unlessEV p e      = whenEV (not p) e
```

One remark may be in place. Events are functors but not monads, since some of the equations of monads do not hold for events. The distinction between computations and events on the level of the type system can therefore be justified semantically. The "ugly" consequence is that the solution requires additional abstractions such as `tryEV`, `unlessEV` and `whenEV` that have computational counterparts within the IO monad.

#### 4.1 Message Queues

Synchronous events may be used to implement *asynchronous buffered communication*. CML has demonstrated how to do this in terms of an active *buffer process* that maintains the queue. Two channels are used for writing and reading elements of the queue. In Haskell, the buffer process looks like:

```
buffer :: Channel a -> Channel a -> Queue a -> IO ()
buffer rch wch q = sync (
    receive rch >>>= (buffer rch wch) . (insertQ q)
  +> unlessEV (isEmptyQ q) (
    send wch (headQ q) >>> buffer rch wch (tailQ q)))
```

A receive guard is set up for inserting new elements into the queue (`insertQ`), whereas a send guard is used for removing elements from the queue (`tailQ`). The send guard is only enabled if the queue is not empty.

#### 4.2 Interactors

Frequently, when developing reactive systems, an agent must be set up to react to an event repeatedly throughout its entire lifetime. The model for selective communication has therefore been extended with a concept of *interactors* providing *iterative choice* over an event (Fig. 3). Interactors provide a refinement of the *Actor model* of Gul Agha [Agh86]:

- the computation `interactor e` creates a new interactor that repeatedly interacts as defined by `e`.
- the computation `become e` changes the behaviour of the current interactor so that the interactor will from now on interact as defined by `e`.
- the computation `self` returns the identity of the current interactor.

The implementation of the `self` operator uses the thread local state to hold the identity of the current interactor (if any). With interactors and first class synchronous events it actually becomes possible to simulate more traditional paradigms to event handling such as callbacks and the Model-View-Controller [KP88] paradigm.

### 4.3 Callbacks

The most common approach to event handling is to associate *callbacks* (i.e. actions) with events of the environment. An *event loop* is then set up that executes the associated callback in response to an incoming event. While this dispatching process is going on, the application may *register* (or *de-register*) callbacks in order to change the reactive behaviour of the system.

Callbacks can be simulated using event values. Each callback is identified by some ordinal value, that is used as a key in calls to the `registerCB` and `deregisterCB` commands. The callback itself is represented in terms of a value of type `EV ()`, which defines the event in question as well as the reaction to the event. The role of the event loop is taken over by an interactor that listens to the sum of all registered events:

```
type EventLoop a = (MVar (RST a), InterActor, Channel ())
type RST a = FiniteMap a (EV ())

eventloop :: Ord a => EventLoop a -> EV ()
eventloop el@(mv,iact,ch) =
    (rst +> receive ch) >>> become (eventloop el)
    where rst = event(withVar mv (return . choose . eltsFM))
```

The implementation maintains an *event registration set* (`RST`), i.e. a mapping from callback identifiers to callbacks. This set may change, and at the end of each iteration, the interactor therefore calls the `become` command to change behaviour. Notice that a channel is used to "wake up" the event loop whenever another agent has changed the registration set.

New callbacks are then registered by calling `registerCB`:

```
registerCB :: Ord a => a -> EV () -> EventLoop a -> IO ()
registerCB oid ev (mv,iact,ch) = do
    changeVar mv (\cbs -> return (addToFM cbs oid ev))
    this <- self
    unless (this == (Just iact)) (sync (send ch ()))
```

The registration set is updated and a notification message is forwarded to the event loop. This message is only generated, if `registerCB` is called by another thread than the event loop. This way we avoid deadlock caused by the event loop attempting to communicate with itself. Deregistration of events is not that much different.

The simulation improves a traditional approach by being composable and generic. Registered events may be composite, and the event loop can be instantiated to work with GUI event, OS events, DBMS events - or a suitable union of these event, as required by the application at hand.

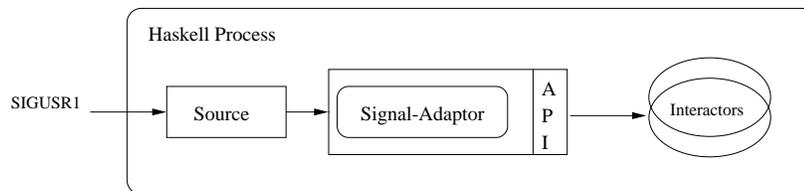
## 5 External Tool Events

A reactive system is characterized by a control component (here the network of executing threads) and an environment of external tools. Basically, the reactive system is event driven.

In a framework to event handling based on synchronous events, we would like to represent tool events as first class composable event values. All a listening thread should do in order to receive an event `e` from an external source would be to call `sync e`. Such tool events could then be combined to form new composite events, and one could freely mix internal channel events and external tool events using guarded choice and the event-action combinator, thus having a uniform and composable framework to event handling that is independent of the actual source of the event. Suppose for example that we would like to wait for an operating system generated `sigUSR1` or `sigUSR2` signal. The following piece of Haskell code should do the trick:

```
sync(signaled sigUSR1 +> signaled sigUSR2)
```

The toolkit is based on a concept of event *sources*, *adaptors* and *interactors* to achieve this degree of abstraction (Fig. 4). The event source is just a primitive sensor that receives events of the environment. The sensor could for example capture OS signals or it could listen to messages over a pipe.



**Fig. 4.** Reactive Systems Architecture

The main component is the adaptor which is interposed between the physical event source (i.e. a GUI, a DBMS or the OS) and the interactors of the

application. It is the role of the adaptor (e.g. the adaptor for OS signals), to turn an external event into a first class composable value of type `EV`, and to delegate such an event, whenever it occurs, to the interactors awaiting the event. It is, on the other hand, the role of the interactors to carry out the reaction to such an event, i.e. to provide the logic of the application.

External events (Fig. 5) are communicated from the event adaptor to the set of listening interactors in terms of tuples  $(\mathbf{eid}, \mathbf{d})$ , where `eid` denotes the unique *event designator* of the event and `d` the *event descriptor*. An interactor is set up to receive a tool event `eid` by synchronizing on the `listen eid r dr` event, where `r` and `dr` are computations that register, respectively de-register, the interactor with the adaptor mediating the event denoted by `eid`. All the *registration command* does, is actually to inform the adaptor that "when this event occurs, please forward it to me". The de-registration command has the opposite effect. Both commands are executed behind the scene when `sync` is called.

```

data EventID    deriving (Eq,Ord)

class EventDesignator e where
  toEventID    :: e -> EventID

type Register = InterActor -> IO ()

listen        :: (EventDesignator e, Typeable a)
              => e -> Register -> Register -> EV a

```

Fig. 5. External Tool Events

All tool events are required to be uniquely identifiable in terms of event designators of type `EventID`. The class `EventDesignator` abstracts over the kind of types that can be used as event designators. The event descriptor carries relevant information about the event in terms of a value of the *dynamic* type (`Dyn`). The injection (`toDyn`) and projection (`fromDyn`) functions are called by the adaptor and the event listener - respectively.

The registration command is used during synchronization to inform the adaptor about the presence of a new listener. The adaptor can then delegate an incoming event from the event source onwards to the interactor. Having received an event  $(\mathbf{eid}, \mathbf{d})$ , the event listener determines the reaction associated with the event designator `eid`, passes it the received event descriptor `d` as actual argument, and executes the resulting computation. The reaction to such an event is, of course, defined by using the event-action combinator.

## 5.1 Delegation

External tool events are delegated from an adaptor to an interactor, either in the form of *synchronous requests* or in the form of *asynchronous notices* (Fig. 6). Delegation is handled by the adaptor by application of the `request` event or the `notify` computation.

```

notify  :: (EventDesignator e, Typeable a)
         => InterActor -> e -> a -> IO ()
request :: (EventDesignator e, Typeable a)
         => InterActor -> e -> a -> EV ()
```

**Fig. 6.** Delegation of Tool Events

Each interactor is actually equipped with a channel for receiving requests and a message queue for receiving notices. External events are communicated to the interactor over these two communication objects.

## 5.2 Adaptor

The primary role of the adaptor is to delegate events from an event source to the set of event listening agents having registered interest in the event. The adaptor provides a decoupling between the event source and the listener. A specific adaptor receives events from the associated event source, and identifies, for each incoming event, the listeners having registered interest in it. The event is then delegated onwards to the listener(s). By enforcing a loose coupling between event listeners and event sources, *customized adaptors* can be interposed that deal with the idiosyncrasies and peculiarities of the event source.

The *standard adaptor* is one such adaptor, which serves as an event broker by delegating events from the external source to the set of listening interactors. The adaptor is parameterized over the type of the event descriptor. It offers the following computations (Fig. 7):

- `emit a eid d` is called by the event source and forces the adaptor `a` to dispatch the event `(eid,d)`.
- `register a eid m c iact`, is called by an interactor `iact` in order to register interest in event `eid`. The mode `m` defines whether delegation should be asynchronous or not, and `c` is a computation that does the physical registration with the external tool.
- `deregister a eid c iact`, is called by an interactor `iact` in order to de-register interest in event `eid`.

```

data Adaptor a
data Mode    = Notice | Request

emit        :: (EventDesignator e, Typeable a)
            => Adaptor a -> e -> a -> IO ()

register    :: (EventDesignator e, Typeable a)
            => Adaptor a -> e -> Mode -> IO () -> Register
deregister :: (EventDesignator e, Typeable a)
            => Adaptor a -> e -> IO () -> Register

```

Fig. 7. Standard Adaptor

The adaptor is by nature very similar to the event loop outlined in the previous section, but with slightly different registration commands and interaction patterns. The registration and de-registration commands are called by the interactors behind the scene, whenever `sync` is called, or whenever an interactor changes its interaction pattern by issuing a call to `become`. The implementation of the interactor, as well as the implementation of the standard adaptor, are therefore to some extent similar to the implementation of event loops, in so far that they all maintain a registration set.

It is quite interesting to observe that this model does not require more than is already in the concurrency toolkit. The communication between the adaptors of a reactive system and the listeners is fully definable using synchronous channel events.

### 5.3 Example

Suppose that we would like to represent OS signals in terms of first class synchronous events. We need to install appropriate C signal handlers to achieve this goal. There is no way (yet) by which we can call Haskell from within a C signal handler, but a work-around would be to install C signal handlers that redirect the signal over a dedicated pipe. An event reading thread can then be set up to fetch signals from this pipe, and delegate them onwards to the *signal adaptor*. We can use the standard adaptor as an event broker for this:

```

instance EventDesignator (Adaptor (),Signal) where {...}

signaled :: Adaptor () -> Signal -> EV ()
signaled a s = listen (a,s) reg dereg
  where reg   = register a (a,s) Notice (setCHandler s)
        dereg = deregister a (a,s) (unsetCHandler s)

```

The event descriptor is the unit type since signals do not carry information. The event designator is defined as a tuple consisting of the handle of the standard adaptor and the `Signal` value.

The definition of the `signaled` event instructs the adaptor to delegate the signal in the form of a notice - asynchronously to all interactors awaiting it. The `setCHandler` computation installs a C signal handler, that instructs the OS to send the signal over the dedicated file descriptor. The event source, defined by the `emitter` computation, reads incoming signals and sends them to the adaptor mediating the signal:

```
emitter :: Adaptor () -> Fd -> IO ()
emitter a fd = forever (readFd fd >>= \s -> emit a (a,s) ())
```

## 6 Performance

We have developed a couple of test programs for comparing the performance of the toolkit with CML. The benchmark (Fig. 8) was run on a Sparc Ultra using GHC version 2.8 and CML version 109.19. The resulting figures, in micro-seconds per operation, are given in the table below. A ratio less than 1 means that Concurrent Haskell is faster than CML.

The cost of shared memory access is determined by updating a `MVar` (benchmark 1) and a `RVar` (2). Notice that reentrant monitors could not be implemented in CML, since it has no operation for retrieving the current thread identity. The cost of message passing is determined by a simple producer-consumer configuration where messages are send over a channel (3) or a message queue (4). Generalized selective communication is measured by a more symmetric configuration using two channels and two threads that send and receive messages over both channels (5).

Benchmark	Scope	Haskell	CML	Ratio
1.	Update of MVar	2.8	45.2	0.06
2.	Update of RVar	10.4	-	-
3.	Synchronous Communication	80	18.9	4.23
4.	Asynchronous Communication	40	38.6	1.04
5.	Generalized Selective Communication	180	40.5	4.44

**Fig. 8.** Performance Benchmark

Concurrent Haskell is by far faster than CML when it comes to shared memory abstractions. Haskell is, on the other hand, significantly slower than CML in dealing with synchronous communication. In other words - the built in abstractions are performing best. It is a minor surprise, that the two systems compare equally well when it comes to asynchronous communication over message queues. The reason for this is that we have hard-wired message queues into the protocol for selective communication using `MVar`'s, rather than emulating them using an active buffer thread.

The overhead of factor 4 when using reentrant monitors, rather than `MVar`'s, seems to be justifiable. CML is faster on pure synchronous communication, but for applications such as reactive systems, with a blend of variable updates, asynchronous and synchronous communication, Haskell is the better choice.

## 7 Related Work

The concurrency toolkit extends Concurrent Haskell [PJGF96] with a concept of dynamic types, thread identity and thread local state. Thread identities and `MVar`'s are then in turn used to implement reentrant monitors, which improves on `MVar`'s by supporting recursion and abstraction.

Concurrent Haskell provides an approach to reactive system development based on stream processors and selective reads [FPJ94]. Generalised selective communication has however proven itself useful in the definition of the standard adaptor, where it is used to keep the adaptor and the interactors "in sync". However, a thorough trade-off between the two approaches still remains to be done.

By going for generalized selective communication, we follow in the footsteps of earlier approaches to higher order concurrency such as PFL [Hol83], Amber [Car86], Facile [GMP89] and CML [Rep92,Rep91,Rep95]. A similar attempt to selective communication in Haskell is described in [GF96], but here the `event` operator, which is almost indispensable in practical life, is not supported.

The major improvement over these systems however, is the handling of tool events of the external environment. The idea of associating registration and de-registration commands with tool events can be traced back to the JavaBeans architecture [Mic96]. The toolkit improves on JavaBeans by providing automatic registration of events, and by representing events in terms of first class, composable values. A functional language is a prerequisite in coming up with these abstractions.

Interactors provide a refinement of the *Actor model* of Gul Agha [Agh86]. Rather than defining the response to an event in terms of a continuation function and an event dispatching case statement, it is defined in terms of an event value that hides the actual dispatching being done. Advantage: event values are first class composable values, whereas case patterns are not! Iterative choice has also been seen in [Pie94].

At the time of writing, the concurrency toolkit has been used to implement a GUI atop of Tk [Ous94,Kar97b], an `expect` [Lib91] like tool for encapsulating interactive Unix tools [Kar97a] as well as an encapsulation of an active DBMS [KW97]. Events of these tools are represented uniformly as external tool events of type `EV`. Reentrant monitors, another key feature of the toolkit, are used extensively, e.g. to represent the state of each widget.

## 8 Future Work

There are several areas where we would like to see improvements to Concurrent Haskell and the UniForM Concurrency Toolkit. One concerns the provision of *daemon* [Ber96] threads, i.e. threads that become garbage collected when there is no other non-daemon thread around. Daemon threads take the role of system (or background) threads that should disappear when there is no proper application thread around to require its services. The `emitter` thread that reads operating system signals should for example be defined as a daemon thread: otherwise the program would never terminate.

A second issue concerns the implementation of generalized selective communication. At the moment of writing, the implementation is not fair. We hope to improve on this in the future. While doing so, some performance improvements could easily be achieved. At the moment, one `MVar` is created for every read guard in a call to `sync`. With dynamic typing, we can get around with one single `MVar` for all read guards. Further improvements with respect to performance are possible, because there are still optimizations that can be done to Concurrent Haskell.

## 9 Conclusion

Concurrent Haskell was thought of as a minimal kernel on which more complicated abstractions could be established. The computation for forking of new threads and `MVar`'s are all what is offered. It is however possible to provide a powerful set of abstractions on this tiny basis such as locks, shared variables and selective communication in its full elegance by extending Concurrent Haskell with a concept of thread identity and thread local state. The minimal model of Concurrent Haskell has therefore passed the test.

The UniForM Concurrency Toolkit provides a number of features that are very useful when developing reactive systems: reentrant monitors, interactors with iterative choice and first class event values. A key contribution is the uniform approach to internal channel events and external tool events in the form of first class composable event values. It is interesting to observe that this model can quite easily be used to model more traditional approaches to event handling such as callbacks and the Model-View-Controller paradigm.

Abstraction is of little use, if it cannot be implemented efficiently. It is actually promising to observe that the toolkit performs very well in comparison to Concurrent ML. For reactive systems, relying on a blend of synchronous and asynchronous point-to-point communication, message broadcasting and shared memory updates, we like to think of the toolkit as a better choice. Specification convenience is furthermore provided by using infix operators whenever possible.

## Acknowledgement

I would like to thank Walter Norzel for implementing selective communication and a couple of the concurrency abstractions of the toolkit, and Kevin Hammond as well as Simon Peyton Jones for implementation hints. Without Concurrent Haskell, this work would not have been possible.

## References

- [ACPP91] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically Typed Language. In *ACM Transactions on Programming Languages and Systems*, pages 13(2):237–268, April 1991.
- [Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, Massachusetts, 1986.
- [Ber96] D. Berg. *Java Threads - A Whitepaper*. California, USA, 1996.
- [BNA91] P. S. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, non-strict Functional Language with State. In *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 523, pages 538–568, Boston, 1991. Springer Verlag.
- [Car86] L. Cardelli. Amber. In *Combinators and Programming Languages*, Lecture Notes in Computer Science 242. Springer Verlag, 1986.
- [Dea89] A. Dearle. Environments: A Flexible Binding Mechanism to Support System Evolution. In *Proc. 22nd International Conference on System Sciences*. Springer Verlag, 1989.
- [FGPS96] T. Frauenstein, W. Grieskamp, P. Pepper, and M. Südholt. Communicating Functional Agents and their Application to Graphical User Interfaces. In *Proceedings of the 2nd International Conference on Perspectives of System Informatics, Novosibirsk*, Lecture Notes in Computer Science. Springer Verlag, 1996.
- [FPJ94] S. Finne and S. Peyton Jones. Programming Reactive Systems in Haskell. In *Proceedings of the Glasgow Functional Programming Workshop*, Ayr, Scotland, September 1994. Springer Verlag.
- [GF96] V. M. Gulias and J. L. Freire. Concurrent Programming in Haskell. Technical report, LFCIA, Department of Computer Science, University of La Coruna, Spain, 1996.
- [GMP89] A. Giacolone, P. Mishra, and S. Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. In *International Journal of Parallel Programming, Vol. 18, No. 2*, 1989.
- [Hol83] S. Holmström. PFL: A Functional Language for Parallel Programming, and its Implementation. Technical report, University of Göteborg and Chalmers University of Technology, Sweden, September 1983.
- [Jon94] M. P. Jones. The Implementation of the Gofer Functional Programming System. Research report yaleu/dcs/rr-1030, Yale University, New Haven, Connecticut, USA, May 1994.
- [JP97] M. P. Jones and J. C. Peterson. *Hugs 1.4, The Nottingham and Yale Haskell User's System*, April 1997.
- [Kar97a] E. W. Karlsen. Integrating Interactive Tools using Concurrent Haskell and Synchronous Events. In *CLaPF'97: 2nd Latin-American Conference on Functional Programming*, September 1997.

- Available at <http://www.informatik.uni-bremen.de/~ewk/papers/-clapf97.ps.gz>.
- [Kar97b] E. W. Karlsen. The UniForM User Interaction Manager. Technical report, FB 3, Universität Bremen, Germany, November 1997. Available at <http://www.informatik.uni-bremen.de/~ewk/papers/uim.ps.gz>.
- [KBPO<sup>+</sup>96] B. Krieg-Brückner, J. Peleska, E. R. Olderog, D. Balzer, and A. Baer. Universal Formal Methods Workbench. In U. Grote and G. Wolf, editors, *Statusseminar des BMBF: Softwaretechnologie*. Deutsche Forschungsanstalt für Luft- und Raumfahrt, Berlin, 1996. Available at <http://www.informatik.uni-bremen.de/~uniform>.
- [KP88] G. Krasner and S. Pope. A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.
- [KW97] E. W. Karlsen and S. Westmeier. Using Concurrent Haskell to Develop Views over an Active Repository. In *IFL'97: Implementation of Functional Languages*, September 1997. Available at <http://www.informatik.uni-bremen.de/~ewk/papers/ifl97.ps.gz>.
- [LHJ95] S. Liang, P. Hudak, and M. Jones. Monad Transformers and Modular Interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, US, January 1995.
- [Lib91] D. Libes. expect: Scripts for controlling interactive processes. In *Computing Systems, Vol 4, No. 2*, Spring 1991.
- [Mic96] Sun Microsystems. *JavaBeans 1.0*. JavaSoft, December 1996.
- [Nor97] W. Norzel. Building Abstractions for Concurrent Programming in Concurrent Haskell. Master thesis (in german), FB 3, Universität Bremen, Germany, April 1997.
- [Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.
- [Pie94] B. C. Pierce. Programming in the  $\pi$ -Calculus: An Experiment in Concurrent Language Design, PICT Version 3.4c. Technical report, Department of Computer Science, University of Edinburgh, Scotland, 1994.
- [PJGF96] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Principles of Programming Languages '96 (POPL'96), Florida*, 1996.
- [PKB<sup>+</sup>91] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS Multi-thread Architecture. In *USENIX, Winter 91*, 1991.
- [Rep91] J. H. Reppy. CML: a Higher-Order Concurrent Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1991.
- [Rep92] J. H. Reppy. *Higher-Order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, 1992.
- [Rep95] J. H. Reppy. First-class Synchronous Operations. In *Theory and Practice of Parallel Programming*, Lecture Notes in Computer Science, Sendai, Japan, 1995. Springer Verlag.