Integrating Interactive Tools using Concurrent Haskell and Synchronous Events

by

Einar W. Karlsen
Bremen Institute for Safe Systems (BISS)
Universität Bremen
Postfach 330 440
D-28329 Bremen
Germany
mail: ewk@informatik.uni-bremen.de
fax: (+49) 421-218-3054

Abstract

In this paper we describe how existing interactive tools can be integrated using Concurrent Haskell and synchronous events. The base technology is a higher-order approach to concurrency as in CML extended with a framework for handling external events of the environment. These events are represented as first class synchronous events to achieve a uniform, composable approach to event handling. Adaptors are interposed between the external event sources and the internal set of listening agents to achieve this degree of abstraction. A substantially improved integration framework compared to existing technology (such as for example the combination of Tcl/Tk with `expect`) is then provided. With this basis it is for example possible to wrap a GUI around the hugs interpreter with very little work required.

Submitted to *CLaPF'97 - 2nd Latin-American Conference on Functional Programming*, October 3-4, 1997, La Plata, Argentina

Integrating Interactive Tools using Concurrent Haskell and Synchronous Events*

Abstract

In this paper we describe how existing interactive tools can be integrated using Concurrent Haskell and synchronous events. The base technology is a higher-order approach to concurrency as in CML extended with a framework for handling external events of the environment. These events are represented as first class synchronous events to achieve a uniform, composable approach to event handling. Adaptors are interposed between the external event sources and the internal set of listening agents to achieve this degree of abstraction. A substantially improved integration framework compared to existing technology (such as for example the combination of Tcl/Tk with `expect`) is then provided. With this basis it is for example possible to wrap a GUI around the hugs interpreter with very little work required.

1 Introduction

There are several ways in which existing tools can be encapsulated to work in an integrated environment. The general solution to interoperability is to use component technologies such as Corba [OMG95] or the Component Object Model [COM95] where client/server stubs are generated on the basis of Interface Definition Language (IDL) signatures. Both systems provide interoperability in a heterogeneous language and platform environment. Haskell programmers may also turn to Green Card [PNR97], which has a more restricted scope by providing interoperability between existing C functions and Haskell.

However, there is no adequate way in Haskell to integrate loosely coupled and *interactive* tools having a textual interface such as `latex`, `ftp`, `telnet` or `hugs`. A solution has been suggested by `expect` [Lib91], which uses Tcl [Ous94] as scripting language and Tk for wrapping GUI's around existing shell tools. `expect` runs the tool in the background and communicates with it in terms of "send command/expect response" sequences. Regular expressions are used to distinguish between different kinds of responses, and expect statements allow the developer to associate reactive behaviour with response patterns. There is one inherent drawback with this approach: Tcl doesn't provide sufficient support

*This work has been supported by the German Ministry of Research (BMBF) on Project Uni-ForM ("Universal Formal Methods WorkBench")

for large scale tool integration due to its lack of strong typing, modules and concurrency.

Concurrent Haskell [PGF96], extended with synchronous events in the style of CML [Rep92], provides a better basis. Command scripts are then defined in terms of computations of type `IO` according to the imperative functional style [PW93]. Events, on the other hand, are represented as first class composable values with the non-deterministic choice and the event-action operator as basic combinators. The overall reactive system is consequently viewed as a network of executing and communicating threads. However, using channel events and selective communication directly to represent external events is not a promising direction to take. Selective communication is too expensive considering the amount of events to be handled in the context of large reactive systems such as integrated Software Development Environments.

The solution is to be found in architectures such as the ToolBus [BK94] or Java Beans [Sun96]. An adaptor is then interposed between event sources (external tools) and the event listeners (threads of the reactive system). It is the responsibility of the adaptor to turn external events of the environment into first class synchronous operations. By using dynamic typing, combined with a dedicated protocol for interaction between the adaptors and the event listeners, the need for selective communication can efficiently be reduced: each listener is equipped with exactly one channel for receiving all kinds of external events.

This basis can in turn be used to provide a functional equivalent to **`expect`**, but with substantial improvements with regard to composability and uniformity. Expecting tool responses is now defined using first class, composable events featuring abstraction and information hiding. The functional **`expect`** tool is backed up with an equally concurrent GUI that interacts with the Tk toolkit through a dedicated adaptor. But unlike Tk, widgets are typed and type classes are used to group different kind of properties. User interactions are furthermore represented in terms of first class synchronous events to achieve a completely uniform framework for presentation and control integration that hides the underlying implementation details.

The remainder of this paper is organized as follows. The next section sets the scene by introducing first class synchronous events. The third section defines the approach to handling external events in terms of event sources, adaptors and event listeners. The fourth section presents the functional **`expect`** tool, which supports encapsulation of interactive tools in Haskell. The fifth section provides a quick overview of Haskell-Tk, mainly focusing on the aspects relevant for representing user interactions as first class synchronous events. The sixth section demonstrates the application of this encapsulation technology on a larger example by wrapping a GUI around the **`hugs`** interpreter. The last section compares the approach to existing technology and discusses the results.

2 Synchronous Events

Concurrent Haskell [PGF96] provides a minimal model to concurrency in terms of commands for forking off new threads that communicate through **MVar**'s. More support is actually required when developing comprehensive reactive systems. We have therefore developed the UniForM Concurrency Toolkit [KN97] offering shared memory protocols, message passing models and various thread abstractions. Whereas the shared memory primitives are quite easily provided on top of **MVar**'s, the message passing model requires a completely different setting. Event handling is done the "process algebraic" way by providing similar basic operations and combinators as CML [Rep92], but in a monadic (and more uniform) specification framework. A concurrent (reactive) system is then 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) when triggered.

The following computations, base events and event combinators are provided:

- **channel** creates a new channel of type **Channel a**.
- **receive ch** denotes the event for reading a value over the channel **ch**. Communication is by handshake between two threads, which means that a sender and a receiver must perform a rendezvous in order to exchange values. Another thread must therefore synchronize on a
- **send ch v** event, which denotes the event for sending a value over channel **ch**.
- **inaction** denotes the empty set of events corresponding to the *null process* of process algebras.

```
data Channel a

channel  :: IO (Channel a)

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

sync     :: EV a -> IO a
poll     :: EV a -> IO (Maybe a)

(>>>=)   :: EV a -> (a -> IO b) -> EV b
(>>>)    :: EV a -> IO b -> EV b
e >>> c  =  e >>>= (\_ -> c)

inaction :: EV a

(+>)     :: EV a -> EV a -> EV a

event    :: IO (EV a) -> EV a

tryEV    :: EV a -> EV (Either IOError a )
```

Signature 1: Synchronous Events

- `e1 +> e2` denotes the non-deterministic choice operator.
- `e >>>= c` is the *event-action* combinator that combines an event `e` with some additional reactive behaviour given in terms of a continuation function `c`.
- `sync e` is the operation that synchronises on the event `e`. Execution of `sync e` will suspend until one of the events denoted by `e` triggers. The result of the `sync` command is the result returned by the triggering event.
- `poll e` attempts to synchronize on the event `e`, and immediately returns the resulting value wrapped with the constructor `Just` if synchronisation is possible. Otherwise, `Nothing` is returned.
- `event c` denotes the event computed by `c`. `c` is a computation that will, when executed during a call to `sync` or `poll`, yield an event value as result. This event is used for synchronization.
- `tryEV e` wraps an error handler around an existing event `e`, and returns either an error or a value depending on the outcome of `e`.

The signature of these operations is defined in Signature 1. The implementation of selective communication in terms of `MVar's` is thoroughly documented in [KN97]. We shall demonstrate the primitives by defining some derived events to be used throughout the rest of this paper. The `choose` combinator turns a list of events into a single event using non-deterministic choice:

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

Promises are asynchronous procedure calls. A `promise` takes a command as parameter, but rather than executing it immediately, it spawns of a server thread to do the job, and returns an event on which the client may synchronise later on in order to get the result.

```
promise :: IO a -> IO (EV a)
promise beh = do { ch <- channel;
   forkIO (try beh >>= sync. (send ch));
   return (receive ch >>>= either fail return)}
```

Awaiting the result of such an (deferred) RPC is represented by the client synchronising on a `receive` event over a reply channel. The server thread will eventually `send` its result over this channel.

Timeouts are realised in the concurrent setting by using promises, where the server thread is set up to delay execution for the specified time interval:

```
timeout :: Time -> EV ()
timeout = event . promise . delay
```

3 External Events

There are several ways by which prefabricated tools can be integrated. An approach that conforms very well with the model of synchronous events has been suggested by the ToolBus architecture [BK94]. Here a reactive system is viewed as a loosely coupled network of executing agents communicating with each other by sending messages or by broadcasting notices. Communication with an external tool goes through an adaptor: agents of the reactive system send requests to the tool through the *adaptor*, and receive events from the external tool - also through the adaptor. The adaptor consequently serves as a *capsule* (or *encapsulation*) of the tool.

A similar approach, but building on objects and method invocations rather than agents and message passing, can be found in the Java Beans architecture [Sun96]. The Java Beans architecture uses events for propagating state change notifications between a *source object* and one or more target *listener objects*. A common event registration mechanism permits the dynamic manipulation of the relationships between event sources and event listeners. Event sources furthermore identify themselves by providing particular *registration* (and de-registration) *methods* that accept references to particular event listener interfaces. In circumstances where a listeners cannot directly implement a particular interface, an instance of a custom *adaptor* may be interposed between a source and one or more listeners in order to establish the relationship or to augment behaviour. The primary role of the adaptor is to conform to a particular event listener interface, and to decouple the incoming event notifications on the interface from the actual listeners. For example, it is up to the adaptor whether events are synchronously unicasted to a single listener or asynchronously multicasted to all listeners.

In a framework of synchronous events, we would like to represent such *external events* (being generated by foreign tools) as first class composable values. All a listening thread should do in order to receive the event from an external source would be to synchronise on it calling `sync`. Registration (and de-registration) of the listener with the adaptor should

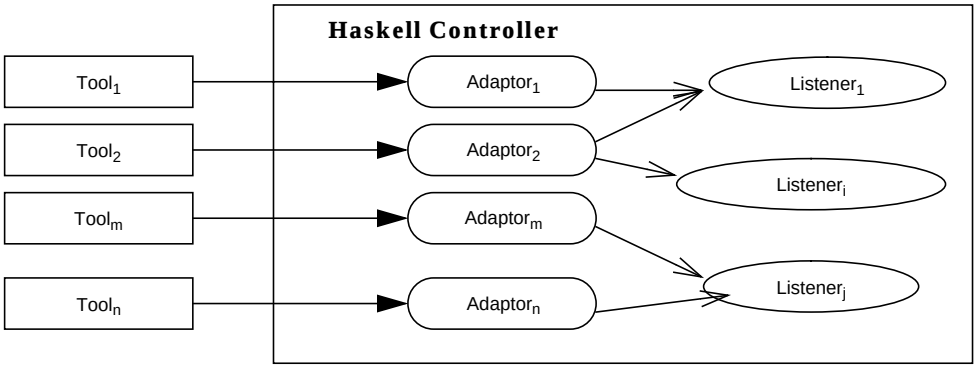

Figure 1: Reactive System Architecture

then happen behind the scene, just as when one synchronises on channel events. Such external events could then be combined to form new composite events, and one could freely mix internal and external events (using `+>`), thus having a uniform and composable framework for event handling independent of the actual source of the event.

Considering the amount of events in systems like integrated SDE's, and knowing the cost of selective communication, it would be quite unrealistic to represent each single external event (e.g. a button click) directly in terms of a unique channel event. Our solution to the problem is to require that every adaptor presents, to the set of listeners, a uniform interface that abstracts over the details (and possible idiosyncrasies) of the encapsulated event source. Similar to the Java Beans architecture, a decoupling between the listeners and the adaptors is enforced. All external events of the universe are required to be uniquely identifiable in terms of values of type `EID`. The events are passed onwards, from an adaptor to the registered set of event listeners, in terms of tuples consisting of the `EID` and the information associated with the event. Event relevant information is in turn represented in terms of values of the dynamic type (`Dyn`).

The need for selective communication can now be reduced since each event listener can use a single private channel for receiving all kinds of external events independent from the actual source of the event. We refer to such a listener as an *interactor*. The event listeners represent by convention an external event in terms of a quadruple consisting of: 1) the event id, 2) a registration command, 3) a de-registration command and 4) a continuation function that specifies the reaction to the event. The latter is, for an event of type `EV a`, of continuation function of type (`Dyn -> IO a`). 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 event listener. When an external event arrives, the listener identifies the associated continuation function, passes the received event information to it as an argument and executes the associated reaction to the event.

The adaptors, on the other hand, act as wiring managers between the sources and the listeners. 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.

It is quite interesting to observe that this model does not require more than is already in the concurrency toolkit (apart of course, from the dynamic types). The communication between the adaptors of a reactive system and the listeners is fully definable using synchronous channel events combined with the core features of Haskell.

3.1 Event Listeners

A generic event for listening to external interactions is provided by the function **listen**. The predefined reaction associated with this event is to project the dynamic value received onto its internal type, or to fail with an error should the internal type of the dynamic value not match the external type expected by **listen** (Signature 2). Additional reactive behaviour can be glued onto such an event by using the event-action combinator (**>>>=**). Rather than identifying events directly, we let the interface go through the class **ExternalEvent**, which provide an injection function into type **EID**. The injection (**toDyn**) and projection (**fromDyn**) methods of type **Dynamic** are provided through operations of class **Typeable**. This approach to dynamic typing within an otherwise statically typed language (also to be found in [JP97]), requires explicit instantiations to be given for each element type.

The semantics of the **Register** and **Deregister** arguments are adaptor dependent (more on this below). The effect of executing the register command is to inform the adaptor mediating the event that the given event listener is willing to receive the event. The external events themselves are communicated as tuples (of type **Event**) consisting of the event identifier and the value associated with the event in the form of a dynamic value.

When a listener executes the **sync e** command, it starts out by creating a channel for receiving external events:

```
type Event = (EID, Dyn)
type Listener = Channel Request
type Request = (Event,Channel ())

sync :: EV a -> IO a
sync (EV be ial) = do {
 ch <- channel; registerEvents ial ch;
 sync_be (
     receive ch >>>= (\msg -> do{deregisterEvents ial ch;serve msg ial})
 +>  choose (map ($ (deregisterEvents ial ch)) be)
 )}
```

The external events denoted by **e** are then registered with the relevant adaptors, and the listener is subsequently set up to await the occurrence of an event. Registration in this

---

```
listen::(ExternalEvent e,Typeable a) => e -> Register -> Deregister -> EV a

class ExternalEvent e where
   toEID    :: e -> EID

class Typeable a where
   typeOf   :: a -> TypeTag
   toDyn    :: a -> Dyn
   fromDyn  :: Dyn -> Maybe a
```

Signature 2: Event Listener Interface

context means telling the adaptor that "when this event occurs, please pass it onwards on my channel", i.e. the event listener identifies itself to an adaptor in terms of its *external event channel*. The events that can be accepted by the listener are either one of the base channel events denoted by **e**, or one of the external events just registered. An external event occurrence has the following effect: first, all registered events are immediately de-registered, then the event is handled by executing the associated reaction to the event.

Synchronous events are represented in terms of two lists: one denoting base channel events (**BE**) and the other denoting external interactions (**IA**) (we ignore computed events for the time being):

```
data EV a = EV [IO () -> BE a] [IA a]
type IA a = (EID, Register, Deregister, Action a)
type Action a = Dyn -> IO a
```

Base channel events are in turn represented as functions parameterised over the actual de-registration command: this command is invoked as soon as one of the base events occurs, but before the associated reaction to the event is computed.

When handling an event coming from an adaptor, an acknowledgement is send back to the adaptor that generated the event (the adaptor was kind enough to pass on a reply channel for this purpose). The action associated with the received event is finally executed. The outcome of this action is the result returned by **sync**:

```
serve :: Request -> [IA a] -> IO a
serve ((e,v),ca) ial = do {sync(send ca ()); reaction e ial v}

reaction eid ial v = let(_,_,_,a) = ia in (a v)
  where ia = head (filter (\(eid',_,_,_) -> eid == eid') ial)
```

The reaction to an event, specified in terms of a continuation function, takes the role of a callback, but unlike callbacks, a useful value is returned to the application. Events are furthermore composable, whereas callbacks are not.

3.2 Standard Adaptor

The role of the adaptor is to delegate events from the event source onwards to any listener that has registered interest in the event. Different kinds of event sources may require different kinds of adaptors that map event handling onto the model of synchronous events. Adaptors can therefore be customised on need. We shall demonstrate the principles of the architecture by defining a *standard adaptor*, which is actually used as a basis for setting up some customized adaptors.

The standard adaptor manages a mapping from event identifiers to the set of listeners willing to receive the event. This adaptor is identified in terms of a channel for incoming events, a channel for registration requests, and the current *event registration set*:

```
data Adaptor a = A (Channel (EID,a)) (Channel (IO ())) Registrations
```

The adaptor is in one of three states. In all states it is willing to serve new (de-)registration requests. The registration request is actually communicated over a channel in the form of an IO computation, and the reaction to the request is simply to execute the computation:

```
registration :: Adaptor a -> EV ()
registration (A _ chr _) = receive chr >>>= id
```

In the first state, the adaptor is set up to accept events from the event source:

```
adaptor1 adp@(A che _ _) = sync (
     registration adp >>> adaptor1 adp
+>   receive che >>>= multiplex adp)
```

Having received an event from the source, the adaptor enters a new state where it attempts synchronization by *multiplexing* the event to one of the listeners having registered interest in the event:

```
multiplex adp@(A _ _ rs) e@(eid,_) =
     getListeners rs eid >>=(adaptor2 adp e)

adaptor2 adp e lst = sync (
     registration adp >>> multiplex adp e
+>   choose(map(delegate e)lst) >>>= adaptor3 adp)
```

Delegation to the listener is provided according to a synchronous RPC protocol. A channel is therefore created so that the adaptor can synchronise on the reply once a listener has accepted the event:

```
delegate (eid,v) ls = event (do {
   ch <- channel;
   return( send ls ((eid,toDyn v),ch) >>> return ch)
   })
```

Having delegated the event to one of the listeners, the adaptor gets into a state where it is awaiting the reply before it goes on reading new events from the source.

```
adaptor3 adp ch = sync (
   registration adp
+> receive ch >>> adaptor1 adp)
```

Note that the unicast pattern is a special case of a multicast pattern and as such allows migration from a unicast to a multicast protocol without breaking existing client code. The actual implementation supports both communication paradigms.

3.3 Event Registration

The missing link between an adaptor and a set of event listeners is the set of registration commands. In order for a potential event listener to establish an event flow from a source to the listener, **sync** must call the methods for registering and de-registering events with the associated adaptor.

Each adapter usually provides its own specialization of the **listen** event that hides the definition of the register and de-register commands to the application. For example, when using the standard adaptor, the following event is used by the event listeners:

```
listen' :: (ExternalEvent e, Typeable a) => Adaptor a -> e -> EV a
listen' a e = listen e (register' a eid) (deregister' a eid)
       where eid = toEID e
```

Concerning the standard adaptor, event registrations are maintained by a mutable variable (**MVar**) holding a mapping from event identifications to the set of listeners that are registered as potential receivers of the event. The registration command is a computation that takes the identity of a listener. The effect of course is to add the given listener as a recipient of the event:

```
type Registrations = MVar(FiniteMap EID (Set Listener))
type Register = Listener -> IO ()

register' (A _ chr rs) eid ls = sync (send chr c)
  where c = updVar rs (addListener eid ls)

registerEvents ial ls = sequence [reg ls | (_,reg,_,_) <- ial ]
```

We omit the function **addListener** - it is trivial. The definition of the de-registration command follows the same pattern.

3.4 Event Source

Events generated by the event source are communicated to the adaptor by sending the event over the adaptors event channel. The **emit** function defines this behaviour:

```
emit :: ExternalEvent e => Adaptor a -> e -> a -> IO ()
emit (A che _ _) e v = sync(send che (toEID e,v))
```

The communication with a loosely coupled tool is frequently established using pipes. A low level *reader thread* is then interposed between the event source and the adaptor that waits for a message to arrive over the pipe (calling **threadWaitRead**). The event at this stage is actually represented as a plain string. The string is consequently parsed (using **show**) before it is forwarded to the adaptor for further delegation.

3.5 Iterative Choice

Calling **sync e** suspends the execution of the calling thread until one of the communications denoted by the event **e** triggers. Synchronisation happens at most once. For many reactive systems an *iterative choice* is more applicable and less expensive. An iterative listener can be created by spawning off a new *interactor* that repeatedly listens to a number of events:

```
interactor:: EV () -> IO ()
interactor e = forkIO (do {iterate e})
```

```
iterate e = sequence (map sync (repeat e))

become :: EV () -> IO ()
become e = do {forkIO (iterate e); suicide}
```

In the presence of iterative choice, it makes sense to provide a command by which an interactor may change its event registration set sometime during its lifetime. This is achieved by the **become** command, similar to the **become** command of the Actors model. An interactor has many similarities with the Actor [Agh86] model, but unlike Actors we use synchronous communication and hide the actual message dispatching by representing external interactions in terms of composable first class events. The concrete implementation is slightly more complicated than described above. An iterative server does not re-register the (external) events at each iteration, rather it keeps a record of the registrations made at the last iteration. Each iteration therefore has to deal with changes with respect to the last iteration only, thus speeding up the handling of (external) events significantly.

4 A Concurrent Expect Tool

A problem with many interactive shell tools, such as for example **ftp**, is that they cannot be run without a user interactively supplying the input. A large number of application programs are written with the same fault of demanding user input: **su**, **telnet**, **passwd** etc. **expect** [Lib91] is a tool designed to control interactive programs. An expect script, based on the Tcl language, defines the dialogue with the interactive tool in terms of send/ expect sequences: the **expect** tool sends commands to the interactive tool, and receives responses in the form of strings. **expect** supports regular expressions and can wait for multiple strings at the same time, executing a different action for each kind of string. Expect may therefore be used to set up new non-interactive versions of existing interactive tools that can be called within programs because they no longer require user interaction.

We have developed a functional tool solving the same set of problems, which uses Haskell and first class synchronous events for matching responses. The functional **Expect** tool offers in essence, 3 commands and 2 events for encapsulating interactive tools:

- The **newExpect** command starts a program and returns a handle to the tool. The call specifies the name of the program as well as additional tool specific parameters.
- The **execCmd** command sends a request to the tool. The request is represented as a plain string.
- The **match** event defines a pattern of interest and thus plays the important part of receiving and reacting to the responses of the external tool. The pattern is expressed in terms of a regular expression. The matched string is returned as the result.

```
data Expect
newExpect    :: FilePath -> [Config OSProcess] -> IO Expect
closeExpect  :: Expect -> IO ()
execCmd      :: String -> Expect -> IO ()
match        :: String -> Expect -> EV String
terminated   :: Time -> Expect -> EV Status
```

Signature 3: Functional Expect Tool

- The **closeExpect** command shuts down the external tool and the corresponding encapsulation on the Haskell side.
- The **terminated** event returns a status value when the **Expect** tool terminates.

An interactive tool can now be controlled by setting up a listener that synchronises on tool events using **match**. The encapsulation in Haskell of archetypical Unix tools such as **latex**, **mail**, **ftp** and **telnet** is, in principle, very similar, but also quite lengthy since we must deal with a number of errors and special cases. We shall therefore demonstrate the concepts using a smaller example, namely that of setting the user identity (**su**).

4.1 Set User Command

A Haskell command that allows one to become another user without logging off can be implemented using the Unix **su** command. The user name is the first argument passed to the **setUser** function, the password the second:

```
setUser:: String -> String -> IO ()
setUser user pwd = do {
  su <- newExpect "su" [arguments [user]];
  sync (   match "^su: Unknown.*$\n" su >>> fail unknownUser
       +>  match "^Password.*" su >>> execCmd (pwd ++ "\n") su);
  sync (   match "^su: Sorry.*$\n" su >>> fail illegalPassword
       +>  timeout (secs 1) >>> done);
}
```

The command for performing background **su** emulates the typical dialogue between a user and **su**. Requests from **su** are matched by appropriate **match** events, and the responses are forwarded to **su** in the form of strings. First, a new **su** process is spawned off to run in the background. The first call to **sync** looks for a password prompt (or an error). The listener responds by sending the password to **su** (or by failing with an error). The second call to **sync** checks whether the password was ok or not. If so, **su** doesn't respond. In situations where there is no prompt from the external tool, a **timeout** can be used: or better, one may attempt to detect that the tool has terminated.

4.2 Termination Events

Process termination can be captured in Unix by calling the **getProcessStatus** command. In a concurrent setting we must take care that this call does not block the thread scheduler. This requires in turn that a polling "jacket" is written which investigates, at regular intervals, whether the process is running or not. The termination event is defined by wrapping a promise around an active polling loop. The **jacket** command actually takes the role of a customized adaptor turning a operating system event into a first class synchronous event:

```
terminated :: Time -> Expect -> EV Status
terminated t =
  event . promise . (jacket t) . (getProcessStatus False True) . getProcessID

jacket :: Time -> IO (Maybe a) -> IO a
jacket d c = do { s <- c;
   case s of
      Nothing -> do {delay d;jacket d c}
      (Just s) -> return s}
```

4.3 Composite Events

In general, it cannot be assumed that a response coming from a interactive tool is terminated by a newline (prompts usually aren't). The **Expect** tool therefore reads in chunks of text from the tool, and attempts the registered set of patterns to see if anything matches. The more frequently a match is found, the less the number of characters buffered and the less the time needed to match a response. In the presence of long input lines, it therefore makes sense to split the line into small fragments until an end-of-line character finally occurs. The following composite event expresses such a line matching pattern:

```
matchline:: Expect -> EV String
matchline exp = match "^.*" exp >>>= sync . (remainder exp)

remainder :: Expect -> String -> EV String
remainder exp l =
      match ".*" exp >>>= sync.(remainder exp).(l++)
  +>  match "$\n" exp >>> return (l++"\n")
```

4.4 Advantages over Tcl/expect

Expect has been fully implemented in Haskell in terms of approximately 100 LOC on top of the UniForM Concurrency Toolkit and the regular expression utility of GHC. The **Expect** adaptor is a slightly customized version of the standard adaptor. First, characters read by the reader thread must now be buffered until a prefix of the buffer matches one of the registered patterns. A rematch should furthermore be initiated every time a new pattern is registered with **Expect** to see if it matches the contents of the buffer.

Compared to Tcl/expect [Lib91], the Haskell variant provides an improved integration

framework by being type-safe and concurrent. However, the major argument in favour of a functional approach is its support for abstraction. Events are fully composable: giving a set of predefined building blocks one may define more derived abstractions, such as `timeout`, `terminated` and `matchline`. Using Tcl/expect, such commands are either built in (`timeout`) or not supported at all (`matchline`). There exist work-arounds to event composition such as the specializations `expect_before` (and `expect_after`) where pattern-action pairs from the most recent `expect_before` are implicitly added to the beginning of any following `expect` command. The choice operator of `Expect` (`+>`) provides a much more general and functional solution for composing events.

5 Graphical User Interface

Once an interactive tool has been encapsulated using `Expect`, one may be tempted to go one step further. Numerous programs are interactive but non-graphic. A prominent example is the hugs interpreter. In many cases, converting these to use graphic user interfaces would be beneficial [Lib94]. Using Haskell, it is actually possible to wrap GUI's around existing tools. The concurrent `Expect` tool is complemented with an (equally concurrent) encapsulation of Tk. Wrapping a functional interface around Tk is not a new direction of research [VST96,LWW96,FGPS96]. HTk (acronym for Haskell-Tk) [Kar97] is distinguished by being a strongly typed, *concurrent* encapsulation of Tk. It has been designed to support abstraction and composability, for what concerns dynamic as well as static aspects of a user interface. Event handling is therefore done by synchronising on first class, composable events making the handling of user interactions uniform to all other kind of events generated within the system. Another advantage is that it eases the development of composite and customised widgets whose events may now be defined by appropriate compositions over the base events of the constituent widgets. An extensive class hierarchy has furthermore been set up, which defines the general commands and configuration options of the GUI. This class hierarchy has been instantiated to provide most of the functionality of the build in widgets offered by Tk as well as composite widgets developed directly in HTk.

In principle, the encapsulation of Tk follows the pattern suggested by the `Expect` tool: commands are forwarded to Tk in the form of strings, and responses are caught and handled by a pattern matching listener. The encapsulation of `wish` is, however, significantly more complicated than that, since a local cache is needed to record information about the widgets and their associated properties. A detailed presentation of these issues (or the overall features provided by HTk) is clearly beyond the scope of this paper. We shall briefly introduce some of the salient concepts of HTk by wrapping a GUI around the `su` command, and then go on with a larger integration case study, namely that

```
setUserGUI = do {
  e <- entry [];
  l <- labelbox e [text "User:"];
  win <- openWin l [title "Set User"];
  usr <- awaitInput e;
  configure l[text "Password"];
  configure e [showText '*', value ""];
  pwd <- awaitInput e;
  catch (setUser usr pwd)
     (\e -> openErrorWin e []);
  destroy win
} where awaitInput e =
    sync (keyPressed e "Return" >>> getVar e)
```

Figure 2: GUI Wrapper for set user

---

of wrapping a graphical user interface around **hugs**.

5.1 Set User (Continued)

A minimal solution for wrapping a GUI around **su** (see fig. 2) is to use one and the same **entry** widget to enter the user identity and the password. The entry is packed into a **labelbox** - a customized packer for associating a label with an arbitrary widget. The window is then opened and the user is queried for the user identity. Having entered the identity, the window is reconfigured to prompt for the password. Finally, the **setUser** command is called, with passes control to an **Expect** tool. Errors are caught and displayed by a dialogue box.

A number of comments may be in order regarding the GUI commands. Each object is identified by a unique and typed handle rather than by name, e.g. the type **Entry a** denotes entries displaying text of type **a**. The handle is returned when invoking the command for creating a new widget of the kind (**entry**). Initial configuration options may be specified at creation time. Later on, the configuration options can be changed by applying the **configure** command.

5.2 User Interaction

Interaction with the user is expressed in terms of first class, synchronous events. The base operation for handling user interaction is the **interaction** event:

```
interaction ::(GUIObject o,GUIEvent e) => o -> e -> EV GUIEventInfo
interaction w e = listen e' (bind w e') (unbind w e')
       where e' = toEID (w,e)
```

Specializations of this event, such as for example those needed to capture key presses or button clicks, are easily provided:

```
clicked :: Button () -> EV ()
clicked w = interaction w Clicked >>> return ()

keyPressed :: GUIObject o=> o -> String -> EV Position
keyPressed w key = interaction w p >>>= selPos where p = KeyPress (Just key)
```

Each interaction is uniquely defined in terms of its origin (an interactive object) and its event pattern. The class **GUIEvent** defines the class of values that can be used to designate Tk event patterns (e.g. a key press, optionally preceded by a list of modifiers). The type **GUIEventInfo** represents the type of information returned when a Tk event occurs, which amounts, among others, to positioning information.

The registration command is a refinement of the standard adaptor command that has the additional effect of passing a **bind** request to Tk should the registration set associated with the event be empty. The bind request instructs Tk to report the event when it occurs. Should the set of listeners associated with this event become empty again, then a similar **unbind** command is send to Tk. The nice side effect of such a scheme is that Tk can only generate events for which there are registered listeners.

A subset of the widgets are equipped with a single standard event, i.e. buttons. These events are defined through the class **Trigger**. Active widgets are polymorphic, a solution that has been inspired by Haggis [FP95]. A base **clickbutton** is of type (**Button ()**), meaning that the unit value is returned when the button event triggers. Buttons are functors however, and can be mapped to return a value more suitable for the application at hand.

```
class Trigger t where
   triggered :: t a -> EV a

data Button a = Button GUIOBJECT (() -> IO a)

instance Functor Button where
   map f (Button b g) = Button b ((map f) . g)

instance Trigger Button where
   triggered (Button b f) = clicked (Button b return) >>>= f
```

Notice that every object has an associated untyped representation in terms of a value of type **GUIOBJECT** (basically an **MVar** providing MT-safe access to the object's attributes).

A menu is an example of a composite widget that consists of a number of menu items. Similar to buttons, menus are parameterised over the type of the value returned when a menu item is invoked:

```
data Menu a = Menu GUIOBJECT (MVar [Button a])
```

The current set of menu items of the menu is associated with the **Menu** handle. A composite trigger matching the selection of one of the menu items is therefore given by the following instantiation:

```
instance Trigger Menu where
  triggered (Menu _ v) = event ( do {
     bts <- getVar v;
     return (choose ((map triggered) bts))
  })
```

6 Hugs Interpreter

The Haskell Users Gofer System (**hugs**) is running as a shell tool. Hugs for windows (**winhugs**) [JP97] offers a GUI front-end for Microsoft Windows. A more portable solution can be achieved by encapsulating the existing hugs interpreter using **Expect** and HTk. The **Expect** tool plays a central role in controlling the dialogue between the user and hugs running in the background. HTk is used to provide an interface featuring a scrolling console window that looks like the normal hugs interface.

A simplified listener for hugs is presented below, in terms of an interactor that repeatedly listens to the events generated by either the user, by hugs or by the operating system. The **newHugs** command creates a new instance of the hugs interpreter and returns a handle to the tool. It starts up a new expect tool to interact with hugs (*model*), creates the window (*view*) and turns the control of the system over to an interactor (*controller*). The interactor handles user interactions, fetches responses from hugs and listens to termination events.

```
newHugs :: IO Hugs
newHugs = do {
   exp <- newExpect "hugs" [];
   (win,tp,mns) <- mkHugsWin exp;
   interactor (
       match "^? " exp >>>= (appendText tp)
   +>  keyPressed tp "Return" >>> do {
         cmd <- getInput tp;execCmd (cmd++"\n") exp}
   +>  matchline exp >>>= (appendText tp)
   +>  choose (map triggered mns) >>> done
   +>  closed exp win >>> do {closeTool exp;suicide}
    );
   return (Hugs exp win tp)
   }
```

The user interface consists of a console window equipped with scrollbars, a menubar of hugs command and a number of torn-off menus. The **mkHugsWin** command sets up the window and the associated menus and returns a handle to the window, the text widget and the list of menus making up the menubar. The various commands and options of hugs are all available as menu items. Dialogue windows are in turn used for entering user input such as the repeat string should the command require interaction with the user.

Expression evaluation and command execution is taken into account by the first 3 events. The **match** event catches the prompts generated by hugs (Version 1.0). The **keyPressed** event is triggered when the user wants an expression to be evaluated pressing the **Return**

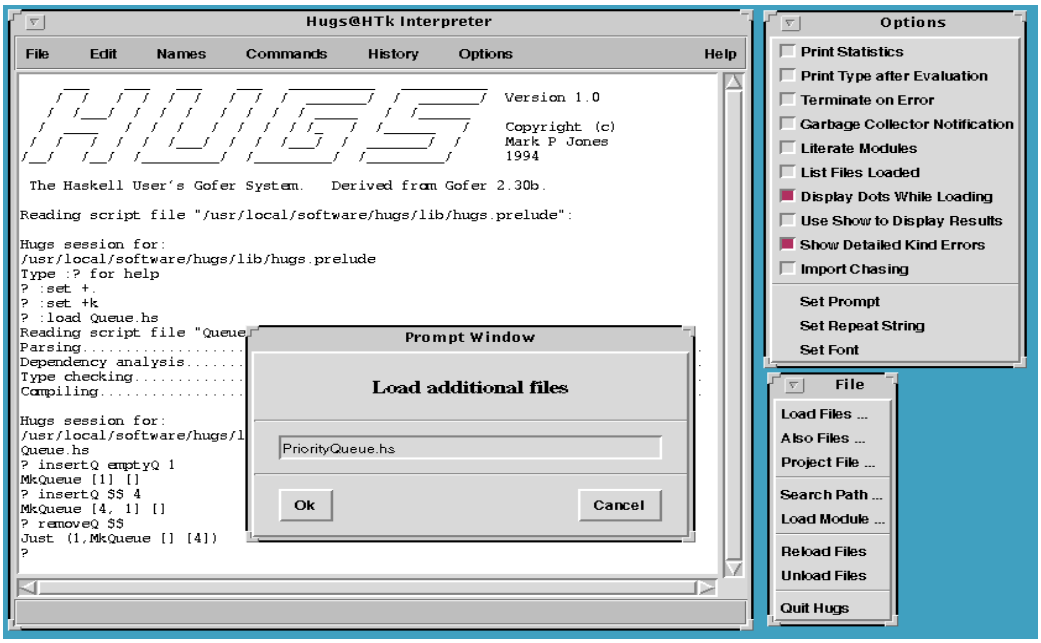

Figure 3: GUI interface to the hugs interpreter

---

key. The controller reads the expression entered by the user and forwards it to hugs for evaluation. The **matchline** event matches the result generated by hugs in response to such a request. Menu invocations are in turn handled by the **choose** event. Given the list of menus, **map** is applied to generate a list of events, each awaiting menu invocations within a single menu.

The last event of relevance is the event signalling that the session with hugs has ended. The **closed** event is actually a composite event since there are at least 3 ways to terminate hugs: by destroying the window, by executing a Unix kill command, or by submitting the "**:q**" command to hugs. Hugs responds to the last request by printing "[Leaving Hugs]". The event triggered by hugs being terminated is therefore defined as:

```
closed exp win =
      destroyed win
 +>   terminated (secs 2) exp >>> done
 +>   match "^[Leaving Hugs]$\n" exp >>> done
```

In all three cases, the reaction to the event is the same. The example is interesting because it demonstrates the uniformity and advantage of having events as first class values.

7 Conclusion

Starting with synchronous events in the style of CML, it is possible to provide an adequate approach to the handling of external events in a reactive system architecture by introducing a concept of adaptors and interactors. The advantage of this approach is that all kind of events of a reactive system can be given a representation as first class, synchronous and composable values, independent of the actual source of the event. A uniform framework for presentation, control and platform integration is consequently achieved. However, for some tools (such as for example Tk), quite an amount of re-engineering is necessary in order to hide the idiosyncrasies of the tool behind more friendly (and safe) abstractions.

We are currently using the technology as a basis for the UniForM WorkBench [KPO+96], which offers a framework for developing integrated Software Development Environments given existing off-the-shelf tools. Haskell plays a major role in this context by being the central integration language. The WorkBench has been established by reusing existing tools such as Tk, the graph visualization system daVinci [FW97], and the Portable Common Tool Environment [ECM93]. Adaptors have been wrapped around the tools, and synchronous events are therefore used to represent operating system signals, database notifications and user interactions, among others.

The generic UniForM WorkBench is currently being instantiated to develop a SDE for specification and proof [KSW96,KLMW97] of Z specifications. This SDE will soon be extended with another track supporting specification and proof of CSP specifications. The `Expect` utility has proven to be extremely useful in this context to develop Haskell wrappers for existing Z tools such as the Z type-checker, configuration manager, prettyprinter and proof tool. The encapsulation of the transformation and proof tool (HOL-Z, a structure preserving encoding of Z in Isabelle [Paul95]) is an interesting case study in itself since it demonstrates how `Expect` can be used to control an application normally running within an SML session.

The encapsulation of hugs is more a side-track - the first prototype was developed within one afternoon just for the fun of it. It has demonstrated to us that using Haskell with the right kind of abstractions is a viable approach to tool integration. The experience would be even more positive with multiple parameter type classes, support for orthogonal persistence and a more general approach to dynamic typing.

Acknowledgments

The author would like to thank Walter Norzel for implementing selective communication and Stefan Westmeier for many discussions on the odds and ends of Tk.

References

[Agh86] G. Agha: *Actors - A Model of Concurrent Computations in Distributed Systems*, The MIT Press, 1986.

[BK94] J. A. Bergstra, P. Klint: *The ToolBus - a Component Interconnection Architecture*, Technical Report P9408, University of Amsterdam, 1994.

[COM95] *The Component Object Model Specification*, Draft Version 0.9, October 24, Microsoft Corporation and Digital Equipment Corporation, 1995.

[ECM93] European Computer Manufactorers Association: *Portable Common Tool Environment (PCTE) - Abstract Specification*. ECMA Report Number 149, 1993.

[FP95] S. Finne. S. Peyton Jones: *Composing Haggis*. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms for Computer Graphics*, Springer Verlag, 1995.

[FGPS96] T. Frauenstein, W. Grieskamp, P. Pepper, 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, LNCS, Springer Verlag, 1996.

[FW97] M. Fröhlich, M. Werner: *daVinci V2.0.3 Online Documentation*. Universität Bremen, http://www.informatik.uni-bremen.de/~davinci, 1997.

[JP97] M. P. Jones, J. C. Peterson: *Hugs 1.4, The Nottingham and Yale Haskell User's System*, User Manual, April 1997.

[Kar97] E. W. Karlsen: *Haskell-Tk - A Concurrent Encapsulation of Tk*. Technical Report, Universität Bremen, 1997.

[KLMW97] Kolyang, C. Lüth, T. Meyer, B. Wolff: TAS and IsaWin: Generic Interfaces for Transformational Program Development and Theorem Proving. In Proc. *TAPSOFT'97*, Lille, France, LNCS 1214, 1997.

[KN97] E. W. Karlsen, W. Norzel: *The UniForM Concurrency ToolKit*, Technical Report, Universität Bremen, 1997.

[KPO+96] B. Krieg-Brückner, J. Peleska, E. R. Olderog, D. Balzer, A. Baer: UniforM, Universal Formal Methods Workbench. In U. Grote, G. Wolf (eds): *Statusseminar des BMBF: Softwaretechnologi*. Deutsche Forshungsanstalt für Luft- und Raumfahrt, Berlin 1996. See also http://www.informatik.uni-bremen.de/~uniform.

[KSW96] Kolyang, T.Santen, B. Wolff: A Structure Preserving Encoding of Z in Isabelle/HOL. In *The 1996 International Conference on Theorem Proving in Higher Order Logic*, Turku, Finland. LNCS 1125, 1996.

[Lib91] D. Libes: expect: Scripts for Controlling Interactive Processes. In *Computing Systems*, Vol. 4, No. 2, Spring 1991.

[Lib94] D. Libes: X Wrappers for Non-Graphic Interactive Programs. *In Proceedings of Xhibition 94*, San Jose, June 20-24, 1994.

[LWW96] C. Lüth, S. Westmeier, B. Wolf: *sml-Tk - Functional Programming for Graphical User Interfaces*, Technical Report, Universitat Bremen, 1996.

[OMG95] The Object Management Group: *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, July 1995.

[Ous94] J. K. Ousterhout: *Tcl and the Tk toolkit*. Addison Wesley, 1994.

[Paul95] L. Paulson: *Isabelle: A Generic Theorem Prover*. LNCS 828, 1995.

[PGF96] S. Peyton Jones, A. Gordon, S. Finne: Concurrent Haskell. In *Principles of Programming Languages '96 (POPL'96)*, Florida, 1996.

[PNR97] S. Peyton Jones, T. Nordin, A. Reid: Green Card: a foreign language interface for Haskell. In *ACM Sigplan Haskell Workshop*, Amsterdam, 1997.

[PW93] S. Peyton Jones, P. Wadler: Imperative Functional Programming. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, 1993.

[Sun96] Sun Microsystems: *JavaBeans 1.0*, JavaSoft, December 4, 1996.

[Rep92] J. H. Reppy: *Higher-Order Concurrency*, Ph. D. Thesis, Department of Computer Science, Cornell University, 1992.

[VST96] T. Vullings, W. Schulte, T. Schwinn: The Design of a Functional GUI Library using Constructor Classes, In *Perspectives of System Informatics*, LNCS 1181, 1996.