Using Concurrent Haskell to Develop Views over an Active Repository

Einar W. Karlsen and Stefan Westmeier

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

Abstract. The UniForM WorkBench is an integration framework with Haskell features in support of data, control and presentation integration. Using the WorkBench, it is possible to implement the entire Software Development Environment for Haskell - using Haskell itself. The paper presents the higher order approach to event handling used within the WorkBench, as well as the persistent repository with version management support. It is then demonstrated how views over this repository are kept consistent, on the fly, in a multiuser environment using the Model-View-Controller paradigm. Interactors are set up to maintain consistency between a view and its underlying repository by coordinating database change notifications and user interactions. These events are represented as first class, composable event values.

1 Introduction

There is a strong tradition within the functional programming community, that the compiler for a functional language (e.g. Haskell) is bootstrapped. Not only the compiler, but even the entire *Software Development Environment* (SDE), with integrated tool support, graphical user interfaces, and version and configuration management, could be implemented very elegantly using Haskell itself, under the provision of course, that the right kind of integration technology were available.

The *UniForM WorkBench* [1] [KPO+96] is such an integration framework offering services for *data integration* (DBMS), *presentation integration* (GUI), and *control integration* in Concurrent Haskell [PGF96]. The WorkBench is set up along the imperative functional programming paradigm [PW93], extended with a higher order approach to event handling inspired by CML [Rep92]. The main achievement over CML, however, is that internal channel events as well as tool events, such as user interactions and database change notifications, are uniformly represented in terms of composable and abstract event values that entirely hide the source of the event [Kar97b].

[1] This work has been supported by the German Ministry of Research (BMBF) on Project UniForM ("Universal Formal Methods WorkBench")

We have used this integration framework to develop a prototypical environment for Haskell program development - the *Hugs WorkBench* - that integrates Haskell tools such as the Hugs interpreter [JP97] around a persistent repository providing version and configuration management of Haskell modules. The first step in achieving such an integration is to wrap Haskell interfaces around the pre-fabricated development tools (e.g. Hugs). Haskell is from then on used to glue the entire system together. In essence, the Hugs WorkBench turns out to be a reactive system, where events amount to user interactions, tool events, operating system events and database change notifications.

This paper is concerned with a central issue of such an integrated environment: the provision of consistent views over the repository. Other issues, related to control integration and tool encapsulation, are for example discussed in [Kar97a]. The intricate aspect of an SDE is that it is inherently multiuser. While one user is viewing a version graph, another may change it by creating or pruning versions. Views over the repository must therefore be maintained dynamically - on the fly. It shall be demonstrated how this can be done very elegantly by using a functional language extended with a higher order approach to event handling.

The remainder of this paper is organized as follows. The architecture of the WorkBench is introduced in the next section. First class synchronous events and tool events are discussed in section three and four - respectively. The fifth section presents the main features of the Repository Manager, in particular its support for attributed and versioned objects. The sixth section demonstrates the application of this technology on a larger example by defining a version graph over the repository. The results are then discussed.

2 Architecture

The UniForM WorkBench has been established on top of existing, public domain tools. Data integration is provided by the *Repository Manager* (RM), which takes its basis in H-PCTE [Cre94] - a persistent object management system that implements an extended subset of the Portable Common Tool Environment [ECM94]. H-PCTE is a distributed and active DBMS for managing persistent graphs.

Presentation integration is provided by the *User Interaction Manager* (UIM) [Kar97c], which consists of Haskell encapsulations of Tk [Ous94] and the graph visualization system daVinci [FW97]. *Haskell-Tk* serves as a general purpose GUI for wrapping graphical user interfaces around non-graphic tools. The graph visualization system *daVinci* is, in turn, used to visualize graph structures of the RM such as version and configuration graphs.

Control integration is provided by a number of concurrently executing *interactors* that are set up to listen to the events of the integrated system. The response to an event is then defined in terms of a computation that usually calls one or more commands of the tools making up the environment.

The Hugs WorkBench is illustrated in Fig 1. A version graph for the Queue module is visualized using daVinci. New revisions are made using a text editor. Attributes of a revision, such as the informal description, are edited using an input form. A graphical interface to the Hugs interpreter serves as a typechecker. All tools are invoked within the daVinci view.

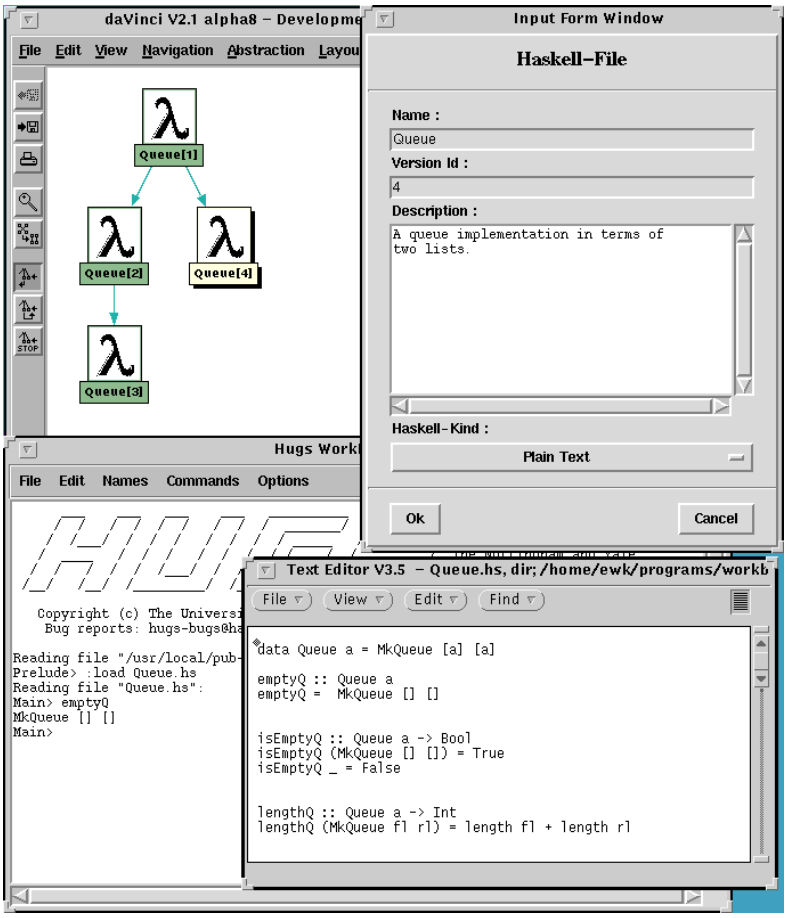

Fig. 1. Hugs Workbench

The Hugs WorkBench is a reactive system designed and implemented according to the *Model-View-Controller* (MVC) [KP88] paradigm. The repository takes the role of the model, the daVinci graph the role of the View and the set of event listening interactors the role of the Controller. When the user starts up the text editor the following happens. A new version is first

created by calling commands of the RM. A change notification is then sent by the RM to all interactors having registered interest in the generated event. The interactor controlling the version graph reacts to this event by adding a revision link and a new version node to the version graph.

By using the MVC paradigm throughout to maintain views over the repository, it does not matter, from the point of view of the implementation, whether the changes have been made (1) within the current view, (2) within another view running in the same WorkBench or (3) by a completely different user at a remote machine. Distribution is achieved, almost for free!

3 Synchronous Events

The *UniForM Concurrency ToolKit* [Kar97b] extends Concurrent Haskell [PGF96] 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. 2):

- `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 synchronises 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.

Communication is by handshake between two threads, which means that a sender and a receiver must perform a rendezvous in order communicate.

3.1 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:

```
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

interactor :: EV () -> IO InterActor
become     :: EV () -> IO ()
stop       :: IO ()
```

Fig. 2. Synchronous Events

- 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 `stop` terminates the current interactor.

Interactors provide a refinement of the *Actor model* of Gul Agha [Agh86]. However, 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!

3.2 Example

Assume for a moment that tool events, such as user interactions and RM change notifications, are represented as values of type `EV`. Suppose then, that we would like to use a counter widget to view and update some integer attribute of a persistent object (Fig. 3). This may sound as a simple and artificial example, but it actually demonstrates the principles used to construct complex views of the repository such as version and configuration graphs.

The counter widget consists of an entry displaying the current count and 2 buttons for initializing and decrementing the count. The count attribute of the persistent object `obj` can be changed, either when the user interacts with the counter widget `cnt`, or when some other application sets the count

attribute of the persistent object calling operations of the RM. An interactor, serving as a controller for the counter, is then defined as:

```
interactor (
      updated cnt          >>>= \v -> setAttr anm v
  +>  objModified obj anm  >>>= \v -> setCount cnt v
  +>  destroyed cnt        >>>  stop
  +>  objDeleted obj       >>>  do {destroy cnt; stop})
  where anm = countAttr obj

updated :: Counter -> EV Int
updated (Counter bs bp ent)  =
      clicked bs  >>> updEntry ent succ
  +>  clicked bp  >>> updEntry ent pred
```

The first event-action pair ensures that the count attribute associated with the persistent object `obj` is set every time that the user changes the count by interacting with the counter widget `cnt`. The `updated cnt` event is a composite event that listens to the button clicks, and returns the new count when triggered (it also updates the entry behind the scene).

The second event-action pair is set up to react to change notifications coming from the RM. The event `objModified obj anm` occurs whenever the attribute `anm` associated with the persistent object `obj` is changed. The reaction to this event is to change the view by updating the count.

The two last event-action pairs are set up to terminate the interactor should the counter widget or the persistent object be destroyed. The interactor will automatically deregister all registered events in such a case.

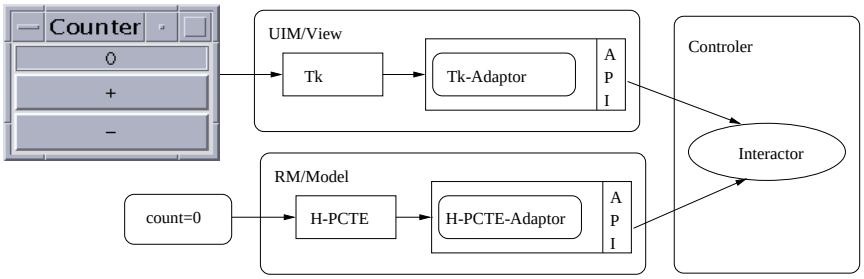

Fig. 3. Model-Controller-View Paradigm in Action

Tk and H-PCTE are foreign tools, implemented in Tcl and C. A Haskell API has then been wrapped around the tools, which defines the types, computations (`IO`) and events (`EV`) of relevance. It is the role of the Tk adaptor and the H-PCTE adaptor to turn physical events of the two foreign tools, whatever representation they may have, into first class composable events.

4 Tool Events

In a framework to event handling based on synchronous events, we would like to represent tool events, such as user interactions and repository change notifications, 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 and external 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.

The Workbench is based on a concept of event *sources*, *adaptors* and *interactors* to achieve this degree of abstraction (Fig. 3). The main component is the adaptor which is interposed between the physical event source (i.e. a GUI or a DBMS) and the interactors of the application. It is the role of the adaptor to turn a tool 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.

Tool events (Fig. 4) are communicated from the event adaptor to the set of listening interactors in terms of tuples `(eid,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 deregister, 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 deregistration command has the opposite effect. Both commands are executed behind the scene when `sync` is called.

```
class EventDesignator e where
  toEventID  :: e -> EventID

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

type Register = InterActor -> IO ()

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

Fig. 4. 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 provided through class `Typeable` [JP97], and 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 `(eid,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.

4.1 Specializations

When a Haskell interface is wrapped around a reactive tool such as a GUI, specializations of the `listen` event are usually defined that hide the registration commands of the adaptor. *Haskell-Tk* defines for example all user interactions in terms of the following specialization:

```
interaction ::(GUIObject o,GUIEvent e) =>
              o -> e -> EV GUIEventInfo
interaction w e = listen eid (regGUI eid) (deregGUI eid)
      where eid = toEventID (w,e)

clicked bt   = interaction bt Clicked >>> return ()
```

Each `interaction` is uniquely defined in terms of its origin (a graphical object) and its (Tk) event pattern. The class `GUIObject` defines the class of graphical objects such as widgets, whereas the class `GUIEvent` defines the class of event patterns (e.g. a key press or a mouse motion). The type `GUIEventInfo` represents the event descriptor returned when a Tk event occurs, which amounts to mouse positioning or key press information. The registration command (`regGUI eid`) registers the event with the Tk adaptor. The events associated with active widgets, such as buttons and menus, are then defined by further specializations of the interaction event.

5 Repository Manager

The *UniForM Repository Manager (RM)* serves as persistent store for software objects like specifications, proofs, programs, test runs, manuals etc. These objects are linked together by *development links* (revision, refinement)

and various *dependency links*. e.g. import links, test-spec links, documentation links etc. Thus, the repository is, from an abstract point of view, a persistent graph machine.

The RM is based on a Haskell encapsulation H-PCTE, which extends a traditional file system with features in support of attributed objects, object types, fine grained links, transactions and last but not least, change notifications. The Haskell API abstracts over the underlying functionality provided by H-PCTE. Classes are set up that provide generalized functionality of relevance to objects, links and attributes. Change notifications from H-PCTE are furthermore represented as first class composable event values by interposing an adaptor between H-PCTE and the interactors of the application.

A subset of the Haskell API is given in Fig 5. We shall briefly introduce some of the commands and demonstrate how they can be used to define some of the computations needed for managing versions of Haskell modules. Essentially, the version graph is a directed acyclic graph, where the nodes denote modules and the edges revision links.

```
class RMObjectC o where
  getAttr   :: (RMObjAttrC a,RMAttrValC av)
            => (a o) -> IO av
  setAttr   :: (RMObjAttrC a,RMAttrValC av)
            => (a o) -> av -> IO ()

class RMObjectC o => RMIdObjectC o where
  getPOId   :: o -> IO POId

class RMLinkC l where
  lnkDelete :: (RMObjectC s,RMObjectC d)
            => (l s d) -> IO ()

getTargets  :: (RMLnkTypeC lt,RMObjectC o,RMObjectC o')
            => lt -> o -> IO [o']
```

Fig. 5. Repository Commands

5.1 Objects

Persistent objects are instances of the class `RMObjectC`. Application objects (`RMIdObjectC`) are furthermore uniquely identified by a *persistent object identifier*. Haskell modules are, for example, persistent objects of type `HaskellObj`:

```
data HaskellObj
instance RMObjectC HaskellObj where {...}
instance RMIdObjectC HaskellObj where {...}
```

Objects are attributed. The computation `getAttr (anm o)` retrieves the value of the attribute `anm` associated with the persistent object `o`, whereas the computation `setAttr (anm o) v` sets the attribute `anm` of object `o` to the value `v`. The computation that retrieves the content of a Haskell module is for example defined as:

```
getContent :: HaskellObj -> IO String
getContent hobj = getAttr (haskellContentAttr hobj)
```

Other attributes are for example used to hold the name, description, version number and creation date of a Haskell module.

Attribute values amount to predefined and scalar H-PCTE attribute types, Haskell strings (any instance of `Read` and `Show` actually), as well as so-called *long field attributes*. Long field attributes are used to represent the content of files, which are stored within the repository in order to maintain the ACID property of transactions for objects as well as for files. Long field attributes can be exported to the file system (and imported again) in order to make the files accessible to Unix tools. For example, the content of a Haskell module is stored in a long field attribute, but exported to the file system whenever a user wants to edit, interpret or type check the file using the appropriate Unix tools for doing so:

```
exportModule :: HaskellObj -> FilePath -> IO ()
exportModule hobj fnm = getContent hobj >>= writeFile fnm
```

5.2 Links

Links are binary relationships parameterised over the type of the source and target object. The class of links is defined by `RMLinkC`. A revision link for Haskell modules is for example defined as:

```
data RevisionLnk a b = RevisionLnk a b
instance RMLinkC (RevisionLnk HaskellObj HaskellObj) ...
```

Deletion works for links by invoking the computation `lnkDelete` l, whereas objects are garbage collected whenever they can't be reached from the persistent root any more. This approach of H-PCTE fits very well with the paradigms of functional programming.

Each link has an associated link type designator that defines the role of the link. The computation `getTargets r o` retrieves the objects reachable from `o` following links with link type designator `r`. The operation `getRevision o`, that returns the revisions of a given Haskell object by retrieving the targets of all `sucessorOf` links emerging from `o`, is quite easily defined as:

```
getRevisions :: HaskellObj -> IO [HaskellObj]
getRevisions = getTargets successorOf
```

5.3 Change Notifications

H-PCTE is an *active database* that allows an application to receive notifications whenever the object base has changed. The kind of events reported by the RM amount to object deletion events (`objDeleted`), link deletion events (`linkDeleted`), link append events (`linkAppended`) and attribute modification events (`objModified`).

The notifications are implemented in terms of a single `changeNotification` event, that is parameterised over the class of RM event designators (Fig. 6). The event is a simple specialization of the `listen` event.

```
class RMEventDesignator e where
  getRMEventID :: e -> IO (EventID)

changeNotification :: RMEventDesignator e => e -> EV RMEventInfo
```

Fig. 6. Change Notification Event

The events provided to the casual application developer are straightforward refinements of the `changeNotification` event. For example, the event that is triggered when an object is garbage collected is defined as:

```
objDeleted :: RMIdObjectC o => o -> EV ()
objDeleted o =
  changeNotification (o,HPcteObjectDelete) >>> return ()
```

5.4 Versioned Objects

Versioned objects are instances of class `RMVersObjectC`, i.e. the version model is parameterised. The following base computations and events are of relevance to version management:

```
class (RMIdObjectC vo) => RMVersObjectC vo where
   revise          :: vo -> IO vo
   merge           :: vo -> vo -> IO vo
   prune           :: vo -> IO ()
   getRevisions    :: vo -> IO [vo]
   getPredecessors :: vo -> IO [vo]
   revised         :: vo -> EV vo
   pruned          :: vo -> EV ()
```

Fig. 7. Versioned Objects

- `revise o` creates a new revision of object `o`.
- `merge o1 o2` merges two revisions `o1` and `o2`.
- `prune o`, removes version `o` from the version graph.
- `getRevisions o` returns the revisions of object `o`.
- `getPredecessors o` returns the parents of `o`.
- `revised o` occurs whenever a new revision has been made to object `o`.
- `pruned o` occurs whenever revision `o` has been deleted.

It is quite straightforward to define the operations in terms of the kernel functionality provided by the RM. For example, the revised operation is defined as:

```
revised = linkAppended o successorOf >>>= getTargetOfLink
    where getTargetOfLink l = ...
```

6 Active Views

So far we have seen fragments of code defining the interfaces to the repository and the approach to event handling. Time has come to glue everything together by defining a view over the repository that can be maintained consistently on the fly in a multiuser setting. The view is constructed by using the services of the *User Interaction Manager*. Interactors are then set up that respond to user interactions and repository change notifications. The principles shall be demonstrated by building and maintaining a *version graph* (see Fig 1).

6.1 Building the Version Graph

The following computation creates a new *daVinci* window that displays the version graph associated with a given Haskell module:

```
buildVersionGraph :: DaVinci -> HaskellObj -> IO Graph
buildVersionGraph dav hobj = do
  g <- graph [title "Development"]
  buildGraph g hobj
  displayGraph g
  mn <- buildHaskellMenu g
  interactor( triggered mn >>> done
      +>    (destroyed g +> destroyed dav) >>> stop)
  return g
```

First the graph is built and displayed. Then the Haskell menu is built. Finally, an interactor is set up that reacts to destruction events as well as menu invocations. The menu provides menu items for creating new revisions of a module (which in turn start's up the text editor), for interpreting entire program configurations (by calling Hugs) and for pruning the version tree.

A version graph is an acyclic directed graph. The view is transitively constructed by retrieving all revisions associated with the versioned Haskell module:

```
buildGraph :: Graph -> HaskellObj -> IO ()
buildGraph g hobj  = do
  revs <- getRevisions hobj
  foreach revs (\rev -> do
    showRevision g hobj rev
    registerEvents g rev
    buildGraph g rev)
```

A revision is visualized in terms of a revision edge pointing from the source to the target object. The new node and edge are added to the version graph by calling the operations `addNode` and `addEdge`, respectively:

```
showRevision :: Graph -> HaskellObj -> HaskellObj -> IO Edge
showRevision g hobj rev = do {
  addNode g rev; addEdge g (RevisionLnk hobj rev)}
```

6.2 Maintaining the Version Graph

Objects and links of relevance to a version graph may come and go during the lifetime of the view. The view is kept consistent with the underlying repository by a bunch of interactors - one for each node of the version graph. Each interactor is set up to listen to three events:

```
registerEvents :: Graph -> HaskellObj -> IO InterActor
registerEvents g hobj = interactor ev
   where ev =
        revised hobj >>>= (\rev -> do
                showRevision g hobj rev
                registerEvents g rev
                redrawGraph g)
    +>  pruned hobj >>> do
                destroyNode g hobj
                redrawGraph g;
                stop
    +>  destroyed g >>> stop
```

When a notification arrives that a new revision has been made, the revision is visualized and a new interactor is created to monitor changes to it. The graph is then redrawn. Similar, when a notification arrives that the revision has been pruned, the *daVinci* node is destroyed and the interactor stops. The interactor will furthermore stop interaction if the graph is deleted, i.e. if the user closes the window displaying the graph.

7 Related Work

The provision of integrated program environments for functional languages have previously been addressed within the Napier project [FDK+92], but in a homogeneous and persistent language setting. The integration of foreign tools has not been taken into account. Nowadays, SDEs are however constructed by integrating off-the-shelf tools in order to reduce development costs [SvdB93]. The architecture of the UniForM WorkBench reflects this trend, by providing generic integration services for data, control and presentation integration.

ToolBus [BK94] is a similar system using adaptors and a network of communicating agents to integrate tools. The integration is expressed in an extremely simple process and term language, which does not compare to Haskell in power and elegance. The idea of associating registration and deregistration commands with tool events can be traced back to the JavaBeans architecture [Mic96]. The WorkBench 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.

The provision of graphical user interfaces has been a vivid track of research over the past years [HC95,NR95,FP95,FGPS96,LWW96,VS96]. The UniForM WorkBench goes one step further by providing a uniform approach to event handling that is independent of the actual source of the event. User interactions, repository change notifications, operating system events and tool events are all represented as first class composable event values.

8 Future Work

An interesting track of future research is to extend the WorkBench with tools for cooperative work and development management. This requires that development rules can be expressed, which in turn suggest that the framework must be extended with "rules", i.e. events triggered by changes to object collections (types) rather than individual instances.

The repository manager could be made more elegant in the presence of orthogonal persistence [Tri90]. Having persistent functions at hand, meta information - such as for example the computation for visualizing Haskell modules - could be stored on the level of the persistent store. A more flexible environment, that could be instantiated on the fly rather than being precompiled, could then be achieved. Multiple parameter type classes and existential types are other features that could, as soon as they appear, lead to an improved framework.

9 Conclusion

The UniForM WorkBench, which provides generic services for data, presentation and control integration in Concurrent Haskell, has proved itself to be

a viable vehicle for developing integrated systems given prefabricated components. A key feature, beyond the usual advantage of using a functional programming language, is the higher order model to event handling, which extends CML with first class, fully composable tool events such as user interactions and database change notifications.

With this technological basis, it is actually possible to take the discipline of implementing functional languages one step further, since not only the compiler, but the entire SDE can be developed using the functional language itself. The paper demonstrates how a single aspect of such a development task, namely that of maintaining consistent views over a distributed multiuser repository, can be very elegantly achieved without loss of abstraction.

Acknowledgement

We would like to thank Walter Norzel for implementing parts of the *UniForM Concurrency Toolkit*, Carla Blanck Purper for implementing some of the underlying features of the *daVinci* encapsulation, and last but not least, Michael Fröhlich and Mattias Werner for providing *daVinci*.

References

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

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

[Cre94] The H-PCTE Crew. H-PCTE vs. PCTE, version 2.6. Technical report, Universität Siegen, 1994.

[ECM94] ECMA. *Portable Common Tool Environment (PCTE) — Abstract Specification.* European Computer Manufacturers Association, 3 edition, December 1994. Standard ECMA-149.

[FDK+92] A. M. Farkas, A. Dearle, G. N. C. Kirby, Q. I. Cutts, R. Morrison, and R. C. H. Connor. Persistent Program Construction through Browsing and User Gesture with some Typing. In *Proc. 5th International Workshop on Persistent Object Systems.* San Miniato, Italy, 1992.

[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, 1996.

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

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

[HC95] M. Hallgren and M. Carlsson. Programming with Fudgets. In *First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science*. Springer, May 1995.

[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 Concurrency Toolkit and its Extensions to Concurrent Haskell. In *GWFP'97: Glasgow Workshop on Functional Programming*, September 1997. Available at `http://www.informatik.uni-bremen.de/~ewk/papers/gw97.ps.gz`.

[Kar97c] E. W. Karlsen. The UniForM User Interaction Manager. Technical report, FB 3, Universität Bremen, 1997.

[KPO+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 Forshungsanstalt 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.

[LWW96] C. Lüth, S. Westmeier, and B. Wolff. sml_tk: Functional Programming for Graphical User Interfaces. Technical Report 8/96, FB 3, Universität Bremen, 1996.

[Mic96] Sun Microsystems. *JavaBeans 1.0*. JavaSoft, December 1996.

[NR95] R. Noble and C. Runciman. Gadgets: Lazy Functional Components for Graphical User Interfaces. In *PLILP'95: Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, volume 982 of *LNCS*, pages 321–340. Springer, September 1995.

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

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

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

[Rep92] J. H. Reppy. *Higher-Order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, 1992.

[SvdB93] D. Schefström and G. van den Broek. *Tool Integration*. Wiley, 1993.

[Tri90] P. Trinder. *A Functional Database*. PhD thesis, Department of Computer Science, University of Glasgow, 1990.

[VS96] T. Vullings and W. Schulte. The Design of a Functional GUI Library using Constructor Classes. In *Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*. Springer, 1996.