List-Graphs
Theory and Implementation

Greg O'Keefe

May 12, 2009

This literate Haskell development introduces and implements a somewhat novel form of Graph. List-Graphs are motivated in the author's PhD thesis [O'K08] as a mathematical foundation for the Unified Modelling Language (UML) [Obj09]. The thesis argues that UML models and system states should be defined as list-graphs, and the dynamic semantics of the language given by a graph transformation system [BH02] over these graphs.

The thesis only discusses the proposed graphs in very general terms. The present work aims to make the proposal for UML semantics more concrete, and demonstrate it with examples. One very specific goal is motivated by the UML Reference Animator project being conducted by a team of software engineering students at ANU. This team have produced a small example model with a sequence diagram. The code presented here will enable us to produce a concrete execution trace of this model which satisfies the sequence diagram. The code will hopefully also serve as a prototype for the core of the interactive model validation tool they are producing.

This is a work in progress. Parts of it are useable, but there is much remaining to be done.

The first section describes the 2 graph input formats and their parsers. Section 2 presents the graph representation, manipulation and display code. The third section builds on the basic code of Section 2 to simplify the construction of system traces, which are sequences of graphs, each being an update of its predecessor. It also enables visual, frame-by-frame display of these traces, with the changes highlighted for easy comprehension. The example which motivates this work is presented in Section **??**. There, we briefly describe the example trace of a UML model of a microwave oven. The trace is intended to satisfy a sequence diagram presented as part of the example model. Section 5 outlines plans for further development work, and Section 6 describes some theoretical questions about list-graphs which we hope to answer. There is currently no conclusion, because the work is far from concluded.

Contents

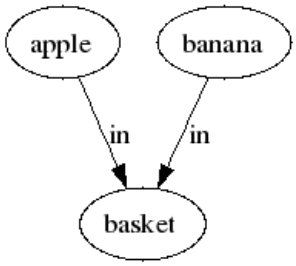

Figure 1: Example graph as displayed by `dot`

1 Input

This section documents the parsers, which are generated by the Haskell parser generator Happy (`http://haskell.org/happy`). These are used to read in graphs from files, in a couple of useful textual formats.

1.1 Dot Format

It is convenient for us to be able to use a simplified format compatible with the graph visualisation program dot (`http://www.graphviz.org`). The parser described in this subsection is generated from the Happy input grammar `DotParser.y`, which defines a useful subest of dot's texual graph language. For further details of dot and its input language, see the website. Here is an example graph expressed using this format.

```
digraph appleInBasket {
        apple -> basket [label=in]
        banana -> basket [label=in]
}
```

This example is in a file called `appleInBasket.dot`. Figure 1 shows this graph as displayed by dot. The diagram is a `.png` file, produced by the following command line.

```
dot -Tpng -o appleInBasket.png appleInBasket.dot
```

Other formats such as `.ps` and `.svg` are available, and there are options which can increase the resolution to remove the aliasing visible here. For more details, see the graphviz website, or the `dot` man page.

The dot format begins with a keyword describing what kind of graph follows. In this case, the keyword is `digraph`, which says that the following graph has directed edges. Although dot can work with undirected graphs, we only consider directed ones. The next word gives a name to the graph, which we ignore. Following this is a list of edges, enclosed in braces { and }. Each edge has the format `sourceNode -> targetNode`, followed by a list of options enclosed in square braces [and]. The only such option we recognise is `label=someLabel`. If it is absent, the edge has the empty string as its label.

The module `DotParser` provides a pair of functions for converting strings in this format into one of the datatypes we use for representing graphs. These types are described in the following Section.

Function `tokenize :: String -> [String]` breaks the input string up into the "words" of the dot language. Function `dotParser [String -> [(String, String, String)]` converts this list of words into a list of graph edges.

In order to read the contents of the file into a string, we must use Haskell's `do` notation, to get into the "IO monad". Readers not familiar with this stuff are urged not to freak out.

Enter the Haskell interpreter `ghci` and load `graph.lhs`. (From the shell command line in the directory with this code, enter `ghci graph.lhs`.) The following command will read and parse the file.

```
do
    dotString <- readFile "appleInBasket.dot";
    return ((dotParser . tokenize) dotString)
```

Which returns a display of the resulting list-of-triples value.

```
[("banana","in","basket"),("apple","in","basket")]
```

1.2 Plain Triples Format

Support for the dot format allows us to work with graphs that can be visualised using dot. However, it is not a very convenient form for data entry. That is the purpose of the parser described briefly in this subsection.

Here is an example graph in the plain triples format, which is available in the distribution as `test.triple`

```
{ a cute example graph }
apple in basket
banana in basket
cow over moon
```

The format is : sourceNode label targetNote, repeat until finished. Comments are allowed, surrounded by , {* and *}, but they are only allowed before or after a triple. That is, no comments in the middle of a triple. Do not put * in your comments, the parser will choke. Separating the triples by newlines is recommended for readability. The following Haskell expression reads the file in, parses it, reformats it as dot input and invokes dot to display it. The showEdgeList function is described, along with other similar functions, in the following section.

```
do
    s <- readFile "test.triple";
    showEdgeList ((tripleParser . tokenize) s)
```

The resulting graph is shown in Figure 2.

1.3 Graph Updates

In order to generate a system trace, we must apply a sequence of collections of updates to an initial state graph. Each collection of updates consists of edges to add and edges to delete. To make this job a bit easier than coding the updates directly in Haskell, we have provided a parser.

Here is an example, taken from the microwave example.

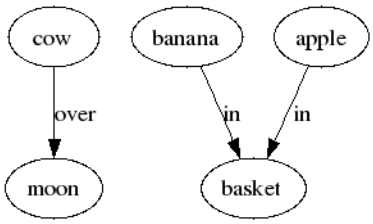

Figure 2: Example graph as displayed by `dot`

```
{* Microwave accepts the cook signal receipt event  *}
add {
  m1classifierBehaviorExecution activeState Cooking
  m1CookingDo1 i BehaviorExecution
  m1CookingDo1 host m1
  m1 execution m1CookingDo1
  m1CookingDo1 behavior CookingDo
  {* I think explicit enabled edges are not necessary *}
}

delete {
  m1 pool c1receipt
  m1classifierBehaviorExecution activeState NotCooking
}

{* Microwave executes the do action of its Cooking state
 (In order to make sure that each action is only executed once, I
think behavior executions will need "execute" edges to their actions
when created. These edges will be removed when the action is executed)
*}

{* read self action *}
add {
  selfPin value m1
}
delete { }
```

The format builds on the triples format. In fact, the two languages are different entry points into the same grammar in `TripleParser.y`. Lists of triples are enclosed in braces, with a heading which indicates whether they are edges to be added or deleted. The file must contain a group of adds then a group of deletes and so on in strict alternation. Notice the final, empty collection of deletes here. Leaving it out would cause a parse error.

Comments can be inserted in the lists of triples wherever legal for that format, and also before a group heading and at the end of the file.

2 Graph Representations, Manipulation and Output

This section presents the module `Graph`. Three different graph types are defined, each useful for some particular purpose. Graph "navigation" is implemented, that is, evaluation of expressions of the form $node.label$. Graph manipulation is provided in the form of edge add and delete functions. A pretty printing function is provided, which outputs graphs in the dot format described in Section 1.1. This enables us to view graphs "graphically" by running them through dot and invoking a viewing program.

The code below imports the required modules. In Section 1, we described two Happy generated parsers and their lexer. These are included to enable graphs to be read from text files. Display of graphs is also provided by pretty-printing graphs to dot's input language, and calling dot using the `system` function, provided by the `System` module. Module `List` provides the function `partition`, which we use to extract association list entries for a given key.

```
module Graph where

import Tokenizer
import DotParser
import TripleParser

import System
import List
```

We introduce a type for list-valued association lists and use it to define general graphs, whose nodes and labels can be any type. However, everything we do from here on uses graphs whose nodes and labels are strings, so we reserve "`Graph`" for this type. When reading and writing graphs, it is convenient to convert them to lists of edges, where an edge is a triple (sourceNode, label, targetNode). General and string-specific versions of this type are defined.

```
type Lasl k d = [(k, [d])]

type GGraph n l = [(n, Lasl l n)]
type GEdge n l = (n,l,n)

type Graph = GGraph String String
type Edge = GEdge String String
```

2.1 Navigation

UML has an instruction-set, which it calls its "Actions". Among the actions are `ReadLinkAction` [Obj09, §11.3.33] and `ReadStructuralFeatureAction` [Obj09, §11.3.37]. These actions read values from an attribute or association end, and are often refered to as "navigation". Action languages sometimes express this using a dot notation, as in `self.subordinate` to read the collection of objects related to the current one across an association called "subordinate".

Defining graphs as nested association lists, as we have done, enables a very simple implementation of navigation. Navigation uses a version of the Haskell prelude function `lookup`, which we have written specifically for list-valued association lists. Rather than return `Nothing` when nothing is found, we return the empty list.

```
llookup :: (Eq a) ⇒ a → Lasl a b → [b]
llookup _ [] = []
llookup key ((k,l):rest)
    | k == key  = l
    | otherwise = llookup key rest

navigate :: (Eq n, Eq l) ⇒ (GGraph n l) → n → l → [n]
navigate graph node label = llookup label (llookup node graph)
```

2.2 Display

The following functions allow the user to view graphs. Evaluating `showGraph g` writes the graph `g` in `.dot` format, has dot render it as an `.svg` and calls `eog` (Eye of Gnome) with the result. Of course, the graphics format and viewer can be changed to suit the local installation.

The function `edgeList` supports this by converting the graph into a list of (sourceNode, label, targetNode) triples, while `dotEdge` converts such a triple into an edge statement for dot.

Since the parsers return graphs in edge-list form, the function `showEdgeList` is also provided. To make it easy to display graph files in plain-triple format, the briefly named function `self` (show edge list file) is also provided. We hope that this does not get confused with the "self" term of the action languages, used to refer to the object executing the behavior it occurs in.

UML system states in the form of these graphs contain a great number of edges showing that one node (object) is an instance of another. These edges are labelled "i" by convention in [O'K08], and `dotEdge` displays them in grey to make the resulting graph more readable.

```
edgeList :: GGraph n l → [GEdge n l]
edgeList graph = [ (s,l,t) | (s, lts) ← graph, (l,ts) ← lts, t ← ts]

dotEdge :: (Show n, Show l) ⇒ GEdge n l → String
dotEdge (source, label, target) =
    concat [show source, " → ", show target,
            " [label = ", show label, igrey, "] \n"]
        where igrey = if show label `elem` ["i", "\"i\""]
                      then ", color=grey"
                      else ""

dotEdgeList :: (Show n, Show l) ⇒ [GEdge n l] → String
dotEdgeList el =
    "digraph anonymous {\n" ++ foldr (++) "}\n" (map dotEdge el)

dotGraph :: (Show n, Show l) ⇒ GGraph n l → String
dotGraph graph = dotEdgeList (edgeList graph)


--  I would like to generalise these two, but don't know how

showEdgeList :: (Show n, Show l) ⇒ [GEdge n l] → IO ExitCode
showEdgeList el =
```

```
    do
      writeFile "tmp.dot" (dotEdgeList el)
      system("dot -Tsvg -o tmp.svg tmp.dot")
      system("eog tmp.svg&")

showGraph :: (Show n, Show l) ⇒ GGraph n l → IO ExitCode
showGraph graph =
    do
      writeFile "tmp.dot" (dotGraph graph)
      system("dot -Tsvg -o tmp.svg tmp.dot")
      system("eog tmp.svg&")

self path =
    do
      fileContents ← readFile path;
      showEdgeList ((tripleParser ∘ tokenize) fileContents)
```

2.3 Add and Delete Edges

All graph modifications are acheived by adding and deleting edges. Nodes are effectively a byproduct of their edges, and there are no explicit operations to add and delete them. To add a node, add some edge to or from it. To delete a node, delete all edges to and from it. This removes the troublesome question of what to do with "dangling" edges, they simply cannot arise. It also makes it impossible to have an isolated node. This could be considered a deficiency, be we think not. We have defined list-graphs essentially as navigation values. Since no isolated edge can ever occur in the value of a navigation, there is no reason for them to exist.

The following code is for adding edges. First a couple of helper functions. The function `insertAt` adds an item to a list at a given position. The function `llookupAndRest` looks up a key in a list-valued association list, returning the result as the first element of a pair. The second element of the pair is the association list elements for other keys. This makes it easier to manipulate the entry for a given key, then put the result back into the remainder of the list.

The add edge function enables us to write a simple conversion from edge-list form to graph form. It is called `graph`.

```
insertAt :: Int → a → [a] → [a]
insertAt pos item list =
    front ++ item:back
        where (front,back) = splitAt (pos - 1) list

llookupAndRest :: Eq a ⇒ a → Lasl a b → ([b], Lasl a b)
llookupAndRest key asl = (llookup key asl,rest)
    where (_, rest) = partition (\(k,_) → k==key) asl

addEdgeAt :: (Eq n, Eq l) ⇒ Int → GEdge n l →  GGraph n l → GGraph n l
addEdgeAt pos (source,label,target) graph =
    let
        (outgoing, otherNodes) = llookupAndRest source graph
        (targets, otherLabels) = llookupAndRest label outgoing
        labelLists = (label, insertAt pos target targets):otherLabels
```

```
    in  (source, labelLists):otherNodes

addEdge :: (Eq n, Eq l) ⇒ GEdge n l → GGraph n l → GGraph n l
addEdge edge graph = addEdgeAt 1 edge graph

graph :: (Eq n, Eq l) ⇒ [GEdge n l] → GGraph n l
graph el = (foldr (.) id (map addEdge el)) []
```

Now we handle edge deletion in a similar manner.

```
deleteAt :: Int → [a] → [a]
deleteAt pos list =
    let (front, victim:back) = splitAt (pos - 1) list in
    front ++ back

deleteEdgeFun :: (Eq n, Eq l) ⇒
                 ([n] → [n]) → n → l → GGraph n l → GGraph n l
deleteEdgeFun delfun source label graph =
    let (labList, otherNodes)    = llookupAndRest source graph
        (targetList,otherLabels) = llookupAndRest label labList
        labelLists = (label, delfun targetList):otherLabels
    in  (source, labelLists):otherNodes

deleteEdgeAt :: (Eq n, Eq l) ⇒ Int → n → l → GGraph n l → GGraph n l
deleteEdgeAt pos source label graph =
    deleteEdgeFun (deleteAt pos) source label graph

deleteEdge :: (Eq n, Eq l) ⇒ GEdge n l → GGraph n l → GGraph n l
deleteEdge (source, label, target) graph =
    deleteEdgeFun (delete target) source label graph
```

2.4 Ideas for Further Work

The edge display function `dotEdge` currently displays "i" edges grey. This hard-coded behavior could be generalised as follows. A list of (predicate, formatter) pairs could be accepted, so that the first predicate satisfied by a given edge gets the format associated with that predicate. This might be a good way to display similar kinds of edges similarly.

3 System Traces

The previous section developed code for representing, manipulating and displaying list-graphs. In this section, we build on this to enable construction and display of sequences of list-graphs. Since the purpose of these sequences is to exhibit the behaviour of a modelled system, we call them system traces.

Each system state transition is given as a pair of lists of edges. The first list is the edges to add to the input graph, the second list is the edges to delete.

Of course, for each edge $(source, label, target)$ to be added or deleted, we should also specify the position it should be added at or deleted from. It could be added anywhere in the list of $label$ edges leaving $source$, and there may be several such edges that are candidates for deletion. For now, we add edges at the first position, and

remove the first matching edge. This is sufficient for UML models whose Properties are unordered, the default value. To handle the full generality of list-graph transitions, the module will need a minor rewrite, adding positions to the edge specifications and using the positional versions of the edge add and delete functions.

In order to easily produce and display a sequence of graphs from a sequence of updates, we have developed the fuction `trace`. It takes a graph and a list of updates, where an update is a pair of lists of edges, the adds and the deletes. It writes the resulting sequence of graphs as `state0.dot` ... `state`n`.dot`.

```
module Trace where

import Graph
import System
import List   -- for nub, which removes duplicates

type GUpdateList n l = [([GEdge n l],[GEdge n l])]

traceHelper :: (Show n, Show l, Eq n, Eq l) ⇒
    (GGraph n l, Int) → GUpdateList n l → IO (GGraph n l, Int)

traceHelper (graph, i) ((adds,deletes):restUpdates) =
    do
        writeFile ("state" ++ show i ++ ".dot") (dotGraph graph)
        traceHelper (graph'', i+1) restUpdates
        where
           graph'  = (foldr (.) id (map addEdge adds)) graph
           graph'' = (foldr (.) id (map deleteEdge deletes)) graph'

traceHelper (graph, i) [] =
    do
        writeFile ("state" ++ show i ++ ".dot") (dotGraph graph)
        return (graph, i+1)

trace :: (Show n, Show l, Eq n, Eq l) ⇒ GGraph n l → GUpdateList n l → IO ()
trace graph updates =
    do
        traceHelper (graph, 0) updates
        return ()
```

3.1 Display

We wish to display traces as sequences of graphs. These graphs will be quite large, for example, the graph for the initial system state for the very simple example UML model has almost 200 edges. We implement a couple of techniques to help make visual sense of these sequences of large graphs.

First, to ensure that nodes and edges do not move around from one state to the next, we create a union graph that has every node and edge from the whole sequence. Instead of a new graph for each state, we recycle this graph, marking the currently existing edges.

The second aid to sequence comprehension also employs this marking idea. Newly created nodes and edges are displayed in green, those that are about to be deleted in red. To enable this, we add a couple of extra markings.

These existence status "markings" are implemented as a datatype `Existence`. "Existential" edges are a pair with an existential-status and a familiar $(source, label, target)$ triple.

Existential status values correspond to display attributes for dot. This correspondence is implemented as the function `existentialAttribs`. The pretty printing functions we saw in Section 2, for "unmarked" triples, are rewriten in a mostly obvious way.

One difference is that we explicitly mention the nodes, so we can give them display attributes. Since the nodes are not explicitly represented, it takes a bit of work to extract them and determine their existential status. The function `existentialNodes` gives each node the "greatest" existential status of its (incoming and outgoing) edges.

```
data Existence =  Void | Destroying | Creating | Persisting
    deriving (Show, Eq, Ord)

type ExistentialGEdge n l = (Existence, GEdge n l)

existentialAttribs :: Existence → String
existentialAttribs status =
    case status of
      Creating → " color = green, fontcolor = green "
      Persisting → ""
      Destroying → " color = red, fontcolor = red "
      Void → " style = invis "

dotExistentialEdge :: (Show n, Show l) ⇒ ExistentialGEdge n l → String
dotExistentialEdge (status,(source, label, target)) =
    unwords [show source, "→", show target,
            "[label =", show label, igrey, attribs, "]"]
        where igrey = if show label `elem` ["i", "\"i\""] && status == Persisting
                      then ", color=grey"
                      else ""
              attribs = existentialAttribs status

existentialNodes :: (Eq n, Eq l) ⇒
                    [ExistentialGEdge n l] → [(Existence,n)]
existentialNodes eedges =
    let source (e,(s,_,_)) = (e,s)
        target (e,(_,_,t)) = (e,t)
        allENodes = (map source eedges) ++ (map target eedges)
        nakedNodes = nub [n | (_,n) ← allENodes ]
        status node = foldr max Void [e | (e,n) ← allENodes, n == node ]
    in [(status n,n) | n ← nakedNodes]

dotExistentialNode :: Show n ⇒ (Existence, n) → String
dotExistentialNode (status, node) =
    unwords [show node, "[", existentialAttribs status, "]"]

dotExistentialEdgeList :: (Eq n, Eq l, Show n, Show l) ⇒
                          [ExistentialGEdge n l] → String
dotExistentialEdgeList el =
    unlines ("digraph anonymous {":(dotNodes ++ dotEdges)) ++ "}"
        where
```

```
        dotNodes = map dotExistentialNode (existentialNodes el)
        dotEdges = map dotExistentialEdge el
```

We now have the ability to display graphs represented as lists of existentially attributed edges. Next we provide the means to create a trace in this format.

```
etraceHelper :: (Show n, Show l, Eq n, Eq l) ⇒ [[ExistentialGEdge n l]] →
                [ExistentialGEdge n l] → GUpdateList n l → [[ExistentialGEdge n l]]
etraceHelper trace state ((adds, deletes) : updates) =
    etraceHelper (deleting:adding:state:trace) state' updates
    where
      adding   = paint Void Creating adds state
      added    = paint Creating Persisting adds adding
      deleting = paint Persisting Destroying deletes added
      state'   = paint Destroying Void deletes deleting

etraceHelper trace state [] = state:trace


paint :: (Show n, Show l, Eq n, Eq l) ⇒
    Existence → Existence → [GEdge n l] →
    [ExistentialGEdge n l] → [ExistentialGEdge n l]

paint fromStatus toStatus (edge:edges) eedges =
    if
        any (==(fromStatus,edge)) eedges
    then
        let (front, (_,e):back) = break (==(fromStatus,edge)) eedges
        in paint fromStatus toStatus edges (front ++ (toStatus,e):back)
    else
        error (unwords ["paint:", show fromStatus, show edge, "does not exist"])


paint _ _ [] edges = edges


etrace :: (Show n, Show l, Eq n, Eq l) ⇒
                [GEdge n l] → GUpdateList n l → [[ExistentialGEdge n l]]

etrace state updates =
    reverse (etraceHelper [] (vadded ++ pstate) updates)
    where
      vadded = map (\edge → (Void,edge)) adds
      adds = concat (map fst updates)
      pstate = map (\edge → (Persisting,edge)) state
```

Finally, a utility to produce a sequence of .dot and .svg files from an existential trace.

```
visualise :: (Eq n, Eq l, Show n, Show l) ⇒
             [[ExistentialGEdge n l]] → IO ()
visualise trace = visualiseHelper 0 trace

visualiseHelper :: (Eq n, Eq l, Show n, Show l) ⇒
                   Int → [[ExistentialGEdge n l]] → IO ()
```

```
visualiseHelper i (state:states) =
    do
      writeFile ("state" ++ show i ++ ".dot") (dotExistentialEdgeList state)
      system ("dot -Tsvg -o state" ++ show i ++ ".svg state" ++ show i ++ ".dot")
      visualiseHelper (i+1) states

visualiseHelper _ [] = return ()
```

3.2 Future Work

Develop and demonstrate proper list-graph (not bag-graph) traces.

Separate production of trace from writing it to disk.

4 Examples

This section presents applications of the graph tools developed in Section 2 and the trace tools developed in Section 3.

The second subsection develops a trace of the UML model, whose initial state graph has almost 200 edges. Before diving into this level of complexity, we present a much simpler example in the first subsection.

4.1 Very Simple Example

This small example models a very optimistic story of the authors aquisition of European languages. First, we define an initial state, then two updates. These are used to construct a trace.

```
module UMLmodelTraceEG where

import TripleParser
import Tokenizer
import Graph
import Trace

initialLang :: Graph
initialLang = [("Greg",[("learning",["French"])])]

edgesToAdd1, edgesToDelete1, edgesToAdd2, edgesToDelete2 :: [Edge]
edgesToAdd1 = [("Greg","learning","Italian"),("Greg", "speaks", "French")]
edgesToDelete1 = [("Greg","learning","French")]
edgesToAdd2 = [("Greg","speaks", "Italian"), ("Greg","learning","Spanish")]
edgesToDelete2 = [("Greg", "learning","Italian")]
langUpdates = [(edgesToAdd1, edgesToDelete1), (edgesToAdd2, edgesToDelete2)]

langTrace = trace initialLang langUpdates
```

The three states of this trace can be seen side by side in Figure 4.

Now, we create a coloured, positionally stable visual animation of this trace.

```
egEtrace = etrace (edgeList initialLang) langUpdates
```

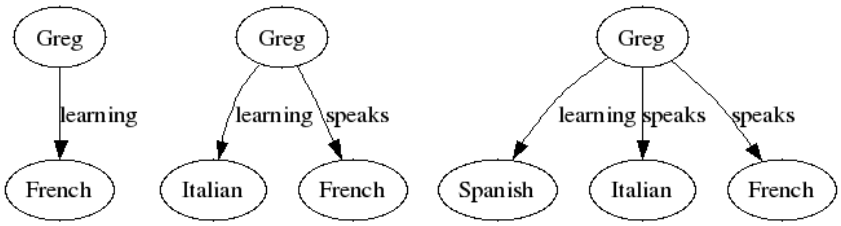

Figure 3: Three successive graph-states

To create a visualisation of this trace, load `UMLmodelTraceEG.lhs` in `ghci`, and evaluate `visualise egEtrace`. This will create files `state0.svg...state6.svg`. Eye of Gnome can be invoked to view all these files with the command `eog state*.svg`, and then the "next" button can be used to step through them.

4.2 Simple UML Example

The motivation for list-graphs is formal semantics for UML. This section presents an execution trace for the example model shown in Figure 3. The trace is intended to satisfy sequence diagram shown there.

```
vt triplesFile updatesFile =
  do
    tfileContents ← readFile triplesFile
    ufileContents ← readFile updatesFile
    let
      iStateTriples = (tripleParser ∘ tokenize) tfileContents
      updates = reverse ((updateParser ∘ tokenize) ufileContents)
      in
      visualise (etrace iStateTriples updates)
```

The initial state of our example is given an a plain-triple format file, `microwave.triple`, and the updates in `microwave.updates`.

Now the trace visualised by evaluating `vt "microwave.triple" "microwave.updates"` in ghci, then viewing the results with `eog state*.svg` in the shell.

5 Modules to be Developed

Here are some thoughts on further modules to be developed. Do not take too seriously, rethink before beginning to code.

Morph graph homomorphism type and predicate, search

Xfrm graph transformation rules and application

UMLmm (useable fragment of) the UML metamodel (a .dot or other graph file)

UMLrt UML run-time structures

UMLrules graph transformation rules for (some of the) actions, and signal handling

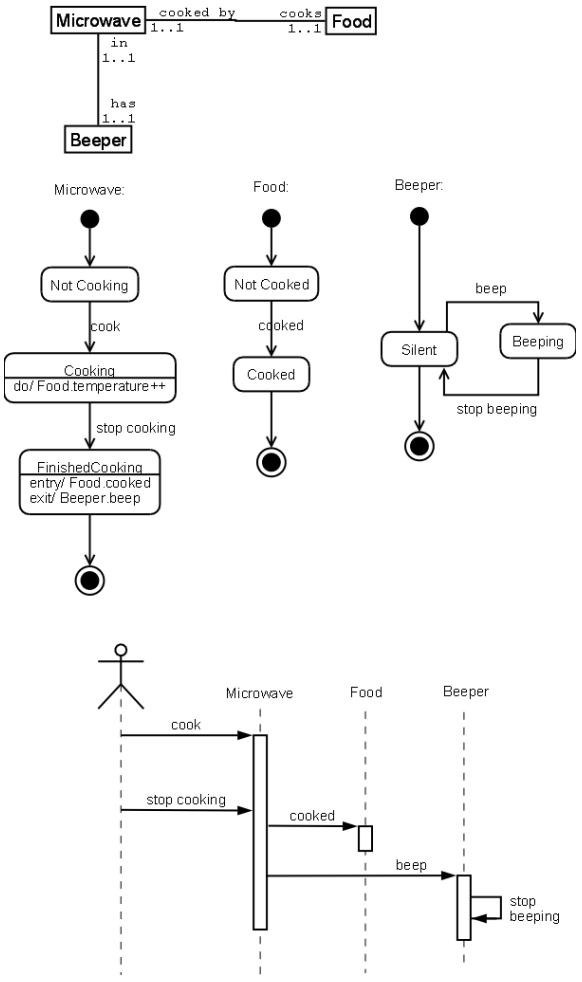

Figure 4: Example UML model

MicrowaveIS microwave UML model, and initial state for execution trace

MicrowaveTrace rule applications to produce execution trace

DisplayUtils filters and usages of dot features to adjust display, visual animation of execution trace using dot (by having everything always there, just hidden sometimes)

6 Open Theoretical Questions

List-graphs have not been studied so far as I know. Here are some theoretical questions about them, that seem important to us.

What is a list-graph homomorphism? It probably involves list homomorphisms. The navigation dot operation should be preserved. What else should be preserved?

What is the difference between (my) list-homomorphisms and (Bird) free-monoid homomorphisms? Do mine preserve concatenation? Can we construct one of mine from an fmh? Are they therefore equivalent?

Can list-graphs be expressed as functors From some category into Set, or perhaps into Set^*?!?

Is there a "forgetful" functor which takes categories to list-graphs (or vice-versa!) ?

Congruent Collection Categories? UML uses 4 kinds of collection, lists, multi-sets, ordered sets and sets. Concatenation is the natural way of combining lists, and union is the natural way of combining sets. If you take the concatenation of two lists, then apply the forgetful functor to obtain a set, that set is the union of the two sets obtained from the two lists using the forgetful functor. Do the other two collection types form categories? What are the morphisms? Is there a natural combination operation for each? Do they commute with the relevant forgetful functors?

Edge Lists For practical reasons, we often represent list-graphs as lists of edges. Obviously several different lists can represent the same graph. To what extent can the operations on list-graphs be mirrored on edge-lists? That is, Can we create an algebra of edge-lists which maps homomorphically into the list-graph algebra?

References

[BH02] Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *Proceedings of the first International Workshop on Theory and Application of Graph Transformation*, pages 402–429, 2002.

[Obj07] Object Management Group. Unified modeling language: Superstructure, version 2.1.2. Technical report, Object Management Group, 2007. `http://www.omg.org/docs/formal/07-11-01.pdf`.

[Obj09] Object Management Group. Unified modeling language: Superstructure, version 2.2. Technical report, Object Management Group, 2009. http://www.omg.org/docs/formal/09-02-03.pdf.

[O'K08] Greg O'Keefe. *The meaning of UML models*. PhD thesis, The Australian National University, 2008.