Clusterware Light: Attempt at QuickCheck Specification

October 23, 2008

1 Introduction

A cluster controller is responsible for managing a collection of applications running on a collection of processors. It responds to events such as changes in a processor's status, by starting, stopping, and moving applications around, according to the constraints in a static *configuration* of the controller. This document is an attempt to specify the behaviour of such a cluster controller, "ClusterWare Light".

The document as a whole is a literal Erlang module, using the QuickCheck API.

```
-module(clware_eqc).
-include_lib("eqc/include/eqc.hrl").
-compile(export_all).
```

2 Notations

This section introduces some useful notations for use later.

We can generate non-empty lists as follows,

```
nonempty(G) ->
    ?SUCHTHAT(L,G,L/=[]).
```

and sorted lists by sorting.

```
sorted(G) ->
    ?LET(L,G,lists:sort(L)).
```

We compare sets and bags by sorting.

```
set_eq(S1,S2) ->
    lists:usort(S1) == lists:usort(S2).

bag_eq(S1,S2) ->
    lists:sort(S1) == lists:sort(S2).
```

Check that Xs contains Ys (Ys is a subset of Xs)

```
contains(Xs,Ys) ->
    Ys--Xs == [].
```

We represent relations as lists of pairs; we may need their domain, range and inverse:

```
dom(R) ->
    remove_duplicates([X || {X,_} <- R]).

ran(R) ->
    dom(inverse(R)).

inverse(R) ->
    [{Y,X} || {X,Y} <- R].
```

The *image* of a set through a relation is the set of values related to elements of the set:

```
image(Set,R) ->
    [Y || X <- Set,
          {X1,Y} <- R,
          X == X1].
```

We may convert relations to and from "functions" that associate elements in the domain with lists of elements in the range.

```
group([]) ->
    [];
group([{X,Y}|R]) ->
    [{X,[Y|[Y1 || {X1,Y1} <- R,
                  X==X1]]}
     |group([{X1,Y1} || {X1,Y1} <- R,
                        X/=X1])].

ungroup([]) ->
    [];
ungroup([{X,Ys}|F]) ->
    [{X,Y} || Y <- Ys] ++ ungroup(F).
```

We define a generator for mappings:

```
mapping(DomG,RanG) ->
    ?LET(Dom,ulist(DomG),
         [{X,RanG} || X <- Dom]).
```

Now we can state inverse relationships between `group` and `ungroup`.

```
prop_group_ungroup() ->
    ?FORALL(F,mapping(int(),nonempty(ulist(int()))),
            group(ungroup(F)) == F).

prop_ungroup_group() ->
    ?FORALL(R,list({int(),int()}),
            bag_eq(ungroup(group(R)),R)).
```

We sometimes need to specify that the elements of a list are *different*:

```
different(L) ->
    lists:usort(L) == lists:sort(L).
```

We can create lists of different elements by removing duplicates from lists with duplicates.

```
remove_duplicates(L) ->
    lists:foldr(fun(H,T)->[H|lists:delete(H,T)] end,[],L).

prop_different_remove_duplicates() ->
    ?FORALL(L,list(int()),
            different(remove_duplicates(L))).

prop_remove_duplicates_different() ->
    ?FORALL(L,list(int()),
            ?IMPLIES(different(L),remove_duplicates(L)==L)).
```

We generate lists of different elements using remove_duplicates:

```
ulist(G) ->
    ?LET(L,list(G),remove_duplicates(L)).
```

3 Basic Concepts

3.1 Applications

The purpose of a cluster controller is to control the execution of applications. An application is a top-level OTP application, represented in this document by an atom. For example, the following are applications:

```
-define(applications,[sip,diameter,megaco,snmp,inets,perf,b2bua,b2bua_sb]).
```

Suitable test data can be generated as follows:

```
application() ->
    elements(?applications).
```

3.2 Rôles

A processor playing a certain rôle will usually run a set of related applications. During the lifetime of a product, the rôles of processors are actually more stable than the particular set of applications that implement that rôle. Therefore, we make the notion of a rôle explicit, and assign rôles to processors, rather than applications.

Rôles are named by atoms, for example:

```
-define(role_names,[basic,oam,active,standby]).

role_name() ->
    elements(?role_names).
```

They correspond to a set of applications. The specification of the rôles in use is a part of the configuration of the cluster controller, represented as a mapping from role names to lists of applications. Each application may appear in at most one rôle.

```
valid_roles(Roles) ->
    different(dom(Roles)) andalso different(ran(ungroup(Roles))).
```

We can generate valid rôle assignments by inverting a mapping from applications to role names.

```
roles() ->
    ?LET(AppRoles,mapping(application(),role_name()),
         group(inverse(AppRoles))).

prop_valid_roles() ->
    ?FORALL(Roles,roles(),valid_roles(Roles)).
```

4 The Meaning of Layout

The layout field of a group specifies how roles should be assigned to slots, based on which of the slots are currently filled. We assume

- `Slots` is a list of the group slots:

  ```
  valid_slots(Slots) ->
      different(Slots).
  ```

- `SlotMembers` is a mapping from slots to processors

  ```
  valid_slot_members(Slots,SlotMembers) ->
      contains(Slots,dom(SlotMembers)) andalso different(ran(SlotMembers)).
  ```

  filling those slots (a list of slot-processor pairs)

- `Layout` is a layout specification as per Ulf's slides.

Then the *meaning* of the layout is a relation between rôles and members, defined as follows:

```
layout_meaning(Slots,SlotMembers,Layout) ->
    [{Role,Proc}
     || Elem <- Layout,
        {Role,Proc} <- layout_element_meaning(Slots,SlotMembers,Elem)].
```

That is, the meaning of a layout is a combination of the meanings of its elements. `on_each` just assigns the rôle to each active slot:

```
layout_element_meaning(_Slots,SlotMembers,{on_each,Roles}) ->
    [{Role,Proc} || Role <- Roles,
                    Proc <- ran(SlotMembers)];
```

Rôles can be explicitly assigned to slots—provided the slot is filled.

```
layout_element_meaning(_Slots,SlotMembers,{Slot,Roles})
  when is_atom(Slot)->
    [{Role,Proc} || Proc <- image([Slot],SlotMembers),
                    Role <- Roles];
```

The `if_active` property is interesting: note that the value of this property, according to Ulf's slides, is just a special case of a layout! We can therefore generalise the property to allow an arbitrary layout specification—and at the same time simplify the meaning.

```
layout_element_meaning(Slots,SlotMembers,{{if_active,N},Layout}) ->
    case length(SlotMembers) of
        N ->
            layout_meaning(Slots,SlotMembers,Layout);
        _ ->
            []
    end.
```

When is a layout *valid*? It seems to me we should require that no rôle is assigned to the same slot more than once—in other words

```
valid_layout_meaning(Slots,SlotMembers,Layout) ->
    different(layout_meaning(Slots,SlotMembers,Layout)).
```

However, I wonder whether this is entirely satisfactory? Note that this meaning function *completely specifies* the rôles that should be assigned to each slot. This means there is little scope for the cluster controller to *vary* the assignment depending on load, for example. Essentially the only way the cluster controller is allowed to influence the rôle assignment is by choosing which slots to fill with members (assuming the group is not packed). Perhaps the controller should be allowed more freedom?

For example, I wonder whether instead of specifying for each slot, a list of rôles to be run there, might it make sense to specify *for each rôle, a list of slots where it might run?*. Such a list could be interpreted in priority order (run on the first slot that is filled), or in non-deterministic order (run on one of these slots, choice left to the cluster controller), or in another specified order (run on the least heavily loaded slot, for example). This kind of specification might be useful for any group, with any number of filled slots, as opposed to the `if_active` properties which are really only sensible for mated pairs.