Change Report for Generic Dream

Kim Berninger, Michael Raulf

In order to expand the Dream framework by a generic overlay there also had some changes to be applied to the underlying infrastructure. Those as well as the final top level changes are to be explained in the following chapters.

1 Events, Values, Attributes and Constraints

When the value of a proxy in the Dream framework is updated the subscribers of that proxy are notified by being passed an event containing the attributes which describe the changes that took place. Also subscriptions are filtered using constraints which just als the attributes depend on the type of the value being observed in that communication.

Prior to our changes those value types were wrapped in a *Value* class in which a fixed set of different types has been hardcoded. We made this class obsolete by introducing a generic type parameter each to the *Attribute* and *Constraint* classes. However we maintained the *Value* class and made *Attribute* and *Constraint* backwards compatible to it since those are both used by the old infrastructure on which our overlay still holds dependencies.

It would be possible to eliminate the old classes with fixed types, i.e. *IntegerObservable*, *DoubleObservable* etc., altogether and with that also completely rewrite the *Constraint* class in order to use extensible predicate operations, what, however, would deprecate most of the code using the old version of Dream.

2 Var, RemoteVar and Signal

Our new overlay classes build upon the old *Proxy*, *AbstractReactive* and *Observable* classes and introduce one generic type parameter each.

2.1 RemoteVar

First the *RemoteVar* class serves as a general proxy for arbitrary types as well as a new foundation for the old fixed proxy classes in order to reduce redundancy. Just as the proxies the remote variables can be used to access a variable on a different component by its name. The value of the variable which is connected to some *RemoteVar* can be accessed using the *get* method just as it is also possible with *Var* and *Signal*. The *getProxy* method of *ProxyGenerator* still returns an object of type *Proxy* in order to be compatible to code using the old specialized proxy classes since it is their most specific common superclass.

2.2 Signal

The new *Signal* class directly implements the abstract *evaluate* method of *AbstractReactive* using a *Supplier* which is passed to its constructor right along with a list of proxies which are saved in the closure of the supplier. This makes it possible to pass the computation of the signal using the new Java 8 lambda style. Also there is a convenience constructor which also accepts a list of any kind of *ProxyGenerator* in order to name the variables the signal depends of, e.g. other signals or local variables.

An example signal generation could look as depicted in the following listing.

```
Var<Integer> a =
  new Var<>("a", Integer.valueOf(1));

Var<Integer> b =
  new Var<>("b", Integer.valueOf(2));

Signal<Integer> c =
  new Signal<>("c", () -> a.get() + b.get(), a, b);
```

2.3 Var

The new *Var* class provides a generic overlay to the old *Observable* class. Since the old specialized observables introduced their own methods for modifying the underlying object we had to think of an alternative solution to emulate that in our generic context. As a solution we proposed two methods: *set* and *modify*. The first one accepts an entire new value for the variable and sets its value to it, the second however accepts a *Consumer* which is applied to the underlying value. That way we can, for example, modify lists stored in variables while still containing just this small set of general methods which can notify the subscribers of a variable without having to be handcrafted for every different type. Another completely new advantage which comes with that approach is that several modifications can be executed at once without causing a new network transmission for each of those by just wrapping them in one consumer. So, we could, for example, insert three new elements in a observable list in an atomic way. That is depicted in the following example.

```
Var<List<Integer>> lst =
  new Var<>("lst", new ArrayList<Integer>());

Signal<Integer> n =
  new Signal<>("n", () -> lst.get().size(), lst);

lst.modify(self -> {
  self.add(1);
  self.add(2);
  self.add(3);
});
```

3 Test cases

In order to test the correctness of our new classes we introduced some new test cases which allocated and modified both local and remote variables as well as signals reacting to their changes. Using those tests we encountered an error in the dependency detection which could be fixed afterwards.

4 Protopeer Simulation

Finally we incorporated our code into a given simulation experiment and were able to test it in a vaster scenario than our own test cases. In order to do so we first had to update the old ANTLR-dependent code in order to be compatible to our new code what proved to need a deep rebuilding of both the reactive layer of the simulation as well as the mechanisms which compute the randomized expressions. In the old version those have been composed as simple strings which was not compatible to our typed version of Dream. After resolving a couple of bugs caused by the integration we finally got the simulation up and running.