Graphical editing support for QuickCheck models

Thomas Arts, Kirill Bogdanov, Alex Gerdes, John Hughes

This project has received funding from the EU FP7 Collaborative project PROWESS, grant number 317820, http://www.prowessproject.eu
Testing with QuickCheck

- QuickCheck permits one to write generators for test data and pre/postconditions.

- The expectation is that user provides a model, based on which test data is randomly generated.

- Illustration of testing a `write` operation:

  ```erlang
  write_args(_) -> [key(), int()].
  write(Key, Value) -> lock:write(Key, Value).
  write_post(_, [Key, Value], Res) -> eq(Res, ok).
  ```
Global state is a record-type of type \textit{state} with element \textit{started}, passed as an argument to all operations.

- \texttt{write\_pre(S) -> S#state.started}
- \texttt{write\_args(S) -> [ key(), int() ].}

- This is a precondition of the global state element of the global state.
- A list of arguments to pass to \texttt{write} of the system under test.
- Returns a generator for keys.
- Returns a generator for integers.
- Returns a generator for arguments.
- A precondition of the global state.
Testing *write* using global state

Assuming *started* is a boolean component of the global state reflecting if the system was started,

\[
\text{write\_args}(S) \to [\text{key()}, \text{int()}].
\]

\[
\text{write}(\text{Key}, \text{Value}) \to \text{lock:write}(\text{Key}, \text{Value}).
\]

\[
\text{write\_pre}(S) \to S\#\text{state}.\text{started}
\]

\[
\text{write\_post}(S, [\text{Key}, \text{Value}], \text{Res}) \to \text{eq}(\text{Res}, \text{ok}).
\]

\[
\text{write\_next}(S, \text{Res}, [\text{Key}, \text{Value}]) \to
\]

\[
S\#\text{state}\{\text{kvs} = [{\text{Key}, \text{Value}} | \text{proplists:delete}(\text{Key}, S\#\text{state}.\text{kvs})]\}.
\]
Locker example

- Can be started/stopped
- Can be locked/unlocked
- Does not include read/write
Part of this diagram in pure QuickCheck

lock_pre(S) -> S#state.started andalso not S#state.locked.
lock_args(S) -> [].
lock_next(S,Res,[]) -> S#state{locked=true}.

unlock_pre(S) -> S#state.started andalso S#state.locked.
unlock_args(S) -> [].
unlock_next(S,Res,[]) -> S#state{locked=false}.

Very easy to make a mistake in one of the above expressions
Now if we are doing something more complex

A lot of effort will go into ‘state maintenance’
Addition of a *read* transition around *unlocked*. 
What we did

• Developed a tool to edit graphical models.

• Names of operations are extracted from Erlang code.

• For the above example, the resulting model is half the size of the traditional model …

  … and much easier to maintain.

• Test failures and frequencies are automatically extracted from results of test execution.
Running tests produces a distribution of transitions
Weights can be updated

Changing weights makes operations of interest run more frequently.
Conclusions

- Existing QuickCheck models are hard to develop for complex state-transition diagrams.

- Developed interface to edit such diagrams.

- This will be part of the upcoming version of QuickCheck.

- Currently working on the case study with an industrial partner - testing of the interface to video on demand system.