Mocha2MV - Using reactive modules with VIS

Sebastian Gfeller
Supervisor: Barbara Jobstmann

January 10, 2008

1 Introduction

Mocha2MV provides a utility that translates a system specification in the Reactive Modules format (as accepted by jMocha) into BLIF-MV, a format accepted by VIS. Furthermore, an extension package allows to directly use Reactive Modules with VIS.

2 Architecture

The main conversion code was written in Java, reusing the jMocha parse tree (this is the only part needed, so if jMocha does not run on your platform, you might still get lucky). The VIS bindings are written in C code and are simply calling the conversion JAR, passing the output to VIS-internal commands.

2.1 Translation Details

BLIF-MV is a fairly low-level format based on models which can contain subcircuits, latches and input-output tables. Reactive Modules are composed of modules containing atoms that control variables. Both in Reactive Modules and in BLIF-MV, variable updates may be nondeterministic, but VIS only supports this in the form of additional pseudo-input variables.

Here is a simple Mocha module that describes a traffic light:

```
type color is { green, orange, red }

module minitraffic is
       interface light: color;
```

```
        lazy atom CtrlLight controls light reads light
             init
[] true -> light' := green;
    update
[] light = green -> light' := orange;
[] light = orange -> light' := red;
[] light = red -> light' := green;
```

This is translated to the following BLIF-MV code:

```
.model minitraffic
# Non-deterministic pseudo-inputs
.inputs nd0
.outputs light light'
.mv light,light' 3 green orange red
.mv nd0 2
.latch light' light
# CtrlLight
.reset light
green
.table nd0 light  -> light'
0 green orange
0 orange red
0 red green
1 green =light
1 orange =light
1 red =light
# Filling in the unused nondet combinations.
.end
```

Variables are represented by latches and the laziness of the atom is translated in a second update choice where the value of the latch stays the same. Module composition is simulated by subcircuits.

2.2 The Translation implementation

The translation is done in multiple steps. In a first step, all types are collected. Afterwards, the models are specified on the basis of the input modules. Finally, the input-output tables are filled by evaluating the guarded expressions and assignments on the set of possible inputs.

While VIS allows for a more compact form, expressing multiple inputs on one line, this does not correspond well to the arithmetic expressivity of reactive modules, so no size optimization is performed. This means that tables can get quite large when using big range types. Usually, this problem can be avoided by letting atoms only read and control few variables.

3 Installation

The Mocha2MV distribution consists of two parts, a JAR and command line tool to translate a reactive modules specification into a BLIF-MV file and bindings for VIS.

3.1 Installing the command line tool

To build the command line tool and corresponding JAR file for a Unix system, type

```
% ./install.sh
```

In the main directory. This launches the ANT task to compile the jar and places an executable (that points to this exact JAR) in /usr/local/bin.

There are no executables provided for Windows, but you can nevertheless use the jar directly to convert reactive modules. For this, just launch the default ant task.

3.2 VIS bindings

The VIS bindings are a bit trickier to install. First, you'll have to check whether you can build VIS on your system. Let $VIS denote the VIS source directory and $MOCHA2MV the Mocha2MV source directory.

1. Copy the directory "mocha" to VIS:

   ```
   % cp -R $MOCHA2MV/mocha $VIS/src/mocha
   ```

2. Let VIS know of the "read_mocha" command: For this, open $VIS/src/vm/vm.h and add the following include in the section "nested includes" (line 50)

   ```
   #include "mocha.h"
   ```

3. Launch the command initialization at VIS startup: in the file $VIS/src/vm/vmInit.c , add the following at the end of the function VmInit:

```
Mocha_Init();
```

4. Add mocha to the list of packages to build in $VIS/Makefile, line 98:

   ```
   ALL_PKGS = ... var vm mocha
   ```

5. Finally, make; make install in $VIS to install the version with VIS.

4 Usage

After the complicated installation, the conversion can be invoked pretty easily. To translate and load the file 3bitcounter.rm, you would type inside VIS

```
vis> read_mocha ./3bitcounter.rm
vis> init_verify
```

It will sometimes occur that Mocha2MV fails to translate the input file correctly. Most of the time, this will fail during the table generation phase. The error thrown should give some indication what construct is missing in the translator. In the following section, some limitations are named. However, it might also be possible, that an incomplete model is generated. In this case, the init_verify command will complain about it.

5 Limitations

Mocha2MV does not yet translate all valid Reactive Module specifications. The following things are missing:

Array types and Bitvectors Array types and bitvectors are not supported at the moment. They could, however easily be represented by multiple latches. Additions to the Expression evaluator would also be needed.

Compact sequence representations In the same fashion, sequence types are used in their extended form and not in a binary representation, which would be much smaller.

Event atoms While event atoms are translated, they don't produce usable output if more than one variable is awaited.