A User's Guide to ATLAS Version 0.2

Bryan A. Brady
bbrady@eecs.berkeley.edu

Sanjit A. Seshia
sseshia@eecs.berkeley.edu

Randal E. Bryant
Randy.Bryant@cs.cmu.edu

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

May 30, 2011

Contents

1 Introduction

ATLAS is a front-end for UCLID. The main purpose of ATLAS is to perform semi-automatic term-level abstraction on a UCLID model. During the implementation of ATLAS we extended the UCLID language to include additional features. The extensions to the UCLID language break backward compatibility and for that reason, we explain the ATLAS-extended UCLID language before we describe how to use ATLAS.

2 The ATLAS Specification Language

This section assumes the reader is familiar with the UCLID specification language. The UCLID specification language can be found in the UCLID userguide at http://uclid.eecs.berkeley.edu/.

The biggest addition to the UCLID language is support for module instantiation. Users familiar with Verilog will see many similarities as syntax added to UCLID is a subset of the syntax allowed in Verilog.

1. Addition of a portlist to module declarations;
2. addition of an `OUTPUT` section;
3. addition of module instantiations to the `DEFINE` section;
4. a global `DEFINE` section, and
5. addition of the `MAIN` module to all UCLID models.

3 Module support

ATLAS allows the user to specify an optional portlist. The original UCLID syntax for a module declaration is:

```
MODULE alu
```

The ATLAS syntax for a module declaration is either the original syntax, without a portlist, or the following syntax

```
MODULE alu (a, b, cntl, out)
```

where `a,b,cntl` are the inputs to the alu and `out` is the alu output. The reason to use a portlist is if you want to specify a certain order of the ports. If the user doesn't specify a portlist, the port order is the inputs followed by the outputs. The inputs and outputs are in the same order as they are declared in the `INPUT` and `OUTPUT` sections. The `OUTPUT` section follows the `INPUT` section in the `MODULE` declaration. For example:

```
MODULE alu

INPUT
a : BITVEC[32];
b : BITVEC[32];
cntl : BITVEC[4];

OUTPUT
out : BITVEC[32];
```

In the example module declared above, the portlist will be `a,b,cntl,out`.

Now that we can declare a module, we need to know how to use it. Module instantiations are defined in the `DEFINE` section in the following manner:

```
DEFINE
...
alu alu_instance_name(in0,in1,in2,out0);
```

When ATLAS sees this module instantiation, it will connect `in0` to `a`, `in1` to `b`, `in2` to `cntl`, and `out` to `out0`.

Often designers will declare global constants which denote things such as ALU op codes. These op codes may be referenced in many places, such as instruction decoding in pipelines, forwarding logic, and inside the ALU. Instead of requiring the designer to redefine these signals in each module, we've added a global `DEFINE` section so that these signals are only declared once. Any signal defined in the global `DEFINE` section will be accessible to every module. For example:

```
MODEL my_circuit

CONST (* global consts *)

DEFINE (* global defines *)
ALU_ADD := 3;

MODULE alu(out,a,b,cntl)

INPUT
a : BITVEC[32];
b : BITVEC[32];
cntl : BITVEC[4];

OUTPUT
```

```
out : BITVEC[32];

DEFINE
out := case
         cntl = ALU_ADD : a +_32 b;
         default : 0;
       esac;
```

There are two things to note in the above example. The first is that ALU_ADD is not defined inside the `alu` module, instead, it's defined in the global `DEFINE` section. The second point to note is that the user specified order of the portlist for module `alu` is different than the default portlist order. ATLAS will use the specified portlist in this case, instead of the default portlist described above.

The last change to the UCLID language due to adding module support is the addition of the `MAIN` module. The `MAIN` module is the top level module in the hierarchy. The example below shows how the `MAIN` module will be used in our running example:

```
MODEL my_circuit

CONST (* global consts *)

DEFINE (* global defines *)
ALU_ADD := 3;

MODULE alu(out,a,b,cntl)

INPUT
a : BITVEC[32];
b : BITVEC[32];
cntl : BITVEC[4];

OUTPUT
out : BITVEC[32];

VAR

CONST

DEFINE
out := case
         cntl = ALU_ADD : a +_32 b;
         default : 0;
       esac;
```

```
ASSIGN

MAIN

INPUT
in0 : BITVEC[32];
in1 : BITVEC[32];
cntl : BITVEC[4];

OUTPUT
output : BITVEC[32];

VAR

CONST

DEFINE

alu my_alu(in0,in1,cntl,output)

ASSIGN
```

`my_alu` is an instance of `alu` defined at the top level. When you run ATLAS on input files it generates a single output file in the original UCLID syntax with all modules flattened into the `MAIN` module.

4 Using ATLAS

General Usage: To run ATLAS use the following command:

```
atlas input_file1.ucl input_file2.ucl ... input_fileN.ucl
```

This will invoke ATLAS on the input files specified. This command will flatten all the modules into a single UCLID module and output it to `atlas.output.ucl`.

Output file renaming: To specify a different output file for ATLAS, use the `-of` switch:

```
atlas input_file1.ucl input_file2.ucl ... input_fileN.ucl -of output_file.ucl
```

ATLAS will generate the file`output_file.ucl`.

To generate an SMT file: Use the following command:

```
atlas input_file1.ucl input_file2.ucl ... input_fileN.ucl -of output_file.ucl -smt
```

This will generate an SMT file for use with an SMT solver. This option is for internal use and only works on UCLID models generated using the printFormulaAndQuit function in UCLID. This requires recompilation of UCLID, so unless you have source code access to UCLID, you can't use this option.

5 Term-level abstraction with ATLAS

If there is a node labed with `/*@UF_term*/`, ATLAS will perform function abstraction on that node and create a hybrid term-level model which is an abstracted version of the input model. If no node exists that's labeled with `/*@UF_term*/` then ATLAS will perform flattening only, and the resulting circuit will be identical to the original circuit (structurally, not syntactically). The resulting circuit will be compatible with the original UCLID syntax that supports hybrid bit-vector and term-level models.

Several examples are provided as part of the distribution. See the `ucl` subdirectory.