

# Averest 2.0 Howto

Matthias Schäfer  
Embedded Systems Group  
Department of Computer Science  
University of Kaiserslautern  
<http://es.cs.uni-kl.de>

December 3, 2009

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Installation and Configuration</b>	<b>2</b>
2.1	Prerequisites . . . . .	2
2.2	Installation . . . . .	2
<b>3</b>	<b>Using Averest</b>	<b>2</b>
3.1	Design Flow . . . . .	2
3.2	Quartz . . . . .	3
3.3	Compilation . . . . .	3
3.4	Simulation . . . . .	5
3.5	Verification . . . . .	6
3.6	Synthesis . . . . .	6
<b>4</b>	<b>Hints</b>	<b>6</b>

# 1 Introduction

The Averest project aims at providing a complete set of tools for the development of reactive systems. Currently, it consists of the following components:

- a compiler for translating synchronous programs to our intermediate format
- a set of transformations
- a symbolic model checker
- a simulator
- tools for hardware/software synthesis

Thus, Averest covers large parts of the design flow of reactive systems from the specification to the implementation. In the following sections, we will give you a brief and rough overview about the Averest tool chain and its usage.

## 2 Installation and Configuration

This section describes the required steps to use Averest properly. This includes the installation process and the configuration. The description assumes that you're using a Unix-like environment (Linux or Mac OS X are fine) and you're familiar with using the command line.

### 2.1 Prerequisites

Since Averest is developed using the programming language Moscow ML, you need to install the Moscow ML runtime environment (*camlrunm*). You can download it for various platforms from <sup>1</sup>.

### 2.2 Installation

First of all, you need to get the latest version of Averest<sup>2</sup>. Unpacking the archive results to a directory, which contains everything you need. The most interesting thing for the end-user is the subfolder `bin`. This directory contains bytecode files as well as short shell scripts for all tools of the Averest system.

You should add the `bin`-directory to your `PATH`-environment variable so that the ML runtime environment is able to find it. If you do not want to do this, you need to modify the shell scripts appropriately. Otherwise, you will not be able to run the programs.

In the following, we assume that Averest is installed (or rather copied) to `/opt/averest`. If you use *bash*, setting the environment variable can be done by the following command:

```
$ export PATH=$PATH:/opt/averest/bin
```

## 3 Using Averest

### 3.1 Design Flow

The design flow used by Averest consists of the following steps (see Figure 1):

1. The system is described as a program in the imperative synchronous language Quartz (see 3.2).

---

<sup>1</sup><http://www.itu.dk/~sestoft/mosml.html>

<sup>2</sup><http://www.averest.org/>

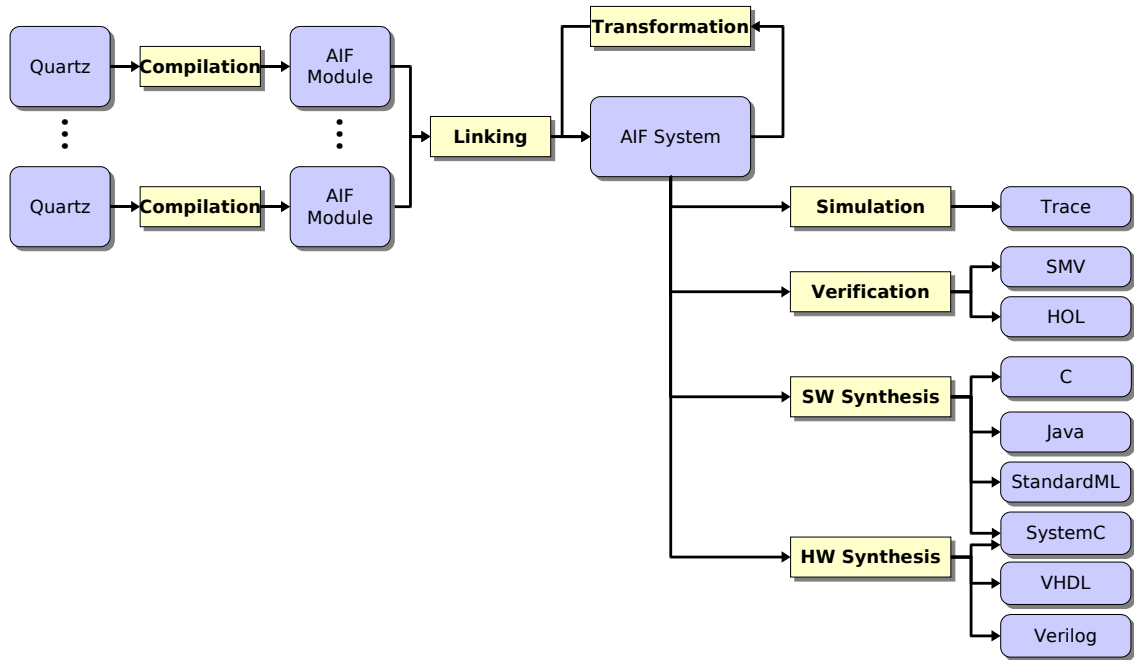


Figure 1: Averest Design Flow

2. The program is translated to a module or a system (see 3.3).
3. Based on the system, one can generate hardware and software implementations, simulate the system and verify the system against its specifications (see 3.4, 3.5, 3.6).

### 3.2 Quartz

The synchronous language Quartz is the basic input language of Averest. This language is used to describe the modules and systems. In the context of this tutorial, further explanations on Quartz would exceed the scope of this document. Hence, we just refer to the official Quartz documentation, which is available on <http://www.averest.org>. We just use a small example for further demonstrations, a simple full adder module implemented in Quartz:

```

package Arithmetic;

module FullAdd (
  event bool ?x, bool ?y, bool ?cin,
  bool !s, bool !cout
) {
  s = x xor y xor cin;
  cout = x and y or (x or y) and cin;
}

```

The `package` statement in the very first line of this file tells us, that this isn't the code for an AIF-system, this is the code for an AIF-module.

### 3.3 Compilation

After we've written our Quartz module for the full adder, we now want to translate it into the Averest Intermediate Format (AIF) for further usage. This is done by calling `qrz2aif`:

```
$ qrz2aif FullAdd.qrz
```

The simple application of *qrz2aif* on the file `FullAdd.qrz` generates a directory called *Arithmetic* which represents the package given in our Quartz source. This directory contains the AIF module called *FullAdd.aifm*. The suffix *.aifm* denotes that this file contains a module. In contrast, the AIF systems, which we will use later, have the suffix *.aifs*.

However, from Figure 1 one sees that a module cannot be used for further steps. Instead, we need a system. Therefore, we build a simple 4-bit adder which uses our `FullAdd` module and which is synchronized through a loop and a *pause* statement:

```
import Arithmetic.FullAdd;

module Adder (
  bool[4] ?x, bool[4] ?y, bool ?cin,
  bool[4] !s, bool !cout
) {
  bool[3] c;

  loop {
    FullAdd(x[0], y[0], cin, s [0], c [0] );
    FullAdd(x[1], y[1], c [0], s [1], c [1] );
    FullAdd(x[2], y[2], c [1], s [2], c [2] );
    FullAdd(x[3], y[3], c [2], s [3], cout);

    pause;
  }
}
```

Invoking the compiler now generates us a AIF-System *Adder.aifs*.

```
$ qrz2aif Adder.qrz
```

The resulting AIF file is fully linked, i. e. the *FullAdd* module is included in this AIF system. Naturally, the *FullAdd* module has to be built before the *Adder* system. Since we have not given a *package* directive, the compiler does not generate a module, but creates a system instead. The system can now be used for further actions like generating C code, translation to VHDL, verification or simulation.

To look at the results of compilation and linking, there is an Averest tool called *aif2txt*. This tool prints out the contents of an AIF module or system in a human readable way. For example, our `FullAdd` module looks like this:

```
$ aif2txt Arithmetic/FullAdd.aifm

module FullAdd:
  interface:
    x: input bool
    y: input bool
    cin: input bool
    s: output memorized bool
    cout: output memorized bool

  locals:
  abbreviations:
    __cvar5 : __cvar2|__cvar3
  strt : __cvar0
  prmt : __cvar1
  susp : __cvar2
  abrt : __cvar3
  strg : __cvar4
  inst : true
  insd : false
  term : false
  surface calls in surface:
```

```

surface calls in depth:
depth calls in depth:
surface:
  control flow:
  data flow:
    __cvar0 => s = !(cin<->!(x<->y))
    __cvar0 => cout = x&y|cin&(x|y)
  assertions:
depth:
  control flow:
  data flow:
  assertions:
transfers:
invar : true
specifications:

```

Assume that you want to transform the system in order to prepare it for further steps (e.g. flatten arrays because the target platform does not provide them). For this case, you can use the tool *aif2aif*. The command

```
$ aif2aif -h
```

prints out all available transformations.

### 3.4 Simulation

Simulating an AIF system is very useful for debugging and performance tests (benchmarks). So this is an important point for working with Averest. The tool for performing the simulation of an AIF system is *aif2trc*. *aif2trc* offers useful options like testbench modules for our system, which generates values for the input variables of our system. This enables us to simulate AIF modules simply by connecting a testbench module to it.

If our system is a *self-contained system*, i. e. it has no input variables, one can simply simulate the system like this:

```
$ aif2trc -s n <AIF system>.aifs
```

The parameter *n* here is an integer which gives the number of simulation steps for the system. For the case that our system has some input variables, we need a trace or a testbench module to control these variables in each step. The latter is a module which has an equally named output variable for each input variable of the simulated module. Lets return our running example, the *Adder* system. Assume that we want to test whether our system can add the two radix-2 numbers 0101 and 0110. Then, we create the file *AdderBench.qrz* as follows:

```

module AdderBench (
  bool[4] !x, bool[4] !y, bool !cin
) {
  // 0101
  x[0] = true;
  x[2] = true;

  // 0110
  y[1] = true;
  y[2] = true;

  pause;
}

```

After compiling the testbench module, the simulation is started by the following command:

```
$ aif2trc -s 1 -t AdderBench.aifs Adder.aifs
```

It returns the following output:

```
past:
  environment of step 0:
    cin : false
    __el10 : false
    cout : false
    x[0] : true
    x[1] : false
    x[2] : true
    x[3] : false
    y[0] : false
    y[1] : true
    y[2] : true
    y[3] : false
    s[0] : true
    s[1] : true
    s[2] : false
    s[3] : true
    c[0] : false
    c[1] : false
    c[2] : false
-----
```

It seems that our adder works as expected.

### 3.5 Verification

Built-in verification is provided by a symbolic model checker for finite and infinite state systems. However, it is also possible to use external symbolic model checker which are based on SMV-files (e.g. NuSMV) by transforming the AIF systems into SMVs by using the tool *aif2smv*. Also a transformation to HOL theorem prover is provided by *aif2hol*.

### 3.6 Synthesis

The synthesis tools *aif2\** translate system descriptions in AIF to hardware or software. Moreover, they can be used as a converter for third-party tools, e.g. other model checkers. Currently, the supported languages include ISO C, LEGO C, SystemC, Java, Standard ML, Verilog and VHDL.

## 4 Hints

The configuration of Averest can be changed by some environment variables. There are more variables which are used by Averest, but we do not need to go that much in detail here. However, one important switch should be mentioned.

Already compiled modules should be made available for all following modules and systems. For this purpose, we create a generally accessible library for them. The path to this library is determined in the environment variable `AIF_LIBRARY_PATH`. For example, you can set up the path as follows:

```
$ mkdir ~/.averest_library
$ export AIF_LIBRARY_PATH=$HOME/.averest_library
```

From now on, all compiled AIF modules will be stored in that folder. When the Linker looks up for some packages and modules, it will also use this prefix for its search.