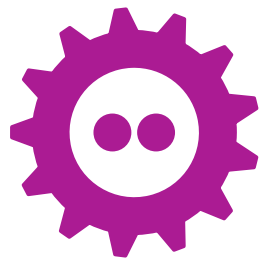
Verilog-AMS in Gnucap

Al Davis, Felix Salfelder

FOSDEM '24



Content

- Gnucap, what is it?
- Some history and what's coming
- Architecture and Plugins
- Verilog-AMS, what is it?
- Modelgen and state of the art
- Some features in detail
- Roadmap

Gnucap, what is it

- Run on hardware too small to run Spice! (originally)
- Beyond Spice – fast, mixed signal
- Updated architecture – C++, plugins
- Model compiler, modelgen

Gnucap, what is it

- Run on hardware too small to run Spice! (originally)
- Beyond Spice – fast, mixed signal
- Updated architecture – C++, plugins
- Model compiler, modelgen

- 1990. *ACS*, Al's Circuit Simulator
- 1992. GPL'd
- 2001. Renamed to *Gnucap*, a GNU project
- 2008-2010. Move to Plugins

Beyond Spice

- Mixed signal – implicit mixed mode
- "fast-spice"
- Large circuits
- Time step control

Mixed-mode

- "Implicit" mixed mode
- Introduced concept of a "connectmodule" (without the name)
- Digital techniques for analog.

Mixed-mode

- "Implicit" mixed mode
- Introduced concept of a "connectmodule" (without the name)
- Digital techniques for analog.

Digital techniques for analog

- Event queue – activity driven
- Partial solutions
- Low rank partial matrix solver
- incremental update
- Queues – load, eval, accept
- Full SPICE accuracy

Time step control

- cross events – separate smoothness and moving events
- real events – once it is scheduled, it's known
- Getting started

Software architecture

- C++
- Plugins
- Library

Program = main + library + plugins

Library

- Matrix solver
- Database
- I/O
- Expression evaluator

Plugins – why?

- Collaboration, modularity enforced
- Quality
- Dependencies
- Anyone can make or modify a plugin

Plugins – how?

- C++ derived classes
- Dynamically loaded (`dlopen`) extensions
- "Dispatcher"
- Wrappers

Plugins – what

- Devices
  - primitives, modelgen, SPICE, Qucsator..
- Commands, algorithms
  - ac, dc, tran, fourier, pz, sparam, postprocessing..
- Source languages
  - Verilog, SPICE, Spectre, gEDA, Qucsator
- Measurements
  - cross, peak, integrate, rms..

Plugins – wrappers

- Interface to foreign code
- Spice(s) – 3e3, 3f5, Jspice, Ngspice
- Qucs (incomplete)
- System-C (possible)
- Python

Model compiler

- Generates C++ from model description
- .. to build a device plugin
- `modelgen`: predates *AMS
- Could generate code for other simulators
- Now: updating to Verilog-AMS

Verilog-AMS: Analog Mixed Signal

- Modelling Language
  - Based on Verilog, IEEE Std 1364-2005
  - Standard in semiconductor industry
  - Conservative and signal-flow disciplines
  - Authored by former SPICE devs

Verilog-AMS: Analog Mixed Signal

- Modelling Language
  - Based on Verilog, IEEE Std 1364-2005
  - Standard in semiconductor industry
  - Conservative and signal-flow disciplines
  - Authored by former SPICE devs
- Features
  - Enables hierarchical modelling
  - Addresses computational efficiency
  - True mixed signal
  - "system-level analog"

Verilog-AMS current implementations

- commercial, costly, closed
- Analog-Subset, "Verilog-A"
  - ADMS (around 2000): generate SPICE models
  - OpenVAF (from 2020), "OSDI", simplified SPICE
  - 2023-24: modelgen-verilog overtaking (taking over?)
  - Enables analog compact modelling
- Beyond Verilog-A
  - from 2000 Gnucap: preparing for post-spice...
  - 2014 Verilog-AMS LRM v2.4
  - 2024: modelgen-verilog, mixed modelling (Funding secured)
  - Will enable system-level analog modelling

Going further

Beyond ADMS/openVAF, we have

- hierarchy
- (compiled) paramset
- binning
- compliant sources
- tolerances
- time step control
- extensibility

Design decisions in Modelgen-Verilog

- Retain proven code bases
- Stick to the architecture
- Make it work, then make it fast.
- Unconstrained by SPICE and/or OSDI
- Focus on new paradigms
- Examples today
  - (compiled) paramset
  - hierarchy
  - extensibility

Paramset 1: module overloading

```
paramset new_component existing_component
 parameter type value = default [range];
[..]
.protoparm(value_expression);
[..]
endparamset
```

Paramset 1: module overloading

```
paramset new_component existing_component
 parameter type value = default [range];
[..]
.protoparm(value_expression);
[..]
endparamset
```

- `paramset` replaces SPICE `.MODEL`
- Build `new_component` from `existing_component`

Paramset 1: module overloading

```
paramset new_component existing_component
 parameter type value = default [range];
[..]
.protoparm(value_expression);
[..]
endparamset
```

- `paramset` replaces SPICE `.MODEL`
- Build `new_component` from `existing_component`
- User defined parameters with ranges
- Model selection and binning
- code re-use

Paramset 2: pruning

Get rid of constants and structures before compilation

```
module capacitor(p,n);
    inout p,n;
    electrical p,n;

    parameter real c = 1. from [0:inf);
    parameter real ic = 0;
    analog begin
    if($param_given(ic) && analysis("ic"))
        V(p,n) <+ ic; // extra stuff, not normally needed
    else
        I(p,n) <+ ddt(c * V(p,n));
    end
endmodule
```

Paramset 2: pruning

Get rid of constants and structures before compilation

```
module capacitor_(p,n);
[..]
   parameter real c = 1. from [0:inf);
   parameter real ic = 0;
   analog begin
   if($param_given(ic) && analysis("ic"))
      V(p,n) <+ ic; // extra stuff, not normally needed
   else
      I(p,n) <+ ddt(c * V(p,n));
   end
endmodule

paramset capacitor capacitor_
   parameter real c = 1. from [0:inf);
  .c(c); // not setting ic, getting rid of it
endmodule
```

Paramset 3: pruning

Imagine...

Paramset 3: pruning

Imagine...

- A million instances of some device

Paramset 3: pruning

Imagine...

- A million instances of some device
- with 10.000 lines of model code each.

Paramset 3: pruning

Imagine...

- A million instances of some device
- with 10.000 lines of model code each.
- Computing the same constant value,

Paramset 3: pruning

Imagine...

- A million instances of some device
- with 10.000 lines of model code each.
- Computing the same constant value,
- in every iteration of your simulation.

Paramset 3: pruning

Imagine...

- A million instances of some device
- with 10.000 lines of model code each.
- Computing the same constant value,
- in every iteration of your simulation.
- (Most of it sits there unused, anyway.)

Paramset 3: pruning

Imagine...

- A million instances of some device
- with 10.000 lines of model code each.
- Computing the same constant value,
- in every iteration of your simulation.
- (Most of it sits there unused, anyway.)
- Now compile/load pruned models,

Paramset 3: pruning

Imagine...

- A million instances of some device
- with 10.000 lines of model code each.
- Computing the same constant value,
- in every iteration of your simulation.
- (Most of it sits there unused, anyway.)
- Now compile/load pruned models,
- get the same results (by design).

Paramset 3: pruning

Imagine...

- A million instances of some device
- with 10.000 lines of model code each.
- Computing the same constant value,
- in every iteration of your simulation.
- (Most of it sits there unused, anyway.)
- Now compile/load pruned models,
- get the same results (by design).

Corollary: "compilation time" is a red herring. Pruned models are small!

Hierarchy 1

Typical "compact modelling" approach

```
module rc_lowpass(out, in)
    electrical out,in; ground gnd;
    analog begin
        I(out, gnd) <+ ddt(1e-6* V(out, gnd));
        I(out, in) <+ V(out, in) / 1e3;
    end
endmodule
```

Hierarchy 1

Typical "compact modelling" approach

```
module rc_lowpass(out, in)
    electrical out,in; ground gnd;
    analog begin
        I(out, gnd) <+ ddt(1e-6* V(out, gnd));
        I(out, in) <+ V(out, in) / 1e3;
    end
endmodule
```

But no need to reimplement components.

```
module rc_lowpass(out, in)
    electrical out,in,gnd; ground gnd;
    capacitor #(.c(1u)) c(1, gnd);
    resistor #(.r(1k)) r(1, 2)
endmodule
```

Hierarchy 2

Model what you need, not what you can.

```
module rc_lowpass(out, in)
    electrical out, in, gnd; ground gnd;
    resistor #(.r(1k)) r(1, 2)
    analog begin
       I(out, gnd) <+ ddt(1e-6* V(out, gnd));
    end
endmodule
```

Hierarchy 2

Model what you need, not what you can.

```
module rc_lowpass(out, in)
    electrical out, in, gnd; ground gnd;
    resistor #(.r(1k)) r(1, 2)
    analog begin
       I(out, gnd) <+ ddt(1e-6* V(out, gnd));
    end
endmodule
```

Why?

- implement models for re-use
- re-use models!
- memory footprint
- compilation time

Hierarchy 2

Model what you need, not what you can.

```
module rc_lowpass(out, in)
    electrical out, in, gnd; ground gnd;
    resistor #(.r(1k)) r(1, 2)
    analog begin
       I(out, gnd) <+ ddt(1e-6* V(out, gnd));
    end
endmodule
```

Why?

- implement models for re-use
- re-use models!
- memory footprint
- compilation time
- validate once.

Extensibility

"The $ character introduces a language construct which enables development of user-defined tasks and functions." (LRM 2.8.3)

Extensibility

"The $ character introduces a language construct which enables development of user-defined tasks and functions." (LRM 2.8.3)

- Some tasks and functions are plugins.

Extensibility

"The $ character introduces a language construct which enables development of user-defined tasks and functions." (LRM 2.8.3)

- Some tasks and functions are plugins.
- All of them will be.
- cos, sin, exp, ln, ddt, idt, ddx, slew, transition, laplace_*, zi_*, $strobe, $monitor, $write, $stop, ...
- No boundary between user extensions and built-in stuff.

Roadmap 2024

- Modelgen-Verilog
  - Essentially complete Verilog-A
  - Logic modelling
  - Digital simulation
  - connect modules
- Simulator
  - Full support for Verilog concepts such as disciplines, natures, connect semantics
  - performance enhancements in the solver
  - vectors, arrays of nodes
- Interoperability and standardisation
  - Verilog as an interchange format for schematic and layout
  - Update and extend device wrappers
  - Target simulators other than Gnucap

Help wanted

- Plugins
- Wrappers
- Data exchange
- Test driving
- Create wishlist