SML

Spice Manipulation Language

Ron Alleyne
Rob Toth
Spencer Greenberg
Michael Apap
What is SPICE?

- **Simulation Program with Integrated Circuit Emphasis**
  - Uses complex systems of equations to solve for voltages at circuit nodes and currents through circuit branches

- SPICE Predecessor
  - CANCER Computer Analysis of Non-Linear Circuits Excluding Radiation (Ron Roher, 1970’s)

- Circuits are described using network lists or netlists and commands are used to invoke simulation
What is SPICE?

• Sample circuit and netlist:

Title – My Circuit’s Spice Code
.options post reltol=1e-6
.op
Vin 1 0 3
R1 1 2 10
R2 2 0 20
.END
Why not SPICE?

• SPICE netlists are difficult to interpret

• SPICE netlists of large circuits are difficult to create

• SPICE lacks programming functionality that an engineer may want before or during simulation
  – SPICE does not allow for dynamic circuit definitions
  – SPICE does not allow for circuit elements to be defined based on mathematical derivations
What is SML?

- **SPICE Manipulation Language** is a wrapper for SPICE code
- Small, yet powerful, C++ like language
- Allows for intuitive definition of circuit topology
- Fully integrated with hspice variant of SPICE
Why SML?

• Offers the engineer the power to:
  – easily manipulate cumbersome circuit designs using SML list objects
  – harness the full strength of the underlying SPICE engine
  – dynamically define circuit element connections and their properties
  – inject SPICE code which can reference SML circuit elements.
SML How-To

• Data Types
  – Simple Types
    • **String** - Stores a sequence of characters
    • **Int** - Stores a c++ long integer value
    • **Float** – Stores a c++ double floating point value
    • **List** – 1-indexed array that can store any SML Object
SML How-To

• Data Types
  – Basic Two Terminal Circuit Elements
    • res
      A resistor object
    • cap
      A capacitor object
    • ind
      A inductor object
    • cs
      A current source object
    • vs
      A voltage source object
SML How-To

• Data Types
  – More Circuit Elements (Semi-Conductors)
    • Diode
      A diode object.
    • Bjt
      A bi-polar junction transistor
    • Mosfet
      A metal oxide semiconductor field effect transistor
SML How-To

• Operators
  – Basic Operators
    • +, -, *, /
    • =, ==, <, >, <=, >=
    • print(object_name)
  – List and String Operators
    • @
    • #
  – Circuit Operators
    • ->
SML How-To

• Circuit Element Properties
  
  c1.capacitance = .00555
  
  c1.initial_voltage = 10
  
  i1.inductance = 4
  
  myVs.voltage = 12

• Connection Circuit Elements
  
  myBjt.base->c1.pos
  myMosfet.source->c1.neg
  
  c1.neg->i1.pos
SML How-To

- Sample input file: myCircuit.sml

```
$[ .op
  .dc #Vin 1 10 1
  .print dc v(#Vin.pos ) v(#r1.pos ,#r1.neg )]$
res r1
res r2
vs Vin
Vin.voltage = 3
r1.resistance = 10
r2.resistance = 20
Vin.pos->r1.pos
r1.neg->r2.pos
r1.neg->r2.pos
Vin.neg->ground
R2.neg->ground
```
SML How-To

• Execution Command:
  sml myCircuit.sml myCircuit.sp

This spice list generated by SML Compiler for myCircuit.sml
*options
  .options post reltol=1e-6
  *simulation commands, all parameters (models, etc)
  .op
  .dc  vVin 1 10 1
  .print dc v( node1 ) v( node1 , node2 )
  rr1 node1 node2 20.00
  rr2 node2 0 10.00
  vVIn node1 0 3.00
  .end
SML How-To

• List Example:

```sml
[$
 .op
 ]$
vs vin
vin.voltage=10
list l
res r1
res r2
res r3
r1.pos -> r2.pos
r1.pos -> r3.pos
r1.neg -> r2.neg
r1.neg -> r3.neg
l @ r1
l @ r2
l @ r3
list g = l
g[1].pos -> l[1].neg
vin.neg->ground
l[1].pos->vin.pos
g[1].neg->ground
```
• List Example Output:

This spice list generated by SML Compiler for inputlist.sml
*options
.options post reltol=1e-6
*simulation commands, all parameters (models, etc)

.op
vvin node1 0 10.00
rl_xyz1 node1 node2 0.00
rl_xyz2 node1 node2 0.00
rl_xyz3 node1 node2 0.00
rg_xyz4 node2 0 0.00
rg_xyz5 node2 0 0.00
rg_xyz6 node2 0 0.00
.end
Resistor Capacitor Bank Example:

```sml
SML

op
.print v( #vin )

vs vin
vin.voltage=10
int a = 1
float f = 0.01
list l
while(a < 25)
{
    res r
    r.resistance = a * 100
    r.pos->vin.pos
    r.neg->ground
    cap c
    c.capacitance = f * 2
    c.pos->vin.pos
    c.neg->ground
    c.initial_voltage= f * f
    l@r
    l@c
    a=a+1
    f=f+0.01
}
vin.neg->ground
```
SML How-To

• Resistor Capacitor Bank Output:

This spice list generated by SML Compiler for inputrescapbank50.sml
*options
.options post reltol=1e-6
*simulation commands, all parameters (models, etc)

.op
.print v( vvin )
vvin node1 0 10.00
rl_xyz1 node1 0 100.00
cl_xyz2 node1 0 0.02 IC=0.00
rl_xyz3 node1 0 200.00
cl_xyz4 node1 0 0.04 IC=0.00
rl_xyz5 node1 0 300.00
cl_xyz6 node1 0 0.06 IC=0.00
rl_xyz7 node1 0 400.00
cl_xyz8 node1 0 0.08 IC=0.00
rl_xyz9 node1 0 500.00
cl_xyz10 node1 0 0.10 IC=0.00
rl_xyz11 node1 0 600.00
cl_xyz12 node1 0 0.12 IC=0.00
rl_xyz13 node1 0 700.00
cl_xyz14 node1 0 0.14 IC=0.00
rl_xyz15 node1 0 800.00
cl_xyz16 node1 0 0.16 IC=0.01
rl_xyz17 node1 0 900.00
cl_xyz18 node1 0 0.18 IC=0.01
rl_xyz19 node1 0 1000.00
cl_xyz20 node1 0 0.20 IC=0.01
rl_xyz21 node1 0 1100.00
cl_xyz22 node1 0 0.22 IC=0.01
rl_xyz23 node1 0 1200.00
cl_xyz24 node1 0 0.24 IC=0.01
rl_xyz25 node1 0 1300.00
cl_xyz26 node1 0 0.26 IC=0.02
rl_xyz27 node1 0 1400.00
cl_xyz28 node1 0 0.28 IC=0.02
rl_xyz29 node1 0 1500.00
cl_xyz30 node1 0 0.30 IC=0.02
rl_xyz31 node1 0 1600.00
cl_xyz32 node1 0 0.32 IC=0.03
rl_xyz33 node1 0 1700.00
cl_xyz34 node1 0 0.34 IC=0.03
rl_xyz35 node1 0 1800.00
cl_xyz36 node1 0 0.36 IC=0.03
rl_xyz37 node1 0 1900.00
cl_xyz38 node1 0 0.38 IC=0.04
rl_xyz39 node1 0 2000.00
cl_xyz40 node1 0 0.40 IC=0.04
rl_xyz41 node1 0 2100.00
cl_xyz42 node1 0 0.42 IC=0.04
rl_xyz43 node1 0 2200.00
cl_xyz44 node1 0 0.44 IC=0.05
rl_xyz45 node1 0 2300.00
cl_xyz46 node1 0 0.46 IC=0.05
rl_xyz47 node1 0 2400.00
cl_xyz48 node1 0 0.48 IC=0.06
.end
Inside SML

• Architecture – Built in C++!!
Inside SML

• **Lexer**
  – Takes a stream of characters and converts them into a stream of tokens.
  – Comments and white space are ignored.
  – Output of the lexer is passed to the parser.

• **Parser**
  – Takes a stream of tokens and builds from it an abstract syntax tree based on our ANTLR grammar.

• **Static and Semantics Analyzer**
  – Walks through the abstract syntax tree.
  – For each node corresponding to a binary operation the types of the left and right expression are tested for consistency.
  – The symbol table is also constructed during this stage and memory is allocated for constant expressions and local variables.
Inside SML

• Interpreter
  – The interpreter walks through the abstract syntax tree after the static and semantic analyzer has verified that it is correct.
  – Objects are set in memory and operators are evaluated.
  – Conditionals, loops, SML connections and objects are evaluated.
  – Symbol table left with all the appropriate information based on the program.
Inside SML

• Code Generator
  – implemented in codeGen.h.
  – Key functions
    • spicify()
      – takes the symbol table of circuit element object as its input and create a netlist of the input circuit’s topology
    • nodeWalker()
      – visits all nodes and invokes nodeCruncher() where necessary
    • nodeCruncher()
      – minimizes the number of circuit element terminals or nodes needed to identify connections in the circuit
    • SMLInjecter()
      – takes injected SPICE and resolves SML object references
Lessons Learned

• Spencer Greenberg
  – crucial to decide how each element of a language will be implemented before deciding on the language itself.
  – learned the extreme importance of code reuse.
  – would have been a good idea to test each finished portion of code.

• Ron Alleyne
  – Our organization was very clear and straightforward; however when we tried to combine our efforts bugs began to develop.
  – I learned that creating your own language requires a different type of workload then regular programming.
  – Weekly deadlines and clear cut goals are a necessity.
Lessons Learned

• Michael Apap
  – I learned that developing a sound plan is a necessity when trying to develop something at this scale.
  – I learned that no matter how much planning and organization is done, when all the parts come together there will be some connection issues.

• Rob Toth
  – After working on the Interpreter for SML I learned a lot on the power of programming languages.
  – I was able to create my own while loops (nested!) and if statements, all at the same time using our SML objects to develop circuits.
THE END