SML
(Spice Manipulation Language)

Spencer Greenberg
Michael Apap
Robert Toth
Ron Alleyne

SPICE Manipulation Language

Abstract

SML (Temp Name) has been proposed as a means of simplifying SPICE coding. By developing a wrapper language that generates SPICE code, the developers of SML have given the typical engineer the power to easily manipulate cumbersome circuit configurations, while harnessing the full power of the latest SPICE engine technologies. SML can be done without the hassle of being distracted by the nuances of any one particular SPICE implementation.

Introduction

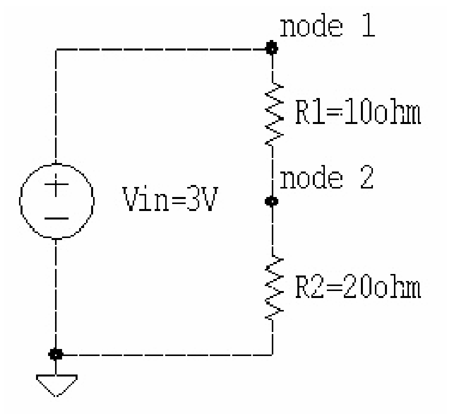
In any engineering discipline, simulation serves as an invaluable tool. The time and energy conserved in early development stages can be re-invested in the overall quality and optimization of the design. For electrical engineering, SPICE (**S**imulation **P**rogram with **I**ntegrated **C**ircuit **E**mphasis) serves as a computer program designed to assist with the simulation of analog electronic circuits. It has become the industrial standard for such experimentation. Early advancement in this area of software engineering saw the development of CANCER (Computer Analysis of Non-Linear Circuits Excluding Radiation) by Ron Roher of the University of California – Berkeley and three other predecessors of modern SPICE variants in the 1970s. In the years since, newer software releases have added greater functionally by introducing support for dynamic and semi-conductor circuit elements and improved analysis.

In order to use these products efficiently, designers must be familiar with the general "nodal" language that is used to create "net-lists" that represent circuits. In addition to being very cryptic, such languages can also vary from one particular SPICE implementation to another. Furthermore, the limitations of the language can make dynamic circuit configuration and manipulation very time-consuming.

SML serves as an interface between the designer's circuit schematic and the subsequent "netlist" that will be pumped into the SPICE translator. It hides the complexity and the implementation-dependent details of any particular SPICE version. It provides portability that allows compliant code to be generated into any particular SPICE dialect. It also provides greater programming functionality to allow for easier manipulation of circuit elements. Furthermore, the readability of the language makes it easier for engineers to collude on simulation parameters and results without spending a lot of time trying to parse a fellow designer's net-list. This language also serves as a viable entry point for further research into the development of high-level languages to model all types of biological and industrial node-based networks.

Background

What else is there to know about SPICE? SPICE simulation files are created by defining all nodes, and the circuit elements between each node. There are many variants of SPICE available to engineers. P-SPICE is a PC version commonly used by many students as a tool to create circuits, whereas H-SPICE is a UNIX based version. SPICE offers different analyses of circuits such as: AC Analysis, DC Analysis, and Transient Analysis. Below is a sample of circuit and the Spice code that would generate it:



Spice Code:

```
Vin 1 0 3

R1 1 2 10

R2 2 0 20

.END
```

As you can see, the code above is not explanatory unless you understand SPICE programming. The syntax is very structurally and without memorizing what goes where, even the code above will have no meaning to someone who has done SPICE programming. Thus SML will offer a simpler and more intuitive approach to simulating circuits by creating a wrapper for SPICE programming.

Language Properties

1) The most important property of SML is the ability to intuitively define a circuit. The core structure is the List object. By using this abstract List objected different components can be added and connected, attendant values can be set and initial conditions can be stipulated.

2) Every object has global scope, so no matter where an object is created its name can be referred to from anywhere in the program.

3) Comments are indicated with "//".

4) Lists are 1-indexed, so L[1] gives the first element in list L.

5) Valid object types are as follows:

float	: a floating point value
int	: an integer value
List	: a list of objects
Res	: a resistor
Cap	: a capictor
Ind	: an inductor
MI	: a mutual inductor
CS	: a current source
VCC	: a voltage controlled current source
VS	: a voltage source
VCV	: a voltage controlled voltage source

Language Operators

typespec name : creates an object of type typespec called name

Example
```
Res r
```

obj1 = obj2 : sets obj1 to a duplicate of obj2 as long as they are of the same type

Example
```
Res r,r2
r.resistance=5
r=r2
```

obj1 == obj2 : returns 1 if obj1 is of the same type and has all the same properties as obj2. For lists, this means that the nth element of each list must have the same properties for all n.

	Example r1==r2

{obj1, obj2, obj3}	: creates a list object containing obj1, obj2 and obj3. Can include any number of objects
	
	Example List Circuit1 = {Res r,Cap c}
	List caps = {Cap c1,c2,c3}

typespec[n]	:creates a list object containing n objects of type typespec. Example List L = Res[5]
list1 U list2	:creates a list which contains the elements of list2 appended to list1 Example List L = Res[5] U {Cap c,Res r}
➔	:creates a physical connection between 2 objects Example r1(+)->r2(-)
Parallel(List L)	:connects the objects in List L in parallel.
Series(List L)	:connects the objects in List L in series.

Properties of circuit objects

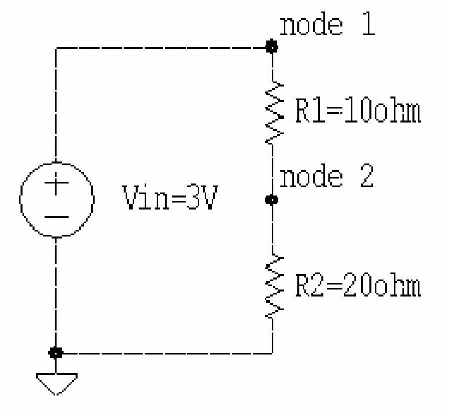
All circuit components have negative and positive terminals in addition to other properties.

obj1(+)	:returns an object representing the positive terminal of obj1
obj1(-)	:returns an object representing the negative terminal of obj1

The basic components of any circuit have inherent properties associated with it, for example:

```
Res res1
res1.resistance = 3
```

SML Sample Circuit



```
Res r1,r2
VS Vin
Vin.voltage=3
r1.resistance=10
r2.resistance=20
Vin(+)->r1(-)
r1(+)->r2(-)
r2(+)->Vin(-)
```

Sample Code Explained

First we create 2 resistors (r1,r2) and a voltage source (Vin). Next, we set the voltage and the resistance for these components. The last 3 lines of code connect the components into a circuit.

References

[1] *UNIVERSITY of PENNSYLVANIA*-DEPARTMENT OF ELECTRICAL ENGINEERING
SPICE - A Brief Overview
http://www.seas.upenn.edu/~jan/spice/spice.overview.html

[2] SPICE TUTORIAL
http://www.brunel.ac.uk/~eestmba/usergS.html

[3] SPICE HISTORY
http://www.ecircuitcenter.com/SpiceTopics/History.htm