SIMetrix Scriptチュートリアル

はじめに

SIMetrixは、シンプルなインタープリタ型スクリプト言語を備えています。これは大体BASICに基づいており、ほとんどのユーザインターフェイスを記述しています。

このマニュアルは、コンピュータプログラミングの概念に共感するユーザが、独自のスクリプトを開発したり、内部スクリプトを変更してユーザインターフェイスを調整したりする手段を提供します。

スクリプト開発について、主要なアプリケーションを以下の3つに分類しました。ただし、他のアプリケーションもあるかもしれません。

1. ユーザインターフェイスの変更。個々の好みまたは特殊なアプリケーションに合わせます。
2. 自動シミュレーション。たとえば、大きな回路について多数のテストを実行する必要がある場合があります。シミュレーションには時間がかかるため、夜間または週末に実行したいでしょう。 簡単なスクリプトでこのタスクを実行できます。
3. 特殊な解析。SIMetrixで提供される曲線解析関数はすべて、スクリプトを使用して実装されています。特殊な機能を実装するために、独自のスクリプトを記述することができます。また、パフォーマンスとヒストグラム解析に使用される目標関数は「ユーザ定義関数」であり、実際にはスクリプトとして実装されます。特殊なアプリケーションには、さらに多くの目標関数を追加できます。

スクリプト言語は、約688個の関数と約323個のコマンドによってサポートされています。これらは、SIMetrixコアへのインターフェイスといくつかの汎用的な機能を提供します。

組み込み関数と同様に、'C'または'C++'で独自の関数を開発できるツールキットが利用可能です。

チュートリアル

ここでは、以下の例を説明します。
例1: Hello World!
例2: ループの入門

例3: クロスプロービング
例4: パーツリストの作成

例1: Hello World!

プログラミング言語'C'を学んだ人なら誰でも、今では有名な"Hello World"プログラムを知っているでしょう。おそらく最も簡単なプログラムです。ここでは、SIMetrixの"Hello World"スクリプトを作成して実行します。

スクリプトは簡単です。

```
echo "Hello World!"
```

上記のスクリプトを作成するには、メニューの**File > New > Script**を選択して、組み込みのスクリプトエディタを開きます。次のとおり入力します。

```
echo "Hello World!"
```

テキストを*hello.sxscr*というファイルに保存します。スクリプトを実行するには、ツールバーのRunボタンをクリックするか、コマンドラインで"hello"と入力します。次のメッセージがメッセージウィンドウに表示されるはずです。

```
Hello World!
```

コマンドラインでファイル名を入力すると、スクリプトが実行されます。ファイルの拡張子が.sxscrの場合、拡張子は省略できます。キーまたはメニューを割り当てて、スクリプトを実行することもできます。 コマンドラインで次のとおり入力します。

```
DefKey F6 HELLO
```

そしてF6キーを押すと、メッセージが再び表示されます。メニューの定義については、User Defined Key and Menu Definitionsを参照してください。

例2: ループの入門

この例では、ベクトル（または配列）内のすべての要素を合計します。ベクトルを作成するには、サンプル回路の1つでシミュレーションを実行します。回路図を手動で開くこと（スクリプトから実行することもできます）を除いて、プロセス全体がスクリプトに入れられます。

まず、サンプル回路*General/AMP.sxsch*を開きます。過渡解析の実行が選択されていることを確認してください。**File > New Script**を選択します。これにより、カレントディレクトリがSCRIPTに設定された状態でテキストエディタが開きます。 次のとおり入力します。

```
Netlist design.net
Run design.net

let sum = 0
for idx=0 to length(vout)-1
let sum = sum + vout[idx]
next idx

echo The sum of all values in vout is {sum}
```

スクリプトをファイル*SUM.sxscr*に保存します。次に、コマンドラインでSUMと入力します。シミュレーションが実行され、次のメッセージがメッセージウィンドウに表示されます。

```
The sum of all values in vout is -6.1663737561
```

回路を変更した場合、または異なるモデルライブラリをセットアップした場合、上記の数値と異なることがあります。このスクリプトは、4つの新しい概念を導入します。

1. Forループ
2. ブレース置換 (最後の行の{sum})
3. ベクトル(または配列)
4. シミュレーションデータへのアクセス

このスクリプトを1行ずつ見ていきます。

最初の2行はシミュレーションを実行します。実際、メニューまたはF9キーを使用してシミュレーションを実行するたびに、同様のことが行われます。`Netlist design.net`は、回路のネットリストを生成し、design.netというファイルに保存します。次に、Run `design.net`は、netlist `design.net`上でシミュレーションを実行します。

下記の行は、最終結果を保持する変数`sum`を作成して初期化します。

```
let sum = 0
```

次の3行は簡単な*for*文です。変数idxは、ゼロから始まりlength(vout)-1で終わるループで毎回1ずつ増加します。voutは変数（実際はベクトル）で、シミュレータによって生成され、VOUTネット上の電圧のシミュレーション値を保持します。このネットには端子シンボルが付けられています。length(vout)はvoutの要素数を返します（idxは0から始まるため、1が減算されます）。

```
let sum = sum + vout[idx]
```

上記の行のvout[idx]はインデックス付きの式であり、ベクトルvoutの番号idxの要素を返します。もちろん、sumは累積の合計です。

```
echo The sum of all values in vout is {sum}
```

最後の行には、*ブレース置換*{sum}が含まれています。sumが評価され、結果が式と中括弧を置き換えます。詳細については、Braced Substitutionsを参照してください。

例3: クロスプロービング

標準のプロットメニューでは、一度に1つの曲線をプロットします。ここでは、マウスが右クリックされるまでクロスプローブ曲線を繰り返しプロットするスクリプトについて説明します。

```
let start=1
do while probe()
if start then
plot {netname()}
else
curve {netname()}
endif
let start=0
probe
loop
```

このスクリプトは、if文、while文、関数、および電圧クロスプローブを可能にする機能、すなわち、関数NetNameとProbeおよびコマンドProbeを導入します。

スクリプトは、probe()関数が0 (=FALSE)を返すまで、do whileとloopの間の文を繰り返し

実行します。Probe関数は、カーソルの形状をオシロスコーププローブに変更しますが、ユーザがマウスの左ボタンまたは右ボタンを押すまで戻りません。ユーザが左ボタンを押すと、関数は1 (=TRUE)を返し、ループ内の文の実行が継続されます。ユーザが右ボタンを押すと、Probe関数は0 (=FALSE)を返し、ループは完了し、スクリプトは終了します。

```
if start then
plot {netname()}
else
curve {netname()}
endif
```

上記の5行では、ループの最初の回でstartは1に等しく、Plotコマンドが実行されます。これにより、新しいグラフが作成されます。続いて、startはゼロに設定され、Curveコマンドが実行されます。すると、作成済みのグラフに新しい曲線が追加されます。

PlotコマンドとCurveコマンドの引数である{netname()}は、前の例で見たブレース置換です。NetName関数は、関数の実行時にカーソルに最も近いネットの名前である文字列を返します。この関数は、ユーザがマウスの左ボタンを押すとすぐに実行されるため、NetNameによって返される文字列は、ユーザが指しているネットになります。NetNameによって返される値は文字列ですが、Plotコマンドには数値表現が必要です。netname()を中括弧に入れると、評価結果が入力されたかのように置換されます。したがって、ユーザがVOUTという名前のネットをポイントすると、netname()は'VOUT'を返し、plotまたはcurveの後に配置されます。すなわち、plot voutが実行されます。

```
probe
```

最後のコマンドは、Probeコマンドを呼び出します。これはProbe関数と同じですが、結果を返しません。Probe関数とProbeコマンドの両方が、マウスのアップクリックとダウンクリックで戻るために必要です。2番目のProbeは、マウスボタンのアップクリックを単に待ちます。

クロスプロービングに使用される関数は、他に4つあります。それらは、GetNearestNet、NearestInst、PinName、およびBranchです。

最後に注意があります。plot {netname()}は、算術文字（たとえば'+' や'-'）などの特定の文字が名前に含まれるベクトルでは機能しません。これらの文字は文字通りの意味として解釈され、通常エラーが発生します。名前にこれらの文字が含まれるベクトルをプロットするには、Vec()関数を使用して、ベクトル名を文字列として指定する必要があります。例えば次の

とおりです。

```
plot Vec(netname())
```

ここでは中括弧が使用されていないことに注意してください。この理由は、Vec()関数が、プロットされる実際のデータを含む数値ベクトルを返すためです。NetName 関数は、実際のデータではなくベクトルの名前を返します。

例4: パーツリストの作成

このスクリプトの例では、現在選択されている回路図のコンポーネントのリストを、その参照と値とともにメッセージウィンドウに表示します。

```
* mk_bom.txt Display parts list in message window
if NOT SelSchem() then
echo There are no schematics open
exit all
endif

let refs = PropValues('ref','ref')

for idx=0 to length(refs)-1

let val = PropValues('value', 'ref', refs[idx])
* check for duplicate ref
if length(val)==1 then
echo {refs[idx]} {val}
else
echo Duplicate reference {refs[idx]}. Ignoring
endif
next idx
```

最初から見ていきます。

```
* do_bom.txt Display parts list in message window
```

上記の行はコメントです。'*'で始まる行は無視されます。

```
if NOT SelSchem() then
```

上記の行は、*if*文の開始です。`SelSchem()`関数は、回路図が開いている場合は1を返し、開いていない場合は0を返します。それで、`if NOT SelSchem()`は、「開いている回路図がないならば」を意味します。これは、ユーザが実際に回路図を開いたかの初期チェックです。

```
echo There are no schematics open
exit all
```

上記の行は、回路図が開いていない場合に実行されます。最初の行は`echo`コマンドを呼び出します。これは、同じ行の後続のすべてのテキストをメッセージウィンドウにエコーします。2番目の行は*exit*文です。この場合、実行が中止され、残りのスクリプトは無視されます。

```
endif
```

上記の行は*if*文を終了します。 すべての`if`に対して、マッチングする`endif`または`end if`が必要です。
通常はもちろん、ユーザが回路図を開いて、スクリプトの残りが実行されることを期待しています。

```
let refs = PropValues('ref', 'ref')
```

上記の行はletコマンドを呼び出します。このコマンドは、評価する代入式を予期します。この場合、関数PropValuesの呼び出しの結果をrefに代入します。この例では、回路図上のすべてのインスタンス（つまりシンボル）について、コンポーネント参照を返します。

```
for idx=0 to length(refs)-1
```

上記の行は*for*ループを開始します。この行とマッチングする`next`の間のブロックは、`idx`の値がループのたびに1ずつ増加し、`length(refs)-1`に達するまで繰り返されます。length関数はrefs変数の要素数を返すので、refsのすべての要素に対してループが繰り返されます。

```
let val = PropValues('value', 'ref', refs[idx])
```

上記の行により、`PropValues`関数が再度呼び出されます。今回は、`refs[idx]`という値を持つプロパティ*ref*を持つインスタンスの*value*プロパティの値を返します。回路図に注釈が付けられていると仮定すると（すべてのコンポーネントに一意の参照が割り当てられている）、

この呼び出しの結果は**val**に割り当てられた単一の値になります。

```
if length(val)==1 then
echo {refs[idx]} {val}
```

上記の**if**文は、valの長さが1であることを確認します。これは、参照が一意であることを意味します。その場合、Echoコマンドが呼び出され、それに続くすべてのテキストがメッセージウィンドウに表示されます。この例では、**echo**コマンドの後に2つの*ブレース置換*が続きます。ブレース置換は、中括弧 '{'および '}'で囲まれた式です。中括弧と囲まれた式は、値が入力されたかのように、式を評価した結果に置き換えられます。ブレース置換は、SIMetrixスクリプト言語の非常に重要な機能です。ここでは、結果はコンポーネントの参照であり、値はメッセージウィンドウに表示されます。

```
else
echo Duplicate reference {refs[idx]}. Ignoring
endif
```

上記はforループの最後の部分です。if式の**length(val)==1**が偽の場合、この部分が実行されます。これは、そのコンポーネント参照を持つコンポーネントが複数あることを意味します。そのコンポーネント参照は無視されることを示すメッセージが出力されます。

```
next idx
```

最後の行はforループを終了します。