プログラミング言語最終課題 課題B

鈴木佑典

課題内容

作成したプログラムは，Verilogプロジェクトにおいてファイルの依存関係を抽出してそれを可視化，シミュレーション用のMakefileを作成してビルドを簡単にするものである．以下，この課題を行った背景と，作成したプログラムについて述べる．

課題背景

SystemVerilogはハードウェア設計の場面で広く用いられているが，SystemVerilogにはいくつか開発に際して不便な点が多々ある．その一つが，シミュレーションやビルド時にC言語のように実行の際に関係しているファイルをすべて記載しなければならない点である．小規模なプロジェクトであれば大きな問題にならないが，大規模なプロジェクトであれば問題となってくる．なぜなら，SystemVerilogは名前空間という概念がなく，ファイルのinclude等の操作も無いため，コードをよく確認しないと依存ファイルが特定出来ないからだ．大規模なプロジェクトを扱う際にはMakefileが必要不可欠であり，それらを自動生成するプログラムには大きな意味がある．故に，プロジェクト内のファイルを読み込み，シミュレーション用Makefileを作成するプログラムを作成した．また，上記の理由から，Verilogではモジュール間の接続を特定するのにも大きな労力がかかる．故に，モジュール間の接続を可視化することも研究開発の助けとなるため，その機能も搭載した．

実装内容

上記の理由から，今回はVerilogの依存関係可視化し，Makefileを作成した．まず，実装に際していくつか外部クレートを用いたので，それらの説明と用いた理由の説明を行う．

外部ライブラリ

今回用いた外部クレートは

- sv-parser
- walkdir
- ptree

の3つである．以下にそれぞれのクレートの説明を行う．

sv-parser

sv-parserは，SystemVerilogのコードからastを生成するクレートである．本来であれば，パーサーから実装するべきであるが，SystemVerilogの使用はBNFでも100ページ存在し，この課題ですべてを実装することは現実的に不可能である．一方，パーサーのサブセットを作成し使用することは可能である．今回であればモジュールの宣言部分だけ抽出するといった機能を持ったもので十分だ．しかし，それでは今後の機能拡張をする際に大幅なリワークが必要となり，拡張性が大きく下がってしまう．また，パーサーを作成すること自体今回の課題のメインでは無い．故に，ここでは完全なSystemVerilogのパーサーを提供しているクレートを利用した．sv-parserは数少ないIEEE基準で完全なSystenVerilogのparserを提供しているクレートであり，このクレートの存在がSystemVerilogの開発ツール作成においてRustを選定する理由になる．

walkdir

walkdirは，クロスプラットフォームでのディレクトリ移動を実装したクレートである．ディレクトリ移動の実装もまた本質的では無いため，ある程度の安全性が保証されている外部クレートを使用した．

ptree

ptreeは，unixのtreeコマンドの様な形式でtreeを出力できるクレートだ．モジュールの階層関係を出力するために使用した．

次に，プログラムの実装について述べる

実装

本プログラムの実装は，

1. sv-parserを用いてSystemVerilogをパースしastを生成する
2. astからモジュールの宣言を抽出し，ファイルの名前に紐づけたHashMapに作る
3. astからインスタンス宣言を抽出し，モジュールの名前に紐つけたHashMapを作る
4. 1 ~ 3の操作を，walkdirを用いてディレクトリ内のすべてのSystemVerilogファイルに対して行い，それらの結果を結合する．結果として，ファイルとモジュールの関係のHashMapとモジュール同士の依存関係のHashMapが生成される．
5. 作成したHashMapのうち，モジュール同士の依存関係のHashMapに対してトポロジカルソートを適用し，それらをptreeを用いてverilog.treeというファイルに出力する．
6. 4で生成された2つのHashMapを利用して，指定したtopモジュールから辿れるファイルのリストを作成する．これが，Make時に必要となるファイルのリストとなる．
7. 最後に，6で生成したリストをもとに，Makefileを作成する．

という流れで実行される．これらの処理を行っているファイルは

- get_module_list.rs (1, 2, 3)
- get_file_module_list.rs (4)
- topological_sort.rs make_tree.rs (5)
- gen_source_list.rs (6)
- makefile.rs (7)

である．

実行

実行は，

```
cargo run [project directory name] [top file name] [run command] [target
file name]
```

サンプルプロジェクトはFFTというフォルダにある．このプロジェクトはFFTのハードウェア実装であり，トップモジュールはtop.sv, トップモジュールのテストベンチがtop_sim.svである．

サンプルの実行結果

サンプルプロジェクトのディレクトリ構造は以下．

```
./FFT
├── FFT.py
├── FFT_stage1.py
├── FFT_stage1_py.txt
├── FFT_unit.sv
├── FFT_unit_sim.sv
├── FIFO.sv
├── a.out
├── bit_rev.py
├── bit_rev.txt
├── bit_rev64.txt
├── butt2.sv
├── const_w.vh
├── draw.py
├── dump.vcd
├── gen_input.py
├── gen_roter.py
├── input.png
├── input.txt
├── input_fft.png
├── input_pyfft.png
├── output.png
├── output.txt
├── plot_input.py
├── reoder.sv
├── roter.txt
├── roter1.txt
├── shift_reg.sv
├── stage1.txt
├── stage2.txt
├── stage3.txt
├── stage4.txt
├── stage5.txt
├── stage6.txt
├── top.sv
└── top_sim.sv
```

この内，Pythonのファイルは SystemVerilogのテストケース作成用のプログラム及び出力の確認用プログラム，.txtが入力及び中間出力の確認用ファイル，.pngがFFTの結果をmatplotlibで出力したものである．今回のプロジェクトでは，モジュール名とファイル名が一致している．それに対して，出力されたtreeとMakefileは以下である．

```
top_sim
└─ top
   ├─ FFT_unit
   │  ├─ shift_reg
   │  ├─ butt2
```

```
    │   └─ FIFO
    └─ reorder
```

```
RUN = iverilog -g2009
SRCS = /home/meltpoint/verirunner/FFT/top.sv
/home/meltpoint/verirunner/FFT/FFT_unit.sv
/home/meltpoint/verirunner/FFT/shift_reg.sv
/home/meltpoint/verirunner/FFT/butt2.sv
/home/meltpoint/verirunner/FFT/FIFO.sv
/home/meltpoint/verirunner/FFT/reoder.sv
/home/meltpoint/verirunner/FFT/top_sim.sv
TARGET = a.out

$(TARGET): $(SRCS)
    $(RUN) $(SRCS)

.PHONY: clean

clean:
    rm -f $(TARGET)

.PHONY: clean_all

clean_all:
    rm -f $(TARGET)
    find . -name "*.vcd" -delete

.PHONY: run

run:
    $(RUN) $(SRCS)
    ./$(TARGET)
```

今後の展望

今回はSystemVerilogのシミュレーション用Makefileを作成するプログラムを実装したが，SystemVerilogにはまだまだ実装すべき機能が多くある．例えば，SystemVerilog向けの高速シミュレータVerilatorの制約を満たすようにファイルを整形する機能である．Verilatorでは定義されているモジュールとファイルの名前が一致していないといけない．その他にもいくつか制約があるので，Verilatorでシミュレーションすることを想定していないプロジェクトに適用するのは難しい．しかし．それを満たすように自動で整形できれば，シミュレーションの効率がより向上するだろう．このように，SystemVerilogの開発環境にはまだまだ開発の余地があり，その恩恵も大きいだろう．故に，今後も開発環境の開発を続ける大きな意味がある．