

A Meta-Interpreter for Circuit-Extraction

Krishnaprasad Thirunarayan
Department of Computer Science and Engineering
Wright State University, Dayton, OH 45435.
email: tkprasad@cs.wright.edu

Abstract

The design of a VLSI circuit consists of a description of the circuit in terms of its components and subcomponents, at various levels of detail. To verify that the layout of a VLSI circuit conforms to its design, one needs to work backwards from the lowest-level description of the circuit and recognize the higher-level components it constitutes. This paper is concerned with the application of logic programming techniques in the formal verification of the structural correctness of the VLSI circuit layouts. In particular, we review Michael Dukes' Generalized Extraction System (GES) that compiles design descriptions into a set of extraction rules, and then study the benefits and the limitations of using a meta-interpreter approach to extraction.

1 Introduction

The design of a VLSI circuit consists of a description of the circuit in terms of

its components and subcomponents, at various levels of detail. To verify that the layout of a VLSI circuit conforms to its design, one needs to work backwards from the lowest-level description of the circuit and recognize the higher-level components it constitutes. In other words, one needs to *extract* top-level components from the transistor netlist using the design description. This paper is concerned with the design of a tool using Prolog [1, 2] for the formal verification of the structural correctness of a VLSI circuit layout with respect to its design specification.

The “hierarchical” knowledge about the structure of the circuit can be formally defined in Horn logic or a suitable subset of VHDL [5] that can be mechanically translated as Prolog facts and rules. Michael Dukes' GES (Generalized Extraction System) [3, 4] uses this kind of declarative specification in Prolog to obtain customized rules for extracting higher-level components from the netlist. However, it is not always necessary to “compile” the declarative specification into the corresponding ex-

traction rules. In fact, we show here that it is possible to write a simple general purpose meta-interpreter in Prolog that uses the declarative specification *directly*, to perform circuit extraction, thereby eliminating the explicit generation of extraction rules. In this paper, we study the meta-interpreter, consider its benefits and its limitations. We also give a few illustrative examples.

2 Review of Dukes' GES

We briefly review GES using an example from [3].

The transistor netlist is represented as a sequence of facts of the form:

```
p(Gate, Drain, Source, X_Loc, Y_Loc).
n(Gate, Drain, Source, X_Loc, Y_Loc).
```

where *p* (resp. *n*) designates a p-type (resp. n-type) transistor. An example CMOS netlist for two invertors, with the output of the first as the input of the second, would look like:

```
p(in1, vdd, out1, 50, 50).
n(in1, gnd, out1, 50, 40).
p(out1, vdd, out2, 100, 50).
n(out1, gnd, out2, 100, 40).
```

The fact that the source and the drain are interchangeable can be expressed as follows:

```
ptrans(G, D, S, X, Y) :- p(G, D, S, X, Y).
ptrans(G, D, S, X, Y) :- p(G, S, D, X, Y).
ntrans(G, D, S, X, Y) :- n(G, D, S, X, Y).
ntrans(G, D, S, X, Y) :- n(G, S, D, X, Y).
```

(We avoid commutativity rules such as `p(G, D, S, X, Y) :- p(G, S, D, X, Y)`, as they lead to nonterminating computations.)

The structure of the inverter can be specified as follows:

```
inverter(In, Out, X, Y) :-
    ptrans(In, vdd, Out, X, Y),
    ntrans(In, gnd, Out, _, _).
```

The extraction rule that recognizes an inverter in the netlist, eliminates the transistors that construct it, and adds a fact corresponding to the inverter, can be written as follows:

```
extract_inverter :-
    ptrans(In, vdd, Out, X, Y),
    ntrans(In, gnd, Out, _, _),
    rem_p(In, vdd, Out),
    rem_n(In, gnd, Out),
    asserta(inverter(In, Out, X, Y)),
    fail.
extract_inverter.
```

The goal `fail` in the body of the first rule ensures that *all* instances of the `inverter` are extracted through backtracking.

The removal of the transistors can be done using the Prolog built-in `retract` that works only on facts as follows:

```
rem_p(G, D, S) :-
    retract(p(G, D, S, _, _)).
rem_p(G, D, S) :-
    retract(p(G, S, D, _, _)).
rem_n(G, D, S) :-
    retract(n(G, D, S, _, _)).
rem_n(G, D, S) :-
    retract(n(G, S, D, _, _)).
```

In general, for every Prolog rule that describes the structure of a higher-level component (in terms of its immediate subcomponents), one needs to write a separate extraction rule. This does not seem very desirable from a practical standpoint. Fortunately, it is possible to automatically translate a rule describing the structure of a component into the corresponding extraction rule. For example, the structure-describing rule template

```
COMPONENT :-
    SUBCOMPONENT_1,
        ...,
    SUBCOMPONENT_n.
```

can be compiled as

```
:- dynamic COMPONENT.
:- dynamic SUBCOMPONENT_1.
    ...,
:- dynamic SUBCOMPONENT_n.

extract_COMPONENT :-
    SUBCOMPONENT_1,
        ...,
    SUBCOMPONENT_n,
    retract(SUBCOMPONENT_1),
        ...,
    retract(SUBCOMPONENT_n),
    assert(COMPONENT),
    fail.

extract_COMPONENT.
```

For run-time efficiency reasons, this approach is satisfactory for low-level components that appear in a device in

large numbers. However, this kind of translation is not necessary for higher-level components that are not used multiple times. In particular, we wish to investigate logic programming techniques that will use the structure-describing rules *directly*, rather than duplicate the information through compilation.

3 The Meta-Interpreter

We apply meta-programming techniques to “effectively generate” the required extraction rules, on demand, from the structure-describing rules.

To extract a component, we need to extract each of its subcomponents *recursively*. The immediate subcomponents are determined by retrieving the corresponding structure-describing rule. This is captured by the following straight-forward meta-program.

```
extract(Comp) :-
    clause(Comp,Sub_Comps),
    Sub_Comps \== true,
    call(Sub_Comps),
    remove_primitive(Sub_Comps),
    asserta(Comp),fail.

extract(_).

remove_primitive((Com1,Com2)) :-
    !, remove_primitive(Com1),
    remove_primitive(Com2).

remove_primitive(Comp) :-
    clause(Comp,true),
    !, retract(Comp).
```

```

remove_primitive(Comp) :-
    clause(Comp,Sub_Comps),
    remove_primitive(Sub_Comps).

```

The `fail`-predicate in the first clause of `extract` enables extraction of all occurrences of a component. Furthermore, before a subcomponent of a component is extracted, a check is made to establish that all the immediate subcomponents of the component are simultaneously present in the netlist. This is done to ensure that the subcomponents are extracted irrevocably in the “right context”. Note also that, in Quintus Prolog, the built-in `retract` can be used on facts containing only *dynamic* predicates.

To connect the declarative specification with the meta-interpreter, the following “interface” rule is added:

```

extract_COMPONENT :-
    extract(COMPONENT).

```

However, this simplistic approach does not work in cases where “normal” Prolog predicates (such as the built-ins or the user-defined predicates other than those that describe the hardware structure) appear in the body of the structure-describing rules, to impose certain additional constraints present in the hardware. To illustrate this point, consider the specification of a tristate-inverter containing connectivity constraints as shown. ($\backslash+$ stands for negation-as-failure in Quintus Prolog.)

```

:- dynamic invZ/6.

```

```

connected([Node|Tail]) :-
    member(Node,Tail), !.
connected([_| Tail]) :-
    connected(Tail).

```

```

invZ(P,N,I,0,X,Y) :-
    ptrans(I,vdd,Q,X1,Y1),
    ptrans(P,Q,0,X2,Y2),
    ntrans(N,0,R,X3,Y3),
    ntrans(I,R,gnd,X4,Y4),
    \+ connected([Q,R,vdd,gnd]),
    X is (X1+X2+X3+X4)/4,
    Y is (Y1+Y2+Y3+Y4)/4.

```

To “protect” the static non-hardware predicates, we introduce a meta-predicate named `constraint`, modify the `invZ`-rule and add another clause for `remove_primitive` as shown below.

```

constraint(Test) :- call(Test).

```

```

invZ(P,N,I,0,X,Y) :-
    ptrans(I,vdd,Q,X1,Y1),
    ptrans(P,Q,0,X2,Y2),
    ntrans(N,0,R,X3,Y3),
    ntrans(I,R,gnd,X4,Y4),
    constraint(\+connected([Q,R,vdd,gnd])),
    constraint( (X is (X1+X2+X3+X4)/4,
                Y is (Y1+Y2+Y3+Y4)/4) ).

```

```

remove_primitive(constraint(_)) :- !.

```

The correctness of the above meta-interpreter crucially depends on the following characteristics of the hardware specifications.

- The Prolog facts represent low-level components (such as the transistors) which are combined to get higher-level components. The predicates in the facts and in the structure-describing rule heads are disjoint. Furthermore, the hardware designs considered here exhibit a hierarchical structure, and so, the corresponding rules are non-recursive.
- Each low-level component is a part of a single higher-level (parent) component. In other words, a given fact is not “shared” by multiple rules whose head and body are both satisfied. Thus, one can delete a fact representing a sub-component when its parent component is identified.
- The above observations imply that the programs can be “stratified” in such a way that the operation of replacing a collection of low-level components with an equivalent higher-level component can be done by deleting all the facts corresponding to the former and adding the fact corresponding to the latter. Furthermore, this process terminates.

Overall, the meta-interpretation strategy seems flexible and reasonably efficient for higher-level components.

4 Conclusions

We studied the application of meta-programming techniques to perform circuit extraction from the declarative specification of the hardware components and the transistor netlist. We argued that Dukes’ approach to compiling the specification is efficient for low-level components, while the meta-interpreter approach advocated here can add flexibility to the system for higher-level components. We also stated the characteristics of the hardware specifications that we exploited in our approach.

References

- [1] Bratko, I., *Prolog Programming for Artificial Intelligence*, Addison-Wesley Publ. Co., 1990.
- [2] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, 1987.
- [3] Dukes, M., “A Generalized Extraction System for VLSI”, TR# AFIT/DS/ENG/91, Air Force Institute of Technology, 1990.
- [4] Dukes, M., Brown, F., and DeGroat, J., “Verification of Layout Descriptions using GES”, 29th DAC, pp. 63-72, 1992.
- [5] Thirunarayan, K., Reintjes, P., and Ewing, R., “VHDL-93 Parser in Prolog”, Wright Laboratory, WPAFB, 1994.