Correct High-Level Synthesis: a Formal Perspective

J.M. Mendías, R. Hermida, M. Fernández
Dpto. de Arquitectura de Computadores y Automática
Escuela Superior de Informática. Universidad Complutense de Madrid. Spain
e-mail: mendias@eucmos.sim.ucm.es, rhermida@eucmax.sim.ucm.es

Abstract

This paper presents a formal synthesis system which delegates the design space exploration to non-formal, and potentially incorrect, high level synthesis tools. With a quadratic complexity, our system obtains either a truly correct-by-construction design, since the formal design process constitutes itself the verification process, or demonstrates that the solution found by the conventional tool was incorrect.

1: Introduction

Behavioural synthesis tools have had to evolve quickly last years, and all this development has got a price to be paid: the complexity of algorithms grows and the supporting data structures becomes more sophisticated. As a consequence, the bugs in the tools proliferate. The effect is that reliance in synthesis tools decreases and nowadays no sensible designer takes the risk to accept a circuit automatically generated without a later validation step.

To address this problem the so-called formal synthesis systems appeared recently [1]. Their aim is to perform all the design steps within a purely mathematical framework, where the synthesis process itself becomes the proof of soundness of the implementation. There are three common characteristics of these systems: i) they just use a single mathematical formalism; ii) they synthesise by sequentially applying a set of behaviour preserving transformations; iii) they are not automatic: although any transformation can be done automatically, the sequence is decided by a designer.

In this paper we will present the main features of the formal tool FRESH (FRom Equations to Hardware) that covers the whole HLS process. Like previous systems, the tool is not self-contained, but besides the possibility of being operated by a designer, it can be driven by a conventional HSL tool, thus creating a framework where the correctness of the design can be ensured automatically. The main advantages of this system are: *Easiness* that means not having to modify conventional tools to adapt them to the mathematical formalism; they simply must deliver the results of their search. *Reliability* that means selecting a kernel of few simple transformations (which minimises the number of error sources) and adopting a declarative representation with first-order formal semantics (which simplifies both designer interpretation and tool processing). *Applicability* is obtained neither by constraining the kind of accepted behaviours nor the kind of reachable designs. *Efficiency* is obtained by specialising the system in order to perform complete formal synthesis processes (from specs to datapath+controller) with quadratic complexity.

2: System representation

It is well-known that a combinational circuit, alike a linear data-flow graph, can be represented by a set of equations describing its structure. If we, additionally, want the set of equations to be able to describe fedback sequential circuits or iterative computations, we must add a temporal operator to represent delay. We call this operator *fby* (followed by) and this sort of representation *equational spec*. The equational spec of a 2nd order recursive filter looks as follows:

$$\{\ out = z - (a1*(0\ fby\ z) + a2*(0\ fby\ 0\ fby\ z))$$
$$z = b1*(0\ fby\ z) + b2*(0\ fby\ 0\ fby\ z) + in \qquad \}$$

An equational spec, besides the schematic interpretation, has a useful semantics in terms of streams. A stream is a sequence (infinite for our purpose) of values belonging to certain domain in correspondence with those ones carried by a signal along time. If a signal denotes a stream then, every combinational operator will denote a pointwise stream function; every constant will denote a stream with a single value infinitely repeated; and *fby* will be the operator that inserts a value in the head of a stream.

Getting equational specs from procedural ones (i.e. VHDL) is not difficult. It is enough to compile the source code and describe the resulting graph in terms of a set of mutually recursive equations. However, in many applications it is easier to bypass the elaboration of procedural code, since equational specs can be directly obtained from alternative specification styles like block diagrams or temporal specs.

To derive a circuit implementation from a given spec, it is necessary that both of them, as well as, all the intermediate states (e.g. partially scheduled or partially allocated graphs) share a single representation formalism. So it is necessary to find some extra temporal operators which can express the key idea of any HLS process: the time multiplexed use of hardware resources.Thus, we will define the operators << (replicate) and >> (sample) to express the frequency ratio between two signals, and the operator || (interleave) to express multiplexed use of resources. In addition we will use the symbol # (wildcard) to represent those values computed but not stored in certain cycle (i.e. calculated by a resource which no operation has been assigned to). And, finally, we will introduce the *next* operator as the inverse of *fby*. As an example, let *index* = <1,2,3,4...> be an stream, then:

(*index* << 2) = < 1, 1, 2, 2, 3, 3, 4 ... >
(2 >> *index*) = < 2, 4, 6, 8, 10, 12, 14 ... >
(# || *index*) = < #, 2, #, 4, #, 6, # ... >
(next *index*) = < 2, 3, 4, 5, 6, ... >

3: High level stages formulation

Once we have got the minimum operator set to express any partial design, we are going to present a set of proved properties that they fulfill. With these properties we intend to formalise mathematically every stage of a HLS process and to behold the task of designing as a simple first order equational calculus process. The properties, shown as families of equations, can be classified as follows:

Inverse operator: states that *next* is the inverse operator of *fby*, and that *sample* is *replicate*'s.

Temporal distributivity: states the distributivity of temporal operators with respect to combinational ones.

Identity elements: states that any constant signal is not affected by temporal operators.

High level synthesis: there are five families,

a) *Temporal multiplexing theorem*, assigns a cycle to compute an operation tagging remaining cycles with *wildcards* for later reuse.

$k>>x = k>>(\#\ \mathrm{fby})^m(\ \#\ \|^{k-m-1}\|\ \#\ \|\ \mathrm{next}^m(x)\ \|\ \#\ \|.^m.\|\ \#\ )$

b) *Architectural delayers replacement theorem*, that allows to use architectural registers to store auxiliary values in those cycles in which the registers cannot be observed.

$(\ \#\ \|^{k-m-1}\|\ \#\ \|\ \mathrm{next}^m((z\ \mathrm{fby}\ (k>>x))<<k)\ \|\ \#\ \|.^m.\|\ \#\ )$

$= (\#\ \mathrm{fby})^{k-m-1} z\ \mathrm{fby}\ (\#\ \mathrm{fby})^m(\ \#\|^{k-m-1}\|\#\|\mathrm{next}^m(x)\|\#\|.^m.\|\#\ )$

c) *Memorization theorem*s, replaces chains of delayers by a feedback one when the values are computed and consumed within the same algorithm initiation:

$(\#\ \mathrm{fby})^{n+1}(\ \#\ \|^{k-m-1}\|\ \#\ \|\ x\ \|\ \#\ \|.^m.\|\ \#\ )$

$\approx \mathrm{fix}(\ \lambda z.(\#\ \mathrm{fby}(\ \#\ \|^{k-m-1}\|\ \#\ \|\ x\ \|\ z\ \|.^n.\|\ z\ \|\ \#\ \|^{m-n}\|\ \#\ ))\ )$

or when they are in different initiations:

$(\#\ \mathrm{fby})^{n+1}(\ \#\ \|^{k-m-1}\|\ \#\ \|\ x\ \|\ \#\ \|.^m.\|\ \#\ )$

$\approx \mathrm{fix}(\ \lambda z.(\#\ \mathrm{fby}(\ z\ \|^{n-m}\|\ z\ \|\ \#\ \|^{k-n-1}\|\ \#\ \|\ \mathrm{x}\ \|\ z\ \|.^m.\|\ z\ ))\ )$

The *fix* operator allows to express anonymous recursivity, and ≈ indicates that both streams never transmit different values (although one of them can transmit a *wildcard* and the other one an ordinary value).

d) *Decomposition theorem*, separates the different RT-level actions included in a high-level operation (i.e. operand selection, computation, and result storage) in order to separately reuse the different hardware modules involved.

$(\ x_1\ \|^{i-1}\|\ x_i\ \|^{k-i}\|\ x_k\ ) = (\ x_1\ \|^{i-1}\|\ (\#\|^{i-1}\|x_i\|^{k-i}\|\#)\ \|^{k-i}\|\ x_k\ )$

e) *Input anticipation theorem*, states that a value read in a slow input port can be anticipated.

$(\ x_1\ \|^{k-m-1}\|\ x_{k-m-1}\ \|\ \mathrm{next}^m(x<<k)\ \|\ x_{k-m+1}\ \|.^m.\|\ x_k\ )$

$= (\ x_1\ \|^{k-m-1}\|\ x_{k-m-1}\ \|\ (x<<k)\ \|\ x_{k-m+1}\ \|.^m.\|\ x_k\ )$

RT-level implementation theorems: formalise the operator mapping into hardware modules [2].

4: A transformational design kernel

Now we need a set of manipulation rules that allow to properly apply the properties to transform an equational spec into another one with the same behaviour, but different cost-performance. The rules presented have been proved to be correct and constitute, as a whole, the only computation mechanism allowed in our formal synthesis system. Given that this kernel is small and simple, we have been able to reduce to a minimum the risk of programming errors. The summary of the set of rules is: a) *Substitution*: to replace any occurrence of a signal by its definition. b) *Rename*: to change the name of a signal. c) *Expansion*: to replace any subterm of a definition by a new signal, if the signal is defined as the subterm to be replaced. d) *Elimination*: to remove any definition not used by any other. e) *Cleaning*: to remove redundant definitions. f) *Replacement*: to replace any *wildcard* by any other term (to be understood in term of reuse). g) *Rewriting* to transform a definition applying a universal first order formula.

5: Automating formal HLS

We have developed an algorithm which governs the right rule application order to perform formal and automatic HLS. This algorithm does not explore the design space, but it applies the decisions already made by a conventional tool in order to prove they are mathematically correct. So its inputs are the equational spec of the circuit and the synthesis decisions externally made, and its output is either another equational spec, representing the designed circuit, or a report about the erroneous resolutions.

The algorithm understands the original specs as a single cycle circuit having all the operators chained. Its aim is to transform this circuit into another one that, keeping the external data sampling frequency, works internally *k* times faster, may have non-chained operators, and may reuse both registers and operators. To do it, the algorithm starts from data sources (input ports, constant and outputs of the architectural delays) and gradually increases the frequency of each operator, scheduling each one in a cycle. The process finishes when the data drains are reached (output ports and inputs of the architectural delays). The scheme of the algorithm is shown as follows:

inputs: spec, design decisions
outputs: circuit, were decisions correct?

normalization	*next* elimination
source multiplexing	scheduling correctness check
architectural delays scheduling	action decomposition
for 1 to critical path length	delayers feedback
sample export	module reuse
sample extract	allocation correctness check
operation scheduling	multiplexer synthesis
end for	control reuse

Its temporal complexity is quadratic respect to the number of nodes in the graph and linear respect to the number of cycles in the scheduling. A 2nd. order filter fully designed (just with RT mapping to go) is next shown:

```
{  out = 4 >> t31
   t40 = t23 - t31
   t39 = mux(t23,t29,t28,t56) + mux(in << 4,t28,t29,t56)
   t38 = mux(a1,b2,b1,a2,t53) * mux(t23,t19,t54)
   t19 = 0 fby mux(t19,t23,t57)
   t23 = 0 fby mux(t23,t39,t58)
   t29 = # fby mux(t38,t29,t61)
   t28 = 0 fby mux(t28,t38,t61)
   t31 = # fby mux(t39,t40,t61)                    DATAPATH
   ------------------------------------------------------
   t53 = ( 0 || 1 || 2 || 3 )
   t54 = ( 1 || 1 || 0 || 1 )                      CONTROLER
   t57 = ( 0 || 0 || 0 || 1 )
   t58 = ( 1 || 0 || 0 || 1 )
   t56 = ( 0 || 1 || # || 2 )
   t61 = ( 0 || 0 || 1 || 1 )          }
```

References

[1] R. Kumar et al. *Formal synthesis in circuit design - A classification and survey*. Proc. Formal methods in CAD, 1996.

[2] J.M. Mendías, R. Hermida, M. Femández. *Algebraic support for transformational hardware allocation*, Proc. VLSI'97.