Formally Analysed Dynamic Synthesis of Hardware

Kong Woei Susanto and Tom Melham

Department of Computing Science
University of Glasgow
{susanto,tfm}@dcs.gla.ac.uk
http://www.dcs.gla.ac.uk/~tfm/dynhw

Abstract. Dynamic hardware reconfiguration based on run-time system specialization is viable with Xilinx XC6200 series FPGAs. The research challenge for formal verification is to help ensure the correctness of dynamically generated hardware. In this paper, the approach is to verify a specialization synthesis algorithm used to reconfigure FPGA designs. The verification approach is based on a deep embedding of a language for netlists and the relational hardware modeling style.

1 Introduction

Most micro-electronic circuit design is done as ASIC design. The design is validated extensively, either by simulation or by formal verification methods, before it is manufactured. The production of the design as a chip will take several months before the designer can test the hardware itself. If the system fails to satisfy the functional specification at the testing stage, the designers must redesign the circuit and redo the whole validation, production, and test cycle.

The flexibility of implementing a design in a short period has made Field Programmable Gate Arrays (FPGAs) popular for rapid system prototyping. The FPGA is an array of generic function cells configured by assigning configuration bits to specify each cell's functionality and interconnection with other cells. The fabrication phase of the ASIC design cycle is replaced by configuration of an FPGA chip, which takes at most a few seconds. Furthermore, FPGAs make it possible to implement designs for small production markets.

Most reconfigurable hardware designs use the reconfigurable capability of FPGAs only by swapping in pre-compiled circuits from a library. Research at the University of Glasgow wants to exploit more advanced capabilities of FPGAs to tune circuits to have better performance at run time. The run-time reconfiguration needed for this is viable now with the Xilinx XC6200 series FPGAs, though future generations of FPGA chips are likely to offer similar capabilities. The XC6200 chip has SRAM reconfigurable cells, so that changing the implemented circuit's functionality is as simple as assigning to a variable in a software system.

Our scheme for run-time synthesis is designed for circuits which have both static and dynamic inputs. Imagine a decryption circuit which has two inputs: the key and the data to be decrypted. Suppose the key changes much less frequently than the data. When the system gets the key, the decryption circuit can profitably be specialized for that specific key. This is done by simplifying components using knowledge of the key value, and reconfiguring the FPGA accordingly. The system will then have a shorter critical path, and so will have a faster decryption process. The key value is known only at run-time, so the system can be specialized only at run-time as well. Our specialization method is adapted from the existing specialization ideas of run-time partial evaluation for software [7].

To ensure the correctness of the system, we can no longer rely on simulation based techniques, which take a long time to execute. Specialization occurs at run-time, and since the specialized circuitry is used immediately, there is certainly no time for simulation. Instead, formal verification will be used to establish the correctness of the specialization process itself. Our verification method is based on formal specification and proof in higher order logic.

The rest of this paper is structured as follows. A brief sketch of partial evaluation techniques for hardware is given in section 2. In section 3, the design flow for our use of FPGAs and the verification approach for this design flow are described. Section 4 explains our verification methodology. In section 5, the FPGA device model used and the embedding of the HDL language semantics are presented. Finally, our conclusions and some ideas for future work are given in section 6.

2 On-line Dynamic Hardware Synthesis

Partial Evaluation is a common technique in software compilations and executions [7]. The method reduces time resource requirements by exploiting the nature of some system inputs which are static for a long period relative to other inputs. Consider a function ***f*** that operates on some data. Suppose the input data can be divided into a static (known data) input ***s*** and a dynamic (unknown data) input ***d***. A partial evaluator is a function ***PE*** that is applied to the (source code for) the function ***f*** and the data value ***s*** to yield a residual function ***fs*** (eq 1). Moreover, ***fs*** has the same action on the dynamic data ***d*** as the generic function ***f*** (eq 2). As the result of partial evaluation, ***fs*** runs much faster for each ***d*** than the original function ***f***.

$$PE(f, s) = fs \qquad \textbf{(eq 1)}$$

$$[fs](d) = [f](s, d) \qquad \textbf{(eq 2)}$$

The basis of our work is the observation that partial evaluation can also be applied to circuit descriptions. Partial evaluation for hardware is done essentially by propagating known input values and specializing the corresponding gates. Similar techniques are known as data unfolding or constant propagation. The idea behind our work is to apply this technique to circuits developed for FPGAs, and to do the specialization at run-time. The specialization occurs by dynamically modifying the configuration data of Xilinx XC6200 chips. Hardware is different than software in the sense that hardware is normally static rather than dynamic. For example, the typical case of dynamic behavior under partial evaluation in software is unfolding of function calls to generate more program text. By contrast, a hardware circuit is always already fixed. Partial evaluation merely specializes the hardware to make some of the circuit disappear, rather than generating new circuitry.

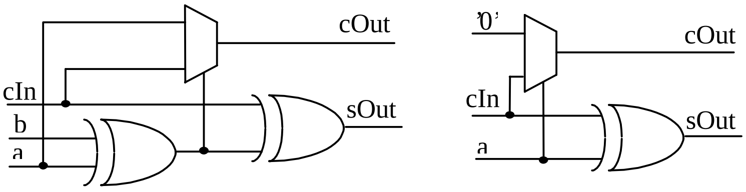
To illustrate partial evaluation, consider the following full-adder, described at the gate level by a Lava [11] function:

```
type FullAdder = ((bit,bit),bit) -> Out (bit,bit)
fa :: FullAdder
fa ((a,b), carryIn)
 = do partSum <- at (0,0) $ xor2 (a,b)
      sumOut <- at (1,0) $ xor2 (partSum, carryIn)
      carryOut <- at (1,1) $ mux2 (partSum, (a, carryIn))
   return (carryOut, sumOut)
```

The ‘at (x,y)’ and ‘$’ notations mean that the gate written after the ‘$’ is placed on the chip at a relative address indicated by the (x,y) coordinates. Suppose the first input ***a*** of the full-adder is known as a static input that will remain the same for many different values on ***b***. The circuit can then be specialized and simplified using this value of ***a***. For example, if the value of ***a*** is known to be zero, the XOR gate can be simplified to a buffer. The input signal ***b*** can directly be propagated to the next component. The overall system is then simplified to a two input function with two components:

```
specialised_fa (b, carryIn)
 = do sumOut <- at (1,0) $ xor2 (b, carryIn)
      carryOut <- at (1,1) $ mux2 (b, (0, carryIn))
   return (carryOut, sumOut)
```

Figure 1 shows the circuit diagram of both full-adders, the original Lava function description and the specialized one. The specialized full-adder has less delay, because it has one less gate along the critical path.



(a) (b)

Figure 1. Circuit diagram of full-adder (a) and specialized full-adder (b).

Another example of the idea is illustrated by the 5 by 6 bit parallel multiplier shown in figure 2. The dynamic input registers represented by ***a0*** upto ***a5*** come from the top of the figure, the static input registers represented by ***b0*** upto ***b4*** come from the right hand of the figure, and the output registers is on the bottom of the figure represented by ***s0*** upto ***s9***.

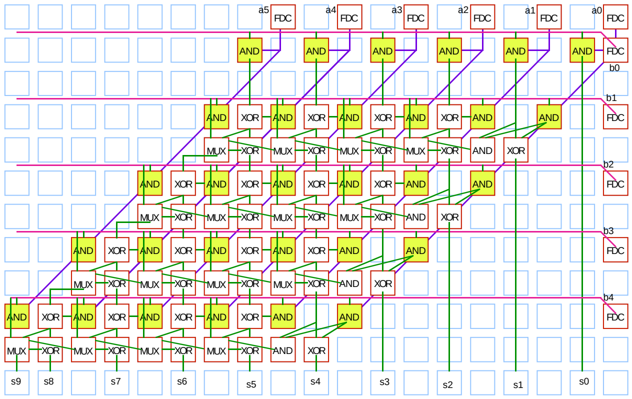


Figure 2. The shift add 5 by 6 multiplier circuit

Assume at run time the input ***b*** is going to have the known value of 6 for many iterations. The circuits can then be specialized and reduced to have only one addition operation and several long wires. The ***b*** register can disappear entirely (figure 3). The resulting circuit, then, can be operated at a higher speed than the original.

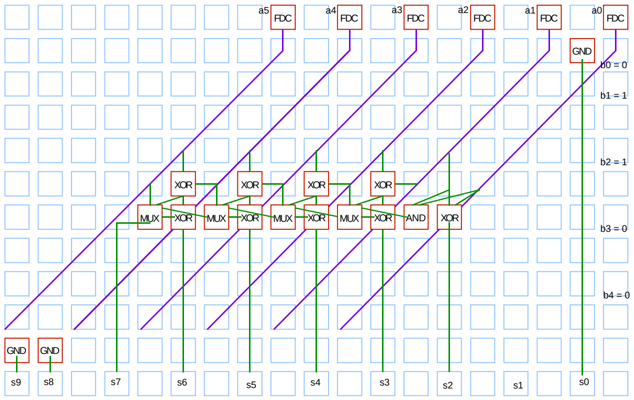


Figure 3. The specialized 5 by 6 multiplier circuit with the ***b*** value of 6

The essential requirement for on-line specialization is that the performance gained by specializing must, over the whole input data, offset the cost of doing the specialization. To illustrate the idea, consider a data stream consisting of some specialization parameters followed by n data items. There are 5 timing parameters to be considered for the specialized system and generic system schemes. For the specialization scheme, we have 3 parameters: the time to synthesize hardware for specialization (***Ts***), the programming time to reconfigure the FPGA (***Tp***), and the cycle time to process the data using the specialized circuit (***Tc***). The generic system has two parameters: the time to load the specialization parameter (***Tk***), and the cycle time to process the data using the generic circuit (***Tg***). The time needed to complete the whole process for both the specialized and generic circuit is presented in equations 3 and 4. A better performance can be achieved either when the number n of data items to be processed is sufficiently large so that the time needed to specialized the circuit (***Ts***+***Tp***) becomes relatively small or when the cost of specialization ***(Ts+Tp)*** over the whole input data is more efficient than the overall cost by the generic circuit.

$$\text{Specialized Circuit} = Ts + Tp + n\,Tc \qquad \textbf{(eq 3)}$$

$$\text{Generic circuit} = Tk + n\,Tg \qquad \textbf{(eq 4)}$$

3 Design Flow and Run-time Partial Evaluation

The Xilinx XC6200 series FPGAs have an SRAM reconfiguration facility which allows them to be dynamically reconfigured in a very short time. The most important feature of this chip is that each cell can be configured individually without having to reconfigure the entire chip. This means that it is possible to reconfigure part of the chip while in other parts of the chip some systems are still running.

A notable feature of FPGA design generally is that placement takes an important role owing to the limited connections between cells available on FPGA chips. Most FPGA system designers therefore already have some kind of placement patterns in mind for their designs. Unfortunately, most Hardware Description Languages (HDLs) do not accommodate this important information. The Department of Computing Science at Glasgow and the University of Chalmers therefore developed an HDL called Lava which incorporates this information [11]. The Lava language is developed as a library in the Haskell programming language; Haskell itself is a pure functional programming language [6].

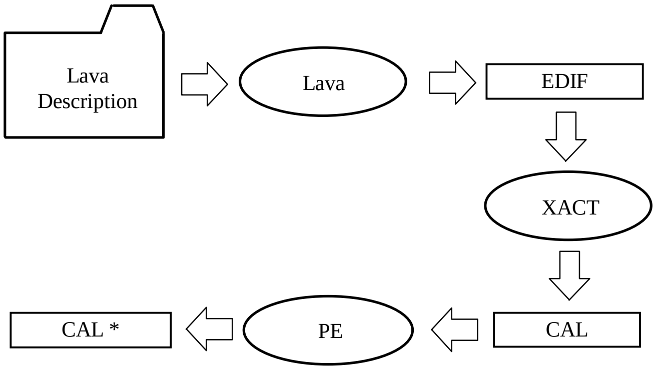


Figure 4. The design flows for the partial evaluation scheme

The overall circuit development environment (design flow) for the partial evaluation scheme is shown in figure 4. The circuit is developed using the Lava programming language. The Lava hardware description is then synthesized to produced a netlist format (EDIF) of the circuit. By using the Xact tools from Xilinx, the circuit at the netlist level is then placed and routed. This process produces a CAL (Configurable Array Logic) description, which consists of the configuration bits of each individual cell in the chip. All these design processes are done off-line. The on-line / run-time reconfiguration part takes place after the circuit is already placed in the chip and is in operation.

In the run-time part of the process, the circuit description in the form of a CAL file is downloaded into the chip's SRAM. At this stage, the chip is ready to perform the system's functionality. When the static input is detected, the partial evaluation program analyses the static value and specializes the circuit already placed on the chip. It immediately reconfigures the circuit by generating new configuration data which specializes the current circuit on the FPGA chip. As the result, the circuit on the chip will be simpler and faster than the generic one. Preliminary results of an experiment for run-time specialization based on a simple constant propagation scheme for partial evaluation are presented in [8].

4 Formal Hardware Verification Approach

The correctness of the overall system depends on the correct functional behavior of the specialized circuit with respect to its generic circuit source for given static values. Normally, the correctness of such a system can be investigated by simulation or by formal verification using equivalence checking or model checking. But in our highly dynamic setting, we may generate hardware and use it for only a few seconds before discarding it. Clearly there is no time for simulation or verification. Our approach is therefore to verify the correctness of the synthesis algorithm for the partial evaluation process. If the correctness of partial evaluation can be assured for any source circuit, then system correctness can be concluded. Of course, the result of partial evaluation will be correct only relative to the correctness of the original design. But this can be checked off-line.

The equation shown below states the correctness criterion in general terms. It is similar to software partial evaluation presented earlier. The specialized circuit will have the same behavior as the generic ones when applied with a specific static value.

$$[\![\text{circuit}]\!](\text{static},\text{dynamic}) \Leftrightarrow [\![\text{PE} (\text{circuit}, \text{static})]\!](\text{dynamic}) \qquad \textbf{(eq 5)}$$

Our verification method is as follows. The circuit at the CAL level is specialized by a partial evaluation algorithm written in C++. The result is a new circuit configuration, which our run-time system places on the FPGA. The partial evaluation part of this is formalized and verified in the PVS environment. The generic function unit that lies within each FPGA cell is modeled using the standard relational modeling style in higher order logic. The syntax and semantics of circuits at the netlist level is then embedded using the deep embedding method [2]. The embedding uses configured generic function unit models as the hardware model. The partial evaluation algorithm is then described abstractly as a PVS function and a proof of the correctness theorem above is done. The same process of specialization as is done by the actual C++ coded partial evaluator is carried out by the abstract representation of this algorithm in PVS. This results in a similar transformation of netlists as occurs at the CAL level (figure 5). Of course, the relationship between our PVS verification and the actual specialization code is only informal, but we aim to make it close enough to justify at least some confidence in the correctness of the actual code.

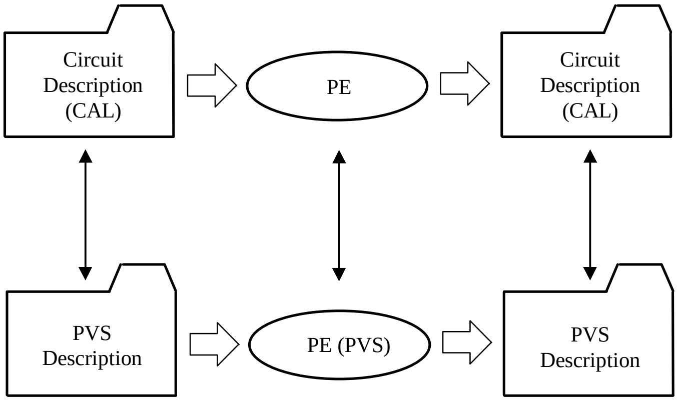


Figure 5. The partial evaluation of circuit at the CAL level and proof of the specialization scheme.

Partial evaluation at the CAL configuration level will eventually have two main components: reconfiguration of cell functionality, and routing reconfiguration. Routing reconfiguration is needed to exploit the connectivity resources of the XC6200 for larger speedups. Verification will cover both parts of the specialization process. At present, our work on verification has addressed only transformations of configuration of the generic function unit. The netlist deep embedding is currently under construction. A bit further in the future will come an extension of the verification to cover routing transformations.

The design flow in figure 4 includes Lava descriptions of the circuits. Preliminary work on a shallow embedding of the Lava language using functional hardware modeling highlighted two problems: Lava does not include routing information, and the Lava circuit model abstraction is too far from the CAL level, resulting a vague relation between algorithms at those two level. For these reasons, our work on formal verification is not be targeted at the high level Lava HDL level, but at the EDIF and CAL level.

5 Verification Models for Netlist HDL

5.1 Hardware Modeling

The Xilinx XC6200 series has a unique hierarchical architecture. The FPGA surface is organized as an array of simple cells. The contents of each simple cell is called a function unit (figure 6). A function unit has six components, five multiplexers and one D-type register. The function unit of each basic cell can be configured either as a logic function or a register by supplying some configuration bits for the multiplexers. Each cell is connected to its four borders (north, east, south, and west). Basic cells are grouped into 4x4 blocks of 16 cells. A 4x4 array of these 4x4 blocks of cells forms a 16x16 block. This hierarchical structure is repeated until 64x64 blocks or 256x256 blocks are formed, depending on the chip series. The whole structure is then surrounded with I/O pads. A similar scheme for routing appears within the hierarchical structure. A 4x4 cell block has its own associated routing resources, which provide fast interconnections. This interconnection capability is known as the length 4 fast wire. A similar routing hierarchical structure appears on the 16x16 block, the 64x64 block, and the 256x 256 block. In addition, the FPGA series we use introduces a special fast wire called a magic wire. A magic wire has the capability to route signals from individual cells to certain points on its 4x4 block border.

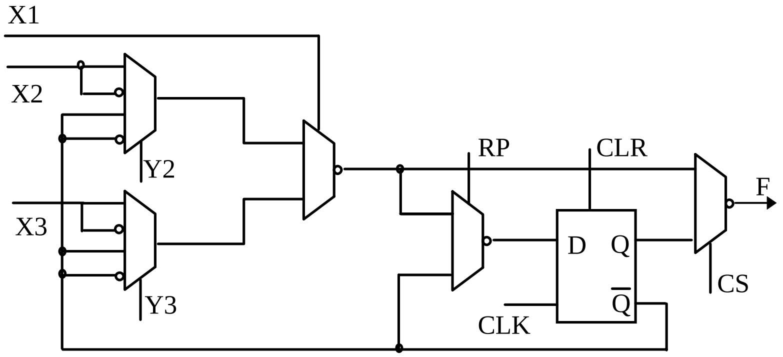


Figure 6. Function unit circuit diagram of the FPGAs XC6200

In PVS, the behavior of the function unit of the basic cell is specified using the relational modeling style. The function unit can be developed by using 3 basic components: the inverter, the two input multiplexer, and the D-type flip-flop. All other components in the function unit can be developed by using this three basic components. One important feature in the model is that every component has time varying inputs and outputs. The formal model employs the usual notion of signal, which is a boolean valued function taking discrete time arguments.

The three basic specifications are shown in the PVS theory in figure 7. The inverter ***inv*** has two ports, a single input ***i0*** and output ***o***. The multiplexer ***mux2*** has four ports, three inputs ***i0, i1, sel*** and a single output ***o***. These two component models are based on the zero time delay assumption. The D-type flip-flop model is implemented using a unit time delay. The system samples the input value ***(din)*** when the clock rises ***(rclk)*** and holds the value on the output **q** until the next rise of the clock. The second output ***(qnot)*** is an inverted output function of ***q***. A more detail explanation of abstract time modelling can be found in [9].

```
i0,i1,sel0,o,din,q,qnot,clk,clr: VAR signal[bool]
t : VAR time
inv(i0,o) = ∀t. o(t) = ¬ i0(t)
mux2(i0,i1,sel0,o) = ∀t. o(t) = (sel0(t) ⇒ i1(t) | i0(t) )
rclk (clk,t) = ¬clk(t) ^ clk(t+1)
rdff(din,clk,clr,q,qnot) =
   (∀t. q(t+1) = (clr(t+1) ⇒ ( c_rclk(clk,t) ⇒ din(t) | q(t) ) | FALSE ) ^
   (∀t. qnot(t) = ¬q(t))
```

Figure 7. The basic relational specifications:
inverter, two input multiplexer and D-type register.

The FPGA function unit relational model is presented in figure 8. The formal implementation description is simply a direct transcription into logic of the circuit diagram on figure 7 (which itself is given in the Xilinx data sheets) [13]. The variables ***x1*** up to ***f*** are the external I/O ports of the function unit. The variables ***sy2_0*** up to ***cs*** are the configuration bits which determine the external behavior of the function unit. The internal interconnection of the system is hidden by the standard method of existential quantification. A more detailed explanation of hardware modeling in PVS is presented in [12].

```
% external I/O ports
   x1,x2,x3,clk,clr,f : VAR signal[bool]
% configuration bits
   sy2_0,sy2_1,sy3_0,sy3_1,rp,cs : VAR signal[bool]
funit (x1,x2,x3,clk,clr,f,sy2_0,sy2_1,sy3_0,sy3_1,rp,cs) =
 ∃(Y2,Y3,RPM,C,S,buf1,buf2,buf3,qnot).
   ∀t. inv(x2,buf1) ^
       inv(x3,buf2) ^
       inv(qnot,buf3) ^
       mux4(x2,qnot,buf1,buf3,sy2_0,sy2_1,Y2) ^
       mux4(x3,buf2,buf3,qnot,sy3_0,sy3_1,Y3) ^
       muxn2(Y3,Y2,x1,C) ^
       muxn2(C,qnot,rp,RPM) ^
       rdff(RPM,clk,clr,S,qnot) ^
       muxn2(S,C,cs,f)
```

Figure 8. The XC6200 FPGA function unit model in PVS

As an example of verification using this model, consider an AND gate implemented by setting configuration bits (figure 9). The signals *sig_zero* and *sig_one* are the *ground* and *vcc* sources respectively. These two signals are used to model the SRAM configuration bits, which have the same behaviour as signal sources. Only some of the input configuration bits are needed to configure the function unit. The unused configuration bits can be ignored by simply existentially quantifying them. Finally, the configured function unit model can be proved correct with respect to a high level behavioural specification and2_spec. At present, 14 possible configurations have been verified in our PVS theory.

```
and2(in1,in2,out1) =
 ∃ (buf2,buf3,sg0,sg1,clk1,clr1).
   ∀t. sig_zero(sg0) ^
       sig_one(sg1) ^
       funit(in1,in2,in1,clk1,clr1,buf3,sg0,sg1,sg1,sg0,buf2,sg1) ^
       inv(buf3,out1)

and2_spec(in1,in2,out1) = ∀t. out1(t) = in1(t) ^ in2(t)

tm_and2: THEOREM and2(in1,in2,out1) = and2_spec(in1,in2,out1)
```

Figure 9. Functional behaviour modeling based on the function unit and proof

5.2 Sematics Embedding Approach

In the design flow in figure 4, the Lava program synthesizes a structural circuit description and produces a flattened description of the circuit in the EDIF Version 2.0.0 syntax. The netlist describes the components as simple gates and models the interconnection between components. In our verification, circuits at the netlist level will be specified in higher order logic notation using the deep embedding methodology. As already mentioned, the choice of a deep embedding of netlists was driven by previous experienced with a shallow embedding of the Lava HDL semantics. The shallow embedding limited the proofs to functional properties. Furthermore, the vague correspondence between the HDL description and what actually happen on the chip gave a less useful verification result.

The netlist syntax we will embed in PVS is presented in figure 10. A netlist description contains a cell library which consists of a collection of cells. The circuit as a whole is also part of the library and is a cell constructed from the predefined basic cells in the library. The Lava netlist generator generates circuits as a single flattened cell. Within the cell, the circuit is described as components and their interconnection relations. This structure at the netlist level is maintained in the higher order logic hardware model. The netlist syntax follows the standard EDIF format, which makes the language well structured. The deep embedding semantics will be implemented as a function which takes a circuit in the netlist form as its argument and produces a circuit in the relational form as a result.

```
cell_library ::=  library_name cell_name [interface]*
                  [instances]*? [net]*?
interface ::=  interface_name [direction]
direction ::= INPUT | OUTPUT | INOUT
instance ::= instance_name cell_name cell_location
net ::= net_name [pin]*
pin ::= interface_name  instance_name
library_name, cell_name, cell_location, interface_name,
  instance_name, net_name ::= identifier
identifier ::= [A-Za-z][A-Za-z0-9_]*
```

Figure 10. Simplified netlist syntax from Lava netlist generator. The optional items are followed by '?', and the repeted items are followed by '*'.

```
C++ :
if (fn == INV)
{ if (input == cell[i][j].input_a)
    { if (value_a == ZERO)
        { cell[i][j].cell_function = ONE;
          setFunction(i, j, ONE, cell[i][j].input_a);
          return ONE;}

PVS :
test_inv_low  : THEOREM cell_inv(0) = cell_high
```

Figure 11. Transformation modelling from C++ to PVS

The run-time partial evaluation algorithm is implemented in the C++ language. The algorithm uses a constant propagation scheme by simply propagated the static value and specializing all the corresponding cells into a simpler cell. Consider an example, an ***inv*** gate cell. If the input of the inverter has a static value of 0, then the cell can be specialized to a vcc source (figure 11). All the possible cell transformations have been implemented in PVS and proved correct (52 cell transformations have been verified). The next step is to develop a high level abstraction of the algorithm which captures all possible circuit netlist transformations in PVS. We will then prove that the partial evaluation function applied to the circuit netlist and the static values will result a specialized circuit netlist whose semantics agrees with that of the original (eq 5).

6 Summary and Future Work

Run-time circuit specialization poses an interesting challenge for verification of the specialization algorithm. Our approach is based on two aspects: hardware component modeling, and semantic embedding of the circuit description language. The problem is addressed in two hierarchical steps, which reflect the design cycles: proving the system correctness at the netlist level and then extending the system with the routing information present in the CAL level. The hardware model at the netlist level is based on a generic function-unit. This function-unit can be configured to perform a certain functional behavior, either to be as a logic function or a register. At the current stages of our work, 14 configurations of basic cell functionality and 52 cell transformations have been verified. The model then will be used in verifying the specialization transformation algorithm based on a high level abstraction of the implemented C++ algorithm. We will do this by developing a deep embedding of a netlist semantics and defining a high level specialization algorithm in the PVS environment.

Acknowledgments

This work is part of a project funded by UK-EPSRC (project reference GR/L38530), the United Kingdom Ministry of Defence, and Xilinx Inc. It is managed by Tom Melham (University of Glasgow), Satnam Singh (Xilinx Inc.), and Derek McAuley (Microsoft Research Labs, Cambridge). The run-time partial evaluation system is developed by Nicholas McKay.

The authors thank the SRI team for making the PVS system available to perform the research presented in this paper. We also thank Richard Boulton for providing us with an unpublished paper on embedding netlists in higher order logic [1], and Paul Jackson for valuable help with PVS.

References

[1] Richard Boulton. ‘A Semantics for a Simple Netlist Language’. Unpublished paper.

[2] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. ‘Experience with Embedding Hardware Description Languages in HOL’, in *Theorem Provers in Circuit Design: Theory, Practice and Experience: Proceedings of the IFIP WG10.2 International Conference, Nijmegen, June 1992*, edited by V. Stavridou, T.F. Melham, and R.T. Boute, North-Holland, 1992, pp:

129-156.

[3] Jim Burns, Adam Donlin, Jonathan Hogg, Satnam Singh, and Mark de Wit. ‘A Dynamic Reconfiguration Run-Time System’ in *IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California, April 1997* : Preliminary Proceedings, pp: 82-91.

[4] Charles Consel and Oliver Danvy. ‘Tutorial Notes on Partial Evaluation’, In *ACM Symposium on Principles of Programming languages, 1993*, pp: 493-501.

[5] Patrick W. Foulk. ‘Data-folding in SRAM configurable FPGAs’, in *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, California, April 1993, pp:163-171.

[6] P. Hudak, J. Fasel, and J. Peterson. ‘A Gentle Introduction to Haskell’, Technical Report YALEU/DCS/RR-901, Yale University, May 1996.

[7] Niel D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*, Prentice Hall International, 1993.

[8] Nicholas McKay and Satnam Singh. ‘Dynamic Specialisation of XC6200 FPGAs by Partial Evaluation’. to appear in proceedings of Field Programmable Logic and Applications ’98 Workshop.

[9] T.F. Melham, *Higher Order Logic and Hardware Verification*, Cambridge University Press, 1993.

[10] Satnam Singh, Jonathan Hogg, and Derek McAuley. ‘Expressing Dynamic Reconfiguration by Partial Evaluation’, in *IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California, April 1996* : Proceedings, edited by Kenneth L.Pocek and Jeffrey Arnold, IEEE Computer Society Press, 1996, pp: 188-194.

[11] Satnam Singh and Mary Sheeran. ‘Designing FPGA Circuits in Lava’, unpublished paper.

[12] Madayam Srivas, Harald Rueβ, and David Cyrluk. ‘Hardware Verification using PVS’ in *Formal Hardware Verification Methods and Systems in Comparison*, edited by Thomas Kropf , July 1997, Springer Verlag LNCS 1287, pp: 156-205.

[13] Xilinx. ‘*XC6200 FPGA Family Data Sheet*’. Xilinx Inc., 1995.