Automated Hardware Synthesis from Formal Specification using SAT solvers.

David Greaves - Univ of Cambridge* / Tenison EDA

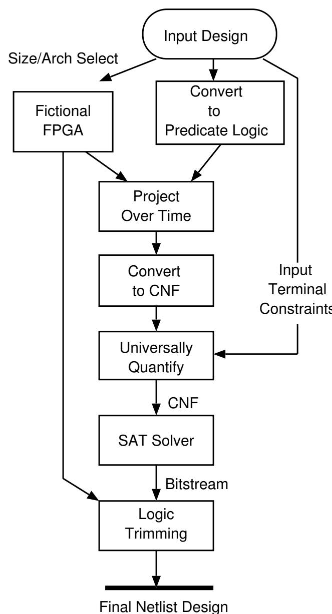

Figure 1. Flow for Logic Synthesis from Formal Design using SAT.

Abstract

System and circuit design can be considered as planning problems, where resources are deployed in time and space to meet a given goal. Recent and continuing developments in the size of SAT problems and other AR problems that can be solved with off-the-shelf tools leads us to consider their direct use in system design. In this paper we start to tackle the design of small hardware subsystems and the generation of glue logic between systems by asking a SAT solver to generate the programming bit stream for a fictional gate array.

1 Introduction

Automatic synthesis of systems and circuits from formal constraint(s) has been considered an attractive alternative to conventional implementations for various reasons, including

- There can be an accuracy benefit from using formal rules,
- Often the detailed implementation does not matter provided it meets generalised interface and functional requirements,
- Both the sending and receiving halves of any interface can be generated from a single specification,
- Specifications in the form of design predicates are composable in any order allowing, in principle, for simple aggregation of design considerations,
- The automated system may be able to do a better job.

Automated refinement from a formal specification has been advocated as one approach [1] and many other seemingly different approaches, such as elaboration of regular expressions are broadly similar, being largely syntax directed. However, a fundamentally different approach is enabled by the recent development in the size of problems that may be tackled with SAT solvers. For instance, problems of thousands of clauses in hundreds of variables are regularly solved at the international SAT solving competitions[2]. If we provide sufficient programmable resources to solve a problem, the SAT solver can generate a programming that implements a solution to the problem. This is exploitable both for hardware and for software synthesis, provided the problem and the possible solutions can be phrased as logic functions. In our work, we have been looking at direct generation of asynchronous logic, system interconnections, protocol designs and machine code for software subroutines using SAT. Here, we present an example of one basic approach to designing a pair of synchronous finite state machines that use a protocol 'of their own choosing' to communicate over a constrained channel. The example is partial

*David.Greaves@djg@cl.cam.ac.uk

in that we are still working on an automated solution to the system start-up.

2 Approach

There are many ways of using a SAT solver to generate hardware designs, but one of the most simple is to postulate a pseudo-FPGA architecture and allow the SAT solver to generate the programming bitstream. In the basic case, the problem to be solved is phrased as a formal logic function and the behaviour of the FPGA under a given programming is expressed similarly. Then a bi-equivalence equation can be solved by a SAT solver. Un-used gates in the FPGA are then removed using standard identities to trim the solution for ASIC or other backend flows. Certain SAT solvers can return dont-care variable settings and these are also an indication of where to trim. Finally, we can select from a number of solutions either by refining the query or selecting from the generated solutions using cost evaluations (e.g. lines of code or gate count). If we provide too-few gates, SAT will fail, and if we provide too many, some may be trimmed by the logic simplification identities or because they lead to no output terminal, but other gates may be wired up in vain. This waste can be minimised by running our process again and again on successively smaller arrays and stopping just before no solution is found.

Regarding design entry, we are not advocating that logic programming should be the only form used in the future. RTL and behavioural expression using threads are well-loved forms, as is structural instantiation of modules (or objects), which is also vital for module reuse and other engineering aspects. In our current approach, owing to growth complexity, we envisage we shall still have to rely heavily on manual partition into separately instantiated modules. We fully support RTL for expressing parts of a design since, in our method, the FPGA is expressed as RTL assignments. Normal elaboration of Verilog RTL constructs [3] such as '`if/then/else`' and '`case`' is unaltered (but currently we only have one clock domain). Further, the recent maturity of formal semantics of programming languages, such as C and Verilog, provides a path to convert most non-RTL forms of behavioural expression into logic programming form, although there is the potential penalty of loss of ingenuity incorporated in the behavioural expression.

3 Details

The basic component of the FPGA is the LUT (look-up-table). We define LUTs using an XFUN macro

$$XFUN(\text{``}xid\text{``}, s_0, s_1, ..., s_{n-1})$$

where the first argument is an user-provided unique identifier and the remaining args are the input net names (n input net names results in an array of 2^n programmable points, or '*fuses*'). The macro expands into a CNF expression of a one-bit wide ROM. For example

$$XFUN(\text{``}xa\text{``}, s_0) := xa0\& \ s_0|xa1\&s_0$$

The variables begining with the user's identifier will remain undriven and be given constant values during SAT. We currently have a convention that all variables beginning with the letter 'x' are fuses, i.e. constants whose values are to be found by SAT.[1]

Our input language supports essentially two core constructs, an assertion, which is a predicate expression, or an RTL definition of the form 'L := R' where L is a single variable name. So that they are always respected in the resulting system, the definitions are internally converted to assertions by replacing the ':=' with a biequivalence '==' but they are also used directly in the expansion of future states as explained below. Finally, when the SAT solution is returned, it is plugged into the definitions to generate the output hardware design.

In this paper we are just designing synchronous finite state machines that use a global clock but there is still considerable flexibility over the structure of FPGA we provide. The wiring interconnection pattern is not restricted by 2-D layout restrictions of real FPGAs, and so could be some sort of n-dimensional hypercube. We use the notation $X(s)$ to denote the output of a D-type flip-flop whose input is 's' and whose clock is the global clock. A fully 'crossbar' FPGA can be expressed as follows:

$$\begin{aligned}
X(s_0) &:= XFUN(\text{``}xs1\text{``}, s_0, s_1, ..., s_{n-1}, D); \\
X(s_1) &:= XFUN(\text{``}xs2\text{``}, s_0, s_1, ..., s_{n-1}, D); \\
... &:= ... \\
X(s_{n-1}) &:= XFUN(\text{``}xsn\text{``}, s_0, s_1, ..., s_{n-1}, D);
\end{aligned}$$

but restricted interconnection with global and local nets is just as expressible and usable. The external inputs, denoted with a vector D, are any free variables: i.e. those not beginning with 'x' and not bound by one of the definitions. The state variables and all expressions can also be input in a vector form and internally expanded to individual nets. Outputs can be taken from any variable.

[1] Certain SAT solvers, especially those based on BDDs, can be asked to prefer solutions where variables are preferentially set to some value, e.g. zero. It is preferable to link this setting with the definition of XFUN so that the SAT solver 'generates' less logic. Equally, some SAT solvers can generate dont-cares in their output and these can be used for logic minimisation in the final output.

3.1 Design Specifications

Any number of predicate calculus rules can be used to express the design. We have implemented a syntactically rich source language [9] and an expander to predicate calculus (that is then expanded to CNF as presented herein). It is also possible to express using direct RTL programming since this is elaborated down to the same form as used in the FPGA definition, but clearly without any SAT fuses in the expressions.

Predicate rules may refer to any variable, v, meaning its value at a current nominal time, or X(v), meaning its value after the next clock edge. Also available is $X^n(v)$ (typed $X(v,n)$) where n is a small integer, denoting clock cycles into the future. We support this by iteratively, symbolically using the ':=' definitions for the supporting variables. For example $X^i(a|b)$ expands to $X^i(a)|X^i(b)$ and $X^{i+1}(s_1)$ expands to

$$XFUN("xs2", X^i(s_0), X^i(s_1), .., .X^i(s_{n-1}), X^i(D));$$

Input variables intrinsically do not possess next state functions available to be used in the expansion of $X^i(D)$, and, in general, inputs may change from one clock cycle to the next, so new net internal names are created for inputs to cover each referenced future timestep. These become further free variables since they are undriven.

As shown in figure 1, once the FPGA and the target design have been converted to predicate calculus, we expand these into a Boolean expression and then into CNF clause form for SAT solving. Rather than expanding out the design fully into CNF, which can generate an exponential number of clauses, a standard trick is deployed: where a disjunction falls below a conjunction an additional SAT variable is used as an intermediate. This generates three clauses from a nested binary conjunction (l.r):

$$(nv + !l + !r)(!nv + l)(!nv + r)$$

Note that the first of these three clause can be dropped without altering sat-ness [7] and that the new variable, nv, has to be uniquified (skolemised) during universal quantification (described below).

3.2 Universal Quantification and SAT solving

The design has a number of free variables which, unless we constrain further, are considered as universally quantified. This means the SAT solution must be for all possible settings thereof. The free variables are the state variables at the nominal reference time, and the external inputs at the nominal time and at all future referenced times. Of course, SAT is targeted at existential problems, so we need a process for explicit universal quantification. In essence, we need to write out all of the clauses for each possible setting of the free variables.

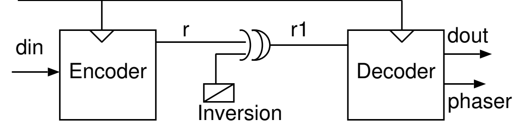

Figure 2. Structure of MFM encoder and decoder demo (clock recovery unit ignored).

It is desirable to reduce the rate of exponential growth during the this phase. A first approach to this can be used when we know that certain input conditions will never occur. Therefore we support a second set of predicates that express things the design specifically does not have to achieve. We do not feed these in to the earlier phase because constraining the free variables with assertions while at the same time as universally quantifying them leads to an instantly invalid design.

To further reduce growth during quantification, when we can group the clauses into sections with totally disjoint support, we can use the identity

$$\forall x, y.(P(x) \wedge Q(y)) \equiv (\forall x.P(x) \wedge \forall y.Q(y))$$

to quantify each section separately. Better still would be to have a SAT solver that accepts variables classed into two partitions: variables in the first partition are solved for, as usual, whereas the solutions found must be valid for all settings of the second partition.

4 Two Examples.

In a first example, we requested a design for a coder and a decoder that implement a serial data modulation format rather similar to Manchester or MFM, as used on magnetic disks. Figure 2 shows the basic setup where every other clock cycle, data from net 'din' is transferred over the channel 'r' to arrive at 'dout'. For the encoder a 'crossbar' FPGA with four internal states was selected. One state was also the output net 'r'.

We impose the main restriction on the encoded function: a run length limitation to a maximum of three consecutive zeros. This is expressed as

$$!r \Rightarrow (X(r) \vee X(r,2) \vee X(r,3));$$

meaning that whenever r did not occur, it will within the next three periods.

For the receiver, we assume a single LUT that operates like an FIR on older copies of the input is sufficient, so we write

$$X(dout) \quad := \quad XFUN("xd", r, X(r), X^2(r), X^3(r));$$

$$X(qual) \;:=\; XFUN(\text{"}xq\text{"}, r, X(r), qual);$$

Such an implementation will demodulate input data and also limit error propagation and latency.

Overall, we need a data conservation property: every other input bit appears on the output after some latency. (As in all MFM codecs, the intermediate input bits can be ignored by the system.) We first define this by implementing a toggler that alternates on each clock cycle

$$X(phaser) \;:=\; XFUN(\text{"}xphix\text{"}, phaser);$$
$$X(phaser) \;==\; !phaser;$$

although this could perhaps more simply have been written using the direct RTL style

$$X(phaser) \;:=\; !phaser;$$

We can now write the overarching system specification in terms of a data conservation rule: one of the following should hold:

$$M_0 \;:=\; din == dout;$$
$$M_1 \;:=\; din == X(dout);$$
$$M_2 \;:=\; din == X^2(dout);$$
$$M_3 \;:=\; din == X^3(dout);$$
$$M_4 \;:=\; din == X^4(dout);$$

The following attempt captures the basic idea that the output data will be the input data, every other bit, after some unspecified delay, but, in this phrasing, the delay is not constrained to be constant from one clock time to another

$$(phaser \vee M_0 \vee M_1 \vee M_2 \vee M_3 \vee M_4);$$

Instead, we allow the SAT solver to choose the constant delay by forcing exactly one of m0, m1, ... m4 to be true using a binary decoding of some further SAT fuses.

$$m0 \;:=\; !xm0 \;\&\; !xm1 \;\&\; !xm2;$$
$$m1 \;:=\; xm0 \;\&\; !xm1 \;\&\; !xm2;$$
$$\;:=\; ...$$
$$m4 \;:=\; !xm0 \;\&\; !xm1 \;\&\; xm2;$$

$$(phaser \vee M_0 \;\&\; m0 \vee M_1 \;\&\; m1 ... \vee M_4 \;\&\; m4);$$

It would be helpful to introduce in our source language an operator that macro expands in this manner.

Additionally, it is necessary to force the modulation scheme to be insensitive to inversions in the channel. Many forms of real channel are subject to this perturbation. To do this, we introduce one further free variable and an exclusive OR gate, as shown in figure 2, and refer to 'r1' in the receiver module.

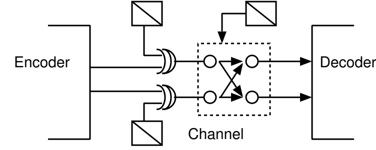

Figure 3. Two Rail Signalling.

$$r1 \;:=\; r \otimes \text{Inversion}$$

Since 'Inversion' is not driven anywhere, the universal quantifier will require a design that operates whether the channel is inverted or not. However, we do not require a design where the channel can change polarity every clock cycle: such a channel would have unity signal to noise ratio and no information capacity! Hence we add the rule:

$$X(\text{Inversion}) == \text{Inversion};$$

4.1 Initial Condition Issues.

It is well known that there are many hardware solutions to the above-phrased problem. They vary in ones-density, disparity control and transition density. For the given conditions, biphase mark Manchester encoding[6] will work as well as standard MFM. This, we demonstrated by using a direct RTL implementations of biphase mark and MFM as our input instead of using the XFUNs.[2] However these codecs all rely on a start-up period during which the not-yet-synchronised receiver may make errors. Hence we could only demonstrate the RTL MFM solution when accompanied with a side condition that forced it to be initially synchronised. The resultant circuits were dumped as a Verilog netlist and an example resultant schematic is shown in figure 6.

Neither SAT solver we tried could find a solution to the problem without the side condition since the presented phrasing does not allow a start-up period and we believe that the solvers have stumbled on a reality: there is no solution that does not have a start-up transient.

4.2 Example 2: Two Rail Coder

The Cambridge Ring [10] and the Inmos Transputer network both used a pair of binary signals between nodes, but were tolerant to the wires being inverted and/or crossed.

[2] When presented with a fully RTL, direct implementation, our tool acts simply as an assertion checker on a hardware design.

```
//--------------------------
// Transmitter encoder
// Encoder state
  node bool: s0, tx1, tx2;

   X(tx1) := XFUN("xtx1", din, tx1, s0);
   X(tx2) := XFUN("xtx2", din, tx2, s0);
   X(s0) := XFUN("xtxs", din, tx1, tx2, s0);

 // Transmitter Constraints
 // Unsuprisingly, this fails if we request an AND in the next line.
 always tx1 != X(tx1) \/ tx2 != X(tx2);

//--------------------------
// Channel Model
// This two wire channel may be permanently
// swapped and/or inverted in one half or the other.
 node bool: rx1, rx2, ch_inv1, ch_inv2, ch_swap;

 X(ch_inv1) := ch_inv1;  // These assigns mean that
 X(ch_inv2) := ch_inv2;  // the initial value of this variable is not known
 X(ch_swap) := ch_swap;  // but that it will not change.

 rx1 := ((ch_swap) ? tx1:tx2) ^ ch_inv1;
 rx2 := ((ch_swap) ? tx2:tx1) ^ ch_inv2;

 //--------------------------
 // Receiver decoder
 node bool: srx;
 X(dout) := XFUN("xDOUT", rx1, rx2, dout, srx);
 X(srx)  := XFUN("xSRX", rx1, rx2, dout, srx);

 node bool: match0, match1, match2, match3, match4, working;

 match1 := X(dout,1) == din;
 match2 := X(dout,2) == din;
 match3 := X(dout,3) == din;

 // By selecting a value for xdel, the tool can decide how much pipeline delay
 // in the coder+decoder.
 node bool: xdel0, xdel1;

 working := (xdel1) ? match3: (xdel0) ? match1 : match2;
 always working;
```

Figure 4. Source Code for the Two Rail Coder.

The seven possible perturbations to the channel can be expressed by adding three free variables as shown in figure 3. Using the code of figure 4, the tool designed a protocol for this with the constraint that one or other channel is changed each clock cycle. This task has no start-up transient problem. One of the resulting circuits has been manually transcribed in figure 5.

5 Results and Growth Rates

A complete hardware synthesis system has been implemented in SML on a 1GHz Linux system. It takes several minutes to convert a source file of the MFM codec complexity, i.e. with about 200 fuses, to clause form. Such a design produces SAT problems of about 20000 clauses of 10000 literals. Nearly all of the SAT literals are the intermediate SAT nets generated during the conversion to CNF and the remainder are the FPGA fuses. We have tried two SAT solvers, Chaff (Version Spelt 3) [4] and Walksat [5]. Both SAT solvers have always returned answers in a second or two. Considering that a proportion of this time must be used to handle a megabyte or so of file I/O rather than SAT solving proper, we conclude we must be creating rather easy SAT problems compared with the normal benchmarks.

If ten thousand clauses is a good upper limit to feed to a contemporary SAT solver, we can estimate the maximum

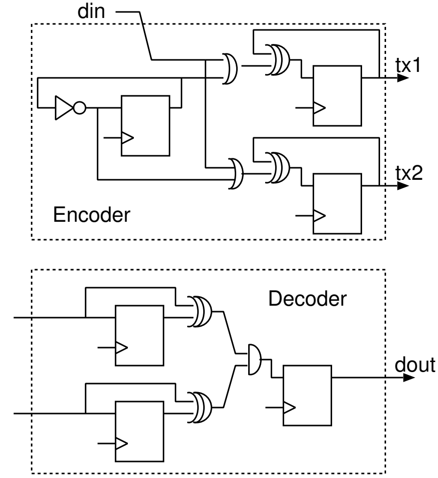

Figure 5. Two Rail Signalling Circuit.

design that can be handled by our approach. We reckon on a factor of five exponential growth from each additional power of X delay and a near doubling of size for each free variable, so a relatively small system with four levels of pipeline and eight universally quantified variables would be above the limit

$$1.5^8 \times 5^4 = 16000$$

but this problem warrants much greater study and ingenuity in its solution.

In all systems, for engineering flow purposes, functionality is encapsulated in component modules or objects. In our system, the very same predicates that describe the behaviour of a component can be used both when instantiating the component in surrounding context, to help synthesise the glue logic, and in the synthesis of the component implementation itself. Predicates that relate to the internal components of a module do not need to be exported, thus providing a significant approach to increasing the capacity of the system. Unfortunately, this approach cannot be applied to our protocol design example, because this currently relies on a full flattening of the coder along with the channel and decoder.

6 Conclusions

In this paper, we have presented the current state of a method for automatic hardware synthesis and protocol design. The system is implemented and being developed further. We allow users to program directly in a variety of styles, while leaving the system to fill in required functionality that is not explicitly expressed. Also, the user can make formal assertions about the parts of the system they have specified structurally or behaviourally and to be informed of violations.

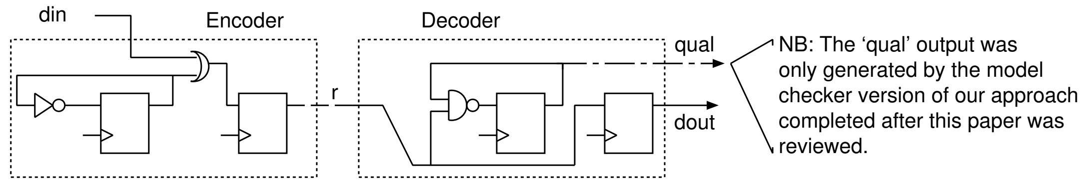

Figure 6. An example MFM coder.

However, the system currently has three shortcomings:

1. There is just one significant error message: 'unsatisfiable', and no feedback as to where or why,

2. A whole additional mechanism really needs to be added to encapsulate the concepts of start up or reset,

3. The depth of design pipeline that can be handled in one run is limited smaller than ideally desired.

The selection of an appropriate pseudo-FPGA architecture is also currently *ad hoc*, especially when we realise that for the approach to be useful to real engineers, we need to have further clock domains and instantiate RAMs, ALUs and even microprocessors in the target hardware, if it is to be a comprehensive solution to today's design tasks.

The main challenge of converting a high-level specification of the problem to a logic function seems to be the inevitable exponential blow-up when projecting through the next-state function and then again when performing universal quantification.

To handle the start up problem, we are now investigating solving the SAT-generated solution only for the reachable states encountered using a symbolic model checker such as SMV [8]. A BDD-based model checker also avoids taking the penalty of conversion to CNF and hopefully allow us to target reasonable-sized problems (by which we mean a task of comparable size to that typically handled in a run of Synopsys Design Compiler).[3]

This is the additional code, using an initial statement to the model checker, that implements a two cycle start up guard:

```
// Reset Logic
    node bool: reset_0, reset_1, resetting;
    initial reset_0 == 1 && reset_1 == 1;
    X(reset_0) := 0;
    X(reset_1) := reset_0;
    resetting := reset_0 | reset_1;
    always !resetting => (phaser != qual);
```

[3]Since this paper was written, the a model checker approach has been tried out and it appears promising. It has cleanly solved the MFM start up problem. The model checker allows an initial true value to be specified for a 'reset' signal, and for reset to be clear thenceforth, or after some number of start up cycles. Although the reset signal does not need to appear in the generated logic, it can be used as a guard for the main system constraints, so they need not be true at start of day. The model checker generates an expression, $Q(s)$ for the reachable states. The synthesised output is a SAT solution of $\forall s.Q(s) \Rightarrow C(s)$

Also, for a tool flow to be practical, there needs to be consistency and reuse of solutions from one run to another. Otherwise, the whole of a design might change as a result of a very minor change. We need to solve that too.

References

[1] Stepwise refinement of action systems.' BACK, R.-J. AND SERE, K. 1991. Struct. Program. 12, 17-30.

[2] 'SATLIB: An Online Resource for Research on SAT' Holger H. Hoos and Thomas Stutzel. In: I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000, pp.283-292, IOS Press, 2 000. SATLIB is available online at www.satlib.org.

[3] "The Verilog Hardware Description Language"Donald E.Thomas and Philip Moorby. Published by Kluwer Academic Publishers. ISBN 0–7923-9126-8.

[4] 'Chaff: Engineering an Efficient SAT Solver' M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, 39th Design Automation Conference, Las Vegas, June 2001.

[5] 'Walksat: Stochastic Local Search for Satisfiability' B Selman, H Kautz. Download form www.cs.washington.edu/homes/kautz/walksat

[6] J Strother Moore 'A Formal Model of Asynchronous Communication and Its Use in Mechanically Verifying a Biphase Mark Protocol' In Formal Aspects of Computing, 6(1) 1994, pp. 60-91.

[7] 'A StructurePreserving Clause Form Translation.' D. A. Plaisted and S. A. Greenbaum. Journal of Symbolic Computation, 2(3):293–304, September 1986.

[8] 'Symbolic Model Checking: An Approach to the State Explosion Problem.' K. McMillan. Technical Report CMU–CS– 92–131, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1992. PhD Thesis.

[9] 'The HPRLS Project: Logic and Embedded System Synthesis' DJ Greaves. www.cl.cam.ac.uk/users/djg/wwwhpr

[10] 'The Cambridge Ring- A Local Network', A. Hopper (1980) in Advanced Techniques for Microprocessor Systems, F. K. Hanna, Ed., Stevenage, United Kingdom: Peter Pergrinus.