A correct-by-construction methodology for designing symmetric circuits

Ashish Darbari

Oxford University Computing Lab
Wolfson Building
Parks Road
Oxford, OX1 3QD
ashish@comlab.ox.ac.uk

1 Introduction

Symbolic trajectory evaluation [1] or STE in short has been successfully used in verification of large industrial sized circuit designs [2]. However, the data abstraction in STE via Xs is often insufficient to bring about any reasonable reduction in the size of the verification of memory based circuits such as random access memories (RAM), content addressable memories (CAM) and caches. Memory based circuits offer a significant opportunity in achieving a reduction in the size of the verification problem due to the inherent symmetry in their structure. Our overall research goal is to develop a symmetry based reduction methodology for STE model checking.

Two key problems have to be addressed in any symmetry based reduction approach for model checking hardware. These are:

1. How can symmetries in circuits be found?
2. How can the symmetry identified in a given circuit lead to a reduction in the size of the verification problem?

This paper addresses the first problem, by presenting a correct-by-construction method of recording and identifying symmetries using an abstract data type for circuit models. In a recent paper [3] we have presented a solution to the second problem by providing a reduction strategy for STE properties based on having identified symmetries using the approach we present in this paper.

2 Design of symmetric circuits

Symmetry has been investigated as a technique for computing reductions for various types of model checking [4–9]. An important lesson learnt is that symmetries are best identified when they are recorded in the structure of the circuit at the time of design. Once symmetries are captured in this manner, then discovering them in the circuit amounts to merely reading them via type checking. This often reduces the otherwise NP-complete problem of discovering symmetries from circuits using sub-graph isomorphism, to a polynomial time type checking problem

of circuit's syntax described at a high-level. Although syntax based approach is very efficient in practice, success has been achieved with automatic symmetry detection as well [10, 11], and symmetry detection using input language restriction [12].

We have worked on the design and implementation of a type system that is used for designing a structurally symmetric class of circuits. The type system is a way of defining an *abstract data type of models*, rather than attempting to design a hardware description language like Verilog, VHDL or SystemC. We have used this to design several symmetric circuits starting from basic logic gates to multiplexers, comparators, registers, RAMs, CAMs and circuits that involve multiple CAMs. These circuits appear regularly in all modern microprocessor designs. Our methodology of designing the type system is based on identifying the following key issues:

1. Propose a sufficiently generic type for circuits.
2. Define the functions that will be used for constructing symmetric circuit blocks.
3. Provide the type judgement rules.
4. Articulate the mathematical definition of structural symmetry.
5. Prove a type soundness theorem, which says that every circuit definition that is well behaved with respect to the typing rules, will have structural symmetry.

A circuit is said to have structural symmetry with respect to a specific set of inputs and outputs when the circuit's behaviour remains unchanged under permutation of those specific set of inputs and outputs. The specific inputs in this case can be termed as symmetric inputs. Every circuit can have some inputs which will be symmetric[1]. There would be some inputs that would not have any role in the symmetry of the circuit, and they are referred to as non-symmetric inputs. Thus a circuit can have the following type:

$$circ : (bool\ list)\ list \rightarrow (bool\ list)\ list \rightarrow (bool\ list)\ list$$

where the type $(bool\ list)$ denotes the set of bit values of any one group of wires (also known as a bus), and the list of Boolean lists $((bool\ list)\ list)$ denotes the collection of several such buses. The first argument of the circuit type denotes the values of non-symmetric inputs, the second argument denotes the values for symmetric inputs and the third argument the outputs of the circuit. The type of symmetric circuit blocks is

$$circ : (bool\ list)\ list \rightarrow (bool\ list)\ list$$

We provide a set of functions that allow us to construct circuits with the above type. The definition of these circuit constructing functions is structured into layers. The first level, called Level 0 in our framework, is the layer that consists of function definitions over Boolean lists, and provides type judgement rules,

[1] In case the circuit has no symmetry the set of symmetric inputs will be empty.

to identify safe functions, that are used to design symmetric functional blocks. The second layer, referred to as Level 1, is the layer where we provide circuit combinators, and a type system that is used for identifying ways in which these circuit combinators are combined to generate circuits that have symmetry. We show that these circuits indeed have symmetry by proving the type soundness theorem. We distinguish between the terms *functions* and *combinators*. The term function is used to mean objects of the type $bool\ list \rightarrow bool\ list$, whilst the term combinators, mean objects of the type ($bool\ list\ list \rightarrow bool\ list\ list \rightarrow bool\ list\ list$), or $bool\ list\ list \rightarrow bool\ list\ list$.

In subsequent sections, we present the details of our approach.

3 Functional blocks — Level 0 framework

We use several well known functions on lists from functional programming [13–15], such as *hd*, *tl*, *append*, *map*, *map2*, *foldr* and so on, as building blocks for defining other higher-order combinators that are used to define symmetric circuit blocks. The definition of symmetry we are going to present relies on the concept of a permutation. Any arbitrary permutation on a set of n elements can be composed from pair-wise distinct swaps on the elements of the set. Since we represent the buses as lists, we are interested in swaps on lists. Thus we define the concept of symmetry in terms of atomic swap operations.

Definition 1. *Swap on lists*

$$\begin{array}{lcl} swap\ \ (i,\ j)\ \ lst & \triangleq & if\ \ (i\ <\ length\ lst)\ \wedge\ (j\ <\ length\ lst)) \\ & & \quad then\ \ (insert\ (el\ j\ lst)\ \ i\ \ (insert\ (el\ i\ \ lst)\ \ j\ \ lst)) \\ & & \quad\ \ else\ \ lst \end{array}$$

The function swap itself is defined in terms of a function called *insert*, that simply inserts (overwrites) a given element at a certain position in a given list. At Level 0 we define the concept of symmetry for functional blocks.

Definition 2. *Symmetry of functional blocks*

$$sym\ c \ \triangleq\ \forall inp\, i\, j.\ c\ \ (swap\ (i,j)\ inp)\ \ =\ \ swap\ (i,j)\ (c\ inp)$$

The functions that constitute the Level 0 framework are *id*, *map*, $\circ$ and *fold*. The function *fold*, folds a given function f onto a functional block c using the familiar function *foldr* (right-fold) on lists; others are described in Table 1. Also shown in the table are the typing rules for Level 0 framework, defined inductively by using the predicate *safe*.

Definition 3. *The function fold*

$$fold\ \ f\ \ (c : bool\ list\ \rightarrow bool\ list)\ \ \triangleq\ \ \lambda inp.\ \ [\,foldr\ \ f\ \ (hd\ (c\ inp))\ (tl\ (c\ inp))\,]$$

The property that all Level 0 functional blocks preserve the symmetry definition shown above, is characterised by the Level 0 safety theorem.

$$\frac{}{safe\ \ id}$$

$$\frac{f : bool \rightarrow bool}{safe\ \ (map\ \ f)}$$

$$\frac{(safe\ \ c)\ \ (f : bool \rightarrow bool \rightarrow bool) \qquad (assoc\ f)\ \ (comm\ f)}{safe\ \ (fold\ \ f\ \ c)}$$

$$\frac{safe\ \ c_1 \qquad safe\ \ c_2}{safe\ \ (c_1 \circ c_2)}$$

Table 1. Rules for safe functional blocks. The function *map* is the polymorphic function that maps a function f onto a list. The function *id* is the identity function on Boolean lists, while $\circ$ is the polymorphic serial composition function. Note that f needs to be associative (*assoc*) and commutative (*comm*) for fold to be a safe functional block.

Theorem 1. *Level 0 Safety Theorem*

$\vdash \ \forall c.\ safe\ \ c\ \supset\ sym\ \ c$

Proof outline: The proof takes place by an induction on the rules defining safe. □

4 Constructing symmetric circuits — Level 1 framework

In this section we shall present the elements of the next layer of our framework of designing structured models. This layer, called Level 1, consists of circuit construction combinators that allow us to construct symmetric circuits. We use the safe functions from Level 0 in the definition of circuit construction, and also define a new set of higher-order combinators that is used for designing bottom-up symmetric circuits. We provide type judgement rules that govern the way symmetric circuits can be constructed, and later on prove that the type judgement rules are sound, meaning that every piece of circuit definition that is well-behaved with respect to the type judgement rules has symmetry.

Before we proceed to show the definitions of our combinators, we define a predicate asserting that all buses in a list are of equal length. This is vital for us since circuits that do bitwise operations on a pair of inputs, require the inputs to be of equal length.

Definition 4. *Buses are of equal length*

$$CheckLength\ \ inp \quad \triangleq \quad \forall l.\ l \in inp\ \supset\ \forall m.\ m \in inp\ \supset\ (length\ l\ =\ length\ m)$$

The combinators used for defining circuit blocks are defined in the subsequent section. Note that all these combinators take a list of Boolean lists and produce a list of Boolean lists as an output. The intuition is that these combinators are all functions from symmetric inputs to outputs, where the type of list of Boolean lists is used for representing the symmetric input and outputs. When defining a circuit with non-symmetric inputs, one can pass them as the first argument and the symmetric inputs can be the second argument. Thus the combinators we define encapsulate the non-symmetric inputs (list of Boolean lists), and hence have the following type:

$$(bool\ list)\ list \rightarrow (bool\ list)\ list$$

4.1 Circuit construction combinators

The combinator *Null* does not produce any output, and is useful for circuit designs, where for a certain configuration of inputs, the user wants to have an empty list of outputs. Then we have the identity or the buffer circuit defined by the combinator *Id*, which simply returns the list of symmetric input buses in the output. The third combinator is the polymorphic combinator *map*. In Level 1, the function *map* maps a safe function c_0, onto a list of symmetric input buses.

$$\begin{array}{lcl} Null & \triangleq & \lambda sym.\ [\,] \\ Id & \triangleq & \lambda sym : (bool\ list)\ list.\ sym \end{array}$$

The serial composition operator $\circ$ is the polymorphic function $\circ$. It is used here to compose two circuits serially.

Parallel composition of two different symmetric circuit blocks is accomplished by a parallel composition operator ($\|$). One important constraint to keep in mind is that the length invariant *CheckLength* should be preserved by buses both before and after the $\|$ combinator is applied. This is important to ensure since we do not want to club unequal buses together in one list because we would like to be able to do a bitwise operation on this list.

$$\begin{array}{lcl} (c1\ \|\ c2) & \triangleq & \lambda sym.\ if\ \ CheckLength(append\ (c1\ sym)(c2\ sym)) \\ & & \qquad then\ \ append\ \ (c1\ sym)(c2\ sym)\ \ else\ \ [\,] \end{array}$$

Given a circuit c, if we want to duplicate it, we can use the *Fork* combinator, the definition of this is presented below.

$$Fork\ \ c \ \triangleq\ \lambda sym.\ \ append\ (c\ sym)\ (c\ sym)$$

Amongst a list of n buses, we may wish to select any one of the n buses, and this is done by the *Select* combinator. Again note that it uses the definition of list selection function *el*.

$$\begin{array}{lcl} Select\ n\ c & \triangleq & \lambda sym.\ if\ (n\ <\ length(c\ sym)) \\ & & \qquad then\ \ [el\ n\ (c\ sym)]\ \ else\ [\,] \end{array}$$

Given a list of buses, we wish to have an operator that can allow us to take all the buses but the first one. This is a feature that is similar to the *tl* function on lists. This is done by the *Tail* combinator. We define the combinator *Tail* in terms of *tl*.

$$\begin{array}{lcl} \mathit{Tail}\ c & \triangleq & \lambda \mathit{sym}.\ \mathit{if}\ (1\ <\ \mathit{length}(c\ \mathit{sym})) \\ & & \quad \mathit{then}\ \ \mathit{tl}\ (c\ \mathit{sym})\ \ \mathit{else}\ [\,] \end{array}$$

Our last combinator definition is also one of the most useful ones. This is the definition of the *Bitwise* operation on a list of buses, all of which are of equal length. The function *Bitwise*, applies the function f bitwise to all the buses, and it does this by using the usual bitwise function on lists, *map2*. *Bitwise* is defined by folding the combinator $(\mathit{map2}\ f)$, onto a list of buses. The starting value for the *foldr* function comes from the head of the list of Boolean lists, and then *Bitwise* traverses the remainder of the list of buses, and using the function f, applies $(\mathit{map2}\ f)$ bitwise to all the buses.

$$\begin{array}{lcl} \mathit{Bitwise}\ f\ c & \triangleq & \lambda \mathit{sym}.\ \mathit{if}\ ((c\ \mathit{sym})\ \neq\ [\,]) \\ & & \quad \mathit{then}\ \ [\mathit{foldr}\ (\mathit{map2}\ \ f)(\mathit{hd}\ (c\ \mathit{sym}))(\mathit{tl}\ (c\ \mathit{sym}))\,] \\ & & \quad \mathit{else}\ [\,] \end{array}$$

Once we define the combinators, we have to establish rules of combining them. These rules lay the foundation of distinguishing the symmetric circuits from non-symmetric ones. By using these rules to type check a given piece of circuit syntax, we can conclude that the circuit has symmetry. This is possible because we prove a type soundness theorem, which we will present later.

4.2 Typing rules for symmetric circuits

In this section we shall present a type system that provides the type judgement rules for building symmetric circuits. The typing rules are defined by inductively defining the predicate *SS*, which represents the concept "symmetry-safe". The rules are shown in Table 2.

4.3 Symmetry of circuits

In this section we will present the definition of symmetry. A circuit is symmetric if its behaviour remains unchanged under permutation of its symmetric input and output states.

Definition 5. *Symmetry of circuits*

$$\begin{array}{lcl} \mathit{Sym}\ \ c & \triangleq & \forall \mathit{inp}.\ \ \mathit{CheckLength}\ \mathit{inp}\ \ \supset \\ & & \quad \forall i\ j.\ \ \mathit{map}(\mathit{swap}(i,j))(c\ \ \mathit{inp})\ =\ c\,(\mathit{map}(\mathit{swap}(i,j))\ \ \mathit{inp}) \end{array}$$

Now we present the most important result of this paper, which is the type soundness theorem. This theorem characterises the property that all Level 1 circuit construction combinators preserve the symmetry property shown in Definition 5.

$$\frac{}{SS\ \ Null} \qquad \frac{}{SS\ \ Id} \qquad \frac{safe\ \ c_0}{SS\ \ (map\ \ c_0)}$$

$$\frac{SS\ \ c_1 \qquad SS\ \ c_2}{SS\ \ (c_1 \circ c_2)} \qquad \frac{SS\ \ c_1 \qquad SS\ \ c_2}{SS\ \ (c_1 \parallel c_2)} \qquad \frac{SS\ \ c}{SS\ \ (Fork\ \ c)}$$

$$\frac{SS\ \ c \qquad n : num}{SS\ \ (Select\ \ n\ \ c)} \qquad \frac{SS\ \ c}{SS\ \ (Tail\ \ c)}$$

$$\frac{SS\ \ c \qquad (assoc\ \ f) \qquad (comm\ \ f) \qquad f : bool \rightarrow bool \rightarrow bool}{SS\ \ (Bitwise\ \ f\ \ c)}$$

Table 2. Type judgement rules for symmetric circuits. Note that for *Bitwise* to be a symmetry-safe (SS) combinator, f needs to be associative and commutative.

Theorem 2. *Symmetry safe circuits have symmetry*

$\vdash \forall c.\ SS\ \ c\ \ \supset\ \ Sym\ \ c$

Proof outline. The proof is done by using rule induction on the predicate SS. For details please refer to [16].

5 Conclusion

The novelty of this paper lies in the design of a framework which allows efficient discovery of symmetry through a correct-by-construction methodology that relies on the design of an abstract data type for circuits. The abstract data type is designed using some basic types of lists, and Booleans, and well-known functions over lists. The heart of the abstract data type is the set of type inference rules that are defined inductively on the circuit combinators (Table 2). On one hand these rules enable type checking and on the other, allow a bottom-up design of symmetric circuits.

Using our approach symmetry detection (via type checking) for a given circuit of different sizes takes a fixed amount of time, which is proportional to the number of terms in the formula of the circuit definition. We have designed and implemented the complete theory presented in this paper in the higher-order logic theorem prover HOL 4 [15]. It is available on request from the author.

We have designed and implemented a framework of reduction where we record symmetries in circuits using the approach presented in this paper, and then these

circuits are synthesized to gate-level netlists that are used for simulation in an STE simulator. Together with the reduction methodology explained in [3], we demonstrate significant reduction in the size of STE model checking for verification of several circuits ranging from simple designs such as comparators, multiplexers, gates and n-bit registers to more complex ones such as random access memories (SRAM), content-addressable memories (CAM), and circuits with multiple CAMS [16].

References

1. C.-J. H. Seger and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Journal of Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, March 1995.
2. Robert B. Jones, John W. O'Leary, Carl-Johan H. Seger, Mark D. Aagaard, and Thomas F. Melham, "Practical Formal Verification in Microprocessor Design," *IEEE Design & Test of Computers*, vol. 18, no. 4, pp. 16–25, July/August 2001.
3. Ashish Darbari, "Symmetry Reduction for STE Model Checking," in *Sixth International Conference on Formal Methods in Computer Aided Design*, 2006.
4. C. Norris Ip and David L. Dill, "Better Verification Through Symmetry," *Formal Methods in System Design*, vol. 9, no. 1/2, pp. 41–75, 1996.
5. Manish Pandey and Randal E. Bryant, "Exploiting Symmetry When Verifying Transistor-Level Circuits by Symbolic Trajectory Evaluation.," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 18, no. 7, pp. 918–935, 1999.
6. K. L. McMillan, "A Methodology for Hardware Verification using Compositional Model Checking," *Science of Computer Programming*, vol. 37, no. 1-3, pp. 279–309, 2000.
7. Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla, "Symmetry Reductions in Model Checking," in *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*. 1998, pp. 147–158, Springer-Verlag, Berlin.
8. A. Prasad Sistla and Patrice Godefroid, "Symmetry and Reduced Symmetry in Model Checking," *ACM Transactions on Programming Languages and Systems*, vol. 26, no. 4, pp. 702–734, 2004.
9. A. Miller., A. Donaldson, and M. Calder, "Symmetry in Temporal Logic Model Checking.," *ACM Computing Surveys*, vol. 38, no. 3, 2006.
10. Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov, "Exploiting Structure in Symmetry Detection for CNF.," in *DAC*, 2004, pp. 530–534.
11. Alastair F. Donaldson and Alice Miller, "Automatic Symmetry Detection for Model Checking Using Computational Group Theory," in *FM*, 2005, pp. 481–496.
12. A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson, "SMC: A symmetry-based model checker for verification of safety and liveness properties," *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 2, pp. 133–166, 2000.
13. L. C. Paulson, *ML for the Working Programmer*, Cambridge University Press, second edition, June 1996.
14. Richard Bird and Philip Wadler, *An Introduction to Functional Programming*, Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1988.
15. "HOL 4," Available from http://hol.sourceforge.net/.
16. Ashish Darbari, *Symmetry Reduction for STE Model Checking using Structured Models*, Ph.D. thesis, Oxford University Computing Lab, Wolfson Building, Parks Road, Oxford, Submitted 2006, Available on request.