MAX-PLANCK-INSTITUT FÜR INFORMATIK

Beyond the Finite in Automatic Hardware Verification

David Basin
Nils Klarlund

MPI–I–96–2–009 September 1996

I N F O R M A T I K

Im Stadtwald
D 66123 Saarbrücken
Germany

Authors' Addresses

David Basin
Max-Planck-Institut für Informatik
Im Stadtwald, D-66123
Saarbrücken, Germany
basin@mpi-sb.mpg.de

Nils Klarlund
AT&T Labs
600 Mountain Ave.
Murray Hill, NJ 07974
klarlund@research.att.com

Publication Notes

The present report has been submitted for publication and will be copyrighted if accepted.

Acknowledgements

The first author was funded by the German Ministry for Research and Technology (BMFT) under grant ITS 9102. Responsibility for the content lies with the authors. The authors thank Harald Ganzinger and Natarajan Shankar for their feedback on previous drafts of this paper and Aarti Gupta for helpful discussions about her work. Tiziana Margaria provided several references.

Abstract

We present a new approach to hardware verification based on describing circuits in Monadic Second-order Logic (M2L). We show how to use this logic to represent generic designs like n-bit adders, which are parameterized in space, and sequential circuits, where time is an unbounded parameter. M2L admits a decision procedure, implemented in the Mona tool [16], which reduces formulas to canonical automata.

The decision problem for M2L is non-elementary decidable and thus unlikely to be usable in practice. However, we have used Mona to automatically verify, or find errors in, a number of circuits studied in the literature. Previously published machine proofs of the same circuits are based on deduction and may involve substantial interaction with the user. Moreover, our approach is orders of magnitude faster for the examples considered. We show why the underlying computations are feasible and how our use of Mona generalizes standard BDD-based hardware reasoning.

1 Introduction

Correctness of hardware systems can be established by enumeration when the possible behaviors are finite, or formal theorem proving, when the possible behaviors are infinite. The finite case arises when reasoning, for example, about combinational circuits: these can be represented as functions in Boolean logic and correctness can be established by enumeration of possible inputs and outputs. Although any hardware system is of finite size, the infinite case may arise in several ways. One may be interested in demonstrating the correctness of an infinite *family* of related systems, for example, families of arithmetical circuits like n-bit adders or n-bit counters, whose description depends uniformly on the parameter n. Alternatively, the behavior of a single circuit may depend not only on current inputs, but on previous values as well. For example, the behavior of a sequential circuit is a function of time, and one may want to establish that the circuit behaves correctly over arbitrarily long time intervals.

When behaviors are finite, arguments based on enumeration are popular due to the optimizations often possible using a symbolic representation like Binary Decision Diagrams (BDDs). A BDD is an automaton-like representation of a finite relation or function. In the BDD method, a symbolic representation of the finite function calculated by a combinational circuit is obtained through operations reflecting the Boolean semantics of the gates. The BDD calculations are often much faster than other mechanized means of reasoning and demand little user intervention.

We present here a generalized method that can automatically establish properties of many infinite relations and functions. Our method is based on a decidable logic, the Monadic Second-order Logic on Strings, abbreviated M2L. In M2L, propositional variables of Boolean logic are generalized to variables that denote strings of bits. Every M2L formula ϕ defines a language over an alphabet $\mathbf{B}^k$, consisting of a cross-product of Booleans: one Boolean for each of the k free variable in ϕ. Strings over this alphabet describe the values of all free variables. The language defined by ϕ then is the possibly infinite set of strings defining values that make the formula true. This correspondence generalizes the way a BDD defines a set of satisfying truth assignments. Moreover, any such language corresponds to a language recognized by a finite-state machine; hence M2L formulas characterize regularity.

We show how to exploit this logical characterization of regularity to reason about parameterized classes of circuit designs and their behavior. The language that a formula defines can represent words of unbounded size (the behaviors of members of a parameterized family of circuits) or how the state of a circuit evolves over time.

An example of a parameterized family of circuits is an n-bit adder. In M2L we can write a formula ϕ (cf. §4) that precisely describes how 1-bit adders are composed in a ripple-carry fashion to form n-bit adders. Under the semantics of M2L, ϕ defines an input-output relation on two inputs A and B of size n, and an output C of size n. This relation can be represented by a language over an alphabet that has three Boolean components so that a string of length n encodes the values of A, B, and C. For example,

	0	1	2	3
A	1	1	0	0
B	1	0	0	0
C	0	0	1	0

defines three rows or *tracks* of bits. The length n of the string is 4. The positions of the string (and of the tracks) are numbered from 0 to $n-1$. If we assume that the least significant bit comes first, then the first track defines $A = 3 = 1100$. Similarly, we read off $B = 1 = 1000$, and $C = 4 = 0010$. Thus, this string defines an interpretation such that the sum of the binary numbers A and B is C. Note that variable A can also be thought of as denoting a subset, namely the set $\{0, 1\}$ of positions where the A-track contains a 1 (similarly, B denotes the subset $\{0\}$, and C denotes $\{2\}$.) Alternatively, we may view the set denoted by A as a predicate $A(p)$ that holds on position p if and only if there is a 1 in the A-track. The predicate $A(p)$ is monadic (i.e., of one argument). Thus, when A occurs in a formal logic as a variable, it is *monadic second-order*.

An example of temporal parameterization is the modeling of an RS flip-flop, where a string of length n with three components models the behavior of the circuit through n time instants, each described by a letter defining the values of the inputs R and S and the output Q. These examples are very easy to formulate in M2L; with a little syntactic sugar, the M2L specifications resemble those used in standard hardware description languages.

Since any M2L formula ϕ can be reduced to an automaton that accepts the satisfying interpretations of ϕ, validity is decidable. A formula ϕ is valid (i.e. always true) if the corresponding automaton accepts all strings. Validity testing can be used to show that the logic of a circuit is consistent with a specification of its behavior. For example, if the formula $\phi_{behavior}$

describes the behavior of an n-bit adder and the formula $\phi_{circuit}$ describes a proposed realization as a parameterized circuit, then the property that the circuit behaves as an adder can be checked by verifying that the automaton corresponding to $\phi_{circuit} \Rightarrow \phi_{behavior}$ accepts all strings. If there is some string that is not accepted by the automaton, then this string encodes a counter-model, which can be used to debug the proposed design.

The MONA tool, described in [16], implements a decision procedure and a counter-model generator for formulas in M2L on strings (and trees, which we do not consider here). MONA supports predicate definitions, libraries, display of automata, and counter-model generation. Its implementation is based on a generalization of BDDs for the representation of automata on large alphabets.

Our contributions

We describe the theory and practice of how M2L, as embodied in MONA, can be used to automatically verify parameterized circuit designs. Our results demonstrate how MONA efficiently generalizes BDDs to handle regular reasoning about infinite domains. The examples we present here offer various techniques for dealing with the infinite in automatic hardware verification.

- Our arithmetic logic unit (ALU) example shows how an infinite family of combinational circuits can be concisely described in M2L.
- Our D-type flip-flop example illustrates how M2L can be used as a succinct temporal logic for analysis of difficult sequential circuits. This example also demonstrates how MONA serves not only as a verification tool but also provides a means to explore and understand circuit behavior.
- Our 74LS163-counter and signal processor examples show how parameterized sequential circuits can be verified.

Our approach applies to any scenario that can be modeled as a regular set over alphabets of the form $\mathbf{B}^k$. Not all parameterized circuits can be so described (e.g., multipliers and grid-shaped circuits with multiple independent parameters). However, our examples indicate that, when applicable, both circuits and their properties can be simply expressed in M2L.

The decision problem for M2L is non-elementary decidable: a formula of size n may require time and space bounded below by an iterated stack of exponentials whose height is proportional to n. Such a staggeringly bad theoretical complexity suggests that any implementation may be unusable in practice. In contrast, Quantified Boolean Logic (QBL), which can formalize combinational logic (and be decided using traditional BDD operations), is only PSPACE-complete.

We explain why M2L is usable in practice despite the worst-case bounds. For the circuits studied both in this paper and in the literature, our approach is orders of magnitudes faster. For hardware problems expressible in QBL, Mona is as efficient as the direct use of BDD-based procedures, since Mona generalizes standard BDD-based hardware reasoning. Moreover, for the parameterized systems considered here, we show that the increased cost of working with a more expressive logic is negligible.

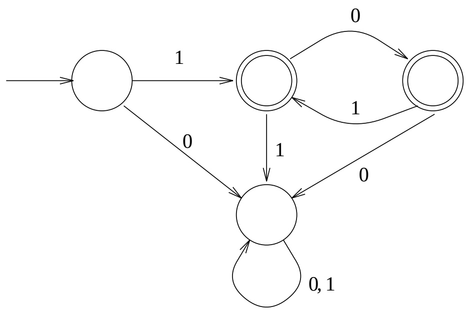
Organization

We proceed as follows. In §2, we introduce M2L. In §3, we present the essentials of the Mona tool and show how it generalizes BDD-based hardware procedures. In §4, we consider specification and verification of parameterized combinational hardware. In §5, we consider timed hardware and use Mona to analyze a D-type flip-flop. In §6, we verify a $4n$-bit counter implemented in terms of n 4-bit 74LS163 counters, and in §7 we present a signal-processing benchmark circuit. These circuits are sequential, but we reason about them as parameterized transition systems. In §8, we give some theoretical justification for why our approach works. Finally in §9, we compare M2L and our use of Mona to other deduction based and automata theoretic approaches.

2 The Second-Order Monadic Logic on Strings

The Monadic Second-order Logic on strings that we use is closely related to S1S, the second-order monadic theory of one successor, and S2S, the second-order monadic theory of two successors, which are among the most expressive decidable logics known (cf. [28]). In these logics, first-order terms are interpreted over positions in an infinite string (S1S) or tree (S2S) and second-order variables are interpreted by subsets of positions. In M2L, first-order terms are interpreted over finite strings. S1S and S2S are more expressive than M2L, but have not been shown to be feasible in practice.

The correspondence between automata and regular languages is well-known. The decidability of the above mentioned logics is based on the well understood (but less widely known) fact that regular languages may be characterized by logics. Consider, for example, the automaton



which accepts the regular language $\{1, 10, 101, 1010, 10101, \ldots\}$. Now assume that X is a variable over binary strings. We say that $X(p)$ holds if the pth position in X is 1. Now, the regular language above can be described in M2L as

$$X(0) \wedge \forall p : p < \$ \rightarrow (X(p) \leftrightarrow \neg X(p \oplus 1)), \tag{1}$$

which states that the first character in the string is 1 and that for subsequent positions p, up to the final position (denoted by the symbol \$), the pth character of X is 1 precisely when the following character is not.

We describe M2L below. It turns out that the logic precisely characterizes regularity: every M2L formula describes a regular set and, conversely, every regular set is described by an M2L formula.

2.1 Syntax

M2L consists of three kinds of entities: first-order terms, second-order terms, and formulas. First-order terms are formed from first-order variables $p, q, \ldots,$ the constants 0 (the first position), \$ (the last position), and the expressions $t \oplus i$ (the ith position to the right from t), where t is a first-order term and i is a natural number. Second-order terms are built from second-order variables $X, Y, \ldots,$ the constants *empty* (the empty set) and *all* (the set of all positions), and they may be combined using $\cap$ and $\cup$. Formulas arise as follows: if t_1 and t_2 are first-order terms and S_1 and S_2 are second-order terms, then $t_1 \in S_1$, $t_1 = t_2$, $t_1 < t_2$, and $S_1 = S_2$ are formulas. Formulas may be combined by the standard connectives $\neg$ and $\wedge$. Quantifiers also build formulas: if p and X are first and second-order variables respectively, and f is a formula, then $\exists^1 p : f$ and $\exists^2 X : f$ are formulas.

The syntax we have given is not minimal, cf. [28]. For example, first-order variables can be eliminated by replacing each first-order variable with a second-order variable that is constrained to be a singleton set. (This is also the way that MONA handles first-order variables.) Also, we will make frequent use of standard definitions and syntactic sugar in the remainder of the paper.

First, the complete set of propositional connectives, inequality, universal quantification and the like are all definable as is standard in a classical logic. For example $f_1 \vee f_2$ is defined as $\neg(\neg f_1 \wedge \neg f_2)$ and $\forall^2 X : f$ is defined as $\neg(\exists^2 X : \neg f)$.

Second, since we can view a second-order variable X as a bit vector, we again write $X(p)$ for $p \in X$.

Third, Boolean variables, connectives and quantification over Booleans values are not part of M2L, but are easily encoded. In particular, each Boolean variable b is encoded by a second-order variable B, and occurrences of b in formulas are encoded as $B(-1)$, where -1 is an extra position, just to the left of the position 0. The position -1 is used solely for the simulation of Boolean variables. (We do not use the position 0 for technical reasons.) In this way, quantification over Booleans ($\forall^0$ and $\exists^0$) is encoded using second-order quantification. For example, the Boolean formula $\forall^0 x, y : \neg(x \wedge \neg y)$ is encoded as the M2L sentence $\forall^2 X, Y : \neg(X(-1) \wedge \neg(Y(-1)))$.

Finally, when the order of a variable can be determined from context then we may omit superscripts on quantifiers. For example, in the expression $X(p) \wedge b$, it must be the case that X, p, and b are second-order, first-order, and Boolean, respectively. To help disambiguation, we use capital letters for second-order variables and lower-case letters like i, j, p, and q for first-order position variables. Remaining lower-case strings like like x, y, cin and $cout$ represent Booleans. With these abbreviations and conventions, (1) is a formula of M2L.

2.2 Semantics

A formula is interpreted relative to a natural number $n \geq 0$, called the *length*, which defines *positions* $\{0, \ldots, n-1\}$. A first-order term denotes a position. Thus, a first-order variable ranges over the set $\{0, \ldots, n-1\}$. The constant

0 denotes the position 0, and \$ denotes $n-1$.[1] The expressions $t \oplus i$ and $t \ominus i$ denote the positions $j+i \bmod n$ and $j-i \bmod n$, where j is the interpretation of t.

A second-order variable P denotes a subset of $\{0, \ldots, n-1\}$. Alternatively, a second-order variable can be viewed as designating a bit pattern $b_0 \ldots b_{n-1}$ of length n, where b_i is 1 if and only if i belongs to the interpretation of P. The constants *empty* and *all* denote the sets $\emptyset$ and $\{0, \ldots, n-1\}$, and the operators $\cap$ and $\cup$ are usual set theoretic operations.

A 0th order (Boolean) variable is simulated by a special second-order variable, which may contain the non-standard position -1 (and this means "true").

The meaning of formulas is straightforward. For example, the formula $t \in S$ is true when the position denoted by t is in the set denoted by S. Propositional connectives have their standard meaning. $\exists^1 p : f$ is true when there is a position i in $\{0, \ldots, n-1\}$ such that the denotation of f is true with i replacing p. Truth of $\exists^2 X : f$ is defined similarly, with X replaced by a subset of $\{0, \ldots, n-1\}$.

A formula ϕ defines a regular language denoting the interpretations that make free variables in ϕ true. In the formula (1), we have one free variable, X, and the interpretations that make ϕ true are exactly the strings in the regular language $\{1, 10, 101, 1010, 10101, \ldots\}$. More generally, if a formula has k free second-order variables (and as noted above, all other variables are encoded using second-order variables), then the language denoted is over the alphabet $\mathbf{B}^k$ consisting of k-tuples of Booleans. As a simple example, the formula ϕ given by $\forall p : P(p) \leftrightarrow \neg Q(p)$ defines a language $L(\phi)$ over $\mathbf{B}^2$ as follows. We make the convention that if the letter $\begin{array}{|c|}\hline a \\ \hline b \\ \hline\end{array} \in \mathbf{B}^2$ occurs in position i, then i is in P iff a is 1 and i is in Q iff b is 1. In this way, a string over $\mathbf{B}^2$ determines an interpretation of P and Q. The language denoted is the set of strings describing interpretations that make ϕ true. For example,

$$\begin{array}{|c|c|c|c|}\hline 0 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 1 \\ \hline\end{array} \in L(\phi) \quad \text{and} \quad \begin{array}{|c|c|c|}\hline 0 & 1 & 1 \\ \hline 0 & 0 & 0 \\ \hline\end{array} \notin L(\phi)\,.$$

[1] When the length n is 0, there are no positions defined. Therefore, 0 and $n-1$ do not make sense. We will not be bothered by this anomaly, since the case $n = 0$ is irrelevant to the kinds of examples presented in this paper.

3 The MONA Tool

The MONA tool implements a decision procedure for M2L. Details can be found in [16]; here, we summarize the main algorithms and data structures.

Input to MONA is a script consisting of a sequence of definitions followed by a formula to be proved. For each formula ϕ in the script, MONA constructs a deterministic automaton recognizing $L(\phi)$. Construction of automata proceeds using standard operations (see [28]) by recursion on the structure of ϕ.

For example, if ϕ is the formula $\phi_1 \wedge \phi_2$, then MONA first calculates the automata A_i recognizing the language corresponding to ϕ_i. Second, MONA calculates the automaton corresponding to ϕ by forming the product automata of the A_i and minimizing the result. In a similar way, negation corresponds to automata-theoretic operation of swapping final and non-final states. Existential quantification corresponds to a projection, followed by a subset construction, and minimization. More precisely, if formula ϕ corresponds to an automaton A that reads strings over the alphabet $\mathbf{B}^k$, then the automaton for the formula $\exists^2 X.\phi$ is built by projection from A by changing it so that it guesses the track corresponding to X. The resulting automaton is non-deterministic and must be determinized in order to be minimized.

Since MONA always stores automata in a minimized form, valid formulas are particularly simple to recognize: they correspond essentially to the trivial automaton whose single state is both the initial and final state with a self-loop as transition on every input. For any formula ϕ that is not valid, MONA extracts from its corresponding automaton a minimal length string defining an interpretation making ϕ invalid. We use this procedure to generate counter-examples to proposed theorems.

3.1 BDD Representation

Although the automata constructions are in principle standard, we note that the exponential size of the alphabet $\mathbf{B}^k$ calls for special consideration—otherwise even the representation of the transition function for an automaton corresponding to a formula with k variables would necessitate space proportional to 2^k. Thus the implementation in [16] uses multi-valued BDDs to compress the representation of the transition function. The exponential blow-up is then often avoided.

To see how this is possible, consider the formula $\phi \equiv x \wedge y \vee A = B$, where

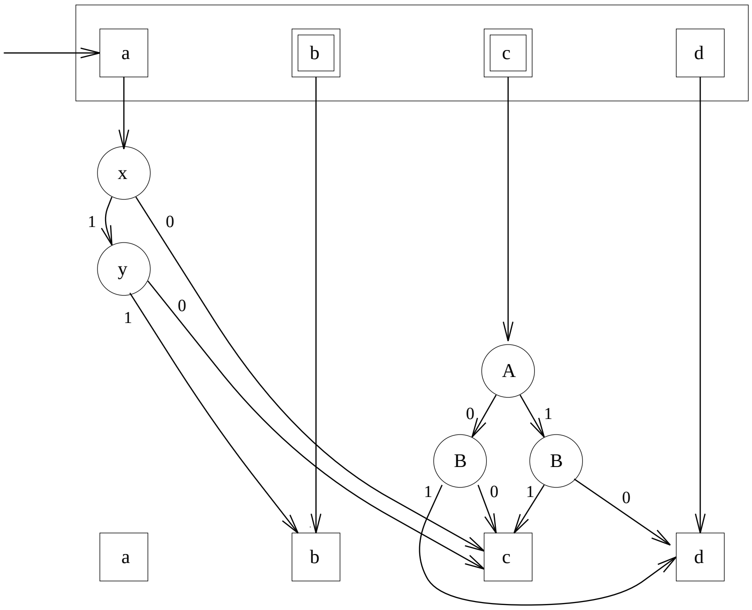


Figure 1: A BDD-represented automaton.

x and y are Boolean variables and A and B are second-order variables. An interpretation of this formula is defined by a string over $\mathbf{B}^4$ whose positions are numbered $-1, 0, \ldots, n-1$ and where we assume that the tracks are in the order x, y, A, B. For example, the string

$$
\begin{array}{c|c|c|c|c|}
 & -1 & 0 & 1 & 2 \\
\cline{2-5}
x & 1 & X & X & X \\
\cline{2-5}
y & 0 & X & X & X \\
\cline{2-5}
A & X & 1 & 0 & 1 \\
\cline{2-5}
B & X & 0 & 1 & 0 \\
\cline{2-5}
\end{array}
$$

defines $x = 1$, $y = 0$, $n = 3$, $A = \{0, 2\}$, and $B = \{1\}$ (X means "don't care"). The automaton that accepts all strings defining satisfying interpretations (i.e., interpretations that make ϕ true) is depicted in Figure 1. The automaton has four states $\{a, b, c, d\}$ shown in the rectangular box. In practice, the states are just entries in an array. Each state contains a pointer to a BDD node. For example, the initial state a points to a decision node for x. Thus if the letter in position -1 has a 1 in the x-component (in the first track), then the pointer labeled 1 is followed, and a decision is then made on the y-component. Consequently, if both the x-component and the y-component have a 1 in the -1st letter, then a leaf marked b is reached upon reading this letter. This leaf signifies that the state entered next is b, which is an accepting state (denoted by an inner square).

From state b, there is a pointer directly to a leaf. We say that the state is *looping*—this means that the letter read is irrelevant. Thus the automaton accepts all strings that define both x and y to be true. If one is false, then the automaton remains in the accepting c state as long as the membership status of the current position is the same for A and B.

Note that by using the position -1 for the Boolean variables, we have avoided the problem that an encoding based on position 0 would lead to an ill-defined semantics for Boolean variables in the case of the empty string (where position 0 does not exist).

3.2 Canonicity of BDD Representation

The automaton shown above is minimal or canonical in two ways: (1) the BDD representation of the transition function is reduced (canonical) and (2) the transition function represented and state space are those of the canonical automaton. The requirement (1) is maintained automatically by the use of BDD algorithms that reduce the representation as the BDD is calculated. Requirement (2) is enforced by the use of a minimization algorithm on each new automaton calculated. The current MONA minimization algorithm [16] is quadratic in the size (the number of nodes and states) of the representation, although in practice minimization is often only about twice as costly as the product and projection routines.

3.3 Relationship to Usual BDDs

If a formula ϕ contains only Boolean variables, then the BDD represented automaton has only three states: the initial state and two looping states, one accepting and one non-accepting. If the pointers of the looping states are deleted, then the resulting graph is identical to the standard BDD representation of ϕ for the given track assignment (ordering of variables). Moreover, for propositional logic, and its extension to Quantified Boolean Logic, the calculations carried out by MONA are essentially identical to those performed by a standard BDD based procedure. In particular, the automaton product algorithm described in [16] essentially degenerates to a BDD binary apply routine. Similarly, the automaton projection essentially degenerates to a BDD projection routine. From this it follows that

Proposition 1 *For any variable ordering chosen for a formula of QBL,* MONA *essentially performs the same calculations as a standard BDD based algorithm.*

4 Parameterized Combinational Hardware

In this section, we show how to specify and verify circuit designs parameterized in their word length. Such parametric designs represent families of circuits. For example, an n-bit adder represents a family of adders, one for each n. Using M2L, we can specify such a family and prove its correctness with respect to a parameterized behavioral specification.

4.1 Preliminaries: Combinational Circuits

We can define in M2L predicates at a level that formalizes appropriate building blocks of circuits. We can represent the behavior of such blocks as functions from inputs to outputs or as relations between external circuit ports. The functional approach is used for example in theorem provers based on equational and other quantifier free logics (e.g., the prover of Boyer and Moore, NQTHM [17]), where primitive components are functions. For example, *and* is a function from two inputs to an output. Larger circuits are built by functional composition.

The relational approach is typically used with first-order or higher-order logic. Basic components are relations which define constraints between port-values. These relations are joined together using conjunction (which combines constraints), and internal wires are represented by shared variables that are existentially quantified. In [4, 11], these two kinds of representation are discussed in detail. Both options are available in our work, and it makes little difference which one we choose.

We follow the relational approach in specifying circuits. We begin by

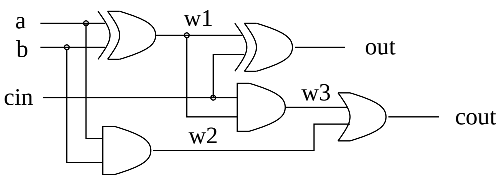


Figure 2: Full 1-bit adder

defining basic gates as relations over Boolean variables. For example:

$$
\begin{aligned}
not(a, o) &\equiv o \leftrightarrow \neg a \\
and(a, b, o) &\equiv o \leftrightarrow (a \wedge b) \\
or(a, b, o) &\equiv o \leftrightarrow (a \vee b) \\
xor(a, b, o) &\equiv o \leftrightarrow ((\neg a \wedge b) \vee (a \wedge \neg b)) \\
and3(a, b, c, o) &\equiv o \leftrightarrow (a \wedge b \wedge c) \\
or3(a, b, c, o) &\equiv o \leftrightarrow (a \vee b \vee c)
\end{aligned}
$$

The left-hand side of each definition names a predicate whose meaning is given by the right-hand side. The actual input to MONA is identical except that ASCII syntax, additional key words, and type declarations are required. The appendices provide scripts from actual MONA sessions, including the verification of the n-bit adder given in this section.

Let us now build a full 1-bit adder from these gates. One such design is given in Figure 2. The top half of the circuit consists of two *xor* gates connected by an internal wire w_1 that computes the sum bit *out*. The bottom half uses the value of internal wire w_1 as well as the two inputs a and b to compute the carry-out bit *cout*. Our definition in M2L conjoins the gate descriptions and projects away the internal wires:

$$
\begin{aligned}
&full\_adder(a, b, out, cin, cout) \equiv \\
&\quad \exists^0 w_1, w_2, w_3 : xor(a, b, w_1) \wedge xor(w_1, cin, out) \wedge and(a, b, w_2) \wedge \\
&\quad and(cin, w_1, w_3) \wedge or(w_3, w_2, cout)
\end{aligned}
$$

Now let us consider our first example of a theorem proved by MONA. Although the adder is specified as a relation, for each set of inputs, it computes unique outputs. That is, *out* and *cout* are functionally determined by a, b,

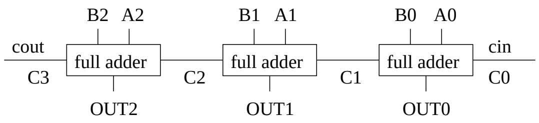


Figure 3: n-bit adder for $n = 3$

and cin.

$$\begin{array}{l}\forall^0 a,\ b,\ cin : \exists^0 out,\ cout : \mathit{full\_adder}(a, b, out, cin, cout) \\ \quad \wedge\ \forall^0 o, co : (\mathit{full\_adder}(a, b, o, cin, co) \rightarrow ((o \leftrightarrow out) \wedge (co \leftrightarrow cout)))\end{array}$$

MONA proves this theorem in 0.25 seconds.[2] This includes parsing all definitions, converting them to automata, and afterwards translating the conjecture into an automaton. In this case, all calculations are equivalent to standard BDD operations, since we are essentially using just Quantified Boolean Logic.

4.2 Correctness of an n-bit Adder

The circuit

We turn now to parameterized hardware and consider an n-bit adder. Figure 3 gives an example of this for $n = 3$. In the general case, an n-bit adder is constructed by (1) wiring together n 1-bit adders where (2) the carry-out of the ith becomes the carry-in of the i+1st. The first and last carry are special cases; (3) the first carry has the value of the carry-in and (4) the last has the value of the carry-out.

It is easy to formalize this kind of ripple-carry connectivity. Let us use C and D to represent the carry-ins and carry-outs, respectively. Then we can formalize the general case as the following predicate, which relates three second-order variables (the two input strings A and B and the output string

[2]All times reported in this paper are measured in CPU seconds on a Sun Ultra-Sparc work station.

Out) and two Booleans (the carry-in cin and carry-out $cout$).

$$\begin{array}{l} n\_add(A, B, Out, cin, cout) \equiv \\ \quad \exists^2 C, D : (\forall^1 p : \mathit{full\_adder}\ (A(p), B(p), Out(p), C(p), D(p))) \\ \quad\quad \wedge\ (\forall^1 p : (p < \$) \rightarrow (D(p) \leftrightarrow C(p \oplus 1))) \\ \quad\quad \wedge\ (C(0) \leftrightarrow cin) \\ \quad\quad \wedge\ (D(\$) \leftrightarrow cout) \end{array}$$

The four lines of the definition body formalize the four requirements listed above. The way we formalize ripple-carry connectivity is independent of the particular component (here a full-adder) that we are iterating. We later use an identical formalization for specifying an n-bit ALU constructed from 1-bit ALUs.

The specification

To verify our circuit, we specify how n-bit binary words are added. Since M2L is a logic about strings and string positions, any arithmetic must be encoded within this limited language. In particular, we encode addition as an algorithm over strings representing bit-patterns, i.e., binary addition. A simple way to do this is to mimic how addition is computed with pencil and paper. The ith output bit is set if the sum of the ith inputs and carry-in is 1 mod 2, and the ith carry bit is set if at least two of the previous inputs and carry-in was set. The 0th carry and the final values must be computed as special cases.

$$\begin{array}{rcl} at\_least\_two(a, b, c) & \equiv & (a \wedge b) \vee (a \wedge c) \vee (b \wedge c) \\ mod\_two(a, b, c, d) & \equiv & a \leftrightarrow b \leftrightarrow c \leftrightarrow d \end{array}$$

$$\begin{array}{l} add(A, B, Out, cin, cout) \equiv \\ \exists^2 C : \\ \quad\quad (\quad \forall^1 p : mod\_two(A(p), B(p), C(p), Out(p)) \\ \quad\quad\quad\quad\quad \wedge\ ((p < \$) \rightarrow (C(p \oplus 1) \leftrightarrow at\_least\_two(A(p), B(p), C(p)))) \\ \quad\quad \wedge \quad (cout \leftrightarrow at\_least\_two(A(\$), B(\$), C(\$))) \\ \quad\quad \wedge \quad C(0) \leftrightarrow cin) \end{array}$$

To give the reader a feel for the complexity involved in translating such specifications to automata, we mention some statistics for this example.

There are, overall, 109 product and projection operations performed, and the average number of states is 5 and BDD nodes is 12. The largest intermediate automaton has 21 states and 71 BDD nodes. We will later return to this example in §8 and analyze more carefully why the state-space does not explode during translation.

Verification

We now have a specification of the implementation of a family of adders built from gates and a specification in terms of its behavior over binary strings. To verify their equivalence, we give Mona the formula

$$\begin{aligned}&\forall^2 A, B, Out : \forall^0 cin, cout : \\ &\quad add(A, B, Out, cin, cout) \leftrightarrow n\_add(A, B, Out, cin, cout)\,.\end{aligned}$$

This formula is verified in 0.41 seconds.

Often we are interested in more than one property of a circuit or its specification. For example, the n-bit adder computes a unique function from its inputs to its outputs.

$$\begin{aligned}&\forall^2 A,\ B : \forall^0 cin : \exists^2 Out : \exists^0 cout : n\_add(A, B, Out, cin, cout) \\ &\quad \wedge \forall^2 O : \forall^0 co : (n\_add(A, B, O, cin, co) \rightarrow (Out = O \wedge (cout \leftrightarrow co)))\end{aligned}$$

We may also check that the addition function defined is commutative.

$$\forall^2 A, B, Out : \forall^0 cin, cout : add(A, B, Out, cin, cout) \leftrightarrow add(B, A, Out, cin, cout)$$

Both of these are verified in under a second.

4.3 Correctness of an n-bit ALU

We now apply our approach to a more complex circuit—a parameterized n-bit ALU. The circuit we analyze is presented in [21]. It is also an interesting theorem for comparison (given in §9), since it has been verified in several theorem proving systems based on induction.

ALU specification

The ALU is designed to perform 8 arithmetic and 4 logical operations. The 12 functions are selected through 3 "selection" lines s_0, s_1, s_2 and the carry-in cin as described in Table 1. For example, if the s_i are 0 and cin is 1, then

Selection				Output	Function
s_2	s_1	s_0	cin		
0	0	0	0	$F = A$	Transfer A
0	0	0	1	$F = A + 1$	Increment A
0	0	1	0	$F = A + B$	Addition
0	0	1	1	$F = A + B + 1$	Addition with carry
0	1	0	0	$F = A - B - 1$	Subtract with borrow
0	1	0	1	$F = A - B$	Subtract
0	1	1	0	$F = A - 1$	Decrement A
0	1	1	1	$F = A$	Transfer A
1	0	0	X	$F = A \vee B$	OR
1	0	1	X	$F = A \oplus B$	XOR
1	1	0	X	$F = A \wedge B$	AND
1	1	1	X	$F = \overline{A}$	Complement A

Table 1: Function Table for ALU

the ALU increments the n-bit input A and places the result in F, producing a carry-out when every bit in F is set.

Let us begin by specifying this behavior: we formalize each functional sub-unit (addition, subtraction, etc.) and specify the function table by case analysis on the values of s_i. The logical sub-units are specified straightforwardly using the previously defined gates.

$$\begin{aligned}
\mathit{transfer}(\mathit{To}, \mathit{From}) &\equiv \mathit{To} = \mathit{From} \\
\mathit{compl}(A, F) &\equiv \forall^1 x : \mathit{not}(A(x), F(x)) \\
\mathit{OR}(A, B, F) &\equiv \forall^1 x : \mathit{or}(A(x), B(x), F(x)) \\
\mathit{XOR}(A, B, F) &\equiv \forall^1 x : \mathit{xor}(A(x), B(x), F(x)) \\
\mathit{AND}(A, B, F) &\equiv \forall^1 x : \mathit{and}(A(x), B(x), F(x))
\end{aligned}$$

For the remainder of the specification, we must develop more arithmetic. We define an auxiliary predicate *one*, which is true when a second-order variable represents the number one, i.e., when only the first bit is set.

$$\mathit{one}(B) \equiv B(0) \wedge \forall^1 p : (p > 0 \rightarrow \neg B(p))$$

We can now define the remaining arithmetic functions using the previously defined relation *add*.

$$
\begin{array}{l}
increment(A, F, cout) \equiv \\
\quad \exists^0 cin : \exists^2 N : one(N) \wedge \neg cin \wedge add(A, N, F, cin, cout) \\
add\_no\_carry(A, B, F, cout) \equiv \\
\quad \exists^0 cin : \neg cin \wedge add(A, B, F, cin, cout) \\
add\_with\_carry(A, B, F, cout) \equiv \\
\quad \exists^0 cin : cin \wedge add(A, B, F, cin, cout) \\
one\_compl\_add(A, B, F, cout) \equiv \\
\quad \exists^0 cin : \exists^2 Comp : \neg cin \wedge compl(B, Comp) \wedge add(A, Comp, F, cin, cout) \\
two\_compl\_add(A, B, F, cout) \equiv \\
\quad \exists^0 cin : \exists^2 Comp : cin \wedge compl(B, Comp) \wedge add(A, Comp, F, cin, cout) \\
decrement(A, F, cout) \equiv \\
\quad \exists^2 V : one(V) \wedge two\_compl\_add(A, V, F, cout)
\end{array}
$$

Now, using the following auxiliary definitions

$$
\begin{array}{rcl}
if_3(a, b, c, d) & \equiv & (a \wedge b \wedge c) \rightarrow d \\
if_4(a, b, c, d, e) & \equiv & (a \wedge b \wedge c \wedge d) \rightarrow e
\end{array}
$$

we encode $alu\_spec(s_0, s_1, s_2, A, B, F, cin, cout)$ by specifying the function table as the iterated conjunction, one conjunction for each function.

$$
\begin{array}{l}
if_4(\neg s_2, \neg s_1, \neg s_0, \neg cin, transfer(A, F)) \wedge \\
if_4(\neg s_2, \neg s_1, \neg s_0, cin, increment(A, F, cout)) \wedge \\
if_4(\neg s_2, \neg s_1, s_0, \neg cin, add\_no\_carry(A, B, F, cout)) \wedge \\
if_4(\neg s_2, \neg s_1, s_0, cin, add\_with\_carry(A, B, F, cout)) \wedge \\
if_4(\neg s_2, s_1, \neg s_0, \neg cin, one\_compl\_add(A, B, F, cout)) \wedge \\
if_4(\neg s_2, s_1, \neg s_0, cin, two\_compl\_add(A, B, F, cout)) \wedge \\
if_4(\neg s_2, s_1, s_0, \neg cin, decrement(A, F, cout)) \wedge \\
if_4(\neg s_2, s_1, s_0, cin, transfer(A, F)) \wedge \\
if_3(s_2, \neg s_1, \neg s_0, OR(A, B, F)) \wedge if_3(s_2, \neg s_1, s_0, XOR(A, B, F)) \wedge \\
if_3(s_2, s_1, \neg s_0, AND(A, B, F)) \wedge if_3(s_2, s_1, s_0, compl(A, F))
\end{array}
$$

ALU implementation

The ALU implementation, as specified in [21], is given in Figure 4. The corresponding M2L formula is encoded analogously to the parameterized

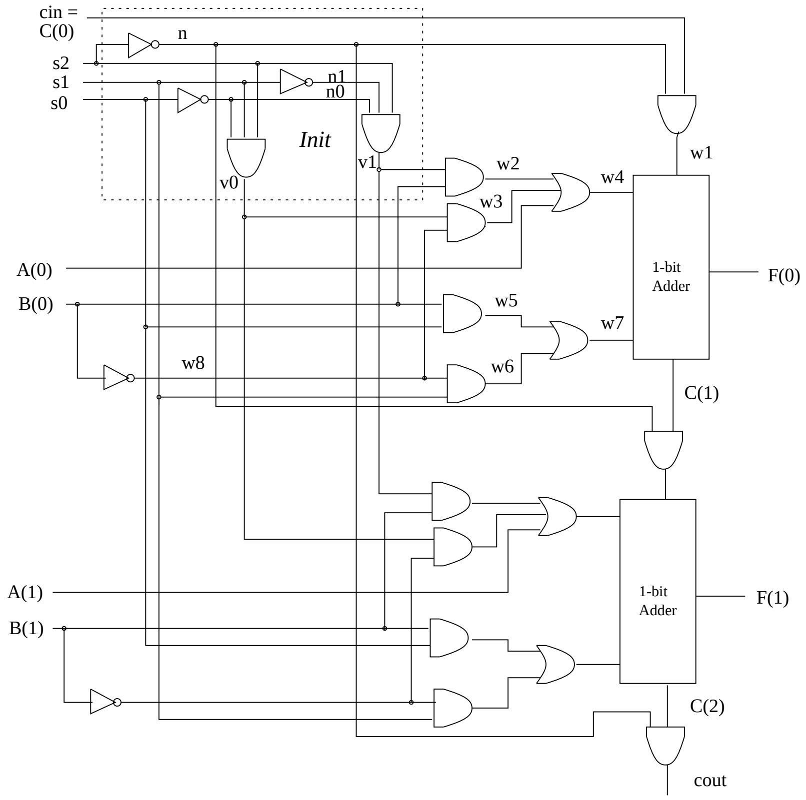


Figure 4: n-bit ALU ($n = 2$)

adder. The only additional complication is that the description consists of two parts: an initialization block and a repeating ALU block. The first part, which we call *init* computes negations of the selection wires and conjunctions of them and their negations.

$$
\begin{aligned}
&\mathit{init}(s_0, s_1, s_2, v_0, v_1, n) \equiv \\
&\quad \exists^0 n_0,\ n_1 : \mathit{not}(s_0, n_0) \wedge \mathit{not}(s_1, n_1) \wedge \mathit{not}(s_2, n) \wedge \\
&\qquad \mathit{and3}(n_0, s_1, s_2, v_0) \wedge \mathit{and3}(n_0, n_1, s_2, v_1)
\end{aligned}
$$

The remainder of the ALU consists of the regular repetition of 1-bit ALU sections. These sections also require the switching wires s_i and the results of the *init* section computed on the wires v_0, v_1, and n.

$$\begin{aligned}
&\mathit{one_alu}(a, b, f, \mathit{cin}, \mathit{cout}, s_0, s_1, v_1, v_2, n) \equiv \\
&\quad \exists^0 w_1,\ w_2,\ w_3,\ w_4,\ w_5,\ w_6,\ w_7,\ w_8 : \mathit{and}(n, \mathit{cin}, w_1) \wedge \mathit{and}(v_1, b, w_2) \\
&\quad \wedge\ \mathit{and}(v_0, w_8, w_3) \wedge \mathit{or3}(w_2, w_3, a, w_4) \wedge\ \mathit{and}(b, s_0, w_5) \\
&\quad \wedge\ \mathit{and}(w_8, s_1, w_6) \wedge \mathit{or}(w_5, w_6, w_7) \wedge \mathit{not}(b, w_8) \\
&\quad \wedge\ \mathit{full_adder}(w_4, w_7, f, w_1, \mathit{cout})
\end{aligned}$$

To specify the parameterized ALU, we combine the *init* block with ripple-carried 1-bit ALU units. The ALU sections are hooked together as were the adder sections in the parameterized adder example.

$$\begin{aligned}
&\mathit{n_alu}(s_0, s_1, s_2, A, B, F, \mathit{cin}, \mathit{cout}) \equiv \\
&\quad \exists^2 C,\ D : \exists^0 v_0,\ v_1,\ n : \mathit{init}(s_0, s_1, s_2, v_0, v_1, n) \wedge \\
&\quad (\forall^1 p : \mathit{one_alu}(A(p), B(p), F(p), C(p), D(p)), s_0, s_1, v_0, v_1, n) \wedge \\
&\quad (\forall^1 p : (p < \$) \rightarrow (D(p) \leftrightarrow C(p \oplus 1))) \wedge (C(0) \leftrightarrow \mathit{cin}) \wedge (D(\$) \leftrightarrow \mathit{cout})
\end{aligned}$$

We may now verify that the ALU implementation satisfies its specification. Namely, when the switches and ports of the ALU take on values consistent with the implementation, the specification is satisfied.

$$\begin{aligned}
&\forall^2 A, B, F : \forall^0 s_0, s_1, s_2, \mathit{cin}, \mathit{cout} : \\
&\quad \mathit{n_alu}(s_0, s_1, s_2, A, B, F, \mathit{cin}, \mathit{cout}) \rightarrow \mathit{alu_spec}(s_0, s_1, s_2, A, B, F, \mathit{cin}, \mathit{cout})
\end{aligned}$$

It takes MONA 2 seconds to verify this. Other properties, such as the functional relation between the inputs and outputs, are also easily checked in about the same amount of time.

Note that we proved only that the implementation satisfies (implies) the specification. We did not prove an equivalence, as we did with the n-bit adder. The reason is that the specification is more abstract than the implementation: it leaves certain port value combinations unspecified. Suppose we did not know this, or perhaps did, but we wanted to determine when the converse fails. If we ask MONA to prove the converse it responds that the formula is not a tautology. If we remove the initial quantifiers, i.e.,

$$\mathit{alu\_spec}(s_0, s_1, s_2, A, B, F, \mathit{cin}, \mathit{cout}) \rightarrow \mathit{n\_alu}(s_0, s_1, s_2, A, B, F, \mathit{cin}, \mathit{cout})\,,$$

then the port values are free variables and MONA produces a counter-example and responds:

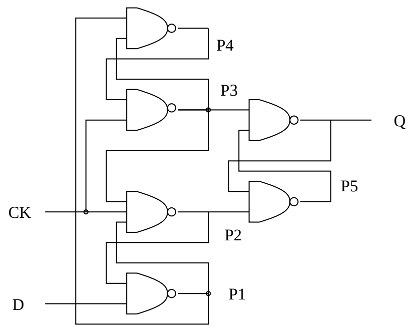


Figure 5: D-type Flip-flop

```
A counter-example of least length (1) is:
Booleans:
cout 1
s2 1
s1 1
s0 1
Second-order:
A          0
B          X
F          1
```

The output tells us that there is a counter-example of length $n = 1$, i.e., consisting of a single 1-bit ALU slice. This counter-example is sensible. The specification only states that when the s_i are all 1, then F is the complement of A. So the specification holds for any value of B and any value of *cout*, in particular $cout = 1$. However, these values are not consistent with the implementation.

5 Sequential Circuits

In the last section, a string represented a sequence of bits, i.e., a word of parameterized length. In this section, a string represents the behavior of a sequential circuit (of fixed bit-width) as it evolves over time. Circuit descriptions are similar to those we have previously seen except that gates are now parameterized by time.

Our example is a standard implementation of a D-type flip-flop, built from 6 *nand* gates, as shown in Figure 5. Although this circuit looks simple, understanding and demonstrating its correctness is difficult. Hanna and Daeche give a thorough and well-written analysis of this flip-flop in [15].[3] They used Veritas, a theorem prover based on a higher-order logic, to give a comprehensive analysis using a partial description of waveforms over the rational numbers. Their analysis is complex, and it took an experienced user a week to construct the proof.

Our starting point is a discrete model of this circuit proposed by Gordon in [11]. He assumed that each gate has a delay of one time unit. Gordon described the behavior of the circuit using HOL formulas, where first-order variables denote time instants. The proof that the circuit meets its specification, which he notes "is fairly complicated" was done by hand only. The flip-flop and Gordon's HOL specification are easily encoded in MONA. To our surprise, MONA calculated a counter-example. We later discovered that Wilk and Pnueli had already reported on the failure of Gordon's specification in [30]. They formulated Gordon's informal requirements in a temporal logic with "quantized" tense operators like $\Diamond^n \phi$, which holds at the present moment if ϕ holds at least once within the next n time units.

Temporal logic, in the sense of tense logic, is based on operators that denote modalities like "it will be the case" and "until". Linear tense logic is PSPACE-complete, and it has been explored intensively [9]. But temporal logic can as well be viewed as simply a first-order logic of natural numbers (if we are content with the natural numbers as a model of time)—which was essentially also Gordon's approach. To our knowledge, this point of view has not been pursued from a practical point of view in verification, maybe because this formulation is non-elementary (as is M2L). We believe that the first-order formulation is more attractive, since many temporal idioms (including the usual tense operators) can easily be expressed as predicates.

To translate the other way, from the first-order formulation to the tense

[3]Hanna and Daeche write about the complexity of the circuit (page 193):

> "It turns out, on analysis, that the *modus operandi* of this circuit is far from simple: in fact, it is unusually complex, and (so the authors found) difficult to understand intuitively. If, like most people, you find this remark difficult to accept at face value, read the rest of this account, then set it aside, and attempt, within (say) one working day, to come up with a carefully justified account of 'how' the proposed implementation is intended to function..."

formulation, is much more difficult and potentially involves a non-elementary blow-up; this is why Wilk and Pnueli could not directly use Gordon's HOL specification, but had to transcribe the informal requirements.

We present next our analysis, which is based on experiments with MONA.

5.1 Temporal Concepts

The temporal concepts needed to reason about the flip-flop are straightforward to express in MONA:

- the value of F is stable in $[t_1, t_2]$:
  $\mathit{stable}(t_1, t_2, F) \equiv \forall^1 t : t_1 \leq t \leq t_2 \rightarrow (F(t) \leftrightarrow F(t_1))$
- t_2 is the first instant after t_1 when F becomes high:
  $\mathit{next}(t_1, t_2, F) \equiv t_1 < t_2 \wedge F(t_2) \wedge (\forall^1 t : t_1 < t < t_2 \rightarrow \neg F(t))$
- F rises at t:
  $\mathit{rise}(t, F) \equiv t > 0 \wedge (\neg F(t \ominus 1) \wedge F(t))$
- F falls at t:
  $\mathit{fall}(t, F) \equiv t > 0 \wedge (F(t \ominus 1) \wedge \neg F(t))$
- F rises at Rise:
  $\mathit{times\_rise}(F, \mathit{Rise}) \equiv \forall^1 t : \mathit{Rise}(t) \leftrightarrow \mathit{rise}(t, F)$
- F falls at Fall:
  $\mathit{times\_falls}(F, \mathit{Fall}) \equiv \forall^1 t : \mathit{Fall}(t) \leftrightarrow \mathit{falls}(t, F)$

5.2 The Circuit

The temporal behavior of a unit-delay nand-gate with inputs I_1 and I_2 and output O is described by

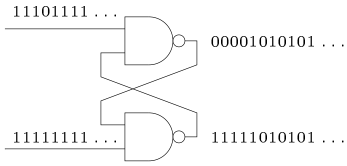
$$\mathit{nand}(I_1, I_2, O) \equiv \forall^1 t : t < \$ \rightarrow O(t \oplus 1) \leftrightarrow \neg(I_1(t) \wedge I_2(t))\,.$$

If we call the corresponding predicate for three inputs $\mathit{nand}3(I_1, I_2, I_3, O)$, then the flip-flop in Figure 5 is described by

$$\begin{aligned} \mathit{dtype\_imp} \equiv\ & \mathit{nand}(P_2, D, P_1) \wedge \mathit{nand}3(P_3, CK, P_1, P_2) \wedge \mathit{nand}(P_4, CK, P_3) \wedge \\ & \mathit{nand}(P_1, P_3, P_4) \wedge \mathit{nand}(P_3, P_5, Q) \wedge \mathit{nand}(Q, P_2, P_5)\,. \end{aligned}$$

5.3 Stability Analysis

In our model, even a simple flip-flop may begin to oscillate due to a single negative spike:



We address this phenomenon (which was not discussed in [11, 30]) to demonstrate how an understanding of the circuit can be achieved by experiments in MONA.

Informally, we would like to argue that if the input signals are kept stable for some time and if the circuit is already stable, then eventually the other signals of the circuit become stable as well. We define

$$\begin{aligned} & \mathit{input\_stable}(t) \equiv t + \mathit{in\_stable\_time} - 1 \leq \$ \\ & \quad \wedge\ \mathit{stable}(t, t \oplus \mathit{in\_stable\_time} - 1, D) \wedge\ \mathit{stable}(t, t \oplus \mathit{in\_stable\_time} - 1, CK) \end{aligned}$$

to denote that inputs are stable for a period of length *in_stable_time*.[4]

We regard the circuit as stable if all outputs of gates are stable for an interval of *circ_stable_time* instants, i.e., if

$$\begin{aligned} & \mathit{circuit\_stable}(t) \equiv t \oplus \mathit{circ\_stable\_time} - 1 \leq \$ \wedge \\ & \quad \mathit{stable}(t, t \oplus \mathit{circ\_stable\_time} - 1, P_1) \wedge \mathit{stable}(t, t \oplus \mathit{circ\_stable\_time} - 1, P_2) \wedge \\ & \quad \mathit{stable}(t, t \oplus \mathit{circ\_stable\_time} - 1, P_3) \wedge \mathit{stable}(t, t \oplus \mathit{circ\_stable\_time} - 1, P_4) \wedge \\ & \quad \mathit{stable}(t, t \oplus \mathit{circ\_stable\_time} - 1, P_5) \wedge \mathit{stable}(t, t \oplus \mathit{circ\_stable\_time} - 1, Q)\,. \end{aligned}$$

Stability preservation of the circuit can be expressed informally as: if the circuit is stable at some t_s and if the inputs are held stable at $t_i \geq t_s$, then there is $t'_s \geq t_i$ such that the circuit is stable at t'_s. Thus, we define

$$\begin{aligned} & \mathit{stability\_preserved} \equiv \\ & \quad \forall^1 t_s : \mathit{circuit\_stable}(t_s) \rightarrow \\ & \qquad \forall^1 t_i : (t_i > t_s \ \wedge\ \mathit{input\_stable}(t_i) \rightarrow \exists t'_s : t'_s \geq t_i \wedge\ \mathit{circuit\_stable}(t'_s))\,. \end{aligned}$$

[4] We here use $+$ instead of $\oplus$ in the formula $t + \mathit{in\_stable\_time} - 1 \leq \$$, which holds if $+$ and $-$ are interpreted in the usual arithmetic sense without "wrap-around". We need the conjunct "$t + \mathit{in\_stable\_time} - 1 \leq \$$" to prevent t from lying too close to the end (in which case there would not be enough remaining time instants to model that the signals are stable for the required amount of time).

Let us try to verify stability preservation as embodied by the formula

$$dtype\_imp \Rightarrow stability\_preserved\,.$$

MONA calculates a counter-example in about 5 seconds (where we have made *in_stable_time* equal 6):

$$\begin{array}{lcl}
D & = & 0111111 \\
CK & = & 0111111 \\
Q & = & 1111010 \\
P_1 & = & 1101010 \\
P_2 & = & 1101010 \\
P_3 & = & 1111010 \\
P_4 & = & 0001010 \\
P_5 & = & 0001010 \\
t_s & = & 1000000 \\
t_i & = & 0100000
\end{array}$$

Here we have made t_s and t_i free variables so that Mona can generate a counter-example that identifies the exact spot of trouble.[5] We see that the simultaneous rise of both the D and CK signals seem to tickle the circuit so that it begins to oscillate despite being stable initially. (Incidentally, this was the problem that Gordon had failed to address in his specification.) Note that the quantification $\exists^1 t'_s$ must succeed before "time runs out," i.e., before the finite segment of time that the logic is interpreted over ends. In other words, we have made the assumption that the stabilization of the circuit takes place while the inputs are kept stable.

5.4 Input Requirements

By experiments that constrain the inputs in different ways, we have arrived at the following requirements on the input signals: the clock signal must not form a negative spike of duration less than *min_clock_low* or a positive spike of duration less than *min_clock_high*. The D signal must be stable for at least

[5]Note that t_s and t_i are first-order position variables. These are actually encoded in Mona as second-order variables ranging over singleton sets. Here t_s and t_i point to positions 0 and 1 respectively.

setup units before CK rises. We define these conditions as

$$\begin{array}{l} \mathit{input\_requirements} \;\equiv \\ \quad \forall^1 t : \; \begin{array}[t]{l} (\mathit{fall}(t, CK) \rightarrow \mathit{stable}(t, t \oplus \mathit{min\_clock\_low} - 1, CK)) \wedge \\ (\mathit{rise}(t, CK) \rightarrow \mathit{stable}(t, t \oplus \mathit{min\_clock} - 1, CK)) \wedge \\ (\mathit{rise}(t, CK) \rightarrow \mathit{stable}(t \ominus (\mathit{setup} - 1), t, D))\,. \end{array} \end{array}$$

(The actual Mona code also contains the test for end of time, which we have omitted here for sake of brevity.) Now, with the choices

min_clock_low	2
min_clock_high	3
setup	3
circ_stable_time	2
in_stable_time	6

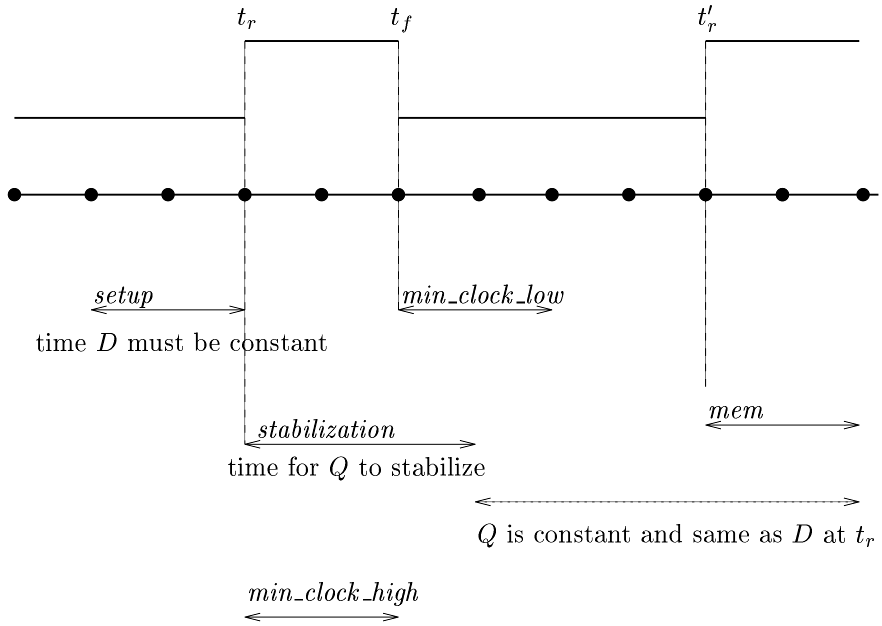
Mona proves the implication

$$\mathit{dtype\_imp} \wedge \mathit{input\_requirements} \rightarrow \mathit{stability\_preserved}$$

in about 2 seconds.

5.5 D-type Flip-flop Behavior

The essential D-type flip-flop behavior is as depicted below: if the clock rises at t_r, then falls at t_f, and then rises again at t'_r, then the value of D at t_r appears at Q at time $t_r \oplus \mathit{stabilization} - 1$ and remains there until time $t'_r \oplus \mathit{mem} - 1$. When we add the input requirements already stated to this set of circumstances, a complicated set of timing relationships is enforced:



Formally, we express the essential flip-flop behavior as

$$
\begin{aligned}
dtype \ \equiv \forall \ \ & t_r, t_f, t'_r : \\
& rise(t_r, CK) \\
& \wedge (\exists^2 P : times\_rise(CK, P) \wedge next(t_r, t'_r, P)) \\
& \wedge (\exists^2 P : times\_fall(CK, P) \wedge next(t_r, t_f, P))) \ \rightarrow \\
& (stable(t_r \oplus stabilization - 1, t'_r \oplus mem - 1, Q) \\
& \wedge Q(t_r \oplus stabilization - 1) \leftrightarrow D(t_r)) \, .
\end{aligned}
$$

This is essentially the same behavior specified by Gordon in [11]. Now, with the additional choices

stabilization	4
mem	2

the implication

$$dtype\_imp \ \wedge \ input\_requirements \rightarrow dtype$$

is verified in about 2 seconds. Experiments show that these values cannot be lowered.

6 Verification of Parametric Iterative Circuits

We have used parameterization to represent both families of combinational circuits and sequential designs. Here we consider the two aspects together: sequential circuits with parametric data-paths. The interesting problem now is that there are two independent parameters: time and word (data-path) length. Both parameters cannot be simultaneously formalized since our second-order variables represent only monadic predicates (which take a single argument).[6] We extend here the well-known idea of reasoning about a sequential circuit in terms of its transition function to circuits with parameterized data-paths. As an illustration, we present a counter architecture based on the 74LS163 4-bit counter with enable and synchronous clear. We begin by specifying the implementation and behavior of the 4-bit basic cell and afterwards we specify a 4×n-parameterized counter constructed as a cascade of 4-bit modules.

6.1 The Behavioral Specification

The counter is given in Figure 6. The inputs are the I_i, four control lines, PEn, SRn, CET, and $CEPT$, and the clock pulse CP. The outputs are the O_i and the terminal count TC. The control lines select one of the four possible operations (*clear*, *parallel_load*, *increment*, or *no_op*) according to the values of the control signals, and the data-path consists of a data register with synchronous clear. In [10] the specification is given by the following mode select table

SRn	PEn	CET	CEP	Action (Effect)
L	X	X	X	Reset (Clear)
H	L	X	X	Load ($I \rightarrow O$)
H	H	H	H	Count (Increment)
H	H	L	X	No Change (Hold)
H	H	X	L	No Change (Hold)

plus the logic equation for the terminal count TC

$$tc = I_o \wedge I_1 \wedge I_2 \wedge I_3 \wedge cet\,.$$

[6] Logics involving binary-predicates, such as logics on grids, are generally undecidable, since Turing Machine computations can be encoded on the grid.

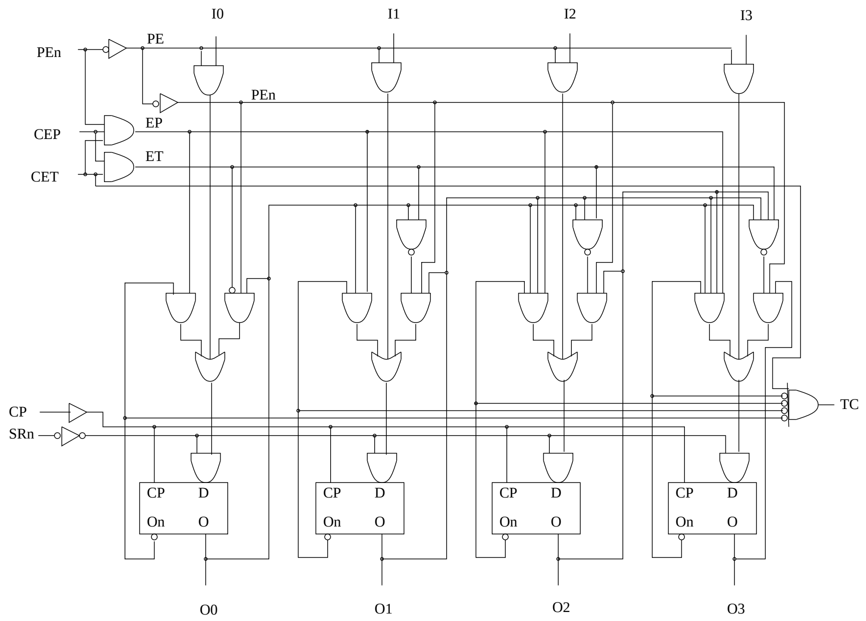


Figure 6: The 74LS163 4-bit Counter

The behavior is parameterized by time: the increment and hold operation depend on the previous output. But all of the operations are also well-defined independent of the width of the data-path. We will give a specification parameterized by the data-path width. It is sufficient to require the consistency of the counter's behavior over any two consecutive time units; that is, for each operation mode, the value of the generated output O (which coincides with the new state) and of the terminal count tc must be consistent with the previous output/state, which we call PO, the current input I and the current values of the control signals as prescribed by the above table.

To encode the behavioral specification, we first define some auxiliary predicates.

$$
\begin{aligned}
\mathit{if}(b,t,f) &\equiv (b \to t) \wedge (\neg b \to f) \\
\mathit{clear}(O) &\equiv \forall^1 p : \neg O(p) \\
\mathit{inc}(O,N) &\equiv \mathit{if}(O = \mathit{all}, N = \mathit{empty}, \exists^1 j : \neg O(j) \wedge (\forall^1 k : k < j \to O(k)) \wedge \\
&\qquad \forall^1 l : (l < j \to \neg N(l)) \;\wedge \\
&\qquad\qquad (l = j \to N(l)) \;\wedge \\
&\qquad\qquad (l > j \to (O(l) \leftrightarrow N(l))))
\end{aligned}
$$

We will also make use of the predicate *transfer*, which was defined for the ALU. Note that instead of using addition to specify incrementation as we did with the ALU, here we specify it directly. If O consists of all 1s, then N will consist of all zeros. Alternatively, there exists a least position j which is zero (i.e. $\neg O(j)$), and then the increment operation should clear all the smaller positions in N, set this position, and leave the rest unchanged.

Now, using PO to represent the output from the previous time unit, we can specify the counter's behavior. We encode the above table using nested *if* statements and the additional condition on tc.

$$
\begin{aligned}
&\mathit{speccount}(\mathit{pen}, \mathit{cep}, \mathit{cet}, \mathit{srn}, \mathit{tc}, I, O, PO) \equiv \\
&\quad \mathit{if}(\neg \mathit{srn}, \mathit{clear}(O), \\
&\qquad \mathit{if}(\neg \mathit{pen}, \mathit{transfer}(I,O), \\
&\qquad\quad \mathit{if}(\mathit{cet} \wedge \mathit{cep}, \mathit{inc}(PO,O), \mathit{transfer}(PO,O)))) \\
&\quad \wedge\, \mathit{tc} \leftrightarrow (\forall^1 j : PO(j) \wedge \mathit{cet})
\end{aligned}
$$

6.2 Verification of the Basic Cell

Our first verification problem is the correctness of the gate-level implementation of the 74LS163 4-bit counter given in Figure 6. Appendix B contains the specification of the implementation, called *count4bit*, which formalizes the counter as a relation in M2L.

The 4-bit implementation is not parameterized. All ports and internal wires are Booleans. Even though our behavioral specification is parameterized, we can still use it to verify the behavior of a counter operating over a 4-bit wide data-path. We must simply insist that the length of the strings is precisely 4 (so the last position, given by $, is 3). Hence we prove the

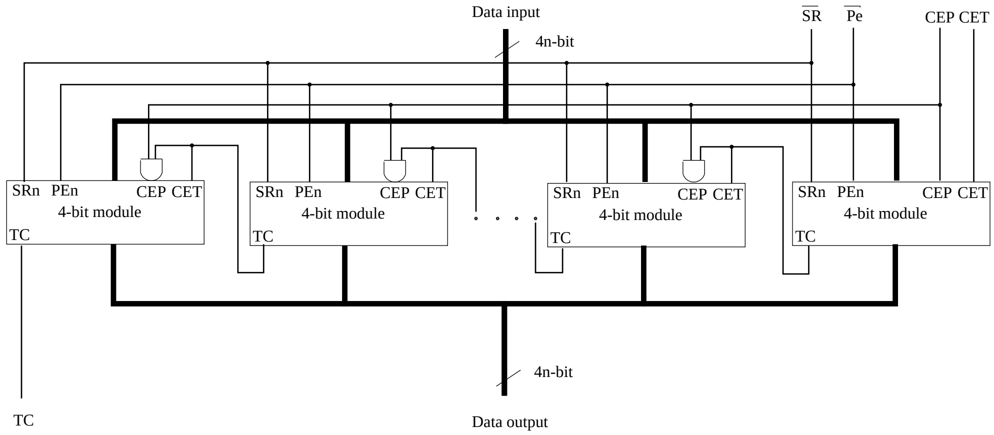


Figure 7: Hierarchical implementation of the $4n$ bit counter

equivalence of the concrete implementation with respect to the parameterized specification as follows.

$$
\begin{aligned}
&\$ = 3 \to \\
&\quad (\mathit{count4bit}(pen, cep, cet, srn, tc, \\
&\qquad I(0), I(1), I(2), I(3), O(0), O(1), O(2), O(3), PO(O), PO(1), PO(2), PO(3)) \\
&\qquad \leftrightarrow \ \mathit{speccount}(pen, cep, cet, srn, tc, I, O, PO))
\end{aligned}
$$

MONA proves this equivalence in under 3 seconds. Note that this is a problem over a finite domain (strings are limited to size 4); thus MONA is again here used essentially as a BDD-based decision procedure for Quantified Boolean Logic.

6.3 A Cascaded Counter Architecture

We can use the 4-bit counter blocks to implement a $4n$-bit counter by cascading together the 4-bit modules, as indicated in Figure 7). To specify this design, we restrict input to strings of size $4n$ and partion the strings into n 4-bit units, one for each 4-bit module. Intuitively, this is possible since automata can count modulo n. We begin by specifying such an automaton with the following predicate, $\mathit{fourth}(p)$, which is true when the position variable p

takes values $3, 7, 11, \ldots$.

$$
\begin{aligned}
& \mathit{fourth}(p) \equiv \\
& \quad \exists^2 S : (\neg S(0) \wedge \neg S(1) \wedge \neg S(2) \\
& \qquad \wedge (\forall^1 p : (p \geq 3 \rightarrow (S(p) \leftrightarrow (\neg S(p \ominus 1) \wedge \neg S(p \ominus 2) \wedge \neg S(p \ominus 3)))))) \\
& \quad \wedge S(p)
\end{aligned}
$$

This predicate will be used to control the iteration that allocates the modules. The following description is a simple generalization of the ripple-carry iteration we have already seen, generalizing such iteration to blocks of constant size (in this case 4).

$$
\begin{aligned}
& \mathit{ripplecount}(pen, cep, cet, srn, tc, I, O, PO) \equiv \\
& \quad \exists^2 CEP, CET, TC : \\
& \qquad \forall^1 p : \mathit{fourth}(p) \rightarrow \\
& \qquad\quad \mathit{count4bit}(pen, cep, cet, srn, tc, \\
& \qquad\qquad I(p \ominus 3), I(p \ominus 2), I(p \ominus 1), I(p), \\
& \qquad\qquad O(p \ominus 3), O(p \ominus 2), O(p \ominus 1), O(p), \\
& \qquad\qquad PO(p \ominus 3), PO(p \ominus 2), PO(p \ominus 1), PO(p)) \wedge \\
& \qquad\quad (p \neq \$ \rightarrow (\mathit{and}(TC(p), cep, CEP(p \oplus 4)) \wedge (CET(p \oplus 4) \leftrightarrow TC(p)))) \wedge \\
& \qquad\quad (CEP(3) \leftrightarrow cep) \wedge (CET(3) \leftrightarrow cet) \wedge (tc \leftrightarrow TC(\$))
\end{aligned}
$$

Here, CEP, CET and TC are internal bit vector variables representing the vectors of intermediate control values that we need to propagate between the modules. For every 4th value of p we instantiate a *count4bit* slice with the preceding 4 input lines. E.g., on the first iteration, for $p = 3$, then $p \ominus 3$, $p \ominus 2$, $p \ominus 1$, and p correspond to to the first 4 positions 0, 1, 2 and 3, and *count4bit* computes the counter relation over these values. The rippling of the terminal count tc to the enabling control lines cep and cet of the next module follows the diagram given in Figure 7. Finally, the internal control signals CEP and CET are connected to the global ones and the global terminal count is defined to be the last position of TC.

We can now verify the equivalence between this cascaded implementation and the behavioral specification *speccount*. We show the equivalence for all inputs whose length is a multiple of 4, which is the case when $fourth(\$)$ holds.

$$
\begin{aligned}
& \forall^2 I, O, PO : \forall^0 pen, cep, cet, srn, tc : \\
& \quad \mathit{fourth}(\$) \rightarrow \\
& \qquad (\mathit{ripplecount}(pen, cep, cet, srn, I, O, PO, tc) \leftrightarrow \\
& \qquad\quad \mathit{speccount}(pen, cep, cet, srn, I, O, PO, tc))
\end{aligned}
$$

MONA verifies that this is valid in 11 seconds. This example was the most time intensive of those considered in this paper. 827 automata are computed in processing this example. The average number of states is 47, and the average number of BDD nodes is 189. The largest intermediate automaton generated contained 3,037 states and 12,865 BDD nodes.

7 A Parameterized Benchmark: the "Min-Max" Circuit

The *Min-Max* signal processor unit was formulated as a benchmark problem for the 1989 IFIP International Workshop on Applied Formal Methods for Correct VLSI Design [7]. Here we study a parameterized version suggested in [25]. This version was specified in the CASCADE Hardware Description Language and verified by means of a theorem prover. We argue that such descriptions can be straightforwardly translated into MONA provided that the arithmetic used is essentially regular.

The unit is controlled by three Boolean signals; in addition, it has a parameterized integer input and output. In its normal mode of operation, the output value is the mean value of the lowest and highest values encountered in the input since the circuit was reset last.

As an example of the transcription into MONA, we reproduce here a submodule of the high-level specification:

```
description LAST (INT N) (in CLOCK H; in BTM0 E, IN_L[0:N-1];
                          out BREG0 OUT_L[0:N-1])
body
  external MUX_N;
  declare BTM0 E_N[0:N-1], OUT_M[0:N-1];
  use MUX_N MUX(E_N, IN_L, OUT_L, OUT_M);
  relation
    E_N = fan N | E,
    !H! OUT_L <= OUT_M;
enddescription
```

This submodule is parameterized by `N` and declares a clock `H`, a Boolean input signal `E`, a parameterized input `IN_L`, and a parameterized register `OUT_L`. The submodule declares parameterized data-paths named `E_N` and

OUT_M, and it instantiates a multiplexer MUX_N, whose output is wired to OUT_M and whose inputs are E_N (which is specified as the signal E duplicated N times), the parameterized input IN_L, and the current value of the parameterized OUT_L register. The submodule also declares that when the clock H rises, the value OUT_M is latched into the register OUT_L.

The corresponding Mona declaration is

$$\begin{aligned}&\mathit{last}(h, e, \mathit{In\_L}, \mathit{Out\_L}, \mathit{Out\_L\_}) \;\equiv\\ &\quad \exists^2 \mathit{E\_N}, \mathit{Out\_M} : \mathit{mux\_n}(\mathit{E\_N}, \mathit{In\_L}, \mathit{Out\_L}, \mathit{Out\_M})\\ &\quad \wedge\, \mathit{fan}(\mathit{E\_N}, e) \wedge \mathit{if}(h, \mathit{Out\_L\_}, \mathit{Out\_M}, \mathit{Out\_L\_}, \mathit{Out\_L});\end{aligned}$$

where the parameterized register variable OUT_L is modeled by two second-order variables $\mathit{Out\_L}$ and $\mathit{Out\_L\_}$ corresponding to the value before and after a clock tick. Here *mux_n*, *fan*, and *if* are Mona predicates defined elsewhere.

We translate both the circuit description `min_max_low` and the high-level description `min_max_high` in a similar fashion (which can be automated). The one exception is that in the high-level description, the mean value is described in terms of usual addition and division on values of the parameterized datapath viewed as integers. As with the ALU, we have to specify these operations bit-wise. Both descriptions concern four Boolean signals (`h`, `clear`, `reset`, and `enable`), the parameterized input (`In_M`) and output values (`Out_M`), and three parameterized registers (`Pastmax`, `Pastmin`, `Last`).

The equivalence of the two descriptions is established if the Mona formula

$$\begin{aligned}&\mathit{min\_max\_low}(h, \mathit{clear}, \mathit{reset}, \mathit{enable}, \mathit{In\_M}, \mathit{Out\_M},\\ &\qquad \mathit{Pastmax}, \mathit{Pastmax\_}, \mathit{Pastmin}, \mathit{Pastmin\_}, \mathit{Last}, \mathit{Last\_})\\ &\Leftrightarrow \mathit{min\_max\_high}(h, \mathit{clear}, \mathit{reset}, \mathit{enable}, \mathit{In\_M}, \mathit{Out\_M},\\ &\qquad \mathit{Pastmax}, \mathit{Pastmax\_}, \mathit{Pastmin}, \mathit{Pastmin\_}, \mathit{Last}, \mathit{Last\_})\end{aligned}$$

is valid. Mona verifies that this is the case in 10 seconds. The description of the circuit and its specification takes five pages of M2L code.

8 Why does it work?

The complexity of deciding the validity of M2L formulas is determined by the complexity of carrying out the operations that translate formulas to automata. Exponential factors arise in two ways. First, as discussed in §3, the transition function of an automaton is exponential in the number of free variables. This is typically not a problem in practice since BDDs often lead to exponential compression whereby the transition function can be represented in polynomial space. The second source of trouble is that each quantifier requires a projection operation followed by an application of the subset construction to determinize the result. The subset construction can lead to exponentially many more states in an automaton. Formulas with alternating quantifiers require iterating this operation (once for each quantifier alternation) and this is responsible for the non-elementary lower-bound associated with M2L and related logics. In what follows, we look more carefully at these operations and argue why a state explosion rarely happens in practice. Indeed, we show that there are particular syntactic and semantic classes of formula (see also §9) where we can guarantee that a blow-up will not occur.

To illuminate why our approach works in practice, we focus on the *add* predicate defined in Section 4.2:

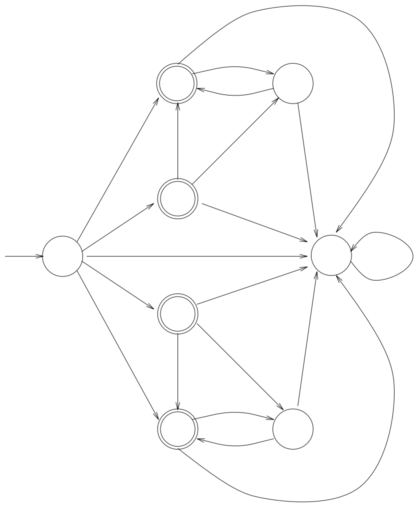
$$
\begin{array}{l}
\mathit{at\_least\_two}(a,b,c) \equiv (a \wedge b) \vee (a \wedge c) \vee (b \wedge c) \\
\mathit{mod\_two}(a,b,c,d) \equiv a \leftrightarrow b \leftrightarrow c \leftrightarrow d
\end{array}
$$

$$
\begin{array}{l}
\mathit{add}(A, B, \mathit{Out}, \mathit{cin}, \mathit{cout}) \ \equiv \\
\quad \exists^2 C : \ \ \$ \geq 0 \ \Rightarrow \\
\qquad\quad (\ \ \forall^1 p : \mathit{mod\_two}(A(p), B(p), C(p), \mathit{Out}(p)) \\
\qquad\qquad\qquad \wedge \ ((p < \$) \rightarrow (C(p \oplus 1) \leftrightarrow \mathit{at\_least\_two}(A(p), B(p), C(p)))) \\
\qquad\quad \wedge \ \ (\mathit{cout} \leftrightarrow \mathit{at\_least\_two}(A(\$), B(\$), C(\$))) \\
\qquad\quad \wedge \ \ C(0) \leftrightarrow \mathit{cin})
\end{array}
$$

Note that we have here added the precondition $\$ \geq 0$ so as to fix the meaning of the formula (to true) for the empty string interpretation; this makes the corresponding automata easier to understand.

A use of second-order quantification

The formula defined by *add* above has the form $\exists^2 C : \phi$. We focus on the computation related to the quantifier $\exists^2 C$, which "guesses" the intermediate carry bits. In theory, the projection and subsequent determinization required to eliminate this quantifier can cause an exponential blow-up in the state space. Here is what happens in practice. The automaton corresponding to the formula ϕ inside the quantifier has 8 states (we have not indicated the 32 BDD nodes of this automaton for the sake of clarity):



The automaton reads a string that defines the interpretations of variables A, B, Out, cin, $cout$ and C. Its shape can be explained as follows. The formula ϕ expresses that each component of the result is the sum of the A and B component and the carry. Thus the automaton counts modulo 2. But it must also remember the value of the carry out $cout$, which can be checked only after the last position has been read. Thus, the automaton has two modulo-2 counters, each having one accepting and one non-accepting state. Since the empty string is always accepted (due to the $\$ \geq 0$ clause), the four different states reached from the initial state upon reading the letter defining the values of the Boolean variables are all accepting. The rightmost state is the one reached in case the carry C or the output Out is wrong at any point. There is no recovery from such an error so this state acts as a sink.

The automaton for $\exists^2 C : \phi$ is obtained by a projection and subset construction that works as follows. Recall that this new automaton reads strings that define A, B, Out, cin, and $cout$, but not C. It must accept if and only if there is some assignment to C that makes the old automaton accept. The first subset constructed is that containing only the initial state. On any transition out of the initial state, another singleton state is reached since the first transition only involves the values of Boolean variables. For any of these four states and any input letter, there are exactly two transitions possible: one to the state that would be reached if the correct value of the carry C was part of the input letter and the sink state corresponding to the situation when C was wrong. Thus, all subsets reached from this point on have exactly two elements: a counting state and the sink state (there is one exception:

the singleton state consisting of the sink state alone is also reachable, for example, if a letter defines the wrong value of Out). As a result, two of the four singleton states reached on the first transition also become two-element states. Thus there are exactly 10 reachable states in the subset automaton.

The arguments above are easily generalized as follows.

Proposition 2 *Let ϕ be a formula of the form $\exists^2 P : \psi(P)$, where P is functionally determined, that is, for any interpretation of the remaining free variables in ψ, there is exactly one interpretation of P making ψ true. Then, the calculation of the subset automaton for ϕ is linear in the size of the automaton for ψ.*

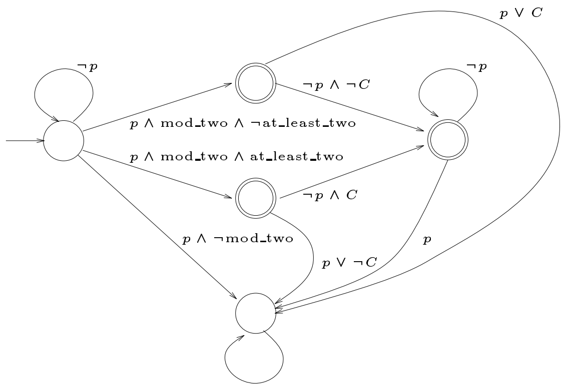
A use of first-order quantification

Recall that each first-order variable is treated as a second-order variable that ranges over a singleton (one element) set. Thus the automaton for $\phi(p_1, \ldots, p_n)$, where $p_1, \ldots, p_n$ are all the free first-order variables in ϕ, recognizes all strings that have exactly one occurrence of a 1 in each p_i-track and that make ϕ true with p_i interpreted by the position of the 1 in the p_i-track.

Returning to the example, we calculate the automaton for $\phi \equiv \forall^1 p : \psi$, where

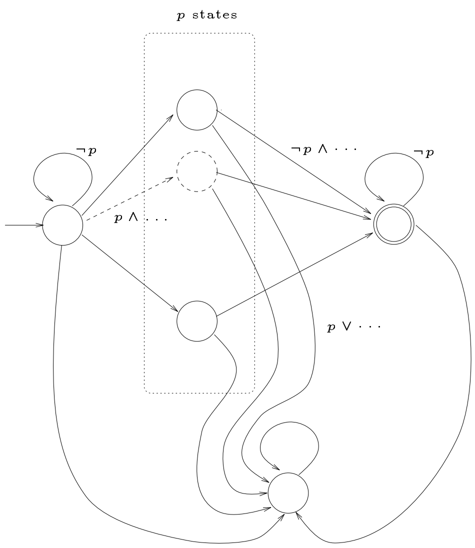
$$\begin{array}{rcl} \psi & \equiv & \mathit{mod\_two}(A(p), B(p), C(p), Out(p)) \wedge \\ & & ((p < \$) \rightarrow (C(p \oplus 1) \leftrightarrow \mathit{at\_least\_two}(A(p), B(p), C(p)))) \end{array}$$

from the automaton for ψ, which looks like:



We have here omitted the initial transition corresponding to the Boolean variables in ψ, since there are none. Intuitively, this automaton waits until it sees the position p; then it either goes to a terminal non-accepting state (if the *mod_two* predicate does not hold at position p), or it branches (if the *mod_two* predicate holds) to a new state that remembers the value of the *at_least_two* predicate at position p. In the latter case, the automaton checks on the next transition, corresponding to position $p+1$, that C has the correct value.

In this example, the subset automaton constructed by projecting out p is also small. (This automaton is constructed from an automaton corresponding to the negation of ψ according to the identity $\forall^1 p : \psi \equiv \neg\exists^1 p : \neg\psi$. The automaton for $\neg\psi$ is the same as the one above, except that accepting and non-accepting states are interchanged and that a few transitions are slightly different.) However, instead of studying the subset construction in detail for the automaton above, we tackle a more general situation. Consider a formula ψ that is (or is equivalent to) a Boolean combination of formulas of the form $p \in X_i$ or $p \leq \$ \Rightarrow p \oplus 1 \in X_i$. Then ψ corresponds to an automaton A that looks like:



This shape is easy to explain: before p occurs, ψ says nothing about any other variable; when p occurs, a new state (inside the dotted box named "p nodes") is reached according to the values of the X_is at p (some of these states may be final, since p might be the last position); and if p is not the last position the truth of ψ is determined by reading the X_is at position $p+1$.

The reachable states of A in the subset construction are those of the form

$$\{s \mid \text{for some } \alpha,\ s \text{ is the state reached when some } p\text{-track is added to } \alpha\},$$

where α determines an interpretation of the X_i. It can be seen that any such set contains at most one state from the box in the figure above. Therefore, we again have an only linear expansion.

Proposition 3 *For a formula of the form $\phi \equiv \exists^1 p : \psi(p, \{X_i\})$, where ψ only mentions p in terms that are of the form p or the form $p \oplus 1$ (where in the latter case, the occurrence is under the provision $p \leq \$$), the calculation of the subset automaton for ϕ is linear in the size of the automaton for ψ.*

This proposition does not directly explain the complexity of the subset construction when there are more than one free first-order variable in the formula. Often, however, the variable that is projected away is tightly constrained by other variables. For example, if we project away the variable z in a formula that contains the clause $x \leq z \leq y$, then the subset construction essentially only explores the situation when $x \leq z \leq y$ holds. Thus, if z is otherwise only used as in the proposition above, we would be able to again establish a linear upper bound.

9 Comparison and Conclusions

Our results constitute a study of automatic verification based on regular classes of circuits. A family of n-bit adders is regular in an informal structural sense (n adders are chained together ripple-carry style), but also in a formal language theoretic sense. Viewing the input/output relation of an n-bit adder as a set of words of length n, we find that the union of the words for $n = 1, 2, \ldots$ is recognizable by a finite-state automaton. The logic of M2L allows us to express regularity in the informal structural sense in a declarative way by stating how an n-bit adder is iteratively built. The decision procedure implemented by MONA reduces analysis of the resulting description of an infinite state space to the analysis of a regular one.

Below we compare our approach with others reported on in the literature.

9.1 Inductive Theorem Proving

Most approaches to reasoning about parameterized systems involve explicit theorem proving: the system is formalized as a recursive (or inductive) definition within a logic like first-order or higher-order logic and explicitly reasoned about by mathematical induction, cf., [1, 3, 8, 11, 15, 17, 18, 20, 24]. For example, to show that a family of circuits C, parameterized by n, with port values given by the vectors $X_1, \ldots X_n$ satisfies a parameterized behavioral specification S, one proves

$$\forall n,\, X_1, \ldots, X_n : C(n, X_1, \ldots, X_n) \rightarrow S(n, X_1, \ldots, X_n)$$

by induction over the parameter n.

The parameterized adder and ALU have been used as test-cases by others in inductive theorem proving, in particular by Cantu *et. al.* using the Edinburgh Clam System [5] and by Cyrluk *et. al.* using PVS [6]. CLAM is a system that generates proofs by induction for a higher-order logic. The development in CLAM of the ALU took over a week and the proof is constructed automatically in 4 minutes and 40 seconds by CLAM, as opposed to 2 seconds by MONA. Their specification shares some similarities to ours, but differs in several important respects. First, they are not limited to specifications expressible within a decidable logic. As a result, they were able to apply their approach to verify circuits such as parameterized multipliers, which cannot be formalized in M2L. Second, they specified the ALU as a recursive function while we specified it as a non-recursive relation. Both are valid representation techniques, but note that we cannot write explicit recursive functions in M2L. On the other hand, if Cantu *et. al.* had formalized the ALU as a recursively defined relation, CLAM would have been unable to construct a proof.[7]

The ALU theorem was also verified using PVS. PVS is a semi-interactive theorem prover that features built-in simplifiers and decision procedures; for example BDDs are used for propositional reasoning. Users can control proof construction by writing proof strategies (similar to tactics in the LCF sense). In [6] the adder and the ALU are verified using the induction, normalization,

[7] To the best of our knowledge, all systems automating proof by mathematical induction reason about recursively specified functions, but not recursively specified relations. Indeed, some provers used for hardware verification, such as NQTHM, are so biased towards functions that they cannot represent hardware specified relationally (e.g., they lack existential quantification).

and BDD features of PVS. The formalization of these circuits is similar to that of Cantu *et. al.* Verification by induction of the parameterized adder is stated to last approximately 2 minutes (as opposed to our time of one second) and their proof of the ALU required 90 seconds, as opposed to 2 seconds in our case.

The signal-processor circuit was verified in NQTHM (the Boyer-Moore theorem prover) and reported on in [25]. The proof required the user to formulate various lemmas. Even with the lemmas, verification required several minutes of CPU time, as opposed to 10 seconds in our case.

These examples suggest that when a parameterized system is formalizable in M2L, then there can be real advantages to our approach. Not only were our verification times typically one to two orders of magnitude faster, but there was no need for search, heuristics, or user interaction. In practice, no theorem proving systems (other than those implementing decision procedures) are fully automatic. Although some systems use powerful heuristics for automating induction (e.g., CLAM, NQTHM, and PVS) or complete proof procedures for semi-decidable logics (e.g., resolution theorem provers like OTTER are typically refutation complete for first-order theories) all such systems require, in practice, user guidance such as suggestion of rewrite rules, lemmas, parameter settings, and the like. This is quite different from our approach where the only possible parameter the user can influence is the variable ordering used in building BDDs. In all our examples, this ordering was picked automatically by Mona.

9.2 Deduction without Induction

An alternative approach to parameterized verification is to fix the parameter to a particular value n. A finite circuit arises that can be analyzed using BDDs. As shown in [23], the circuits that allow BDD representations whose size is linear in n are those with a bounded amount of information flowing through any cross section. Similarly, it is not hard to see that the corresponding parameterized circuit is representable in M2L. The point at which the instantiated description becomes larger than the parameterized description will depend on variable orderings and the chosen representation of automata.

Deduction (without induction) is another possibility for reasoning about non-parameterized circuits: even when BDDs are applicable, one may still prefer to reason within an axiomatic system. An example of this is the 74LS163-counter, which was verified using the OTTER theorem prover in

[22]. In this proof 4 4-bit counters were chained together to form a 16-bit counter, which was verified. Unlike in our work, the proof was partially interactive (simplification rules, equivalences, and the like were initially proven) and there was no induction involved since only a particular case, $n = 16$, was verified.

Although replacing a parameter with a constant may be satisfactory for reasoning about circuits whose size is parameterized, it can lead to incorrect results when reasoning about circuits whose behavior should hold over all instants of time. The problem is that one cannot easily bound how many time instances must be reasoned about to establish correctness; the counter-examples produced in our flip-flop example provide some evidence of the difficulty of this problem. One alternative, discussed above, is to retreat to an undecidable formalism and use induction to explicitly reason about the parameter. Another alternative is to use a decidable temporal logic.

As indicated in §5, both of the above approaches have been pursued in verification of flip-flops. Flip-flops have been laboriously verified interactively in theorem provers based on higher-order logic. In contrast, our fully-automated verification took 2 seconds. A competitive approach is model checking using decidable temporal logics. A temporal logic solution for the flip-flop we analyzed was presented in [30]. Verification took 20 seconds. We have translated the specification given in [30] directly into Mona; our verification time is around 2 seconds—a figure comparable to those of the original solution, since computers are now much faster than in 1989, when [30] was published.

9.3 Combined Induction/Deduction

It is possible to combine induction with non-inductive methods such as decision procedures like Mona or model checkers. In our work, we combined induction and deduction when reasoning about parameterized sequential circuits: an inductive step was performed (which was not formalized in a formal metalogic) to eliminate a parameter (in our case, time) and thereby reduce the problem to one which can be solved by Mona. Such a reduction can be formalized in an interactive theorem proving environment. For example, Kurshan and Lamport combined COSPAN (a model checking system) with TLP (a theorem prover based on Lamport's Temporal Logic of Actions) and used induction to decompose the verification of a parametric multiplier to the verification of 8-bit multipliers, which is then verified automatically [19].

Other researchers have investigated explicit induction principles for reasoning about networks of processes where the base case and the inductive steps are reduced to decidable problems. Such approaches test sufficient conditions for the correctness of the overall system. Kurshan and MacMillan have incorporated reasoning by induction into the COSPAN system [20], which is used to check ω-regular properties of processes; this allowed them to verify safety and liveness properties of a non-trivial version of the Dining Philosophers problem that was parameterized by the number of processes. These ideas have been further extended [26] and similar ideas have been developed in other settings, cf. [31].

9.4 Linearly Inductive Functions

The work closest to ours is that of Gupta and Fisher [12, 13] who, from a rather different starting point, have also developed a BDD-based formalism closely connected to regular languages. They define two classes of inductively defined Boolean functions: Linearly Inductive Functions (LIFs) and Exponentially Inductive Functions (EIFs). Both classes consist of Boolean formulas defined by restricted forms of recursion. For example, the following equations define a family of n-bit adders as two LIFs, one for *sum* and one for *carry*.

$$\begin{array}{lrl}
\text{for } i = 1 & sum_1 = & a_1 \oplus b_1 \oplus cin \\
 & carry_1 = & (a_1 \wedge b_1) \vee ((a_1 \vee b_1) \wedge cin) \\
\text{for } i > 1 & sum_i = & a_i \oplus b_i \oplus carry_{i-1} \\
 & carry_i = & (a_i \wedge b_i) \vee ((a_i \vee b_i) \wedge carry_{i-1})
\end{array}$$

These equations can be expressed in M2L as follows.

$$\begin{array}{l}
add(A, B, Sum, Carry, cin) \;\equiv \\
\quad \forall^1 i : \\
\quad\quad i = 0 \rightarrow \\
\quad\quad\quad Sum(0) \leftrightarrow xor(A(0), xor(B(0), cin) \wedge \\
\quad\quad\quad Carry(0) \leftrightarrow (A(0) \wedge B(0)) \vee ((A(0) \vee B(0)) \wedge cin)) \\
\quad\quad 0 < i \rightarrow \\
\quad\quad\quad Sum(i) \leftrightarrow xor(A(i), xor(B(i), Carry(i \ominus 1))) \wedge \\
\quad\quad\quad Carry(i) \leftrightarrow (A(i) \wedge B(i)) \vee ((A(i) \vee B(i)) \wedge Carry(i \ominus 1))
\end{array}$$

Gupta and Fischer provide algorithms for converting function definitions of this particular form into a *Function Descriptor* (FD) representation. A function descriptor is essentially a state of a BDD-represented automaton (cf. §3.1), but it is associated with two BDDs: a *basis* BDD, which is Boolean-valued BDD followed when the last letter in the string is read, and a *linear inductive BDD*, which is a multi-valued BDD whose value is either a state or a Boolean. A Boolean leaf, which signifies reject or accept, is encountered when the following letters have no significance as to whether the string is accepted—in the usual automaton, this situation corresponds to a looping state.

As shown in [12], the FD representation is in essence an automaton. A precise relation with our framework can be establised as follows:

Proposition 4

1. *For any regular language $L \subseteq \mathbf{B}^k$, the FD representation is isomorphic, modulo a couple of nodes, to the BDD-represented automaton A' for the language L' consisting of all $\alpha \in \mathbf{B}^{k+1}$ such that α projected on the first k tracks is in L and the $k+1$-track is of the form 0^*1. Moreover, the FD representation is linear in the size of an automaton A recognizing L.*

2. *Vice versa, an FD representation can be converted to a BDD-represented automaton with at most a quadratic increase in size.*

Proof: 1) The states in A' have two kinds of transitions: those corresponding to letters with a 1 in the $k+1$-track and those with a 0. All states corresponding to a situation where the 1 in the $k+1$-track has not yet occurred can then be viewed as FDs according to the two kinds of transition, which correspond to the inductive case and the base case, respectively. In addition, to get a proper FD representation, the looping non-accepting state in A' is replaced by a leaf labeled 0. The looping accepting state contains a transition to the looping non-accepting state on a 1 in the $k+1$-track (since no more 0s are expected in this track). This piece of the transition graph is replaced by the leaf with value 1.

To see that the FD description is linear in the original BDD-represented automaton A recognizing L, we note that every state of this automaton can be converted to a (non-reduced) FD descriptor by letting the inductive part be the original transition function and by letting the base part be the BDD that represents the transition function from the state with every leaf replaced by 0 or 1 according to whether the leaf is labeled with an accepting or non-accepting state.

2) The other direction is proven in a similar manner. To go from an FD descriptor to a state with an associated transition BDD, we must make a BDD product of the base case and inductive case BDD of the FD descriptor. The details are omitted. □

The algorithm for translating linearly inductive functions to FD descriptions as described by Fischer and Gupta is based on representing the reverse language. That is, the base case is represented as the last letter in the string. For certain circuits, like shifters, this representation is sometimes exponentially more succinct. Note that the MONA description above can easily be dualized to achieve a representation of the reverse language: simply exchange 0 with \$ and -1 with $+1$. The resulting MONA automaton is then in a relationship with the FD description as explained in the above proposition.

If the FD description is desired as the direct output of the MONA translation, a simple formula for the $k+1$-track in the Proposition above could be easily added so that the automaton A' is calculated. This trick is an instance of padding regular languages to the languages described so that state spaces decrease in size for the padded representations.

The above demonstrates that MONA generalizes the LIF framework as a succinct representation formalism for regular languages. It is also the case that one does not pay a price, from a computational theory point of view, for using MONA to compute automata for LIFs. In particular, any LIF is translated to a formula with a single first-order quantifier (for the parameter i), whose quantifier-free matrix is a Boolean formula built using very limited arithmetic (subtraction by 1, and test against zero). An automaton for the matrix can be computed in exponential time in the worst case using arguments as in §8. This bound is similar to that of Gupta and Fisher, where the the worst-case complexity of their algorithms is doubly-exponential in number of LIF variables (as in our case); [14] does not contain an explicit discussion of the size of the FDs in terms of the input size, but it is not hard to see that this explosion is exponential only. Note that it is an open question as to which approach offers better performance in practice, since the algorithms used to build BDD-represented automata in the two approaches are different.

In the LIF framework multiple automata can be specified at the same time and their representation can be shared; this idea can lead to compact representations that are currently not supported by MONA.

We believe that the MONA approach to specifying hardware is often more natural than the LIF approach since the latter—judging from examples in [12]—sometimes requires substantial amount of reasoning at the meta-level to even see that a circuit can be brought into the form of an LIF. On the other hand, the LIF approach generalizes the Mona approach in that it offers some interesting ways of attacking the problem of simultaneous induction in more than one parameter—something that goes beyond regularity [12, 14].

9.5 Other Approaches to Regularity

Independently of [14], Vuillemin studied relationships between 2-*adic integers* and sequential circuits in [29]. A 2-adic integer is essentially an infinite string of bits that is regarded as a rational number (only certain rational numbers can be represented in this way). In this somewhat abstract setting, Vuillemin showed that synchronous circuits can be synthesized from descriptions in a language named 2Z. The circuits are represented by *Synchronous Decision Diagrams* or *SDDS*, which are essentially equivalent to the function descriptor representation of Gupta and Fisher. Vuillemin did not study algorithmic issues such as minimization of SDDS. In [2], the problem of solving equations involving 2-adic integers was studied, and it was noted that SDDS provide another representation of regular languages.

A different approach to automatic verification based on regularity has been studied by Rho and Somenzi in [27]. They investigate automatic verification of what we called parameterized sequential circuits: networks built by iterating cells where each cell is a finite state transition system. As noted in §6, such systems have multiple parameters and their properties are in general undecidable. Rho and Somenzi show that for certain classes of parameterized systems there are algorithms which can sometimes compute an automaton model of these systems: they infer the automaton model from observations of the systems behavior for $n = 1, 2 \ldots$ until some technical conditions indicate a fixed point. The existence of such a model (*boundedness* in their terminology) is undecidable and their algorithm, when it terminates, provides sufficient conditions for determining simple properties of networks, such as their equivalence.

References

[1] David Basin and Peter DelVecchio. Verification of combinational logic in Nuprl. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Ithaca, New York, 1989. Springer-Verlag.

[2] Wolfram Büttner and Klaus Winkelmann. Equation solving over 2-adic integers and applications to the specification, verification and synthesis of finite state machines. unpublished, 1995.

[3] Albert Camilleri, Mike Gordon, and Tom Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67. Elsevier Science Publishers B. V. (North-Holland), 1987.

[4] Albert John Camilleri. *Executing Behavioural Definitions in Higher Order Logic*. PhD thesis, University of Cambridge, 1988.

[5] Francisco Cantu, Alan Bundy, Alan Smaill, and David Basin. Experiments in automating hardware verification using inductive proof planning. In *Formal Methods in Computer-Aided Design (FMCAD-96)*, Palo Alto, USA, 1996. To appear.

[6] D. Cyrluk, S. Rajan, N. Shankar, and M.K. Srivas. Effective theorem proving for hardware verification. In *Second International Conference On Theorem Proving In Circuit Deisgn: Theory, Practice and Experience*, Bad Herrenalb, Germany, September 1994.

[7] H. De Man D. Verkest, L. Claesen. Special benchmark session on formal system design. In *Proceedings of the IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design, Leuven*, pages 75–76, Nov. 1989.

[8] L. Claesen D. Verkest, P. Johannes and H. De Man. Correctness proofs of parameterized hardware modules in the Cathedral-II synthesis environment. In *Proceedings of the European Design Automation Conference*. IEEE Computer Society Press, 1990.

[9] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. MIT Press/Elsevier, 1990.

[10] *Databook of Analog and Synchronous Components.* Fairchild, 1993.

[11] Michael J. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design.* North-Holland, 1986.

[12] Aarti Gupta. *Inductive Boolean function manipulation.* PhD thesis, Carnegie Mellon University, 1994. CMU-CS-94-208.

[13] Aarti Gupta and Allan L. Fisher. Parametric circuit representation using inductive boolean functions. In C. Courcoubetis, editor, *Computer Aided Verification, CAV' 93, LNCS 697*, pages 15–26. Springer Verlag, 1993.

[14] Aarti Gupta and Allan L. Fisher. Representation and symbolic manipulation of linearly inductive boolean functions. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 192–199. IEEE Computer Society Press, 1993.

[15] F.K. Hanna and N. Daeche. Specification and verification using higher-order logic: a case study. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 179–213. Elsevier Science Publishers B.V., 1986.

[16] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1996.

[17] Warren Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[18] Warren J. Hunt. The mechanical verification of a microprocessor design. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 89–129. Elsevier Science Publishers B. V. (North-Holland), 1987.

[19] Robert Kurshan and Leslie Lamport. Verification of a multiplier: 64 bits and beyond. In *Proceedings of the Conference on Computer-Aided Verification*, LNCS 697, pages 166–179. Springer-Verlag, June 1993.

[20] Robert Kurshan and Ken McMillan. A structural induction theorem for processes. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 239–247. ACM Press, 1989.

[21] M. Morris Mano. *Digital logic and computer design.* Prentice-Hall, Inc., 1979.

[22] T. Margaria. Hierarchical mixed-mode verification of complex FSMs described at the RT level. In *Proceedings of the International Conference on "Theorem Provers in Circuit Design", Nijmegen, IFIP Transactions A-10*, pages 59–76. North-Holland, 1992.

[23] Ken McMillan. *Symbolic Model Checking.* Kluwer, 1993.

[24] T. F. Melham. Using recursive types of reasoning about hardware in higher order logic. In *International Working Conference on The Fusion of Hardware Design and Verification*, pages 26–49, July 1988.

[25] L. Pierre. From a HDL description to formal proof systems: Principles and mechanization. In *Proceedings of CHDL'91, Marseille France*, pages 21–42. IFIP Transactions, North-Holland, April 1991.

[26] A. Orda R. Kurshan, M. Merritt and S. Sachs. A structural linearization principle for processes. In *Proceedings of the Conference on Computer-Aided Verification*, LNCS 697, pages 491–504. Springer-Verlag, June 1993.

[27] J.-K. Rho. and F. Somenzi. Automatic generation of network invariants for the verification of iterative sequential networks. In C. Courcoubetis, editor, *Computer Aided Verification, CAV' 93, LNCS 697*, pages 123–137. Springer Verlag, 1993.

[28] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.

[29] Jean E. Vuillemin. On circuits and numbers. *IEEE Transactions on Computers*, 43(8):868–879, 1994.

[30] A. Wilk and A. Pnueli. Specification and verification of VLSI systems. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 460–463, 1989.

[31] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 68–80. Springer-Verlag, June 1989.

Appendix A: The Parameterized Adder

The following is the entire script given to Mona for the parameterized adder. It includes the formalization of gates, the description of the implementation of a full 1-bit adder and parameterized adder, and the behavioral specification described in section 4. The only differences are small notational ones. Definitions are preceded by the keyword `pred` and type declarations are given. Concrete syntax is used for operators, e.g., `&` for conjuntion.

```
pred and(var0 a,b,o) = o <=> (a & b);
pred or(var0 a,b,o) = o <=> (a | b);
pred xor(var0 a,b,o) = o <=> ((~a & b) | (a & ~b));

pred full_adder(var0 a,b,out,cin,cout) =
     ex0 w1, w2, w3 : xor(a,b,w1) & xor(w1,cin,out) & and(a,b,w2) &
                      and(cin,w1,w3) & or(w3,w2,cout);

pred at_least_two(var0 a,b,c) = (a & b) | (a & c) | (b & c);
pred mod_two(var0 a,b,c,d) = (a <=> b <=> c <=> d);

pred add(var2 A,B,Result, var0 cin, cout) =
        ex2 C: (all1 p: mod_two(p in A,p in B,p in C,p in Result)
     & ((p < $) => ((p +o 1 in C) <=> at_least_two(p in A, p in B, p in C))))
     & (cout <=> at_least_two($ in A, $ in B, $ in C))
     & (0 in C1 <=> cin);

pred n_bit_adder(var2 A, B, Out, var0 cin, cout) =
     ex2 C, D:
        (all1 p: full_adder(p in A, p in B, p in Out, p in C, p in D))
      & (all1 p: (p < $) => (p in D <=> (p +o 1 in C)))
      & (0 in C <=> cin)
      & ($ in D <=> cout);
```

```
all2 A, B, Out: all0 cin, cout:
  (add(A,B,Out,cin,cout)  <=> n_bit_adder(A,B,Out,cin,cout));
```

The last line is the theorem to be proven. MONA responds:

```
Formula with free variables is a tautology
The total time elapsed: 0:00:01
```

Appendix B: The 74LS163 4-bit counter

Below is the MONA specification of the the 4-bit counter given as a relation over its external ports.

```
count4bit(pen, cep,cet,srn,tc,i0,i1,i2,i3,
          o0,o1,o2,o3,oo0,oo1,oo2,oo3) =
  (Ex pe, ep, pen1, et:                 # set combinations of switch values
     (not(pen,pe)  & and3(pen,cep,cet,ep) & and(cep,cet,et)
      & not(pe,pen1)) &

  (Ex ga0, gb0, gc0, gi0, h0 :          # set first output
      and(pe,i0,gi0) &  not(et,h0) &
      and(~oo0,ep,ga0) &  and3(h0,pen1,oo0,gb0) &
      or3(ga0, gi0, gb0,gc0) & and(srn, gc0, o0))  &

  (Ex ga1, gb1, gc1, gi1, h1:           # set second output
      and(pe,i1,gi1) &  nand(oo0,et,h1) &
      and3(~oo1,oo0,ep,ga1) &  and3(h1,pen1,oo1,gb1) &
      or3(ga1, gi1, gb1,gc1) & and(srn, gc1, o1))  &

  (Ex ga2, gb2,gc2, gi2, h2:            # set third output
      and(pe,i2,gi2) &  nand3(oo0,oo1,et,h2) &
      and4(~oo2,oo0,oo1,ep,ga2) & and3(h2,pen1,oo2,gb2) &
      or3(ga2, gi2, gb2,gc2) & and(srn, gc2, o2))  &

  (Ex ga3, gb3, gc3, gi3, h3:           # set fourth output
      and(pe,i3,gi3) & nand4(oo0,oo1,oo2,et,h3) &
      and5(~oo3,oo0,oo1,oo2,ep,ga3) & and3(h3,pen1,oo3,gb3) &
```

```
    or3(ga3, gi3, gb3,gc3) & and(srn, gc3, o3))  &

(Ex w0, w1, w2, w3:                     # set tc
    not(~oo0,w0) &  not(~oo1,w1) & not(~oo2,w2) &
    not(~oo3,w3) &  and5(w0,w1,w2,w3,cet,tc)));
```