Structural and Behavioral Modeling with Monadic Logics

Abdelwaheb Ayari, David Basin and Stefan Friedrich
Institut für Informatik, Albert-Ludwigs-Universität Freiburg,
Universitätsgelände Flugplatz, 79110 Freiburg i. Br., Germany.

Abstract

Logic offers the possibility of modeling and reasoning about hardware and software. But which logic? We propose monadic logics of strings and trees as good candidates for many kinds of discrete systems. These logics are natural, decidable, yet substantially more expressive, extensions of Boolean logic. We motivate their applicability through examples.

1 Introduction

Transistors, pins, and power supplies are the stuff from which computers are made. However, this view is not very helpful for understanding what computers actually do. For this we require a language for modeling the structure and behavior of systems and for relating these models, i.e., for showing that the system modeled has the intended behavior. This requirement is fulfilled by logic.

Boolean logic is the traditional logic of switching theory. It is often applied to model and reason about discrete systems. From the engineering perspective, it is a logic for expressing operations on bits. For instance, a combinational circuit may be represented in Boolean logic by a formula whose syntax reflects its structure, e.g., an *and*-gate in the circuit corresponds to a logical and-operation. Moreover, one can model the circuit's behavior in the logic and verify that the two models are equivalent. When the models are accurate, the result predicts the behavior of actual circuits.

The primacy of Boolean logic for such tasks is understandable. It is conceptually simple and adequate for modeling the structure and behavior of many kinds of discrete systems. Moreover, verification is decidable and, using data-structures like BDDs, many problems have acceptable running times. However there are drawbacks. Certain kinds of systems, even when discrete (or viewed as such) cannot be modeled in Boolean logic. Moreover, even when they can, the Boolean model may differ substantially from an engineer's model, and this makes modeling error prone. A solution is to use more expressive logics like first-order logic, trading increased expressiveness for the loss of decidability. Here we propose a middle way, namely, decidable monadic second-order logics of strings and trees [5, 8, 14, 17].

We motivate this proposal by examining the potential and limitations of structural and behavioral modeling in a progression of logics, from Boolean logic to extensions with quantifiers and variables ranging over strings and trees of Booleans. The string and tree logics may seem exotic, but they are natural and significant extensions of Boolean logic and have a much wider range of applications. Moreover, like Boolean logic, they are decidable and hence offer the possibility of automated verification. Thus, although monadic logics may not challenge the primacy of Boolean logic, we believe that they are well-suited for reasoning about discrete systems and deserve a place in the repertoire of engineers working with such systems.

Although the use of the monadic logic of strings in modeling parameterized combinational and sequential systems has already been established (see [3, 4]), the exposition here is from a different perspective, centered around exploring tradeoffs and providing support for the above proposal. In addition, the use of a monadic logic of trees for hardware modeling and verification is new and presents a new approach to reasoning about hierarchical, tree-structured systems.

2 Boolean Logic

We begin with Boolean logic and its application to structural and behavioral modeling. Although this logic is adequate for many problems, it has limitations, which motivate generalizations.

The *formulae* of Boolean logic are built from *variables* from a set $\mathcal{V} \equiv \{x_1, x_2, \dots\}$ and *functions* (also called *connectives*) like $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$, and $\neg$. Formulae are interpreted in the set $\mathbb{B} \equiv \{0, 1\}$ of *truth values.*

An *evaluation* $\sigma : \mathcal{V} \to \mathbb{B}$ is a mapping from variables to truth values that is homomorphically extended to formulae. When $\sigma(\varphi) = 1$, for some σ, we say that φ is *true under* σ and that φ is *satisfiable*. Since the satisfiability of $\varphi(x_1, \ldots, x_n)$[1] depends only on the assignments for the variables $x_1, \ldots, x_n$, we can represent an assignment as a tuple $\bar{b} \equiv (b_1, \ldots, b_n)$, for $b_i \in \mathbb{B}$, which we call a *Boolean model*. A formula is a *tautology* (*contradiction*) if it is true in all (no) models.

For example, the formula $x_1 \oplus x_2$ defined by $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$ semantically defines a function that is true in the models $(1, 0)$ and $(0, 1)$. Alternatively, since a function on $\mathbb{B}^n \to \mathbb{B}$ can also be viewed as a relation on $\mathbb{B}^n$, $\oplus$ defines the relation $\{(1, 0), (0, 1)\}$. As the standard Boolean connectives are *functionally complete*, every relation over $\mathbb{B}^n$ can be defined by a formula in the logic.

2.1 Example: Gate Level Modeling

Boolean logic can be used to model discrete systems. This is best seen with the example of combinational circuits. Structurally, we can model a circuit C with inputs $i_1, \ldots, i_n$ and outputs $o_1, \ldots, o_m$ by defining m functions on $\mathbb{B}^n \to \mathbb{B}$ that model the computations required to compute each output. By this we mean that the structure of the circuit is reflected in the functions used to compute it, e.g., that each and-gate in the circuit corresponds to an $\wedge$-connective in the function definition.

Note that when we talk about modeling systems, our use of the term *model* is different from the previous semantic use (Boolean model). In the terminology of formal methods practitioners, a model is a construct or mathematical object that describes a system or presents a theory that accounts for properties of a system. Of course, these two uses are connected: a structural or behavioral model has a semantics given by the Boolean model. The role of logic then is to connect these two. By proving that different structural and behavioral models are equivalent, we show that they have the same Boolean models.

Consider a simple example. Figure 1 describes a circuit for a 1-bit full-adder. The two outputs, *sum* and *cout*, are functions of the inputs a, b, and cin. We can model the structure of this circuit in Boolean logic by formulae that reflect how *sum* and *cout* are computed.

$$
\begin{aligned}
sum(a, b, cin) &\equiv (a \oplus b) \oplus cin \\
cout(a, b, cin) &\equiv (cin \wedge (a \oplus b)) \vee (a \wedge b)
\end{aligned}
$$

[1] i.e., φ has $x_1, \ldots, x_n$ as variables. For logics with quantifiers, we mean the *free* variables of φ.

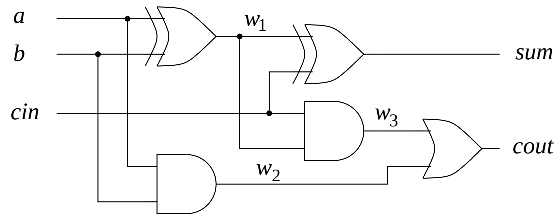


Figure 1. 1-bit full adder

In this structural model, each gate corresponds to a Boolean function and circuit composition is modeled by function composition.

The semantics of these formulae *are* a model of the behavior of the circuit. However, it is often useful to formalize a behavioral model *within* Boolean logic so that we can establish theorems about the equivalence of different structural and behavioral models. For example, we can model the behavior of $sum(a, b, cin)$ by a formula $sum\_beh(a, b, cin)$ that enumerates the inputs for which the output bit *sum* is set (*cout_beh* is defined analogously), i.e.,

$$
\begin{aligned}
&(a \wedge \neg b \wedge \neg cin) \vee (\neg a \wedge b \wedge \neg cin) \vee \\
&(\neg a \wedge \neg b \wedge cin) \vee (a \wedge b \wedge cin) .
\end{aligned}
$$

Although *sum_beh* is logically equivalent to *sum*, it is perhaps more perspicuous as a behavioral description since it amounts to a tabular description of the function computed. Note that the functional completeness of Boolean logic means that every Boolean function can be described by a formula; so Boolean logic is adequate for this kind of behavioral modeling.

Given a formalization of both the structure and the behavior of a circuit in Boolean logic, we can use a decision procedure to validate that the two formalizations are logically equivalent, and thus define identical functions semantically. In this case we verify that

$$
\begin{aligned}
&(sum(a, b, cin) \leftrightarrow sum\_beh(a, b, cin)) \wedge \\
&(cout(a, b, cin) \leftrightarrow cout\_beh(a, b, cin)) .
\end{aligned}
$$

2.2 Limitations

The above example suggests that, for certain kinds of discrete systems, namely combinational circuits, Boolean logic is adequate to model behavior and structure and to demonstrate their equivalence. However, even for such simple systems, there are inadequacies. In particular, the connection between the model (the formula) and what is modeled (the circuit) may be less direct than we would like. For example, although each gate in the circuit corresponds to a function in the formula, shared components must be duplicated in the model. So the corresponding formula can be significantly larger and the relationship between subcircuits and subformulae obscured. Moreover, circuits must be modeled as functions when in practice some are better modeled as relations, e.g., bi-directional circuits.

3 Quantified Boolean Logic

A simple extension to Boolean logic is to add the *Boolean quantifiers* $\forall$ and $\exists$; the result is called *quantified Boolean logic* (or QBL). In the semantics, quantifiers are interpreted as ranging over $\mathbb{B}$. The remaining details are identical to Boolean logic.

It follows semantically that we can eliminate quantifiers: in any model $(b_1, \dots, b_n) \in \mathbb{B}^n$, $\exists x_{n+1}.\, \varphi(x_1, \dots, x_n, x_{n+1})$ is semantically equivalent to $\varphi(x_1, \dots, x_n, 0) \vee \varphi(x_1, \dots, x_n, 1)$. Replacing $\vee$ with $\wedge$ yields a similar equivalence for $\forall$. We can eliminate quantifiers syntactically too: if we replace 0 (respectively 1) in the above expansion by any suitable contradiction (respectively tautology), then we can use the resulting equivalence to eliminate the Boolean quantifiers in any QBL formula. The resulting formula though may be exponentially larger. This difficulty is reflected in complexity of QBL: it is PSPACE-complete. So, provided that $P \neq NP$, theorem proving for QBL is harder than for Boolean logic.

Although quantifiers do not give QBL any more expressive power than Boolean logic, there is a sense in which QBL is more powerful: some Boolean functions can be expressed more succinctly and, moreover, QBL formulae can better capture the structure of the system modeled. The next example illustrates this.

3.1 Example: Gate Level Modeling

In QBL, we can formalize circuits as relations over their external ports. Circuits are built from relations representing primitives, such as transistors or gates, and are combined with conjunction and 'wired together' using existential quantification. This style of representation is standard in the theorem proving community and scales well to complex systems [6].

For example, to model the adder given in Figure 1 we begin by defining the following gates.

$$\begin{aligned}
\mathsf{and}(a, b, o) &\equiv o \leftrightarrow (a \wedge b) \\
\mathsf{or}(a, b, o) &\equiv o \leftrightarrow (a \vee b) \\
\mathsf{xor}(a, b, o) &\equiv o \leftrightarrow ((a \wedge \neg b) \vee (\neg a \wedge b))
\end{aligned}$$

We then compose these to model the adder. The top half of the adder consists of two *xor*-gates, connected by an internal wire w_1, which compute the sum bit *out*. The bottom half uses the internal wire w_1, as well as the two inputs a and b, to compute the carry-out bit *cout*. Our definition conjoins the gate descriptions and projects away the internal wires:

$$\begin{aligned}
&\mathsf{full_adder}(a, b, out, cin, cout) \equiv \exists w_1, w_2, w_3. \\
&\quad \mathsf{xor}(a, b, w_1) \wedge \mathsf{xor}(w_1, cin, out) \wedge \\
&\quad \mathsf{and}(a, b, w_2) \wedge \mathsf{and}(cin, w_1, w_3) \wedge \mathsf{or}(w_3, w_2, cout)
\end{aligned}$$

This formula is a better structural model of the circuit pictured in Figure 1 than the previous one in Boolean logic. Of course, it is still only a model and it abstracts away aspects of actual circuits such as layout. This is acceptable when we want to reason about functional behavior at the level of bits, but unacceptable for other properties, e.g., delay. Modeling other aspects requires either formalizing them in QBL or a richer logic.

3.2 Limitations

Although QBL offers advantages over Boolean logic, many kinds of discrete systems cannot be modeled in it. In our example, we modeled the *static behavior* of a system, i.e., the relationship between its inputs and outputs, which constitute the *system state*. However, it is difficult to model *dynamic behavior*, that is, how the state changes over time.

We can only approximate this. Namely, rather than formalizing a predicate φ of the state variables $\overline{x}$, we instead formalize a Boolean 'before-after' predicate $\varphi'(\overline{x}, \overline{x}')$ that contains a 'primed' version x' of each state variable x and formalizes how the state changes over one unit of time. In this way we can formalize dynamic behavior over any fixed number of time intervals; however formalizing behavior over arbitrary time intervals is impossible.

A second limitation is that these logics cannot be used to model families of related systems. For example, our 1-bit adder specifications can be generalized to adders over 2, 3, ... bit words. Hardware description languages such as VHDL [1] typically provide some kind of parameterization mechanism which, combined with iteration, allows parametric descriptions. Again, this is impossible using the weak logics that we have so far considered.

It turns out that a generalization of the approximation idea described above can be used, in many cases, to solve both problems. Rather than writing a formula $\varphi(\overline{x})$ with models $\overline{b} \in \mathbb{B}^n$, we instead let $\overline{x}$ range over sequences of Boolean models $\overline{b}(0)\overline{b}(1)\dots\overline{b}(m)$ of arbitrary, but finite, length. That is, a model is a string that states how the values (Boolean model) of the n

state variables evolve over m time intervals. Alternatively, the model can represent the behavior of a system with n state variables, each ranging over values of an arbitrary 'length' m. This generalization requires generalizing QBL, which is a logic of bits, to a logic of bit sequences, or strings.

4 Monadic Logic of Strings

We present here a monadic logic of strings called WS1S. Work on WS1S and related monadic theories goes back to Büchi [5], Elgot [8], and Trakhtenbrot [18]. We briefly review syntax and semantics.

Let x and X range over disjoint sets $\mathcal{V}_1$ and $\mathcal{V}_2$ of first and second-order variables. The language of WS1S is described by the following grammar.

$$\begin{array}{lcl} t & ::= & x \mid \mathsf{0} \mid \mathsf{s}(t) \\ \varphi & ::= & X(t) \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x.\, \varphi \mid \exists X.\, \varphi \end{array}$$

Hence terms are built from first-order variables, the constant $\mathsf{0}$, and the successor symbol. Formulae are built from atoms $X(t)$ and are closed under conjunction, negation, and quantification over first and second-order variables. Other connectives and quantifiers can be defined using standard classical equivalences, e.g., $\forall x.\, \varphi \equiv \neg\exists x.\, \neg\varphi$.

Semantically, WS1S formulae are interpreted in $\mathbb{N}$. $\mathsf{0}$ and s are zero and the successor function and $X(t)$ is true when the number denoted by t is in the set of numbers denoted by X. First-order quantification is quantification over natural numbers, whereas second-order quantification is quantification over *finite* sets of natural numbers.

WS1S stands for the *weak second-order monadic theory of one successor*. It is monadic as all atoms are unary predicates. It is weak because second-order variables are interpreted over finite (instead of arbitrary subsets) of the domain. And it has only a single successor function.

Although WS1S is a logic of numbers and sets, we motivated it as a logic of strings, as this emphasizes that WS1S is a natural extension of (quantified) Boolean logic.[2] The relationship is as follows. Any finite string $b(0)b(1)\ldots b(m)$ over $\mathbb{B}$ encodes a finite set of positions, namely $\{p \in \{0, \ldots, m\} \mid b(p) = 1\}$. More generally, we can encode n strings over $\mathbb{B}$ as a single string over $\mathbb{B}^n$. Hence, if $\varphi(\overline{X})$ is a formula whose free second-order variables are $\overline{X} \equiv X_1, \ldots, X_n$, a WS1S *model* can be encoded by a finite string over the alphabet $\mathbb{B}^n$.

As a simple example, the formula $\varphi(X, Y)$ given by

$$X(\mathsf{0}) \wedge \forall p.\, (X(p) \leftrightarrow Y(\mathsf{s}(p)))$$

states that the string X begins with a 1 and that Y equals X shifted one position to the right. A model for this formula is a string $\overline{b}(0)\overline{b}(1)\ldots\overline{b}(m)$ where each $\overline{b}(i)$ is a letter in $\mathbb{B}^2$. To visualize this, we write letters (b_1, b_2) vertically; the first row encoded in the string determines an interpretation for X, and the second for Y. Two such models are

$$\begin{array}{c|c|c|c|c|} X & 1 & 0 & 1 & 0 \\ Y & 1 & 1 & 0 & 1 \end{array} \quad \text{and} \quad \begin{array}{c|c|c|c|} X & 0 & 1 & 0 \\ Y & 0 & 0 & 0 \end{array}.$$

The first model, for example, interprets X as $\{0, 2\}$ and Y as $\{0, 1, 3\}$. φ is true in the first model and false in the second.

Since models are strings, the set of models that satisfy a formula constitutes a language. A fundamental theorem for WS1S is that this logic precisely captures the regular languages: the language associated with each WS1S formula φ is regular, and vice versa. This is the key to decidability. Using automata theoretic techniques, given a formula φ, we can construct an automaton that recognizes the models that satisfy φ. A formula $\varphi(\overline{X})$ is a tautology when the corresponding automaton accepts the universal language on $\mathbb{B}^n$. The 'vice versa' part of this theorem gives us an analog of functional completeness for WS1S: we can model the behavior of any system whose behavior is regular in the language theoretic sense.

WS1S is substantially more expressive than QBL, but this comes at a high price. WS1S is nonelementary decidable [12]: a formula of size n may require time and space bounded below by an iterated stack of exponentials whose height is proportional to n.

4.1 Tools for WS1S

Despite the high complexity of WS1S, several groups have implemented decision procedures for it [10, 11, 13] that work surprisingly well on many nontrivial problems. All examples in this paper have been run in the MONA system [10], which implements a decision procedure for WS1S and its generalization to trees considered in the following section.

MONA offers various kinds of syntactic sugar that are useful in large specifications, which we employ in our examples. For example, first-order variables are actually defined in terms of second-order variables (where a first-order variable is interpreted by a string with a

[2] This view is also taken by Thomas in [16] where Boolean logic is used to motivate a closely related monadic logic of strings, which he calls Büchi-Elgot-Trakhtenbrot logic.

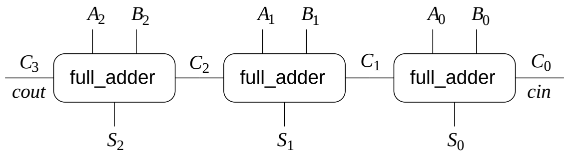


Figure 2. An n-bit adder instance, for $n = 3$

1 in only a single position, giving the value of the first-order term). Hence formulae may have free variables of either order. We also employ sugared notation for QBL, which is definable in WS1S by reasoning about the 0th position of second-order variables. For example, the QBL formula $\forall p.\, \exists q.\, p \leftrightarrow q$ can be defined by the WS1S formula $\forall P.\, \exists Q.\, P(0) \leftrightarrow Q(0)$. MONA supports the use of Boolean variables and Boolean quantifiers. To distinguish between Boolean, first, and second-order variables we use the following naming convention: Second-order variables are capitalized, first-order variables are lower-case letters with names like i, p, and q, and Booleans are lower-case strings like a, b, cin and $cout$. Moreover, we superscript quantifiers with either a 0, 1, or 2 to indicate Boolean, first, and second-order quantification respectively.[3] Finally from the minimal syntax of WS1S we have given, one can define additional operators with their expected meaning, e.g., comparison operators on terms like $=$ and $\leq$. We use such operators freely.

4.2 Example: Parameterized Ripple-Carry Adder

Here we give a simple example of how WS1S can be used to structurally and behaviorally model a parameterized family of circuits. For more sophisticated examples, as well as examples of modeling dynamic (sequential) systems, see [4].

Let us start with a structural model. We have previously shown how to model a 1-bit adder. Figure 2 suggests how an adder can be iterated, in a 'ripple-carry' way, to construct an n-bit adder, for $n = 3$. In general, we can model an n-bit adder by wiring together n 1-bit adders where the carry-out of the ith adder is the carry-in of the $i{+}1$st. The first carry has the value of the carry-in and the last has the value of the carry-out.

It is easy to formalize a ripple-carry architecture by a WS1S formula. If we use C to represent the sequence of carries, we can formalize the general case by the following formula, which relates three strings (the inputs A and B and the output S) and two Booleans (the carry-in cin and carry-out $cout$); the number of bits added is given by the parameter n.

$$
\begin{array}{l}
\mathsf{adder}(n, A, B, S, cin, cout) \equiv \\
\quad \exists^2 C.\, (C(0) \leftrightarrow cin) \wedge (C(n) \leftrightarrow cout) \wedge \\
\qquad \forall^1 p.\, p < n \rightarrow \\
\qquad\quad \mathsf{full\_adder}(A(p), B(p), S(p), C(p), C(p+1))
\end{array}
$$

This is a direct formalization of our natural language description where quantification over positions p formalizes iteration (generalized conjunction) over the n 1-bit adders. Note that given the encoding of Boolean logic in WS1S, we can use the previously given structural model of a 1-bit full adder.

Let us now turn our attention to a behavioral model in which we state how strings, representing n-bit binary numbers, are added. WS1S is a logic of strings and arithmetic must be encoded within its limited language. We model addition by formalizing the standard algorithm for adding base-two numbers: 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 are set. Since we specify the behavior of an n-bit adder, we must restrict the addition modulo 2^n and compute the carry values as special cases.

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

$$
\begin{array}{l}
\mathsf{adder\_beh}(n, A, B, S, cin, cout) \equiv \\
\quad \exists^2 C.\, (C(0) \leftrightarrow cin) \wedge (C(n) \leftrightarrow cout) \\
\qquad \forall^1 p.\, p < n \rightarrow \\
\qquad\quad \mathsf{at\_least\_two}(A(p), B(p), C(p), C(p+1)) \\
\qquad\quad \mathsf{mod\_two}(A(p), B(p), S(p), C(p))
\end{array}
$$

This specifies a language over $\mathbb{B}^6$, which encodes interpretations for n, A, B, S, cin and $cout$. For example, one string in this language is:

	0	1	2	3	4
n	0	0	0	0	1
A	1	1	0	0	0
B	1	0	0	1	0
S	1	0	1	1	0
cin	1	0	0	0	0
$cout$	0	0	0	0	0

(1)

Recall that in the interpreting string for first-order variables like n, exactly the nth bit is 1. Moreover, cin and $cout$ are true iff the 0th bit of their interpreting string is 1. Hence the above encodes that the bit-width $n = 4$. Reading Boolean strings as binary numbers from left to right (up to the nth position) the next three lines encode $A = 3$, $B = 9$, and $S = 13$. Since the carry-in

[3] Any remaining ambiguity (indeed, even without any of these conventions) can be resolved from context.

is set and the carry-out isn't, this string does is indeed in the addition relation modeled.

The adder's structural and behavioral models are similar in that they both iterate components that add a single digit. Using similar models may be natural, but it runs the risk of making similar mistakes in both.

An alternative behavioral model that avoids this problem is based on the equation

$$\begin{aligned} val(n,S) + 2^n vl(cout) = \\ val(n,A) + val(n,B) + vl(cin)\,, \end{aligned} \tag{2}$$

where $val(n,S)$ computes the value of a string S as an n-bit word, and vl converts a Boolean to a number. With this model we define once and for all an encoding of natural numbers as strings, together with predicates encoding (Presburger) arithmetic operations. This encoding of arithmetic can be reused when specifying the behavior of other circuits, like counters, comparators and the like.

$$\begin{aligned}
\mathsf{val}(n,X,Y) &\equiv \forall^1 p.\, Y(p) \leftrightarrow (p < n \wedge X(p)) \\
\mathsf{powof2}(n,b,X) &\equiv \forall^1 p.\, X(p) \leftrightarrow (p = n \wedge b) \\
\mathsf{vl}(b,N) &\equiv \forall^1 p.\, N(p) \leftrightarrow (p = 0 \wedge b) \\
\mathsf{add}(A,B,S) &\equiv \exists^2 C.\, \neg C(0) \wedge \\
&\forall^1 p.\, \mathsf{at_least_two}(A(p),B(p),C(p),C(p+1)) \\
&\quad \wedge \mathsf{mod_two}(A(p),B(p),S(p),C(p))
\end{aligned}$$

We now directly encode (2) as

$$\begin{aligned}
&\mathsf{adder_beh}(n,A,B,S,cin,cout) \equiv \\
&\exists^2 A',B',S',CIN,COUT,U,V,W. \\
&\qquad \mathsf{vl}(cin,CIN) \wedge \mathsf{powof2}(n,cout,COUT) \\
&\quad \wedge \mathsf{val}(n,A,A') \wedge \mathsf{val}(n,B,B') \\
&\quad \wedge \mathsf{val}(n,S,S') \\
&\quad \wedge \mathsf{add}(S',COUT,U) \wedge \mathsf{add}(A',B',V) \\
&\quad \wedge \mathsf{add}(V,CIN,W) \wedge U = W\,.
\end{aligned}$$

4.3 Verification

We now have both a structural model and two equivalent (and this equivalence has been automatically checked by MONA), but differently motivated, behavioral models for a family of n-bit adders. We can formalize the correctness of the structural model, with respect to either behavioral model, as follows.

$$\begin{aligned}
&\forall^1 n.\, \forall^2 A,B,S.\, \forall^0 cin, cout. \\
&\qquad \mathsf{adder}(n,A,B,S,cin,cout) \leftrightarrow \\
&\qquad \mathsf{adder_beh}(n,A,B,S,cin,cout)
\end{aligned}$$

This is verified by MONA in under a second on a Sun Ultra-Sparc workstation. This tells us that any adder in this family, e.g., a 64-bit adder, will operate as expected, provided the gates it is built from function as specified. Such a parameterized statement cannot be expressed in QBL.

4.4 Limitations

Any structural or behavioral model formalized in WS1S defines a regular language over a fixed alphabet $\mathbb{B}^n$. This limits what can be modeled.

To begin with, observe that there is an asymmetry: given a formula, the alphabet size and thus the 'width' of any string, is fixed, while the length of any string is finite, but a priori unbounded. As motivated in Section 3.2, we can use the one 'unbounded dimension' to model the dynamic (sequential) behavior of systems or, alternatively, families of related systems, e.g., parameterized by their bus-width. However, we cannot overcome this asymmetry and model systems where both the width and length dimension are unbounded, i.e., problems that can be modeled on two-dimensional grids instead of one-dimensional strings.[4] As a result, many practical problems concerning discrete systems cannot be modeled. For example, we cannot reason about families of sequential systems such as n-bit registers or counters.

Furthermore, we can only model systems with regular behavior. A parameterized adder defines a regular language; the intuition for this is that an automaton can read a string of bits like that in (1) and it need only propagate bounded information across each letter (e.g., information about the current carry) to determine if a string is in the language. However not all parameterized systems have this property. For example, the language associated with a family of n-bit multipliers is not regular.

Despite these limitations, WS1S is quite expressive. Since every regular language can be represented by a WS1S formula, any discrete system whose behavior is regular, or that can be structurally modeled as a finite state transition system, can be formalized in WS1S. And this covers many systems considered in practice. Furthermore, in some cases, we can model two-dimensional systems that use the second dimension in a limited way. For example, those systems that correspond not to problems on grids, but to problems on trees. Such systems play an important role, e.g., in designing efficient circuits using divide and conquer techniques.

5 Monadic Logic of Trees

In Section 4 we modeled an n-bit ripple-carry adder by cascading n adders. This is correct although inefficient since a delay, linear in n, is required to propagate

[4] Modeling problems on grids requires a stronger logic, e.g., with dyadic predicates. This leads to undecidability as one can encode problems about Turing machine computations in them.

the carry bits between stages. An alternative design is a tree-structured circuit with a logarithmic delay.

We cannot model tree-like systems using a logic of strings. Fortunately, there is a simple generalization that suffices. Rather than having a single successor for the next position in the string, we have two successors, for 'left' and 'right'. Then, just as finite sets of positions over one successor are encoded by strings, sets of positions over two successors are encoded by binary trees. Working out the details yields the *weak second-order monadic theory of two successors* (or WS2S) [9, 15, 17].

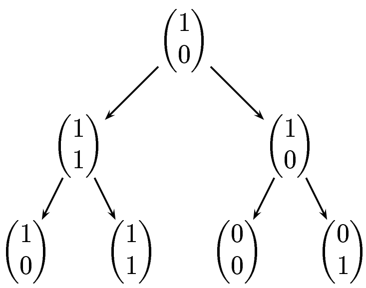
The syntax of WS2S is similar to WS1S except we use ε (instead of 0) to denote the root address in a tree and we now have two successor .0 (left child) and .1 (right child). Semantically, WS2S formulae are interpreted in the domain $\mathcal{D} = \{0,1\}^\star$ and ε denotes the empty string, .0 denotes concatenation with 0, and .1 denotes concatenation with 1. All other details are identical to WS1S.

WS2S generalizes Boolean logic to a logic of trees because second-order variables are interpreted by finite subsets $M \subset \mathcal{D}$ and any such M can be encoded by a Boolean-labeled, binary branching tree. The decidability of this logic follows by generalizing the decision procedure for WS1S: the models of any formula $\varphi(x_1, \ldots, x_n)$ define a regular tree language over the alphabet $\mathbb{B}^n$ and given φ we can construct a tree automaton accepting this language. As with WS1S, WS2S is non-elementary.

The following is a simple example of a specification in WS2S.

$$X(\varepsilon) \;\wedge\; (\forall^1 p.\, X(p.0) \leftrightarrow X(p.1)) \;\wedge\; \forall^1 p.\, \neg Y(p.0) \vee \neg Y(p.1)$$

This states that X contains the root position ε and that a position p is in X iff its brother is also in X. Moreover, for any p, Y contains at most one of p's successors. The interpretation $\{\varepsilon, 0, 1, 00, 01\}$ for X and $\{0, 01, 11\}$ for Y satisfies this formula and is encoded by the following tree (where the first component of each pair encodes the interpretation of X and the second encodes Y).



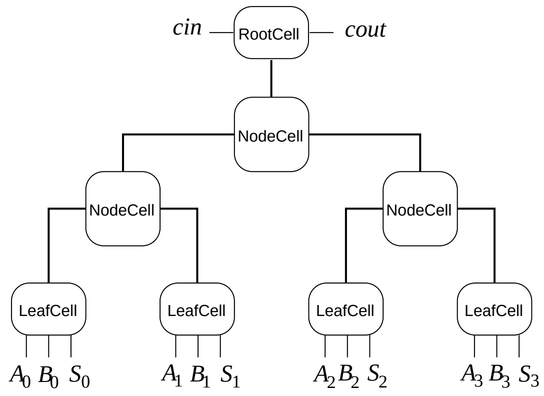


Figure 3. An n-bit CLA instance, for $n = 4$

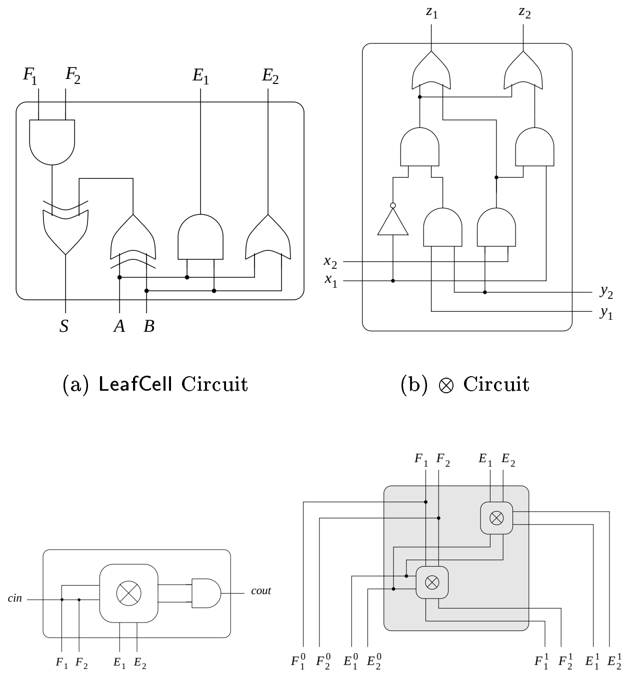
5.1 Example: Parameterized Carry-Lookahead Adder

We begin by describing a family of n-bit *carry-lookahead adder*s (or CLAs) whose overall structure is given (for $n = 4$) in Figure 3. We do not describe the adder in detail (see [7]) and restrict ourselves to a few comments. The adder operates in two phases: an *upward* phase and *downward* phase. In the upward phase, a carry status is computed for each internal node. This indicates whether an incoming carry is *killed*, *propagated*, or a new carry is *generated*. The carry status is given by the wires E_1 and E_2. At the leafs of the tree, the carry status for each digit is computed using the circuit depicted in Figure 4(a). At inner nodes, an operator $\otimes$ is used to combine the carry statuses of the successor nodes (see Figures 4(d) and 4(b)).

In the downward phase, an inner node passes the incoming carry bits F_1 and F_2 unchanged as signals F_1^0 and F_2^0 to its left successor; the carry bits F_1^1, F_2^1 for the right successor are computed using the operator $\otimes$ from the carry statuses and the incoming carry bits (see Figure 4(d)). As indicated in Figure 4(c), the incoming carry at the root is given by *cin* and the outgoing carry by *cout*. At the leaf nodes, the incoming carry bits F_1 and F_2 are used to compute the sum bit as depicted in Figure 4(a).

5.2 Structural Modeling of the CLA

Structural modeling in WS2S is identical to WS1S except now we use two successors to index into trees. Hence we translate the circuits in Figures 4(a)–4(d) directly into the following WS2S formulae. Note that LeafCell and NodeCell take an extra argument p that indicates which leaf or node values are used.



(a) LeafCell Circuit (b) $\otimes$ Circuit

(c) RootCell Circuit (d) NodeCell Circuit (op)

Figure 4. Components of the $\textsc{Cla}$ **Circuit**

$$
\begin{aligned}
&\mathsf{LeafCell}(A, B, S, F_1, F_2, E_1, E_2, p) \equiv \\
&\quad \mathsf{and}(A(p), B(p), E_1(p)) \;\wedge\; \mathsf{or}(A(p), B(p), E_2(p)) \;\wedge \\
&\quad \exists^0 w_1, w_2.\, \mathsf{and}(F_1(p), F_2(p), w_1)) \;\wedge \\
&\qquad\qquad \mathsf{xor}(A(p), B(p), w_2) \;\wedge\; \mathsf{xor}(w_1, w_2, S(p))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{NodeCell}(F_1, F_2, E_1, E_2, p) \equiv \\
&\quad (F_1(p.0) \leftrightarrow F_1(p)) \;\wedge\; (F_2(p.0) \leftrightarrow F_2(p)) \;\wedge \\
&\quad \mathsf{op}(F_1(p), F_2(p), E_1(p.0), E_2(p.0), F_1(p.1), F_2(p.0)) \;\wedge \\
&\quad \mathsf{op}(E_1(p.0), E_2(p.0), E_1(p.1), E_2(p.1), E_1(p), E_2(p))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{RootCell}(F_1, F_2, E_1, E_2, cin, cout) \equiv \\
&\quad (cin \leftrightarrow F_1(\varepsilon)) \;\wedge\; (cin \leftrightarrow F_2(\varepsilon)) \;\wedge \\
&\quad (\exists^0 a, b.\, \mathsf{op}(F_1(\varepsilon), F_2(\varepsilon), E_1(\varepsilon), E_2(\varepsilon), a, b) \;\wedge \\
&\qquad\quad \mathsf{and}(a, b, cout))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{op}(x_1, x_2, y_1, y_2, z_1, z_2) \equiv \exists^0 w_1, w_2, w_3, w_4, w_5. \\
&\ \ \mathsf{not}(x_1, w_1) \;\wedge\; \mathsf{and}(y_1, y_2, w_2) \;\wedge\; \mathsf{and}(x_2, y_2, w_3) \;\wedge \\
&\ \ \mathsf{and}(w_1, w_2, w_4) \;\wedge\; \mathsf{and}(x_1, w_3, w_5) \;\wedge \\
&\ \ \mathsf{or}(w_4, w_5, z_1) \;\wedge\; \mathsf{or}(w_4, w_3, z_2)
\end{aligned}
$$

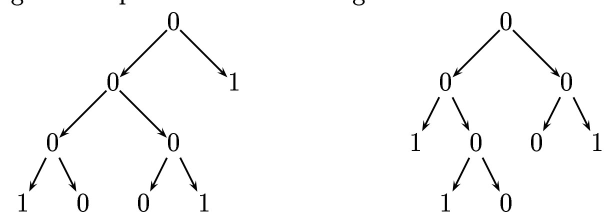
The overall circuit, given in Figure 3, is modeled by the predicate

$$
\begin{aligned}
&\mathsf{cla}(A, B, S, E_1, E_2, F_1, F_2, cin, cout) \equiv \exists^2 T. \\
&\ \mathsf{RootCell}(F_1, F_2, E_1, E_2, cin, cout) \;\wedge \\
&\ (\forall^1 p.\, (\mathsf{leaf}(p, T) \rightarrow \mathsf{LeafCell}(A, B, S, F_1, F_2, E_1, E_2, p)) \;\wedge \\
&\qquad (\mathsf{node}(p, T) \rightarrow \mathsf{NodeCell}(F_1, F_2, E_1, E_2, p))) \;\wedge \\
&\ \mathsf{shape\_cond}(A, B, S, T)\,.
\end{aligned}
$$

RootCell initializes the carry-in bit and computes the carry-out bit. The next two lines correspond to a for-loop with discrimination (pattern matching) for each position p. The first implication gives the base case: each leaf of the circuit is LeafCell. The second gives the step case: each inner node of the circuit is NodeCell. The predicate shape_cond fixes the shape of the inputs; we explain it shortly.

5.3 Behavior

In WS1S we encoded numbers as bit-strings. In WS2S we have binary trees and encode numbers using the labels on a tree's frontier. For example, the following trees represent the bit-strings 10011 and 11001.



An important requirement in specifying tree structured adders, is that, once we fix the format of the adder to be a particular shape, both inputs and the output must also be of that shape; that is, numbers must be encoded at leaf nodes in the same positions in these three trees. This kind of requirement, which corresponds roughly to a kind of 'type-correctness' for the inputs, is easy to specify in a high-level programming or specification language where recursive data-types can formalize this type (or 'shape') constraint. In our setting, we define a predicate (shape_cond, which we used in the specification of cla above) that enforces this requirement.

We proceed in several steps. First, we characterize those trees with a particular shape as those T where:

1. T is not emtpy.
2. T is closed under the parent relation: if a position p is in T then its parent, denoted by $p\uparrow$, is in also in T.
3. If an inner node p is in T then its both successors $p.0$ and $p.1$ are in T too.

This 'type' is formalized by

$$
\begin{aligned}
&\mathsf{shape}(T) \equiv (\exists^1 p.\, T(p)) \;\wedge \\
&\quad \forall^1 p.\, (T(p) \rightarrow T(p\uparrow)) \;\wedge (T(p) \;\wedge\; T(p.0) \rightarrow T(p.1)) \;\wedge \\
&\qquad (T(p) \;\wedge\; T(p.1) \rightarrow T(p.0)).
\end{aligned}
$$

Second, we say that a tree X has shape T (X is of type T), if all its positions constitutes a subset of the leafs of T.

$$\mathsf{is\_of\_shape}(X,T) \equiv \forall^1 p.\, X(p) \to \mathsf{leaf}(p,T).$$

Finally, we combine these to formalize shape_cond that holds when the trees A, B, and S, which represent base-two numbers, have the same shape T.

$$\begin{aligned}&\mathsf{shape\_cond}(A,B,S,T) \equiv \mathsf{shape}(T) \ \wedge\ \mathsf{is\_of\_shape}(A,T) \wedge \\ &\quad \mathsf{is\_of\_shape}(B,T) \ \wedge\ \mathsf{is\_of\_shape}(S,T).\end{aligned}$$

To complete our behavioral model, we define several auxiliary predicates in Figure 5 that allow us to traverse the leaves of a valid input tree from left to right. For a tree T satisfying the shape predicate and a position p, $\mathsf{first}(p,T)$ checks if p is the left-most leaf in T and $\mathsf{last}(p,T)$ checks if p is the right-most leaf in T. The predicate $\mathsf{next}(p,q,T)$ checks if p and q are leaves in T and q is the next leaf to the right of q. Using these, our behavioral model can be defined analogously to the first behavioral model in Section 4.

$$\begin{aligned}&\mathsf{adder\_beh}(A,B,S,cin,cout) \equiv \exists^2 T,C.\\ &\quad \mathsf{shape\_cond}(A,B,S,T) \wedge\\ &\quad \forall^1 p.\, \mathsf{leaf}(p,T) \to \mathsf{mod\_two}(A(p),B(p),C(p),S(p)) \wedge\\ &\quad \exists^1 p.\, \mathsf{first}(p,T) \ \wedge\ (cin \leftrightarrow C(p)) \wedge\\ &\quad \exists^1 p.\, \mathsf{last}(p,T) \ \wedge\ \mathsf{at\_least\_two}(A(p),B(p),C(p),cout) \wedge\\ &\quad \forall^1 p,q.\, \mathsf{leaf}(p,T) \ \wedge\ \mathsf{next}(p,q,T) \to\\ &\qquad\quad \mathsf{at\_least\_two}(A(p),B(p),C(p),C(q))\ .\end{aligned}$$

5.4 Verification

We can now verify that our structural model of the CLA is equivalent to our behavioral model, i.e., that a CLA of any size actually adds its inputs. We formalize this as

$$\begin{aligned}&\forall^2 A,B,S.\, \forall^0 cin, cout.\\ &\ \mathsf{cla}(A,B,S,cin,cout) \leftrightarrow \mathsf{adder\_beh}(A,B,S,cin,cout)\,.\end{aligned}$$

This formula is proved by MONA in one second.

6 Conclusion and Prospects

Monadic logics have been studied by logicians for over forty years, but are almost unknown to engineers. We have argued that these logics are natural generalizations of Boolean logic that can be motivated from an engineering perspective and used to model many more kinds of systems. They are decidable, which means that verification can be automated, and they have an operational side as formulae correspond to automata over strings or trees. This opens possibilities for designing modeling environments that support a variety of formal activities including verification, simulation, and testing.

$$\mathsf{leaf}(p,T) \equiv T(p) \ \wedge\ \neg T(p.0) \ \wedge\ \neg T(p.1)$$

$$\mathsf{node}(p,T) \equiv T(p) \ \wedge\ T(p.0) \ \wedge\ T(p.1)$$

$$\begin{aligned}&\mathsf{path}(p,X) \equiv \mathsf{leaf}(p,X) \wedge\\ &\quad \forall^1 x.\, (X(x) \to X(x\uparrow)) \ \wedge\ (X(x.0) \to \neg X(x.1)) \wedge\\ &\qquad (X(x.1) \to \neg X(x.0))\end{aligned}$$

$$\begin{aligned}&\mathsf{next}(p,q,T) \equiv p \neq q \ \wedge\ \mathsf{leaf}(p,T) \ \wedge\ \mathsf{leaf}(q,T) \wedge\\ &\quad \exists^2 P,Q.\, \mathsf{path}(p,P) \ \wedge\ \mathsf{path}(q,Q) \wedge\\ &\quad \exists^1 s.\, \exists^2 S.\, P(s) \ \wedge\ Q(s) \ \wedge\ P(s.0) \wedge\\ &\qquad\qquad Q(s.1) \ \wedge\ \mathsf{path}(s,S) \wedge\\ &\quad \forall^1 u.\, (P(u.0) \ \wedge\ u \neq s \to S(u.0)) \wedge\\ &\qquad\quad (Q(u.1) \ \wedge\ u \neq s \to S(u.1))\end{aligned}$$

$$\begin{aligned}&\mathsf{first}(p,T) \equiv \mathsf{leaf}(p,T) \ \wedge\ \exists^2 X.\, \mathsf{path}(p,X) \wedge\\ &\quad \forall^1 u.\, X(u.1) \to (\forall^1 v.\, u.0 \leq v \to \neg T(v))\end{aligned}$$

$$\begin{aligned}&\mathsf{last}(p,T) \equiv \mathsf{leaf}(p,T) \ \wedge\ \exists^2 X.\, \mathsf{path}(p,X) \wedge\\ &\quad \forall^1 u.\, X(u.0) \to (\forall^1 v.\, u.1 \leq v \to \neg T(v))\end{aligned}$$

Figure 5. Auxiliary predicates for the CLA

One drawback is that, in some cases, problems must be modeled at too low a level of abstraction: in WS2S everything must be mapped onto binary branching Boolean trees. This can make models difficult to design and understand (witness the concern with 'shape constraints' in Section 4.2) and lead to modeling errors.

Our current work aims at building a high-level specification language that eliminates this drawback. We have developed a language that allows users to specify tree languages in WS2S using data-types like those found in modern programming languages [2]. Moreover, we are building a translator from the high-level hardware description language VHDL to WS1S and integrating the MONA system with other development and verification tools [3].

References

[1] ANSI/IEEE Standard 1076–1993. *IEEE Standard VHDL Language Reference Manual.* IEEE, New York, USA, June 1994.

[2] A. Ayari, D. Basin, and A. Podelski. Lisa: A Specification Language Based on WS2S. In *CSL'97*, volume 1414 of *LNCS*. Springer, 1997.

[3] D. Basin and S. Friedrich. Combining WS1S and HOL. In *Frontiers of Combining Systems, Second International Workshop, Amsterdam, September 1998*, Applied Logic Series. Kluwer Academic Publishers, 1998. To appear.

[4] D. Basin and N. Klarlund. Automata based symbolic reasoning in hardware verification. *The Journal of Formal Methods in Systems Design*, 13(3):255–288, November 1998.

[5] J. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[6] A. J. Camilleri, M. J. C. Gordon, and T. F. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*. North Holland, 1986.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[8] C. C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the AMS*, 98:21–52, 1961.

[9] Y. Gurevich. Monadic second-order theories. In J. Barwise and S. Feferman, editors, *Model-Theoretic Logics*, pages 479–506. Springer, 1985.

[10] N. Klarlund and A. Møller. *MONA Version 1.3 User Manual*. BRICS Notes Series NS-98-3 (second revision), Department of Computer Science, University of Aarhus, October 1998.

[11] Z. Manna et al. STeP: The Stanford Temporal Prover. In M. Mosses, M. Nielsen, and M. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE*, volume 915 of *LNCS*, pages 793–794. Springer, 1995.

[12] A. Meyer. Weak monadic second-order theory of successor is not elementary-recursive. In *LOGCOLLOQ: Logic Colloquium*. Lecture Notes in Mathematics, No. 453, Springer-Verlag, 1975.

[13] F. Morawietz and T. Cornell. On the recognizibility of relations over a tree definable in a monadic second-order tree description language. Research Report SFB 340-Report 85, Sonderforschungsbereich 340 of the Deutsche Forschungsgemeinschaft, February 1997.

[14] M. O. Rabin and D. Scott. Finite automata and their decision problems. In E. F. Moore, editor, *Sequential Machines: Selected Papers*, pages 63–91. Addison-Wesley, Reading, MA, 1964.

[15] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem in second-order logic. *Math. Systems Theory*, 2:57–81, 1968.

[16] W. Thomas. Die Logiken von Boole und Büchi-Elgot-Thraktenbrot in der Beschreibung diskreter Systeme. Manuskript, Lehrstuhl für Informatik VII, RWTH Aachen.

[17] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 4, pages 133–191. Elsevier Science Publishers B. V., 1990.

[18] B. A. Trakhtenbrot. Finite automata and the logic of one-place predicates. *Sib. Math. J.*, 3:103–131, 1962.