Higher-Level Specification and Verification With BDDs *

Alan J. Hu, David L. Dill, Andreas J. Drexler, and C. Han Yang

Department of Computer Science, Stanford University

Abstract. Currently, many are investigating promising verification methods based on Boolean decision diagrams (BDDs). Using BDDs, however, requires modeling the system under verification in terms of Boolean formulas. This modeling can be difficult and error-prone, especially when dealing with constructs like arithmetic, sequential control flow, and complex data structures. We present new techniques for automatically translating these constructs into BDDs. Furthermore, these techniques generate Boolean next-state relations in a form that allows efficient image computation without building the full BDD for the next-state relation, thereby side-stepping the commonly-encountered BDD-size blowup of next-state relations.

1 Introduction

With the high complexity of hardware designs and protocols, improved debugging tools and methodologies are critical to avoiding the expenses and delays resulting from discovering bugs late in the design phase [9, 4]. Simulation catches some problems, but bugs frequently slip through. The increasing concurrency and complexity of hardware designs exacerbates this problem: detecting by simulation every bug resulting from the complex interaction of concurrent events becomes highly improbable (or prohibitively time-consuming). This situation has prompted interest in verification techniques.

Verification by state enumeration is particularly attractive because it is highly automatic: Given a specification, the verification system performs the verification with no further user intervention [4]. The main disadvantage of this technique is the potentially huge number of possible states. To address this state-explosion problem, many recent results have used symbolic expressions to specify sets of states (e.g. [2, 5, 4, 9]). In fact, all of the above cited works use a particularly promising data structure for the symbolic expressions: Boolean decision diagrams (BDDs) [7].

One drawback of BDDs is the requirement to describe the system being verified using Boolean formulas. This modeling is difficult, time-consuming, and

* This research was supported by the National Science Foundation (grant number MIP-8858807), the Defense Advanced Research Projects Agency (contract number N00014-87-K-0828) and by gifts from the Powell Foundation and Mitsubishi Electronics. The first author was supported by an ONR Graduate Fellowship. Most of this work was done using equipment generously donated by Sun Microsystems.

error-prone [17]. Worse, revising the description to, for example, scale the number processes, enlarge buffers, or increase word sizes, entails a laborious rewriting of the entire description. Working with common programming language constructs like scalar-valued variables [18], arrays, records, and sequential assignment would be much more natural.

Another problem confronting verification with BDDs is that the next-state relation of the system being verified frequently results in a BDD that is too large to build [10, 19, 3]. For verifying deterministic finite-state machines (*e.g.* a digital synchronous circuit), Boolean Functional Vectors [10] have proven quite successful. Unfortunately, higher-level specifications tend to be non-deterministic, necessitating other approaches to avoiding this problem.

We are working on two languages to address these problems. One, called Murφ, is a high-level language much like common structured programming languages. We can compile Murφ both into C++ (for simulation and non-symbolic verification) and also into our second language. We discuss Murφ in another paper [11]. The second language, called Ever, is a BDD-based verifier. In addition to the usual Boolean and verification operators common to other BDD-based verifiers, however, Ever provides direct support for many higher-level features, including scalars, arrays, records, and sequentiality. These higher-level features, besides providing expressive power, allow us to avoid building the full BDD for the next-state relation. Supporting these features involves some novel techniques that we discuss here.

2 Design Objectives

We started with some abstract design objectives. As Ever is intended to be an intermediate language, it must be easily machine-generated. Furthermore, the language must allow compact expression of the semantics of the higher-level language. On the other hand, the language should be human-readable and human-writable to allow writing Ever code directly, facilitate modifying automatically-generated code, and ease debugging the automatic generators.

Like other BDD-based verifiers, Ever must support the usual Boolean and temporal logic operators, as well as standard verification operations like reachability and trace generation. Our experience also indicated that parameterized predicates would be convenient.

Complex data structures are imperative. For example, while verifying a real, industrial link-level protocol, we needed to specify queues that were arrays of packets. Each packet, in turn, was a record including fields for packet type, sequence number, and other data. The fields were integers, enumerated types, or records. Clearly, then, we must provide scalar values and variables, with the concomitant arithmetic and relational operators, as well as array and record constructors, to allow arbitrarily complex data structures. Beyond expressive convenience, these higher-level data structures provide an additional pay-off: scalability. By simply changing a few constants and array bounds, we can easily take a large and detailed description of a system and scale it to a smaller version that can be fully verified.

One particularly difficult aspect of modeling with Boolean formulas is an instance of the famous Frame Problem in artificial intelligence [16]. We refer to our instance as the problem of *stability:* for a given operation, how does one determine which variables retain their current values, which variables change, and which variables are non-deterministic. In a normal programming language, a variable that is not explicitly modified keeps its current value. In the Boolean context, however, any variable that is not constrained can take on any value. This semantic difference gives Boolean formulas much of their expressive power, but is also the source of many subtle specification bugs. For example, it is easy to see that an imperative statement like:

```
x := 17;
```

corresponds to the Boolean next-state relation:

```
(next(x)=17) AND (everything except x doesn't change).
```

It is not obvious how to correctly translate a code fragment (taken from an industrial directory-based cache-coherence protocol modeled in our imperative specification language Murφ) like:

```
If (i < Homes[h].Dir[a].Shared_Count) & (Homes[h].Dir[a].Entries[i] = n)
Then
  -- overwrite this entry with last entry.
  Homes[h].Dir[a].Entries[i] :=
    Homes[h].Dir[a].Entries[Homes[h].Dir[a].Shared_Count-1];
  -- clear last entry
  Homes[h].Dir[a].Entries[Homes[h].Dir[a].Shared_Count-1] := 0;
  Homes[h].Dir[a].Shared_Count := Homes[h].Dir[a].Shared_Count-1;
Endif;
```

To handle such expressions, then, we felt it necessary to support deterministic assignment, which handles stability exactly as the user would expect, *directly* in Ever, along with facilities to build correct next-state formulas for complex statements from the formulas for simpler ones.

3 A Simple Example

An example is perhaps the easiest way to acquaint oneself with the Ever language. We will look at a simple link-level protocol. (See Figure 1.) There are four processes: a source, a transmitter, a receiver, and a destination. These four operate asynchronously, handshaking via ports.[2] The source has a string of data, which it sends a character at a time to the transmitter. The transmitter packs the characters into packets, appends a checksum, and sends the packet to the receiver. The receiver unpacks the characters, and sends them one at a time to the destination. The destination receives characters and stores them in a buffer.

2 The computational model underlying Ever is a non-deterministic finite-state machine. We can easily model the asynchronous, interleaved concurrency in this example using non-deterministic choice. Synchronous parallelism currently requires some contortions.

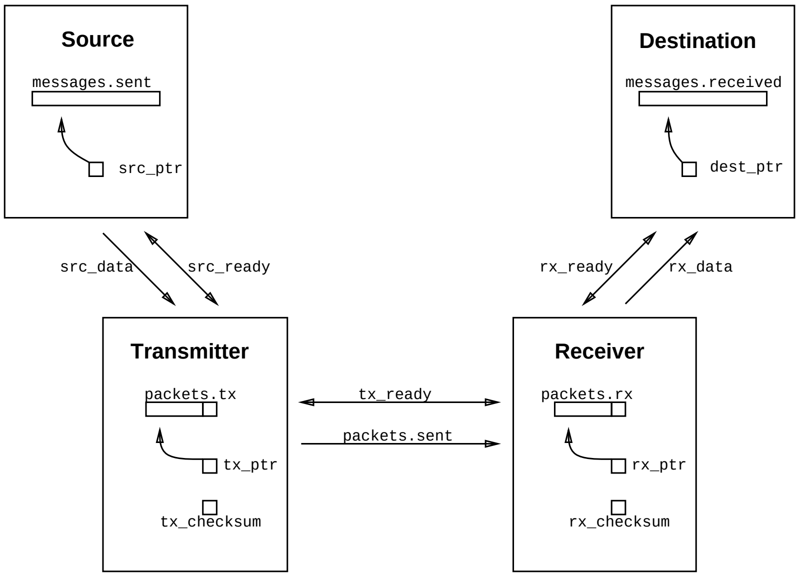

Fig. 1. In this simple example, a message travels from source to destination. The transmitter and receiver handle packets and checksums.

The syntax is simple in order to simplify automatically generating Ever code. A program consists of a sequential list of declarations and commands. Within a declaration or command, expressions are in LISP-like prefix notation. Array and record accesses are denoted by square brackets and periods, as in Pascal or C.

The first few lines:

```
deftype signal (bits 1 "low" "high");
deftype char (bits 1 "a" "b");
deftype integer (bits 4);
deftype packet (record data (array 0 3 char) check char);
```

declare some types. In Ever, we declare scalar types by the number of bits needed. (The strings following the bit width are used for output.) Thus, types `signal` and `char` are both 1 bit, type `integer` is 4 bits, and type `packet` is a record with two fields: `data`, a four-element array, and `check`, of type `char`. The next few lines declare variables for the source and destination:

```
defvar src_ready signal;  -- Variable src_ready is of type signal,
defvar src_ptr integer;   -- and so forth.
defvar src_data char;

defvar dest_ptr integer;

defvar messages (interleaved record
                                sent (array 0 7 char)
                                received (array 0 7 char)
                )；
```

The source sends the contents of `messages.sent` to the destination, which writes the data into `messages.received`. We will discuss the `interleaved` keyword later. We then declare variables for the transmitter and receiver.

```
defvar packets (interleaved record
                        tx packet
                        sent packet
                        rx packet
                );

defvar tx_ptr integer;
defvar tx_checksum char;
defvar tx_ready signal;

defvar rx_ptr integer;
defvar rx_data char;
defvar rx_checksum char;
defvar rx_ready signal;
```

The next lines define the set of possible start states:

```
defprop StartState (and
                        (eq src_ptr^c 0)   -- src_ptr must be 0
                        (eq src_ready^c 0) -- src_ready must be 0
                        (eq tx_ptr^c 0)    -- etc.
                        (eq tx_ready^c 0)
                        (eq rx_ptr^c 4)
                        (eq rx_ready^c 0)
                        (eq dest_ptr^c 0)
                    );
```

The suffix `^c` specifies the current value of the variable, whereas the suffix `^n` specifies the next value. The formula initializes the control variables, but leaves the data unspecified, allowing them to assume any value.

Next, we define the next-state relation. The definition is in parts. First, we define the source:

```
defprop Source (if (and (le src_ptr^c 7)
                        (eq src_ready^c 0)) -- If we can send another char
                   (compose        --then send it to Tx
                        (becomes src_data^n messages.sent[src_ptr^c]^c)
                        (becomes src_ptr^n (add src_ptr^c 1))
                        (becomes src_ready^n 1)
                   )
                   FALSE           --else block
                );
```

which simply writes the next character to a port if the port is ready. Note that scalar arithmetic makes it easy to maintain counters. The `becomes` operator provides deterministic assignment: it generates the BDD next-state relation that assigns an expression to a variable, while keeping all other variables constant. Combining the `becomes` operator with the `compose` operator, which provides next-state relation composition, produces code resembling sequential execution, but which actually defines a next-state relation for *all* legal executions. Next, we define the transmitter:

```
defprop Tx (if (and (le tx_ptr^c 3)
                    (eq src_ready^c 1)) -- If incoming char
```

```
                (compose        --then add it to packet and update checksum
                    (becomes packets.tx.check^n (add packets.tx.check^c
                                                     src_data^c))
                    (becomes packets.tx.data[tx_ptr^c]^n src_data^c)
                    (becomes tx_ptr^n (add tx_ptr^c 1))
                    (becomes src_ready^n 0)
                )
                (if (and (gt tx_ptr^c 3)
                         (eq tx_ready^c 0)) --else if ready to send packet
                    (compose    --then send it to Rx
                        (becomes packets.sent^n packets.tx^c)
                        (becomes packets.tx^n 0)
                        (becomes tx_ptr^n 0)
                        (becomes tx_ready^n 1)
                    )
                    FALSE       --else block
                )
            );
```

The transmitter reads characters from the source, appends them to a packet, and computes a checksum. Whenever it has a complete packet and the channel to the receiver is clear, it sends the packet. The receiver unpacks characters from the packets:

```
defprop Rx (if (and (gt rx_ptr^c 3) (eq tx_ready^c 1)) --If we can receive packet
                (compose        --then get the packet
                    (becomes packets.rx^n packets.sent^c)
                    (becomes packets.sent^n 0)
                    -- Error detection and recovery code would go here.
                    (becomes rx_ptr^n 0)
                    (becomes tx_ready^n 0)
                )
                (if (and (le rx_ptr^c 3)
                         (eq rx_ready^c 0)) --else if we have data for Dest
                    (compose    --then send it
                        (becomes rx_data^n packets.rx.data[rx_ptr^c]^c)
                        (becomes rx_ptr^n (add rx_ptr^c 1))
                        (becomes rx_ready^n 1)
                    )
                    FALSE       --else block
                )
            );
```

In a real link-level protocol, we would insert code to check the checksum and handle errors at the indicated point. The destination is easy:

```
defprop Destination (if (eq rx_ready^c 1) --If Rx has data for us
                        (compose         --then get it
                            (becomes messages.received[dest_ptr^c]^n rx_data^c)
                            (becomes dest_ptr^n (add dest_ptr^c 1))
                            (becomes rx_ready^n 0)
                        )
                        FALSE            --else block
                    );
```

We define the next-state relation for the entire system

```
defprop NextState (or (Source) (Tx) (Rx) (Destination));
```

as the non-deterministic choice of the preceding next-state relations.
The remaining lines of the program:

```
defprop Ok (if (gt dest_ptr^c 7) (eq messages.sent^c messages.received^c));

printtrace (StartState) (NextState) (not (Ok));
```

define a simple verification condition and invoke the reachability verifier. We use verification conditions to check safety properties. The state-reachability computation starts from all states that satisfy the start-state formula and, using the specified next-state relation, outputs a trace that reaches the end condition (or is as long as possible).

This program takes thirteen and a half minutes on a SUN 4/75, using less than 6.4MB of memory, to compute all reachable states and to find and print a longest acyclic trace (37 states long). Interestingly, even this short example has 84 billion reachable states (out of a state space of 1.8×10^{16} states) putting it well-beyond the reach of non-symbolic state-enumeration verifiers.

4 Translation into Logic

Much of the Ever language is similar to other BDD-based verifiers. We will only briefly mention these aspects here. Logical operations are directly supported by BDDs [7]. Our implementation uses Brace *et al.*'s BDD implementation [6]. Symbolic verification algorithms (reachability, trace generation, etc.) are standard and can be found in several sources [9, 2, 4, 5]. Very recently, Clarke *et al.* [8], as part of a larger work on building approximate abstract models of programs, have proposed a solution to the problems of stability and sequentiality that is essentially a special case (no complex data structures) of the deterministic assignment rules in Section 4.3 combined with the sequentiality rule in Section 4.4.

4.1 Scalars

We represent a scalar variable as a vector of Boolean functions. We map scalars to Booleans using the binary representation of the ordinal value of the scalar. A variable a would be represented by (a_3, a_2, a_1, a_0), where the a_i are the Boolean variables corresponding to a. The scalar quantity $a+4$ would be $(a_3 \oplus a_2, \overline{a_2}, a_1, a_0)$. This mapping permits, for instance, using a simple ripple-carry adder to compute the sum $z = x + y$ of two scalar variables:

$$z_i = x_i \oplus y_i \oplus c_{i-1}$$
$$c_i = x_i y_i + c_{i-1}(x_i + y_i).$$

Similar expressions correspond to other operators. We have implemented a larger set of arithmetic and relational operators than Srinivasan *et al.* [18], including comparison, addition, and subtraction between arbitrary scalar-valued expressions. Note that these expressions are computed logically on the Boolean characteristic functions, so that, for example, if $P(x)$ and $Q(y)$ specify the subsets of possible values for x and y, we can express that z can be *all possible values of the sum* by writing $\exists x, y[P(x) \wedge Q(y) \wedge (z = x + y)]$.

4.2 Data Structures

As illustrated in the example in Section 3, Ever permits Pascal-like records and arrays. Implementing the variable declarations is much like compiling the variable declarations of a structured programming language, except that instead of allocating bits of memory for storage, we allocate entries in an array of BDD variables. Let us call this array V. We can easily compute the size of V using a standard recursive computation: the size of a scalar is the number of bits used to represent it, the size of a record is the sum of the sizes of its fields, and the size of an array is the product of the number of elements in the array and the size of an array element.

A more interesting problem is to generate the correct vector of Boolean formulas to correspond to a variable reference. For a simple variable reference, we look up the offset to the correct BDD variables much as a compiler would generate the offset to the start of the correct block of memory. The BDD variables starting at that offset form the correct vector of Boolean formulas. For a record access, we start with the base variable as before, add the field offset, and proceed as in the case of the simple variable reference. Array indexing is the most complex. If the index were a constant, we could proceed as for records. The index, however, can be an arbitrary scalar-valued expression, which is a vector of Boolean functions denoting a *set* of possible values. For example, an expression like $x[a+4]$ denotes a vector of Boolean formulas that, when restricted to having $a = 0$, are equivalent to the Boolean formulas for $x[4]$; when restricted to having $a = 1$, are equivalent to $x[5]$; and so forth. If $f(a)$ is the proposition $1 < a < 5$, then the proposition $(z = x[a + 4]) \wedge f(a)$ says that z can be equal to any of $x[5]$ through $x[9]$. Therefore, the formulas we generate for an array access must perform a case analysis for each possible value of the array index.

In more formal terms, define a *modifier* as either a field name or an array-indexing expression. We will consider a variable name to be a field name in a global record. In this notation, a variable reference is simply a string of modifiers. For any field name or constant array index, define OFFSET(*modifier*) to be the offset from the beginning of the record or array to the start of the referenced field or array entry. OFFSET is a common computation in compilers: the offset for a record field is the sum of the sizes of the fields that precede it, and the offset for the ith array entry is the product of i (minus the array lower bound) and the size of an array element. Denote by $V(n, s)$ the s BDD variables in the storage array V going from $V[n]$ to $V[n + s - 1]$. Since this quantity is a vector of Boolean formulas, we can consider it to be an Ever scalar. For notational convenience, a Boolean operator applied to a vector of Boolean formulas is assumed to apply in parallel to each element. Given an arbitrary variable reference ρ, define s_ρ to be the number of bits that ρ describes, or equivalently, the size of the referenced variable. The vector of Boolean formulas for the variable reference ρ is given by

BDD$(0, \rho)$, where the function BDD is defined by:

$$\begin{aligned}
\mathrm{BDD}(n, \epsilon) &= V(n, s_\rho) \\
\mathrm{BDD}(n, \mathit{field_name}\ \sigma) &= \mathrm{BDD}(n + \mathrm{OFFSET}(\mathit{field_name}), \sigma) \\
\mathrm{BDD}(n, \mathit{index_expr}\ \sigma) &= \bigwedge_{i=l}^{u} [(\mathit{index_expr} = i) \Rightarrow \mathrm{BDD}(n + \mathrm{OFFSET}(i), \sigma)]
\end{aligned}$$

where σ is a (possibly empty) string of modifiers, ϵ is the empty string, and l and u are the lower and upper bounds of the array. Note that these expressions generate a vector of complex Boolean formulas, rather than a set of particular BDD variables. Intuitively, the generated formulas are multiplexors whose select lines are driven by the array-indexing expressions and whose inputs are the BDD variables. (The formula for array indexing can be considered a scalar-valued generalization of work by Beatty *et al.* [1].)

4.3 Deterministic Assignment

To implement deterministic assignment, we use a computation similar to the variable reference computation described above, except that we must specify that every BDD variable that isn't referenced must keep its current value. Also, for variable referencing, we are generating a vector of Boolean formulas; in the case of deterministic assignment, we want to generate a single Boolean relation between the current state of the system and the next state of the system that is true iff the current and next states correspond to the deterministic assignment.

For records, generating this relation is straightforward. For each field not being accessed, we AND into the Boolean relation being generated the further requirement that the field not change. For the field that we do access, we equate a variable reference expression like that generated in the previous section with the right-hand side of the assignment. For arrays, we must again perform a case analysis.

Formally, the next-state relation for deterministic assignment of an expression *new_value* to a variable reference ρ is given by BECOMES$(0, \rho, \mathit{new_value})$, where the function BECOMES is defined by:

$$\mathrm{BECOMES}(n, \epsilon, v) = (V(n, s_\rho) = v)$$

$$\mathrm{BECOMES}(n, \mathit{field\_name}\ \sigma, v) = \bigwedge_{f \in \mathit{record\_fields}} \left[\begin{array}{l} \text{if } (f = \mathit{field\_name}) \\ \text{then} \\ \quad \mathrm{BECOMES}(n + \mathrm{OFFSET}(f), \sigma, v) \\ \text{else } V(n + \mathrm{OFFSET}(f), s_f) \\ \qquad \text{remains constant.} \end{array}\right]$$

$$\mathrm{BECOMES}(n, \mathit{index\_expr}\ \sigma, v) = \bigwedge_{i=l}^{u} \left[\begin{array}{l} \text{if } (i = \mathit{index\_expr}) \\ \text{then } \mathrm{BECOMES}(n + \mathrm{OFFSET}(i), \sigma, v) \\ \text{else } V(n + \mathrm{OFFSET}(i), s_a) \text{ remains constant.} \end{array}\right]$$

where s_f is the size of record field f, s_a is the size of an array element, and the other variables are defined as before.

4.4 Stability

Once we have deterministic assignment, the problem of stability becomes easy. Here are the rules we need to build larger next-state relations from smaller ones with correct stability. (Correctness follows trivially by a structural induction.) If p is some code fragment in a higher-level language, define EVER(p) to be the corresponding Ever next-state relation.

Assignment: If p is an assignment statement "variable := expression", then EVER(p) is "(`becomes` variable expression)."

Sequentiality: If p is a sequence of statements "s_1; s_2; ...; s_n", then EVER(p) is "(`compose` EVER(s_1) EVER(s_2) ... EVER(s_n))." (The `compose` operator is simply Boolean relation composition. For example, if $N_1(x, y)$ and $N_2(x, y)$ are two Boolean relations, we can compute $N_2 \circ N_1(x, y) = \exists z[N_1(x, z) \wedge N_2(z, y)]$.)

Conditional: If p is the conditional statement "if c then s_1 else s_2 endif", then EVER(p) is "(`if` c EVER(s_1) EVER(s_2))."

Non-Determinism: If p is the non-deterministic choice between s_1 and s_2, then EVER(p) is simply "(`or` EVER(s_1) EVER(s_2))."

Note that these rules essentially provide a denotational semantics for a higher-level, sequential specification language in terms of Ever. We are currently using a structured protocol description language Murφ that, by applying these rules, compiles efficiently into Ever for symbolic verification. For example, the complex statement presented in Section 2 is a Murφ statement that compiles by a trivial rewriting into the following Ever code:[3]

```
(if
  (and
    (lt i^c Homes[h^c].Dir[a^c].Shared_Count^c)
    (eq Homes[h^c].Dir[a^c].Entries[i^c]^c n^c)
  )
  (compose                        -- if body
    (becomes
      Homes[h^c].Dir[a^c].Entries[i^c]^n
      Homes[h^c].Dir[a^c].Entries[(sub Homes[h^c].Dir[a^c].Shared_Count^c 1)]^c
    )
    (becomes
      Homes[h^c].Dir[a^c].Entries[(sub Homes[h^c].Dir[a^c].Shared_Count^c 1)]^n
      0
    )
    (becomes
      Homes[h^c].Dir[a^c].Shared_Count^n
      (sub Homes[h^c].Dir[a^c].Shared_Count^c 1)
    )
  )
  CurNextEq                       -- else body
)
```

3 The code fragment given in Section 2 is actually from a procedure, which Murφ inlines. Therefore, in the translation of the real protocol, the parameters i,h,a, and n are replaced by the complex expressions in the procedure call. This situation presents no problem to Ever, but would needlessly complicate our illustrative example.

(CurNextEq is a next-state relation that requires the state not to change.) Translation from other protocol description languages into Ever should also be straightforward.

4.5 Image Computation

The preceding techniques are sufficient to convert a high-level specification into a Boolean next-state relation expressed as a BDD. Given a next-state relation $N(x, x')$, computing foward and backward images of characteristic functions $\chi(x)$ and $\chi'(x')$ is easy and efficient: the forward image is simply $\exists x[\chi(x) \wedge N(x, x')]$, and the backward image is simply $\exists x'[\chi'(x') \wedge N(x, x')]$. Efficiently computing these operations is important because they are the basis for most verification algorithms. For large problems, however, the BDD for $N(x, x')$ is frequently too large to compute [10, 19, 3]. Because of this problem, many researchers have turned to Boolean Functional Vectors [10, 14, 13] or closely related methods [19] to successfully verify large gate-level digital circuits.

Unfortunately, these methods, in addition to complicating image computation, do not support non-deterministic next-state relations. In the domain of gate-level digital circuits, this limitation does not matter; a gate-level circuit is typically considered deterministic. To model and verify higher-level protocols, however, non-determinism is essential, both to model unpredictable events (e.g. noise corrupting a data transmission) and to allow abstraction (e.g. modeling a cache without full details of the line-replacement policy).

Fortunately, the translation of higher-level language constructs into Ever automatically gives us the next-state relation in a form that allows efficient computation of both forward and backward images without building the BDD for the full next-state relation. Let Image(χ, N) be the forward image of the characteristic function χ under next-state relation N. Recall from Section 4.4 that we generate Ever code from a higher-level language using a set of recursive rules. Therefore, an Ever next-state relation always has one of the following forms:

- $N(x, x')$ is a deterministic assignment. This case is the basis. We must build the BDD for N and compute Image$(\chi, N) = \exists x[\chi(x) \wedge N(x, x')]$.
- $N(x, x') = (N_n \circ \cdots \circ N_1)(x, x')$, where the operator $\circ$ denotes composition: $(N_2 \circ N_1)(x, x') = \exists x''[N_1(x, x'') \wedge N_2(x'', x')]$. In this case, Image$(\chi, N)$ is just the forward image of Image(χ, N_1) under $(N_n \circ \cdots \circ N_2)(x, x')$.
- $N(x, x') =$ if $C(x)$ then $N_1(x, x')$ else $N_2(x, x')$. In this case, Image(χ, N) equals Image$(\chi \wedge C, N_1) \vee$ Image$(\chi \wedge \overline{C}, N_2)$.
- $N(x, x') = N_1(x, x') \vee \cdots \vee N_n(x, x')$. This case is Burch *et al.*'s [3] disjunctive partitioned transition relation. We compute Image(χ, N) as simply Image$(\chi, N_1) \vee \cdots \vee$ Image(χ, N_n).

The computation of backward images is almost identical. These rules require building the BDD only for the next-state relations corresponding to each individual assignment, thereby eliminating the problem of building an enormous BDD for the entire next-state relation.

This technique represents an instance of the classic space-time trade-off. The number of nodes in the BDD for the full next-state relation is, in the worst case, the product of the number of nodes in the component next-state relations (yielding exponential behavior). Using the above technique, the number of BDD nodes is the *sum* of the number of nodes in the component next-state relations (yielding linear growth). On the other hand, one image computation on the full next-state relation becomes one image computation for *each* of the component next-state relations, yielding slower execution. Finally, in some instances, the BDD for the full next-state relation is substantially smaller than the worst-case behavior. Ever provides an evaluate operator to force the construction of the BDD for a given expression. Using this operator gives flexibility in trading space for time. For the small example described in Section 3, forcing complete evaluation of the next-state relation only increased memory usage to 6.8MB while cutting runtime to four and a half minutes. In contrast, on a large model of a real directory-based cache-coherence protocol (approximately 1000 lines of Murφ code and 10^{34}-state state space), both the fully-evaluated next-state relation and the disjunctive partitioned next-state relation were unable to build the required BDDs (with a 60MB memory limit), whereas the technique described above enabled reachability computation and verification using less than 24MB memory and one hour of CPU time. (All figures are for a SUN 4/75.)

5 Future Work and Conclusion

One issue that definitely needs further research is the problem of variable ordering. The size of the BDD representation of a Boolean function depends critically on the ordering of the variables [7]. One heuristic that has proven useful for scalars is to interleave the corresponding bits (most significant bits first, followed by the next most significant bits, etc.) [18]. In many cases, this ordering produces a substantially smaller BDD. For example, to check the equality of two scalars $x = y$, this ordering $(x_n, y_n, \ldots, x_1, y_1)$ produces the optimum $3n$-node BDD, whereas a naïve ordering $(x_n, \ldots, x_1, y_n, \ldots, y_1)$ produces an exponential-sized BDD. Jeong *et al.*'s Non-Interleaving Lemma [14] provides theoretical justification for this heuristic.[4] The `interleaved` keyword mentioned in Section 3 implements this ordering on each subtype to which it is attached. We plan to introduce additional variable-ordering heuristics to permit verification of larger examples.

Another issue is the question of relating specifications at different levels of abstraction. For example, we may wish to check that a synthesized circuit corresponds to a higher-level specification. We have developed an efficient technique to find *simulation preorders*, which check that one specification implements an-

4 In an unfortunate clash of terminology, what we and Srinivasan *et al.* call an interleaved variable order, because the bits comprising the scalars are interleaved, creates what Jeong *et al.* call a non-interleaved order, because the Boolean relations for the individual bit-slices have disjoint supports and are grouped together.

other [15, 12]. We are currently investigating the integration of this technique into Ever.

We have demonstrated the efficient implementation of higher-level language features in a BDD-based verifier and expect that these techniques will be applicable to other verifiers as well. These features have greatly simplified writing specifications for verification, as they provide a much more natural means of expression. Furthermore, they can also increase the efficiency of the verification by obviating the expensive computation of the full BDD for the next-state relation. In addition, the translation from a higher-level specification language like Murφ is straightforward. As we have already been using Murφ for several real, industrial problems, we are excited by progress in this direction.

References

1. Derek L. Beatty, Randal E. Bryant, and Carl-Johann H. Seger, "Synchronous Circuit Verification by Symbolic Simulation: An Illustration," *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference,* William J. Dally, ed., MIT Press, 1990.
2. S. Bose and A. Fisher, "Automatic Verification of Synchronous Circuits Using Symbolic Logic Simulation and Temporal Logic," *IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design,* Luc J.M. Claesen, ed., North Holland, 1989.
3. J.R. Burch, E.M. Clarke, and D.E. Long, "Symbolic Model Checking with Partitioned Transition Relations," *VLSI '91: Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration,* Edinburgh, Great Britain, 1991.
4. J.R. Burch, E.M. Clarke, K.L. McMillan, and David L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," *27th ACM/IEEE Design Automation Conference,* 1990, pp. 46-51.
5. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," *Proceedings of the Conference on Logic in Computer Science*, 1990, pp. 428–439.
6. Karl S. Brace, Richard L. Rudell, and Randal E. Bryant, "Efficient Implementation of a BDD Package," *27th ACM/IEEE Design Automation Conference,* 1990, pp. 40–45.
7. Randal E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, Vol. C-35, No. 8 (August 1986), pp. 677–691.
8. Edmund M. Clarke, Orna Grumberg, and David E. Long, "Model Checking and Abstraction," *Proceedings of the ACM Symposium on Principles of Programming Languages,* 1992, pp. 343–354.
9. Olivier Coudert, Christian Berthet, and Jean Christophe Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution," *Automatic Verification Methods for Finite State Systems*, J. Sifakis, ed., Lecture Notes in Computer Science Vol. 407, Springer-Verlag, 1989.
10. Olivier Coudert, Christian Berthet, and Jean Christophe Madre, "Verification of Sequential Machines Using Boolean Functional Vectors," *IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design,* Luc J.M. Claesen, ed., North Holland, 1989.

11. David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang, "Protocol Verification as a Hardware Design Aid," to appear in *IEEE International Conference on Computer Design,* 1992.
12. David L. Dill, Alan J. Hu, and Howard Wong-Toi, "Checking for Language Inclusion Using Simulation Preorders," *Computer-Aided Verification: Third International Workshop,* July 1–4, 1991, K.G. Larsen and A. Skou, eds., Lecture Notes in Computer Science Vol. 575, Springer-Verlag, published 1992.
13. Thomas Filkorn, "Functional Extension of Symbolic Model Checking," *Computer-Aided Verification: Third International Workshop,* July 1–4, 1991, K.G. Larsen and A. Skou, eds., Lecture Notes in Computer Science Vol. 575, Springer-Verlag, published 1992.
14. S.-W. Jeong, B. Plessier, G.D. Hachtel, and F. Somenzi, "Variable Ordering for FSM Traversal," *Proceedings of the International Workshop on Logic Synthesis*, MCNC, Research Triangle Park, NC, May 1991.
15. Paul Loewenstein and David Dill, "Formal Verification of Cache Systems using Refinement Relations," *IEEE International Conference on Computer Design,* 1990, pp. 228-233.
16. J. McCarthy and P.J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," in *Machine Intelligence 4*, B. Meltzer and D. Michie, eds., Edinburgh University Press, 1969.
17. K. L. McMillan and J. Schwalbe, "Formal Verification of the Gigamax Cache-Consistency Protocol," *Proceedings of the International Symposium on Shared Memory Multiprocessing,* Information Processing Society of Japan, 1991, pp. 242–251.
18. Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton, "Algorithms for Discrete Function Manipulation," *IEEE International Conference on Computer-Aided Design,* 1990, pp. 92–95.
19. Herve J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines using BDD's" *IEEE International Conference on Computer-Aided Design,* 1990, pp. 130–133.