Symbolic Representation Approaches for IO Automata*

Behnam Bani-Eqbal, Howard Barringer, Graham Gough
Brian Monahan, Martin Peim, Alan Williams

Department of Computer Science
University of Manchester†
Oxford Road, Manchester, UK.
email: howard@cs.man.ac.uk

January 1992

†This work has been partially supported via SERC project GR/F 09570, SERC/DTI project IED4/1/1357 and ESPRIT BRA project 3096 (SPEC)

Abstract

This report considers different approaches for representing efficiently an IO Automaton semantics for the ELLA hardware description language, which has been developed in earlier work. The problem is to develop a sufficiently compact representation for the semantics which also allows efficient application of formal verification techniques, such as trace-equivalence.

The utilisation of Binary Decision Diagrams (BDDs) is considered, and efficient methods for performing n-ary BDD-compositions, $n > 2$, are given. A representation for IO automata is presented together with associated automaton operations, such as *causal product*. This representation is then used to reify the automaton semantics of Boolean kernel ELLA, a covering subset of ELLA. A symbolic equivalence algorithm for deterministic finite IO automata using the BDD representation is given.

A significant difficulty with BDDs is to decide upon an ordering for the BDD-variables — a bad choice will lead to an explosion in the size of the BDD. Two alternatives to the normal BDD approaches are described.

The body of this report presents an overview of this work; technical details are provided in separate papers included as Appendices.

1 Context and Introduction

At Manchester the so-called "combinatorial state explosion" features most prominently in the temporal logic and automata-based verification techniques being developed for and applied to digital hardware (VLSI) circuit design.

Typically, automata, of various sorts, shapes and sizes, are used to model synchronous and asynchronous systems. However, rather than create yet another home-grown-and-applied hardware design and description language (HDDL), our approach is to work with and develop existing (commercially available) HDDLs and systems, thereby providing formal support for the practicing design engineers within their familiar environment. So, for our work in modelling / verifying synchronous systems, we have developed automata-based methods for the HDDLs ABEL and ELLA. In particular, we have supplied a formal compositional semantics for ELLA, where the semantic objects are input-output labelled finite automata. Then, based on the automaton semantics we have prototyped a compiler which translates ELLA text into IO-automaton models. Currently, the compiler uses an explicit state and transition enumeration of IO automata. This is fine as an exercise to demonstrate and/or animate the semantic approach but, because of the "state explosion", it is obviously hopeless for (large scale) practical use. Interestingly, our particular use of IO automata leads more to "transition" rather than "state" explosion.

What approaches might be suitable for easing the problem? The following three are being investigated,

1. on the fly generation of state + transition space,
2. non-global compositional model checking, and
3. compact representation of state + transition space,

though, for the present, we are primarily concerned with methods to obtain compact representations.

Binary decision diagrams (BDDs) [Bry86] have been shown, in several cases, to be rather successful in providing compact representations of very large state spaces. Claims of handling systems with 10^{50} and even 10^{120} reachable states have been made [BCL91a]. However, the gain does depend critically on the particular system being modelled and the way in which they are used; such statements should be read together with a sort of "government health warning". Indeed, one major difficulty with the use of BDDs is how to determine an ordering for the BDD-variables — a bad choice will lead to an explosion in the size of the BDD. In later sections of this report we consider two alternatives to the normal BDD approach.

The rest of this document is structured as follows. Section 2 presents a brief overview of the state of the art in the area of BDDs. Section 3 considers the use of BDDs as a representation vehicle for ELLA. In particular, there is a short motivational discussion on our use of IO automata in the semantics for HDDLs, a presentation of basic higher level operations on BDDs and then an example of their use in representing and defining ELLA semantic objects. The section concludes with a comparison of the two approaches adopted so far for ELLA, explicit enumeration and symbolic representation of the state and transition space. Section 4 briefly motivates two alternative symbolic representations; the first is based on conditional expressions and the second on a canonical exclusive or normal form. Directions of future work are briefly outlined in section 5. Finally, the appendices present further technical details which support the work reported here. The two which detail the alternative representations of section 4 are, in fact, separate reports in their own right.

2 BDDs: State of the Art

The form of BDD with which we deal in this paper was proposed by Bryant [Bry86] in 1986, drawing on earlier work of Lee [Lee59] and Akers [Ake78]. A later paper [BRB90] describes some improvements in data representation and algorithms for handling BDDs. This paper and others by various authors [MIY90a, MIY90b, MB88] describe modifications to the basic BDD concept such as putting extra information into the graph edges and incorporating truth tables directly into a BDD.

Since the introduction of BDDs there has been considerable interest in using them to represent and verify combinational systems. Some experimental results on the use of BDDs in tautology checking can be found in [FB90, Mad90, MIY90b, MIY90a].

Various groups have worked on applying BDD techniques to the verification of sequential circuits. In [CBM89, CBM90] Coudert *at al.* describe a verification method based on equivalence checking between deterministic Moore machines. Bose and Fisher [BF90] have used a BDD-based logic simulator to implement a temporal logic model checker. Fujita and Fujisawa [FF90] have implemented a model checker by incorporating state information directly into BDDs. Clarke *et al.* [BCM$^+$90, BCMD90] present a very general model checking scheme, based on the Mu-Calculus, in which the entire transition relation is represented as a BDD.

Some work has been done on "working around" the inherent limitations of BDD representations. Bryant pays some attention in [Bry86] and [Bry91] to the problem of representing multiplier circuits with BDDs. Burch [Bur91] describes a technique for verifying multipliers by slightly strengthening the condition to be verified, and dividing the BDD into several BDDs representing parts of the circuit. In a similar vein, Clarke *et al.* [BCL91a, BCL91b] have used multiple small BDDs to replace one large one in model checking applications.

Aside from the work described in this paper on applications of BDDs, the authors have investigated new BDD construction algorithms [Pei92] and alternative (though related) representations (see §4) of boolean functions based on conditionals [Mon92a] and exclusive-or [BE92].

3 ELLA Representation Issues

A semantics for Boolean Kernel ELLA (BKE), a covering subset of ELLA, has previously been given in [MDED91]. It is based directly on state machine descriptions, i.e. the semantic objects *are* IO automata

— a particular form of transition-labelled finite state machine. Thus, the semantic objects themselves can be directly used for verification purposes.

In general, state-based hardware models such as IO automata will suffer from the usual problem of state or transition explosion, if the automaton is represented using the usual enumerated sets for transitions and states. In an effort to reduce the size of these structures and to improve efficiency of verification algorithms, symbolic representations for IO automata are now considered, based on BDDs.

First, we introduce IO automata and describe their use in the semantics of hardware descriptions. BDD representations for sets and relations are then given, which are used as the basis for representing IO automata using BDDs. Algorithms for checking the equivalence of deterministic finite automata are described. Finally we present an overview of the compositional semantics for BKE using IO automata as the underlying semantic objects.

3.1 IO Automata Models

The development of IO automata is part of the adaptation of well-known ideas from automata and language theory to the description of general system behaviour[VW86]. In some sense, they provide a generalisation of the *Mealy* finite state automaton from standard automata theory[HU79, Shi87]. They have some similarity to the structures of the same name studied by Lynch and Tuttle [LT88]. More formally, we take a (basic) IO automaton as a 7-tuple comprising a set of states, an input alphabet, an output alphabet, an input/output-labelled transition relation, a distinguished set of states — the initial states, a distinguished set of states – the terminal states, and a set of subsets of states — the acceptance subsets. The latter component is a generalised form of Büchi-style acceptance condition and allows for the definition of restricted sets of infinite runs of the automata. For the purpose of giving meaning to ELLA components, this aspect can be ignored as we require all infinite runs to be accepting. In fact, in the definition given in §3.2.2 we omit both terminal and acceptance sets.

The following examples show how IO automata can be used to represent design components.

3.1.1 Examples: NOR Gate and Unit-Delay

Consider a NOR gate (without delay) over the two valued set $\{0,1\}$. We model it by a single-state automaton with four transitions as shown in Figure 1(a).

The automaton has state set $states = \{n\}$ and initial state set $start = \{n\}$. The labelling on the transitions in each automaton has been simplified for presentation purposes. Each mapping from label to value has been replaced by a value only, whose position determines the label to which it is assigned. In Figure 1(a) each label has the form ab/c, which represents input labels x, y and output label o, having values a, b and c respectively. From the given labellings it is clear that the output on any transition is the "nor" of the two inputs.

Now consider a unit delay function which delays the given input by one (notional) step. It can be modelled by a two-state automaton as shown in Figure 1(b). Let the initial state be state 0. Consider the possible transitions. Starting in the initial state, if the input is 0, then the output is 0 and the state does not change. If, however, the input had been 1, then 0 would have again been output but the new state would have been state 1. The idea is that state 0 corresponds to having input a 0 on the previous transition, i.e. all incident transitions have input 0, and its emergent transitions have output 0, thus effecting a unit delay on 0 inputs. Furthermore, the incident transitions of state 1 all have input 1 and the emergent transitions have output 1, thus effecting unit delay for 1 inputs.

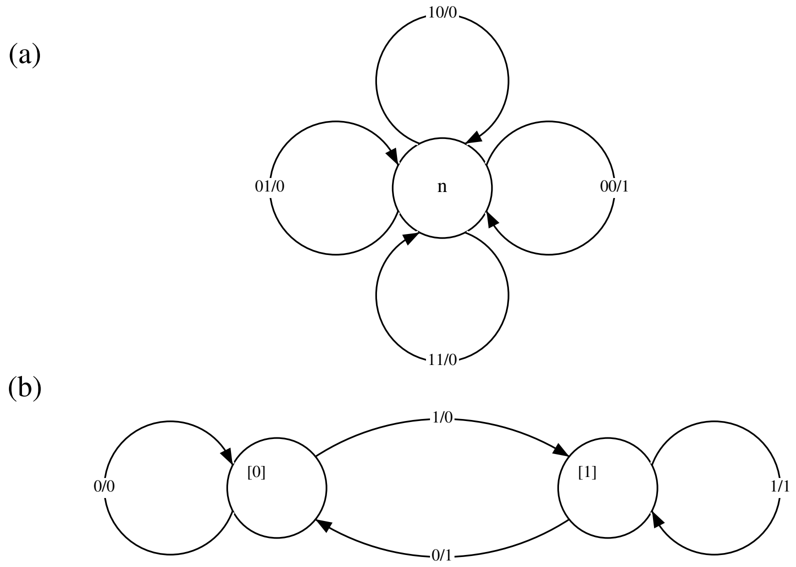

Figure 1: IO Automata for (a) Nor Gate (b) Unit-Delay

3.1.2 Basic Operations on IO Automata

The three IO automaton operations of *causal product*, *rename* and *restrict* form the basis for the construction of semantic objects for BKE.

Given two, or more, automata representing the behaviour of certain hardware components, we require some form of composition on automata which will correspond to "connecting" the hardware components together. For example, we might wish to connect a Unit Delay onto the output of a NOR gate forming a unit-delay-NOR gate. The composition required is a form of *product*. There are, in fact, many ways of forming products of automata that may be of interest in automata-based verification. In the following semantic descriptions, we use a *causal* product as the principal operation for constructing product automata. It is a "synchronous" product in that the resulting automaton produces a state transition only when each of the participants (in the product) have taken a transition of their own. In particular, it takes the usage of inputs and outputs between participants into account, so involving the notion of *causality* between participants. Furthermore, any input of one that happens to be an output of the other is hidden within the resulting product automaton. Therefore all communications exchanged between the participants are hidden from third-party external observers.

A second operation on an IO automaton that is needed is to *rename* its outputs. This may be complicated because some of the new output names may already be inputs to the automaton. In this case any resulting conflicts in the new set of transitions must be removed.

Finally, a *restriction* operation is needed so that certain automaton outputs are ignored and only a subset of the outputs are visible.

3.2 BDD Representations

In this section we present some of the issues involved in designing a layer of software on top of a BDD package à la Bryant [Bry86], which is capable of supporting the implementation of IO automaton verification tools.

3.2.1 Basic Operations

3.2.1.1 Low-level view of BDDs In Bryant's work and in our Common LISP implementation of the basic BDD operations, it is assumed that we are dealing with a fixed set of n boolean variables which have been given an ordering, so that we can replace the variables with integers from 0 to n-1. It is left to users of the package to provide a mapping between semantic objects and boolean vectors, so that these variable indices have a meaning.

3.2.1.2 High-level operations required The following two examples indicate the motivation for the introduction of the data types in §3.2.1.3. They are typical (if highly simplified) instances of the kinds of operation we need to perform in model-checking.

3.2.1.2.1 An example: DFA equivalence Consider the problem of determining equivalence between deterministic finite automata A_1 and A_2 with the same input alphabets (we ignore outputs for the moment). By "deterministic" we mean that each state has *exactly* one transition for each element of the input alphabet. To solve this problem, we compute a relation R_n between the state spaces of A_1 and A_2: $R_n(x,y)$ is true if there is some input string s of length $k \leq n$ such that the execution paths of A_1 and A_2 on input s end in states x and y respectively. For large enough n, we have $R_n = R_{n+1}$. Call this stable relation R_∞. Then A_1 and A_2 are equivalent if

For all x, y, if $R_\infty(x,y)$ then x is an accepting state if and only if y is.

Suppose we represent the transition function for a DFA A by a relation $N \subseteq S \times I \times S$, where S is the state set and I the input alphabet. We can compute R_∞ inductively, using the facts that $R_0(x,y) \equiv x = s_1 \wedge y = s_2$, where s_i is the start state of A_i, and

$$R_{k+1}(x,y) \equiv R_k(x,y) \vee \exists x', y', i \cdot R_k(x',y') \wedge N_1(x',i,x) \wedge N_2(y',i,y) \qquad (1)$$

In fact the whole computation can be performed "symbolically"—in terms of disjunction, conjunction and quantification, without explicitly enumerating the state space or the input alphabet—provided we represent all the relevant sets and relations by BDDs.

The relation R_k appears twice on the right hand side of equation (1) with different arguments. So we need to be able to substitute variables into a BDD. We have a method for substituting a sequence of BDDs into the function represented by some other BDD, but it is very expensive in general. In the kind of situation we have here, it is possible to do the substitution much more directly. To represent the right hand side of (1) we need to order (or index) the variables corresponding to x, y, x', y' and i. For any $z \in \{x, y, x', y', i\}$, let $V(z)$ be the set of boolean variables corresponding to v. If we order variables in such a way that

- The inherited orderings on $V(x) \cup V(y)$ and $V(x') \cup V(y')$ correspond after replacing variables by their 'primed' counterparts, and are the same as the ordering which has been used to compute the BDD for R_k.

- The inherited ordering on $V(x') \cup V(i) \cup V(x)$ is the ordering which has been used to compute the BDD for N_1.
- As above with y in place of x and N_2 in place of N_1.

Then the BDDs for the two occurrences of R_k have the same structure and we can compute one from the other by simply copying the graph, re-indexing the vertices as we go. So if we can choose an ordering which satisfies the criteria given above, and which keeps the BDD representations of the R_k and N_i reasonably small, we can save a great deal of both space and time.

Such considerations of variable ordering have little or nothing to do with abstract operations on relations. But it is possible that the only way to find a good ordering is by some heuristics based on wherever we got the equivalence problem from in the first place—the application. So it must be possible for the user of the relational manipulation package to explicitly set variable orderings.

3.2.1.2.2 Example: Transitive Closure The situation is even more complicated in this case. Suppose we have a set S and a binary relation R over S. We want to compute the transitive closure of R. Let R_n be the relation connecting those elements of S which are joined by R in at most n steps. That is, $R_n(x,y)$ holds if there is a $k < n$ and $x_1 \ldots x_n \in S$ such that $R(x,x_1)$ and $R(x_1,x_2)$ and $\ldots$and $R(x_k,y)$. For large enough n, $R_n = R_{n+1} = R_\infty$, say. We can compute R_∞, the transitive closure of R, inductively, using the recurrence $R_1 = R$ and either

$$R_{n+1}(x,y) = R_n(x,y) \vee \exists u \in S \cdot R_n(x,u) \wedge R(u,y) \tag{2}$$

or

$$R_{2n}(x,y) = R_n(x,y) \vee \exists u \in S \cdot R_n(x,u) \wedge R_n(u,y) \tag{3}$$

The right-hand side of equation (3) has three occurrences of the relation R_n. Furthermore the result, which is to be assigned to the left hand side, will be used again on the right hand side in the next iteration, so it too may have to be re-indexed.

To show that we need not abandon the wishes expressed at the end of §3.2.1.2.1, let us examine one way to perform this computation. Suppose the elements of S are defined in terms of boolean variables $v_1,\ldots,v_n$, and suppose we have a BDD for R which uses the ordering $x_1,y_1,\ldots,x_n,y_n$, where the x_i correspond to the v_i in the role of x, and similarly for the y_i. Then, letting $u_1,\ldots,u_n$ play the role of u, we can use the ordering $x_1,u_1,y_1,\ldots,x_n,u_n,y_n$ for the right hand side of equation (2) or (3). Notice that we can then compute the right hand side using only monotonic re-indexings.

So we need to define higher-level operations which will support this kind of idea—computing a composition of two relations using some particular variable ordering for the intermediate results.

3.2.1.3 Some data types This section will provides a brief description of work in progress on constructing some data types on top of a BDD package, to support the kinds of operation described above.

3.2.1.3.1 Domains and subsets In order to use BDDs to represent sets, we need to represent the (potential) elements of our sets by boolean vectors. The *Domain* data type is intended to capture the translation between the application's objects and the boolean vectors on which we can evaluate our BDDs.

$$\mathit{BoolVec} = \mathbb{B}^*$$

$$\begin{aligned} \mathit{Domain} :: \quad \mathit{arity} &: \mathbb{N} \\ \mathit{coding} &: \mathit{BoolVec} \xleftrightarrow{\mathsf{map}} \mathit{Object} \\ \mathit{used} &: \mathit{BDD} \end{aligned}$$

where

$$inv\text{-}Domain : \mathbb{N} \times (BoolVec \xleftrightarrow{\text{map}} Object) \times BDD \to \mathbb{B}$$

$$inv\text{-}Domain(ar, cod, usd) \quad \triangle \quad \mathsf{dom}\, cod = \{v \in BoolVec \mid \mathsf{len}\, v = ar \wedge bdd\text{-}eval(usd, v) = \mathsf{true}\}$$

We have omitted the formal definition of the *BDD* data type itself. The function *bdd-eval* evaluates the boolean function represented by a *BDD* on a vector of boolean values (a *BoolVec*). Note that the *used* field of *Domain* is included to distinguish those vectors which are "valid" for the translation—those which actually represent objects.

Apart from constructors, there are two principal *Domain* operations, one for each direction of translation.

$domain\text{-}object\ (d{:}\,Domain, v{:}\,BoolVec)\ o{:}\,Object$

$\mathsf{pre}\ v \in \mathsf{dom}\, coding(d)$

$\mathsf{post}\ coding(d)(v) = o$

$domain\text{-}vector\ (d{:}\,Domain, v{:}\,BoolVec)\ v{:}\,BoolVec$

$\mathsf{pre}\ o \in \mathsf{rng}\, coding(d)$

$\mathsf{post}\ coding(d)(v) = o$

Note that this specification omits any discussion of responsibility for error checking.

The sets we are modeling will be subsets of domains. A *subset* consists of a domain and a BDD which represents the characteristic function of the set. That is, the BDD evaluates to true on those boolean vectors which encode elements of the set.

$$\begin{aligned} Subset\ ::\ superset &: Domain \\ char\text{-}fn &: BDD \end{aligned}$$

For subsets of a fixed domain, we can perform the obvious operations of union, intersection, complementation, membership testing, containment testing etc. The Common LISP implementation also provides a control structure for iterating over the elements of a subset.

3.2.1.3.2 Products and relations Just as a (BDD-represented) set is a subset of a fixed domain, so a relation is a subset of a cartesian product of domains. First, we describe how we intend to model cartesian products.

The factors of a *Product* are identified by *Name*s. A variable in the product structure (a *ProdVar*) can be identified by the *Name* of the factor from which it comes and its offset within that factor. Given two domains, of arity n and m, we can of course represent elements of the product domain by boolean vectors of length $n+m$. But this does not tell us how the components of these product vectors should be mixed together. This is the purpose of the *combiner* field in *Product*. To put it another way, the *combiner* provides the ordering for the product (boolean) variables.

$$Index = \mathbb{N}$$

$$\begin{aligned} ProdVar\ ::\ factor &: Name \\ offset &: Index \end{aligned}$$

$$Product :: \quad factors : Name \xrightarrow{\text{map}} Domain$$
$$combiner : ProdVar^*$$

The interpretation of the combiner is that if the ith element of *combiner* is *mk-ProdVar*$(name, j)$ then the ith product variable corresponds the jth variable of the factor called *name*. It seems reasonable to impose an invariant that the ordering on the product variables should be consistent with the orderings on the factors. This can be expressed in terms of monotone sequences.

The *Product* data type should provide operations for translating between product objects and boolean vectors. Here a *product object* is either a sequence of objects in the appropriate domains or a map $Name \xrightarrow{\text{map}} Object$. We also need operations for extracting a sub-product of a given product (projection), restricting either to or away from a given set of field names, and for renaming the fields of a product (see below).

A relation, then, is a subset of a product

$$Relation :: \quad fields : Product$$
$$char\text{-}fn : BDD$$

The operations of intersection, union, and so on, are essentially the same for relations as for subsets. However, we need some operations which convert between relations defined over different products. For instance the projection operations induce operations in the opposite direction on relations. If we have a relation defined over a sub-product q of a product p, then it can be thought of as a relation over p by simply ignoring the extra fields. The BDD for this new relation will be a re-indexing of the old one.

To see why we need these operations on relations, consider equation (3). Suppose the first variable in $R_n(x,y)$ is named FROM and the second is named TO. We can construct R_{2n} as follows. First we construct relations X, which is R_n with TO renamed to VIA, and Y, which is R_n with FROM renamed to VIA. We then build a product P such that

$$factors(P) = \{\text{FROM} \mapsto D, \text{VIA} \mapsto D, \text{TO} \mapsto D\}$$

and such that the sub-products of P having fields $\{\text{FROM}, \text{VIA}\}$, $\{\text{VIA}, \text{TO}\}$ and $\{\text{FROM}, \text{TO}\}$ are equivalent to the product structures used for X, Y and R_n. The latter requirement just means that the variable orderings are consistent as described in §3.2.1.2.2. We then "promote" X and Y to be relations over P, take the conjunction and quantify over $\{\text{VIA}\}$. The resulting relation is over the same product as R_n and we conclude by taking its disjunction with R_n.

3.2.2 Representing IO Automata and their Operations

We now show how BDDs can be used to represent IO automata, taking the simple IO automata for NOR-gate and unit delay from §3.1 as examples. The basic IO automaton operations introduced in §3.1.2 are then described using this BDD representation.

First, two additional fields are added to the 7-tuple IO automaton structure described in §3.1, to give sets of input and output *values*. An IO automaton, without acceptance and terminal sets, is now also a 7-tuple:

$$
\begin{aligned}
IOA \ :: \quad states &: Domain \\
inputs &: \mathbb{P}(Name) \\
outputs &: \mathbb{P}(Name) \\
transitions &: Relation \\
iset &: Domain \\
oset &: Domain \\
start &: Subset
\end{aligned}
$$

The invariant on *IOA* states that the start states are genuine states of our machine and that outputs may not also be inputs. Furthermore, the transition relation is over a product whose fields are named *from*, *ins*, *outs* and *to*, which map to the appropriate state-, input- or output-sets, and the input and output mappings for each transition must have appropriate domains.

$$inv\text{-}IOA : IOA \to \mathbb{B}$$

$$
\begin{aligned}
& inv\text{-}IOA(a) \quad \underline{\triangle} \\
& \quad \mathsf{let}\ mk\text{-}IOA(ss,i,o,tr,is,os,st) = a\ \mathsf{in} \\
& \quad st \subseteq ss \land st \neq \emptyset \land i \cap o = \emptyset \land \\
& \qquad factors(fields(tr)) = \{from \mapsto ss, ins \mapsto is, outs \mapsto os, to \mapsto ss\} \land \\
& \qquad \mathsf{rng}\, coding(is) = i \xrightarrow{\mathsf{map}} Val \land \mathsf{rng}\, coding(os) = o \xrightarrow{\mathsf{map}} Val
\end{aligned}
$$

3.2.2.1 Examples: Nor-Gate and Unit Delay Examples are given here for the BDD representations of the IO automata for a Nor-gate and unit delay, first described in §3.1. Only the BDDs representing the transition relations for each automaton are shown.

A BDD representing the transition relation for the NOR-gate can be constructed, if the ordering for its variables is supplied. There are three variables involved, two inputs x, y and an output n. There are no state-variables because the NOR-gate only has a single state. For the variable ordering $x < y < n$ the resulting BDD is shown in Figure 2(a). The valid input-output transitions in the IO automaton for NOR-gate are those where this BDD evaluates to true for a given variable assignment. For example, when the BDD is applied to $\{x \mapsto 0, y \mapsto 1, n \mapsto 0\}$, it evaluates to T, meaning that $01/0$ is a valid transition of the IO automaton as required. However if the BDD is applied to $\{x \mapsto 0, y \mapsto 0, n \mapsto 0\}$ it evaluates to F meaning that $00/0$ is not a valid transition in the automaton. In each transition, the initial and final states are of course the same, and have been omitted.

A BDD representation for the transition relation of unit delay can be constructed using four variables: an input i, output o, a 'from-state' s and a 'to-state' ps. If these take the order $i < ps < s < o$ then the BDD is shown in Figure 2(b), evaluating the characteristic function $i \Leftrightarrow ps \land o \Leftrightarrow s$. A valid transition results from the variable assignment $\{s \mapsto 0, i \mapsto 1, o \mapsto 0, ps \mapsto 1\}$, while an invalid transition is $\{s \mapsto 1, i \mapsto 1, o \mapsto 0, ps \mapsto 1\}$, as required.

3.2.2.2 Causal Product A presentation of causal product using a BDD representation of IO automata is now given. First, the new input and output sets are formed, with possibly some inputs being redesignated as outputs. Then the two BDDs representing the transition relations are updated so that they use a common variable indexing. Finally the new transition relation is formed by taking the conjunction of the component BDDs.

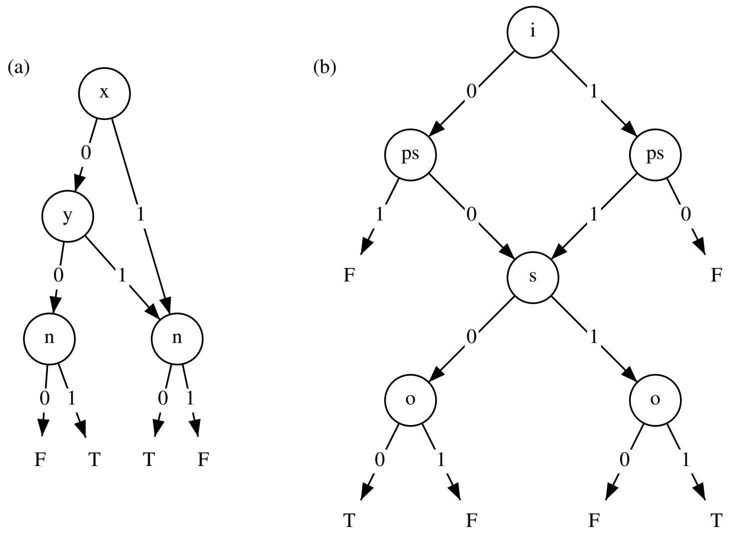

Figure 2: BDDs for (a) Nor Gate Transition Relation, (b) Unit-Delay Transition Relation

$causal\text{-}prod : IOA \times IOA \to IOA$

$causal\text{-}prod(a_1, a_2) \quad \underline{\triangle}$

let $mk\text{-}IOA(ss_1, i_1, o_1, tr_1, is_1, os_1, st_1) = a_1$ in

let $mk\text{-}IOA(ss_2, i_2, o_2, tr_2, is_2, os_2, st_2) = a_2$ in

let $combiner = make\text{-}combine(ss_1, ss_2, is_1, is_2, os_1, os_2)$ in

let $ss = product(ss_1, ss_2)$ in

let $vm = map\text{-}union(is_1, is_2, os_1, os_2)$ in

let $prod = mk\text{-}Product(\{from \mapsto ss, inout \mapsto vm, to \mapsto ss\}, combiner)$ in

$causal\text{-}product\text{-}aux(a_1, a_2, prod)$

$causal\text{-}prod\text{-}aux : IOA \times IOA \times Product \rightarrow IOA$

$causal\text{-}prod\text{-}aux(a_1, a_2, product) \quad \triangleq$

$\quad$ let $mk\text{-}Product(elts, indx) = product$ in

$\quad$ let $\{from \mapsto S, inout \mapsto VM, to \mapsto S\} = elts$ in

$\quad$ let $mk\text{-}IOA(ss_1, i_1, o_1, tr_1, is_1, os_1, st_1) = a_1$ in

$\quad$ let $mk\text{-}IOA(ss_2, i_2, o_2, tr_2, is_2, os_2, st_2) = a_2$ in

$\quad$ let $o = o_1 \cup o_2$ in

$\quad$ let $i = (i_1 \cup i_2) - o$ in

$\quad$ let $ss = product(ss_1, ss_2)$ in

$\quad$ let $os = map\text{-}union(os_1, os_2)$ in

$\quad$ let $is = map\text{-}subtract(map\text{-}union(is_1, is_2), os)$ in

$\quad$ let $st = product(st_1, st_2, indx)$ in

$\quad$ let $\tilde{N}_1 = extend(product, tr_1)$ in

$\quad$ let $\tilde{N}_2 = extend(product, tr_2)$ in

$\quad$ let $new\text{-}indx = update\text{-}index(product, i, o)$ in

$\quad$ let $prod = mk\text{-}Product(\{from \mapsto S, ins \mapsto VM_{|is}, outs \mapsto VM_{|os}, to \mapsto S\}, new\text{-}indx)$ in

$\quad$ let $tr = mk\text{-}Relation(prod, many\text{-}and\text{-}bdd(\tilde{N}_1, \tilde{N}_2))$ in

$\quad mk\text{-}IOA(ss, i, o, tr, is, os, st)$

pre $outputs(a_1) \cap outputs(a_2) = \emptyset$

The operations *product*, *map-union*, *map-subtract* perform the necessary operations to combine BDD representations of sets. The function *make-combine* produces the indexing for a product of its argument factors, and *extend* modifies the BDD representation of the characteristic function of a transition relation so that is operates over a re-indexed and possibly 'extended' product. If the re-indexing does not change the original order of variables used in the transition relation, then *extend* simply has to monotonically re-index the BDD; if the variable order is changed, then the BDD must be rebuilt. The notation '$x_{|y}$' for domains x, y means that only those variable names appearing in dom rng $coding(y)$ are 'visible', but they have a BDD-indexing given in x. Clearly dom rng $coding(y) \subseteq$ dom rng $coding(x)$ for this to be possible. In the function *causal-prod-aux*, *VM* contains all of the input and output names of the two IO automata being composed, but this must be restricted to the names in the new input or new output sets in the product for the new IO automaton. The operation $update\text{-}index(p, i, o)$ modifies a product p so that the set associated with the field *ins* contains only variable names in the set i, and the set associated with the field *outs* contains only variable names in the set o. Finally, the BDD operation *many-and-bdd* forms the conjunction of two or more BDDs.

3.2.2.3 Rename The function *rename* renames the outputs of an IOA a from $outputs(a)$ to the names contained in a list xl. It does this by constructing an IO automaton called *copy* which simply copies inputs named $outputs(a)$ to outputs named elems xl, and then takes the causal product of a and *copy*. Finally, the outputs of the IO automaton are restricted to those having names in elems xl:

$rename : SemObj \times Name^{+} \rightarrow IOA$

$rename(a, xl) \quad \underline{\triangle}$

let $n = \mathsf{len}\, xl$ in

let $outs = [1, \ldots, n]$ in

let $new\text{-}outs = \mathsf{elems}\, xl$ in

let $copy = copy\text{-}machine(outs, xl)$ in

$restrict(causal\text{-}product(a, copy), new\text{-}outs)$

pre $outputs(a) = \mathsf{inds}\, xl \wedge \mathsf{card}\,(\mathsf{elems}\, xl) = \mathsf{len}\, xl$

When performing *rename* a copying IO automaton will be needed which copies inputs with name-list *il* to outputs with name-list *ol*:

$copy\text{-}machine : Id^{+} \times Id^{+} \rightarrow IOA$

$copy\text{-}machine(il, ol) \quad \underline{\triangle}$

let $n = \mathsf{len}\, il$ in

let $ss = make\text{-}map\text{-}set(0, \{0\}, T)$ in

let $st = ss$ in

let $is = make\text{-}map\text{-}set(n, \mathsf{elems}\, il, T)$ in

let $os = make\text{-}map\text{-}set(n, \mathsf{elems}\, ol, T)$ in

let $factors = \{from \mapsto ss, ins \mapsto is, outs \mapsto os, to \mapsto ss\}$ in

let $combiner = make\text{-}combine(factors)$ in

let $pr = mk\text{-}Product(factors, combiner)$ in

let $tr\text{-}bdd = mk\text{-}BDD\text{-}Fn(pr, equal(il, ol))$ in

let $tr = mk\text{-}Relation(pr, tr\text{-}bdd)$ in

$mk\text{-}IOA(ss, \mathsf{elems}\, il, \mathsf{elems}\, ol, tr, is, os, st)$

pre $\mathsf{len}\, il = \mathsf{len}\, ol$

The operation $make\text{-}BDD\text{-}Fn(p, e)$ creates a BDD to represent a boolean expression e which uses variables contained in a *product* p. The operation $make\text{-}map\text{-}set(n, v, e)$ creates a BDD representation of a mapping using a domain with arity n, which has variable names contained in v, with legal evaluations determined by expression e. The operation $equal(x, y)$ produces a boolean expression $x(1) \Leftrightarrow y(1) \wedge \ldots \wedge x(n) \Leftrightarrow y(n)$ for two variable-name sequences x and y of length n.

3.2.2.4 Restrict The function *restrict* modifies the transitions of IO automaton a so that it only outputs those values associated with the output names contained in xs:

$restrict : IOA \times \mathbb{P}(Name) \rightarrow SemObj$

$restrict(a, xs) \quad \underline{\triangle}$

let $mk\text{-}IOA(s, i, o, tr, is, os, st) = a$ in

let $remove = o - xs$ in

let $s = arity(os)$ in

let $k = \mathrm{card}\, xs$ in

let $new\text{-}os = map\text{-}restrict(xs, os)$ in

let $new\text{-}tr = quantify(remove, tr)$ in

$\mu(a, transitions \mapsto new\text{-}tr, outputs \mapsto xs, os \mapsto new\text{-}os)$

pre $xs \subseteq outputs(a)$

The function *quantify* quantifies out the BDD-variables associated with names in the set *ns* from the transition relation *tr*; it updates as necessary the product over which *tr* is taken. The function $map\text{-}restrict(ns, d)$ removes input/output names which are not in the set *ns* from the domain *d* representing the set of possible input/output mappings, while retaining the ordering on BDD variables which provide the value for each name.

3.3 BDD Representation of the IO Automaton Semantics for BKE

BKE includes constructs for constants, variables, case, recursive let, function application, two types of delay and certain built-in operators such as *nor*. An overview of the BDD representation of the IO automaton Semantics for BKE is now given, although no attempt is made to cover the BDD representation of the complete semantics — see [Wil91] for further details. It should be stressed that the semantics for BKE is not being changed here, but only the method used for representing the IO automata involved.

Having already given the BDD representation for the basic IO automaton operations of *causal product*, *rename* and *restrict*, then for the function composition operations in BKE, such as *recursive let*, no further changes are required to presentation of the semantics given in [MDED91].

Changes do have to be made to the descriptions of the semantics for the atomic function constructors such as *delay* or *case*; they follow a common pattern, similar to that illustrated by the above definition of the operation *copy-machine*. They first require BDD representations to be created for the underlying sets of states, input and output maps, and set product. Then the BDD representation of the transition relation is formed over the product; component transition relations are first created for each state- or output-bit involved, and then the conjunction of these relations is taken to form the complete transition relation for the function.

3.4 Results and Comparisons

3.4.1 Compiling by Enumeration - Efficiency

Here we briefly review our experience concerning compilation by enumeration.

The denotational semantics of the language BKE was defined in terms of IO automaton structures [BGL+91, MDED91]. This showed how BKE constructs could be mapped into IO automaton structures in a compositional manner. This approach has similarities with Milner's use of CCS to give semantics to a simple concurrent language [Mil90].

This semantics was formulated through using various primitive operations on automata such as *causal product* and *renaming*. In turn, these operations were defined in terms of their effects upon the natural set-theoretical structures, such as partial mappings and relations.

As the structures defined via BKE (which includes recursive let constructs) are all of finite size, the formulation enabled an executable version of the full semantics to be directly constructed within Common LISP.

3.4.1.1 Testing the Semantics Several benefits followed from the executability of the semantic description. Firstly, the mechanised version semantics could then be used to compute results from very simple test cases and provide an indirect test of the semantics. Secondly, it enabled the generation of test data for our automata equivalence checking procedures. As a result of mechanising the semantics, we were therefore able to examine some of its consequences directly and so increase our confidence in it.

Clearly, such a level of mechanisation is but a pale reflection of that demanded to support an adequate proof theory, and is clearly no substitute for it. However, even with such limitations, it was possible to use this trial compiler[1] to expose flaws and inaccuracies in the semantics as given and to provide an opportunity to correct them.

3.4.1.2 Enumeration and Performance As we have indicated above and in earlier sections, the trial compiler uses enumerative algorithms to compute the automaton structures corresponding to BKE texts. Such a technique proceeds by building appropriate automata for atomic terms, such as variables and constants and also atomic operators such as boolean connectives and user-defined functions. More complex constructs such as case, let, the delay operators and function combination then essentially provide ways of combining their arguments using causal product, renaming and so on. Enumerative algorithms for these primitives also follow directly from their definitions. The compiling algorithm itself operates by recursing over the syntax tree and then using appropriate calls to semantic operators.

Although not completely naïve, the enumerative algorithms were intended to reflect the semantics in a transparent fashion. Such an approach enables the location of errors to be more easily associated with either the particular code or the semantics. As anticipated, this implementation approach leads to severe performance problems and can only be used for reasonably small examples. For instance, the largest example we have compiled is a 4-bit ALU, whose semantics is a deterministic, single state IO automaton with 14 inputs, 5 outputs and 2^{14} transitions. This is illustrative of how a so-called 'state explosion' is traded for a 'transition explosion' — the ELLA HDDL is especially suited to generating functionally complex systems which may also have a substantial use of memory. The important issues of state/transition explosion in the context of BDD representations are discussed in the following section.

3.4.2 BDD Efficiency Issues

There are several reasons for preferring to work with a symbolic representation of IO automata, such as BDDs, rather than a more direct representation which lists all the states and transitions in some way.

One of the advantages of representing circuits by IO automata is that we are often able to model systems by automata with fewer states than if we were using, say, Moore machines. In the representation of a given system there will tend to be more transitions (with different input/output values) between a given pair of states. So for each pair of states there is a (possibly empty) relation from input values to output values. We would like to use a compact representation of these relations.

[1]It is called this because of the large amount of time taken to produce results on 'large' examples such as a 4-bit ALU!

On the other hand, some systems are inherently state-bound. That is, they have large numbers of states which are only loosely connected, in that almost all the transition sets between pairs of states are empty. We do not want these large state sets with all these empty mappings and relations to be represented explicitly. In the following paragraphs we present an example at each end of this state *versus* transition spectrum, to demonstrate how the BDD representation is able, in favourable circumstances, to cope with both problems at once.

Some systems have large numbers of transitions but small state sets. A purely combinatorial BKE description (one with no delays) translates to an IO automaton with a single state. The transition relation for such an automaton reduces to an input-output relation (which for a deterministic description must be a mapping). Consider the case of an n-bit adder. The IO automaton for such a circuit has one state with 2^{2n} self-transitions. With a judicious choice of variable ordering, scalable for different n, we can represent the transition relation of this automaton with a BDD whose size increases linearly with n.

Other systems have large numbers of states but very regular transition structures, so that again the BDD representation may be very compact. For instance, let A be a counter automaton which simply outputs n-bit binary numbers in sequence from 0 to 2^n-1 and then starts again from 0. Then A must have at least 2^n states (and transitions). Yet here again we can represent the transition relation of the minimal model of A by a BDD whose size is linear in n. Explicit enumeration of the state space would result in using a large amount of space to represent a small amount of information.

4 Alternative Approaches

4.1 if - Conditional Switching Operators

In this section, an approach is described which tries to lessen one of the drawbacks of BDDs – finding a suitable variable ordering upon which to build a canonical representation.

The idea is to avoid using *canonical* forms until they are needed, such as when checking two expressions for equivalence. We do this by relaxing the form in which a propositional expression is expressed during intermediate computations. However, it is still necessary to impose structural invariants to achieve normal and canonical forms – the approach simply allows a choice of *when* the ordering invariants are imposed.

The starting point is to take the well-known 'if-then-else' conditional operator (denoted by **if**):

$$\mathbf{if}(\mathit{Condition}, \mathit{Then}, \mathit{Else})$$

and to develop its algebra to give representations of boolean functions. For the implementation, we intend to adopt a DAG representation of **if** terms and other boolean terms so that potential sharing is exploited as far as possible, using standard implementation techniques (cf. [BRB90]). In the following section, we discuss the variable ordering problem and the intuition behind our approach to lessening its influence upon predicate representations.

4.2 The Problem

To construct a BDD representation for a predicate, a variable ordering is needed to ensure that the representation is canonical. Typically, this ordering is determined prior to the building of a representing BDD. Different orderings would produce differently shaped representative BDDs which may involve different numbers of internal nodes.

We believe that this forces a premature choice of variable ordering precisely when least is known about the predicate's semantics. Moreover, it is upon this semantics that a reasonable choice of variable ordering should be based, unless heuristic techniques are employed. But this is immediately a double bind, since the associated analysis will itself need a semantical representation of the predicate e.g. a BDD representation or similar. From this we see that an inexpensive choice of variable orderings is only possible when relatively little analysis is done beforehand. As such, any choice of variable ordering can only be influenced by attributes that are directly and cheaply extracted from the syntactic presentation. The upshot of this is that it is hard to find variable orderings that give reasonable representations of arbitrarily given predicates.

In fact the situation is worse than this — there may be *no* reasonable BDD representation at all for particular predicates. For example, Bryant has shown that all BDD representations of the n-bit integer multiplication function are always of size exponential in n, no matter which variable ordering is used [Bry91]. The result is robust in the sense that minor tinkering with the definition of BDDs is unlikely to produce much of an improvement, if any. Furthermore, an argument due to Shannon suggests that, since there are 2^{2^n} boolean function of arity n, only a vanishingly small percentage of the boolean functions of given arity could have a reasonably sized canonical form[Sha49]. Such results suggests to us that the problem lies not just with the use of BDDs, but with the management and usage of canonical forms in general.

Further evidence that canonical forms are hard to manage comes from the fact that the combination or processing of several BDD structures usually requires that all such structures have compatible variable orderings. To ensure that this is the case may then involve the rearrangement or at least the re-indexing of one or more structures, which can be an expensive process in the worst case. In any system giving verification support for non-trivial applications, semantic representations (such as BDDs) are very likely to come from a variety of independent sources and as such are therefore likely to use incompatible variable orderings. In short, the guaranteed compatibility of variable orderings is unlikely to be achievable in practice.

Our approach involves the investigation of a more general class of representation which has the following objectives and properties.

- To obtain a semantic representation of a given predicate without concern for structural invariants such as variable orderings and so on. In particular, it should be possible to calculate this representation without incurring excessive overheads due consistency or optimal selection requirements.

- To in principle combine arbitrary structures without consistency requirements on the constituents. Although it may be more efficient to exploit such requirements during processing, it should not be strictly necessary to do so.

  In short, input structures need not be in canonical form for operations to be generally applicable.

- To permit a degree of latitude in the representation of structures so that implementations are free to make expedient choices on-the-fly, during computation. This permits implementations to choose the representations of intermediate values appropriate to the context and the natural flow of computation.

- To ensure that BDD structures are included within the representation.

This last property says that a generalisation of BDDs is needed. Fortunately, a suitable candidate is easily found, in the form of the **if** expression. Given a BDD in the form of an acyclic graph, it easy to see how to construct, by bottom-up traversal, an equivalent **if** expression representing the predicate denoted by the BDD.

4.3 Anticipated Benefits and Drawbacks of the Approach

We anticipate the following benefits from adopting an approach based on the above requirements.

- Flexibility of structural representation during processing. This permits implementations to avoid putting the results into unnecessarily restrictive normal forms for later processing.
- On-the-fly selection of equivalent representatives is permitted, so allowing implementations to make choices according to internal criteria.
- Use of particular normal forms (see below) is not forced and can therefore be chosen by the implementation.
- Representatives of input expressions can be found that are of similar complexity to the given expression. Although such forms are likely to be far from being in any normal form, basic logical operations can be performed on them nonetheless.

The main disadvantages we anticipate are:

- computational effort is needed to place formulae into canonical forms when required, such as when testing for equality.
- non-optimal sharing of representations of equivalent sub-expression due to the use of a representation that is potentially non-canonical which, in the worst case, can lead to much larger structures than for conventional BDDs.

We anticipate that the first disadvantage is acceptable because tests for equality are relatively rare in comparison to other logical operations. The second disadvantage is potentially far more serious. We suspect that the flexibility gained by not fixing an ordering early on in the lifetime of a representation will have to be paid for by introducing *ordering constraints* later on dynamically. Further investigation of these and other factors therefore needs to be undertaken.

4.4 Summary of Technical Results

A detailed presentation of the work derived so far is given in a draft working paper [Mon92a]. We give a brief summary of what has so far been achieved.

- Definition of the **if** operator and analysis of its variants, the propositional completeness of the **if** algebra, and rules for transforming **if** expressions into other **if** expressions. In addition, we document the interactions between **if** expressions and other boolean connectives and operators.
- A number of normal forms are developed for the **if** algebra, culminating in Bryant's canonical form for conventional BDDs (i.e. Reduced, Ordered Normal Form (RONF)). It turns out that we do not need to maintain this normal form during calculations and that a somewhat weaker normal form is often all that is needed (i.e. Literal Normal Form (LNF) or Reduced Normal Form(RNF)). The disadvantage is that neither LNF nor RNF is canonical and so do not admit a cheap and direct form of equivalence check in the way that RONF does.
- Provision of various meta operations such as evaluation and substitution (vital for later applications). Rules for efficient simultaneous substitution (based on the use of mappings) are given.

- Quantified Boolean Algebra is introduced and rules for efficient multiple quantification are given. The interaction between other operators is developed.
- A theory of existential and universal composition forms (represented as boolean expressions) is briefly developed, together with rules for efficient implementation.

In another working paper[Mon92b], a form of Relational Calculus is defined in which boolean predicates and combinators can be hierarchically defined and used. Predicates may be defined by simple recursion — mutual recursive definition is not supported. Such definitions can be represented as 'parametric' **if** expressions containing application terms.

We have also started work on providing a Vector Relational Calculus which introduces sum and product types whose values are nested, boolean tuples with a possible labelling. Further work will extend these working documents and also consider implementation strategies.

4.5 Xor Representation

This section describes an alternative approach to the problem outlined in the previous section, namely, providing an alternative to BDDs. One particular problem addressed by the approach outlined here is that of the exponential complexity of the BDD representation of the the n-bit multiplier. The fact that this complexity is *independent* of the variable ordering chosen was conjectured by Bryant and then subsequently proved by the same author in [Bry91] where a lower bound of $2^{n/8}$ is obtained for the n-th bit graph. Experimental results however indicate that the bits have much more complex BDDs (probably double exponential in size). For instance for n=8, the graph has a size of around 5000 vertices, but for n=10 this grows to more than 100 000 nodes and thus the BDD approach becomes impractical.

An examination of the integer multiplier, using the shifted partial products method indicates that the essential operation is multiple addition. Now the most appropriate boolean function for addition is the exclusive or. Furthermore the exclusive or enjoys many useful properties, and in fact any boolean function has a canonical exclusive normal form (XNF). We can therefore modify the Bryant's BDDs to take advantage of this canonical form, to obtain the exclusive decision diagram (XDD) of boolean functions. We have implemented a program to determine the XDDs for the integer multiplier output bits for word sizes up to 10. We found that the graphs are much more compact than the respective BDDs, though they still grow exponentially with a rate of around 3. Our method does have disadvantages over the BDD approach. In particular, the operations of OR or AND on two XDDs could result in an exponential blow-up in the worst case. However XDDs genuinely differ from the BDDs. In fact it will be shown that there are simple functions with a small XDD but which convert to an exponential BDD, and conversely.

5 Results, Future Directions and Conclusions

This report has described work at Manchester which is very much in progress. A BDD representation for IO automata is given, and is used to provide a BDD-based semantics for the HDDL called BKE. An algorithm for a symbolic equivalence-checker for deterministic finite automata is presented. Finally, two alternative representations for Boolean functions have been introduced.

There is presently no compiler to take BKE expressions and directly produce BDDs representing their semantics. However, a translator has been built that produces BDD representations of IO automata from the 'enumerated' representations produced from BKE-text by the prototype compiler. This allows a mechanism for checking BKE designs using the BDD-based equivalence-checker.

Encouraging results have been obtained with this compiler/translator system — the prototype BKE compiler will now be modified so that it produces BDDs *directly* from BKE. Verification tools are to be extended so that they can perform containment checks on designs and produce trace-backs to assist in the case when a verification check fails. The **if** approach to boolean function representation will be prototyped and then benchmarked for comparison with the BDD approach. The work so far deals only with synchronous systems but it is being broadened to cover asynchronous models.

References

[Ake78] S. B. Akers.
Binary decision diagrams.
IEEE Transactions on Computers, C-27(6):985–999, 1978.

[BCL91a] J. R. Burch, E. M. Clarke, and D. E. Long.
Representing circuits more efficiently in symbolic model checking.
In *DAC91*, 1991.

[BCL91b] J. R. Burch, E. M. Clarke, and D. E. Long.
Symbolic model checking with partitioned transition relations.
In *VLSI91*, Edinburgh, 1991.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang.
Symbolic model checking: 10^{20} states and beyond.
In *LICS Proceedings*, 1990.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and David L. Dill.
Sequential circuit verification using symbolic model checking.
In *27th ACM/IEEE Design Automation Conference*, 1990.

[BE92] Behnam Bani-Eqbal.
Exclusive normal form of boolean circuits.
Working paper, University of Manchester, 1992.

[BF90] Suomitra Bose and Allan L. Fisher.
Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic.
In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification—VLSI Design Methods II*. Elsevier Science Publishers B. V. (North-Holland), 1990.

[BGL+91] H. Barringer, G. Gough, T. Longshaw, B. Monahan, M. Peim, and A. Williams.
Semantics and verification for Boolean Kernel ELLA using IO automata.
In *Advanced Research Workshop on Correct Hardware Design Methodologies (CHARME'91)*, Turin, Italy, May 1991.

[BRB90] K.S. Brace, R.L. Rudell, and R.E. Bryant.
Efficient implementation of a BDD package.
In *27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[Bry86] Randal E. Bryant.
Graph-based algorithms for boolean function manipulation.
IEEE Transactions on Computers, C-35(8):677–691, 1986.

[Bry91] Randal E. Bryant.
On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication.
IEEE Transactions on Computers, 40(2):205–213, 1991.

[Bur91] Jerry R. Burch.
Using BDDs to verify multipliers.
In *DAC91*, 1991.

[CBM89] Oliver Coudert, Christian Berthet, and Jean Christophe Madre.
Verification of synchronous sequential machines based on symbolic execution.
In *Automatic Verification Methods for Finite State Machines*, pages 365–373, 1989.
Springer Lecture Notes in Computer Science, Volume 407 (Proceedings of the Workshp on Automatic Verification Methods for Finite State Systems, Grenoble 1989).

[CBM90] Olivier Coudert, Christian Berthet, and Jean Christophe Madre.
Verification of sequential machines using boolean functional vectors.
In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification—VLSI Design Methods II*. Elsevier Science Publishers B. V. (North-Holland), 1990.

[FB90] Allan L. Fisher and Randal E. Bryant.
Performance of COSMOS on the IFIP workshop benchmarks.
In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification—VLSI Design Methods II*. Elsevier Science Publishers B. V. (North-Holland), 1990.

[FF90] Masahiro Fujita and Hisanori Fujisawa.
Specification, verification and synthesis of control circuits with propositional temporal logic.
In J. A. Darringer and F. J. Rammig, editors, *Computer Hardware Description Languages and their Applications*. Elsevier Science Publishers B. V. (North-Holland), 1990.

[HU79] J.E. Hopcroft and J.D. Ullman.
Introduction to Automata Theory, Languages and Computation.
Addison Wesley, 1979.

[Lee59] C. Y. Lee.
Representation of switching circuits by binary-decision programs.
Bell System Technical Journal, 38:985–999, 1959.

[LT88] N.A. Lynch and M.R. Tuttle.
An Introduction to Input/Output Automata.
Technical Report MIT/LCS/TM-373, MIT, November 1988.

[Mad90] Jean Christophe Madre.
Benchmarks for tautology checking—experimental results.
In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification—VLSI Design Methods II*. Elsevier Science Publishers B. V. (North-Holland), 1990.

[MB88] J-C. Madre and J-P. Billon.
Proving circuit correctness using formal comparison between expected and extracted behaviour.
In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 205–210, 1988.

[MDED91] University of Manchester and RSRE Malvern DRA Electronics Division.
Semantics and Verification Foundations for ELLA.
Technical Report IED 4/1/1357 Deliverable D1.1+D2.1, May 1991.

[Mil90] R. Milner.

Concurrency and Communication.
Prentice Hall, 1990.

[MIY90a] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima.
Fast tautology checking using shared binary decision diagram—benchmark results.
In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification—VLSI Design Methods II*. Elsevier Science Publishers B. V. (North-Holland), 1990.

[MIY90b] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima.
Shared binary decision diagram with attributed edges for efficient boolean function manipulation.
In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification—VLSI Design Methods II*. Elsevier Science Publishers B. V. (North-Holland), 1990.

[Mon92a] B. Monahan.
Logical representations using **if**.
Working paper, University of Manchester, 1992.

[Mon92b] B. Monahan.
Relational calculus using **if** expressions.
Working paper, University of Manchester, 1992.

[Pei92] M. D. Peim.
Algorithms for BDD construction.
Working paper, University of Manchester, 1992.

[Sha49] C.E. Shannon.
The synthesis of two-terminal switching circuits.
Bell Syst. Tech. J., 28:59–98, 1949.

[Shi87] M. Shields.
An Introduction to Automata Theory.
Blackwell, Oxford, 1987.

[VW86] M.Y. Vardi and P. Wolper.
Automata Theoretic techniques for Modal Logics of Programs.
Journal of Computer and System Sciences, 32(2):183–221, April 1986.

[Wil91] Alan Williams.
A BDD Representation of the IO Automaton Semantics for Boolean Kernel ELLA.
Technical Report IED 4/1/1357 D2.2a(1), in preparation, Department of Computer Science, University of Manchester, December 1991.