Abstract BDDs: A Technique for Using Abstraction in Model Checking ⋆

Edmund Clarke, Somesh Jha, Yuan Lu, and Dong Wang

Carnegie Mellon University, Pittsburgh, PA 15213, USA
{emc,sjha,yuanlu,dongw}@cs.cmu.edu

Abstract. We propose a new methodology for exploiting abstraction in the context of model-checking. Our new technique uses abstract BDDs as its underlying data structure. We show that this technique builds a more refined model than traditional compiler-based methods proposed by Clarke, Grumberg and Long. We also provide experimental results to demonstrate the usefulness of our method. We have verified a pipelined carry-save multiplier and a simple version of the PCI local bus protocol. Our verification of the PCI bus revealed a subtle inconsistency in the PCI standard. We believe this is an interesting result by itself.

Keywords: Abstract BDDs, Model checking, and abstraction.

1 Introduction

Model-checking has attracted considerable attention because of existence of tools that can automatically prove temporal properties about designs. However, the state-explosion problem still remains a major hurdle in dealing with large systems. Most approaches for solving the state-explosion problem fall into two major categories: *efficient data-structures* and *state reduction techniques*. Symbolic model checking, which uses Binary Decision Diagrams(BDDs) [1, 2, 14] is an example of the first approach. State reduction methods apply transformations to the system and model-check the transformed system instead of the original system. Examples of such techniques are abstraction [6], symmetry reduction [4, 7, 10], and partial order reduction [8, 15].

Among the state-reduction approaches mentioned above, manual abstraction is the most widely used technique. However, this method is ad hoc and error-prone. Furthermore, since different properties of large systems usually require different abstraction techniques, manual abstraction is often difficult to use. For this reason, property driven automatic abstraction techniques are desirable for verifying actual hardware designs. Clarke, Grumberg and Long [6] have proposed

⋆ This research is sponsored by the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294 and the National Science Foundation (NSF) under Grant No. CCR-9505472. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, or the United States Government.

a compiler-based abstraction technique. Their technique applies abstraction directly to variables during the compilation phase. It is automatic and efficient.

However, there are true ACTL properties which cannot be proved using their techniques since the compiler-based abstraction introduces many spurious behaviors. In this paper, we propose an automatic post-compilation abstraction technique using *abstract BDDs* (aBDDs) to avoid such problems. Our approach is closely related to those discussed in [6] and [13]. The difference is that we apply abstraction after the partitioned transition relation for the system has been constructed. The advantage of our approach is that we produce a *more refined* model of the system than theirs. Therefore, we can prove more properties about the system. Note that extraction of a partitioned transition relation is usually possible even though model checking may not be feasible.

Intuitively, an abstract BDD collapses paths in a BDD that have the same abstract value with respect to some abstraction function. They were originally used to find errors in combinational circuits. In this paper, we show how they can be used to provide a general framework for generating abstract models for sequential circuit designs. Our methodology consists of the following steps:

- The user provides the abstraction function along with the partitioned transition relation of the system represented in terms of BDDs. The user generally indicates how a particular variable should be abstracted.
- Next, the *abstract BDDs* corresponding to the transition relation are built. The procedure for building the abstract BDDs will be described in detail later in this paper.
- ACTL properties of the system are checked using the *abstract BDDs* determined by the transition relation.

We have modified SMV [14] in order to support our methodology. Notice that once the user provides the abstraction function, the entire methodology is automatic. We have selected two different designs to demonstrate the power of our approach. The first is a pipelined carry-save multiplier similar to the one described in Hennessy and Patterson [9]. The second is the PCI Local Bus Protocol [16]. We show that our technique can be used to establish correctness of certain key properties of these designs.

The paper is organized as follows. In Section 2, we provide a brief overview of how abstraction is used in model checking. We also explain how abstract BDDs are constructed. Section 3 provides a modified definition for abstract BDDs that is more suitable for the purposes of this paper. Section 4 discusses how abstract BDDs can be used to build an abstract model of a sequential circuit. Experimental results are provided in Section 5 which demonstrate that our method of using abstraction with model checking is superior to the one described in [6]. Section 6 concludes with some directions for future research.

2 Background

Throughout this paper, we assume that there are n state variables $x_1, \cdots, x_n$ with a domain $D = \{0,1\}^k$. The abstract state variables $\hat{x_1}, \hat{x_2}, \cdots, \hat{x_n}$ take values in an arbitrary domain A. For each x_i there is a *surjection* $h_i : D \rightarrow A$, which is the *abstraction function* for that variable. When it is clear from the context, we will suppress the index and write the abstraction function simply as $h : D \rightarrow A$. The abstraction function h induces an equivalence relation $\equiv_h$ on D as follows:

$$(d_1 \equiv_h d_2) \leftrightarrow h(d_1) = h(d_2).$$

The set of all possible equivalence classes of D under the equivalence relation $\equiv_h$ is denoted by $[D]_h$ and defined as: $\{[d] | d \in D\}$. Assume that we have a function $rep : [D]_h \rightarrow D$ that selects a unique representative from each equivalence class $[d]$. In other words, for a 0-1 vector $d \in D$, $rep([d])$ is the unique representative in the equivalence class of d. Moreover, the abstraction function h generates an abstraction function $\mathcal{H} : D \rightarrow D$ as follows:

$$\mathcal{H}(d) \;=\; rep([d]).$$

We call $\mathcal{H}$ the *generated abstraction function*. From the definition of $\mathcal{H}$ it is easy to see that $\mathcal{H}(rep([d])) = rep([d])$. Notice that the image of D under the function $\mathcal{H}$ is simply the set of representatives. The set of representatives will be denoted by $Img(\mathcal{H})$.

2.1 Abstraction for ACTL

Given a structure $M = (S, S_0, R)$, where S is the set of states, $S_0 \subseteq S$ is a set of initial states, and $R \subseteq S \times S$ is the transition relation, we define the *existential abstraction* $M_h = (S_h, S_{0,h}, R_h)$ as follows:

$$S_{0,h} = \exists x_1 \cdots x_n [h(x_1) = \hat{x}_1 \wedge \cdots \wedge h(x_n) = \hat{x}_n \wedge S_0(x_1, \cdots, x_n)]$$

$$R_h = \exists x_1 \cdots x_n \exists x_1' \cdots x_n' [h(x_1) = \hat{x}_1 \wedge \cdots \wedge h(x_1') = \hat{x'}_1 \wedge \cdots \wedge R(x_1, \cdots, x_1', \cdots)]$$

In [6], the authors define a relation $\sqsubseteq_h$ between structures. For a structure $\tilde{M} = (\tilde{S}, \tilde{S}_0, \tilde{R})$, if

1. $S_{0,h}$ implies $\tilde{S}_0$ and
2. R_h implies $\tilde{R}$

then we say that $\tilde{M}$ *approximates* M (denoted by $M \sqsubseteq_h \tilde{M}$). Intuitively, if $M \sqsubseteq_h \tilde{M}$, then $\tilde{M}$ is *more* abstract than M_h, i.e., has more behaviors than M_h.

Since the number of states in the abstract structure M_h is usually much smaller than the number of states in M, it is usually desirable to prove a property on M_h instead of M. However, building M_h is often computationally expensive. In [6], Clarke, Grumberg and Long define a practical transformation $\mathcal{T}$ which applies the existential abstraction operation directly to variables at the innermost level of the formula. This transformation generates a new structure $M_{app} = (\mathcal{T}(S), \mathcal{T}(S_0), \mathcal{T}(R))$ and $M \sqsubseteq_h M_{app}$. As a simple example consider a system M which is a composition of two systems M_1 and M_2, or in other words $M = M_1 \| M_2$. In this case M_{app} is equal to $M_{1,h} \| M_{2,h}$. Note that the existential abstraction operation is applied to each process individually. Since $\mathcal{T}$ is applied at the innermost level, abstraction can be performed before building the BDDs for the transition relation. This abstraction technique is usually fast and easy to implement. However, it has potential limitations in checking certain properties. Since M_{app} is a coarse abstraction, there exist many properties which cannot be checked on M_{app} but can still be verified using a finer approximation. The following small example will highlight some of these problems.

A sensor-based traffic light example is shown in Figure 1. It includes two finite state machines (FSMs), one for a traffic light and one for an automobile. The traffic light M_t has four states $\{$*red, green1, green2, yellow*$\}$, and the automobile M_a also has four states $\{$*stop1, stop2, drive, slow*$\}$. M_t starts in the state *red*, when it senses that the automobile has waited for some time (in state *stop2*), it triggers a transition to state *green1* which allows the automobiles to move. M_a starts from state *stop1* and transitions according to the states of M_t. The safety property we want to prove is that when traffic light is red, the automobile should either slow down or stop. The property given above can be written in ACTL as follows:

$$\phi \ \equiv \ \mathbf{AG}[\neg(State_t = red \wedge State_a = drive)]$$

The composed machine is shown in Figure 1(c). It is easy to see that the property ϕ is true. Let us assume that we want to collapse states $\{$*green1, green2, yellow*$\}$ into one state *go*. If we use the transformation $\mathcal{T}$, which applies abstraction before we compose M_t and M_a, property ϕ does not hold (the shaded state in Figure 1(d)). On the other hand, if we apply this abstraction after composing M_t and M_a, states *(green2, drive)* and *(yellow, drive)* are collapsed into one state(Figure 1(c)), and the property ϕ still holds. Basically, by abstracting the individual components and then composing we introduce too many spurious behaviors. Our methodology remedies this disadvantage by abstracting the transition relation of the composed structure $M_t \| M_a$.

The methodology presented in this paper constructs an approximate structure $\tilde{M}$ which is more precise than the structure M_{app} obtained by the technique proposed in [6]. All the transitions in the abstract structure M_h are included in both $\tilde{M}$ and M_{app}. Note that the state sets of M_h, $\tilde{M}$ and M_{app} are the same. The relationship between M, M_h, $\tilde{M}$ and M_{app} is shown in Figure 2. Roughly speaking, $\tilde{M}$ is a more refined approximation of M_h than M_{app}, or $\tilde{M}$ has less extra behaviors than M_{app}.

2.2 Abstract BDDs

In this subsection, we briefly review abstract BDDs. Additional information about this data structure can be found in [11]. Intuitively, an abstract BDD collapses paths in a BDD that have the same abstract value with respect to some abstraction function. The concepts underlying abstract BDDs are most easily explained using Binary Decision Trees (BDTs) but apply to BDDs as well. A binary decision tree (BDT) is simply a BDD in which there is no graph sharing.

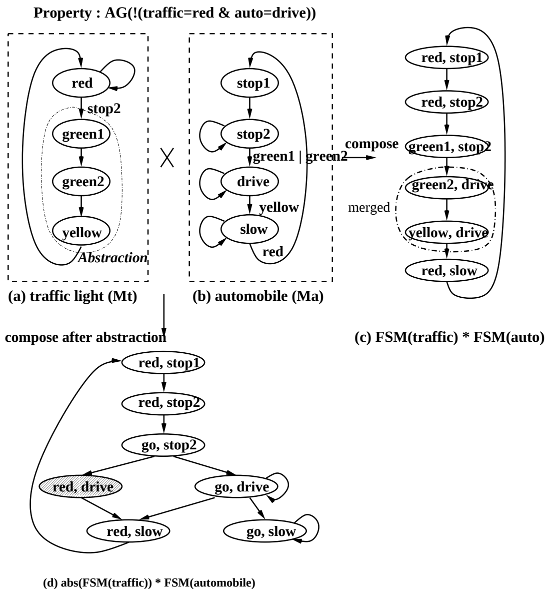

Fig. 1. Traffic light example

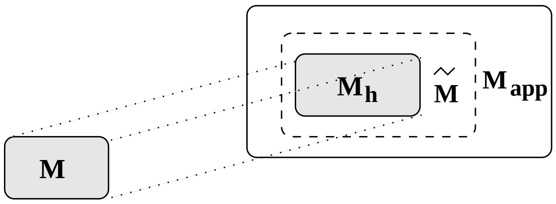

Fig. 2. Relationship between different structures

Given a boolean function $f : \{0,1\}^n \to \{0,1\}$ and its corresponding BDT T_f, let $\boldsymbol{v}$ denote the path from root to the node v at level $k+1$. It is easy to see that the path is a 0-1 vector in the domain $D = \{0,1\}^k$, i.e. $\boldsymbol{v} \in D$. As we described before, an abstraction function $h : D \to A$ induces a generated abstraction function $\mathcal{H} : D \to D$. Assume that $\boldsymbol{w} = rep([\boldsymbol{v}])$, then the path $\mathcal{H}(\boldsymbol{v}) = \boldsymbol{w}$ ends at a node w, which is at the same level as v. Intuitively, in the abstraction procedure, The BDT rooted at v is replaced by the BDT rooted at w. We call node w the *representative* of node v. More formally, the abstract BDT $\mathcal{H}(f)$ of T_f rooted at v is defined as

$$\mathcal{H}(f)(\boldsymbol{v}) = f(\mathcal{H}(\boldsymbol{v})).$$

In [11], the authors show a procedure which can build the abstract BDD of a function directly from the BDD of f instead of building the BDT of f. We also prove the following formula

$$f = p \circ q \rightarrow \mathcal{H}(f) = \mathcal{H}(p) \circ \mathcal{H}(q)$$

where $\circ$ is any logic operation. Notice that this means that the abstract BDD for f can be built incrementally, i.e., we do not have to build the BDD for f and then apply the abstraction function. Details can be found in [11].

The construction of an abstract BDD is illustrated by the following example. Assume that we have a boolean function $f(x_1, x_2, x_3) = (x_1 \wedge \neg x_3) \vee (x_2 \wedge x_3)$ (see Figure 3(a)). Consider the abstraction function $h(x_1, x_2) = x_1 + x_2$ (where "+" is ordinary addition). The abstraction function h induces an equivalence relation $\equiv_h$ on 0-1 vectors of length 2. Note that vectors of length 2 terminate at nodes of level 3. Therefore, we have $\boldsymbol{B} \equiv_h \boldsymbol{C}$ since $h(\boldsymbol{B}) = h(\boldsymbol{C}) = 1$. Assume that $\boldsymbol{B}$ is chosen as a representative, i.e. $\mathcal{H}(\boldsymbol{B}) = \mathcal{H}(\boldsymbol{C}) = \boldsymbol{B}$. In other words, B is a representative node. Then the directed graph after abstraction is shown in Figure 3(b) and the final reduced BDD in Figure 3(c). Intuitively, the construction maintains some "useful" *minterms* and ignores other "uninteresting" *minterms*.

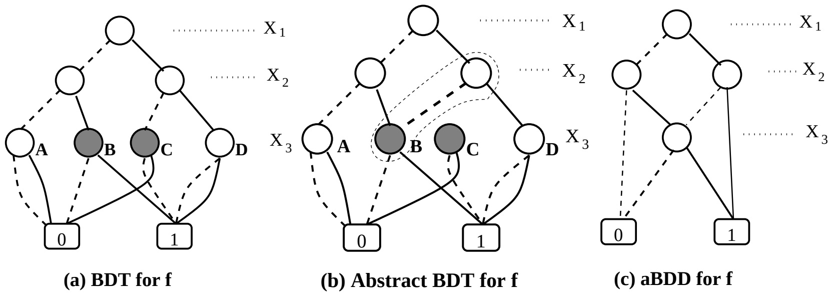

Fig. 3. Abstract BDD for f

Notice that the BDD of a representative node remains unaltered under the present definition. The non-representative nodes, however, are replaced by the corresponding representative nodes. In this paper, we extend aBDDs by allowing other operations on the nodes in an equivalence class. Details are described in next section.

3 New definition of abstract BDDs

Next, we define a new kind of abstract BDD which is more suitable for the purposes of this paper. The BDDs rooted at node v defines a boolean function $f(\boldsymbol{v})$. Recall that $\boldsymbol{v}$ is the path from the root to node v. The abstract BDD $\mathcal{H}(f)$

corresponding to the boolean function $f(\boldsymbol{v})$ is defined by the following equation:

$$\mathcal{H}(f)(\boldsymbol{v}) \;=\; \begin{cases} \bigvee f(\boldsymbol{v'}) & \boldsymbol{v} = \mathcal{H}(\boldsymbol{v'}) \\ 0 & \text{otherwise} \end{cases}$$

Notice that if v is a representative node, then $\mathcal{H}(\boldsymbol{v}) = \boldsymbol{v}$. Basically, the boolean function corresponding to a representative node is the *or* of all the boolean functions corresponding to the nodes in the same equivalence class. For non-representative nodes, the boolean function is defined to be **false** or 0. We use the example discussed previously to illustrate the new definition. Consider the boolean function $f(x_1, x_2, x_3) = (x_1 \wedge \neg x_3) \vee (x_2 \wedge x_3)$, $\boldsymbol{B} \equiv_h \boldsymbol{C}$ and $\boldsymbol{B}$ is the representative vector, so $\mathcal{H}(f)(\boldsymbol{B}) = f(\boldsymbol{B}) \vee f(\boldsymbol{C}) = 1$ and $\mathcal{H}(f)(\boldsymbol{C}) = 0$ (Figure 4).

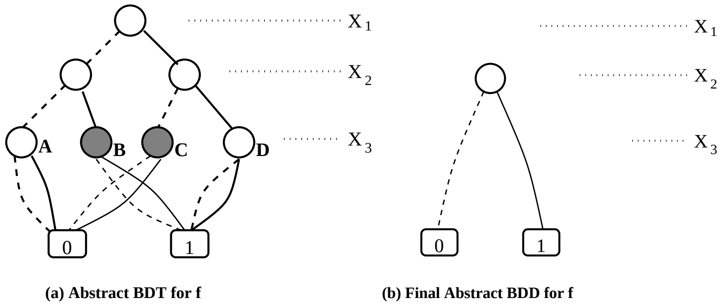

Fig. 4. New Abstract BDD for f

The next lemma describes how the generated abstraction function $\mathcal{H}$ interacts with conjunction and disjunction.

Lemma 1. *Let $f, p, q : \{0,1\}^n \to \{0,1\}$ be boolean functions, and let $\mathcal{H} : D \to D$ be the generated abstraction function corresponding to the abstraction function $h : D \to A$. The following equations hold:*

$$(f = p \vee q) \to (\mathcal{H}(f) = \mathcal{H}(p) \vee \mathcal{H}(q))$$

$$(f = p \wedge q) \to (\mathcal{H}(f) \to \mathcal{H}(p) \wedge \mathcal{H}(q))$$

The proof of this lemma is similar to the proof of Lemma 1 in [11]. A formal proof is provided in an appendix to this paper. The new definition of abstract BDDs can easily be extended to deal with multiple abstraction functions. For example, assume that we have m generated abstraction functions $\mathcal{H}_i : D \to D$ and a boolean function $f : D^m \to \{0,1\}$, the new abstract BDD of f can be defined as

$$\mathcal{H}(f)(x_1, \cdots, x_n) \;=\; \begin{cases} \bigvee f(y_1, \cdots, y_n) & x_1 = \mathcal{H}(y_1), \cdots, x_n = \mathcal{H}(y_n) \\ 0 & \text{otherwise} \end{cases}$$

Vectors x_i and y_i belong to the domain D, and $\mathcal{H}(y_i) = x_i$ implies that x_i is the representative in the equivalence class of y_i. It is easy to prove that Lemma 1 holds for multiple abstraction functions.

Lemma 2. *Given a boolean function $f(x_1, \cdots x_n)$ and an abstraction function $\mathcal{H} : D \to D$, the following formula holds*

$$\mathcal{H}(f) = \exists x_1, \cdots, x_n[\mathcal{H}(x_1) = y_1 \wedge \cdots \mathcal{H}(x_n) = y_n \wedge f(x)]$$

Lemma 2 states that the aBDD $\mathcal{H}(f)$ of the boolean function f corresponds to applying the existential abstraction operation to f (see the first paragraph in Section 2.1).

4 Methodology using abstract BDDs

In this section, we will discuss how to use abstract BDDs to construct an abstract Kripke structure. As we discussed in Section 2.1, $M_h = (S_h, S_{0,h}, R_h)$ is the abstract Kripke structure corresponding to $M = (S, S_0, R)$ using the abstraction function $h : D \to A$. Next we define an abstract structure $M_H = (S_H, S_{0,H}, R_H)$, which we can construct using aBDDs. The structure M_H is defined as follows:

- *State set* S_H is the image of S under the generated abstraction function $\mathcal{H}$.
- *Initial set of states* $S_{0,H}$ is the image of S_0 under the function $\mathcal{H}$. Notice that if S_0 is represented as a boolean function, then $S_{0,H}$ corresponds to the aBDD $\mathcal{H}(S_0)$ (see Lemma 2).
- *Transition Relation* R_H is the image of R under the function $\mathcal{H}$. Notice that if R is represented as a boolean function, then R_H corresponds to the aBDD $\mathcal{H}(R)$ (see Lemma 2).

Lemma 3. $M_h \cong M_H$

Lemma 3 states that M_h and M_H are *isomorphic structures*, i.e., there is a bijection $u : S_h \to S_H$ such that $u(S_0) = S_{0,H}$ and $u(R_h) = R_H$. Lemma 3 is proved in the appendix.

In general, it is intractable to build directly the BDD for transition relation R. Instead, the transition relation is usually partitioned [2]. Suppose that the transition relation R is partitioned into m clusters $R_1, \cdots, R_m$. Each cluster R_i is the transition relation of the composition of some set of components of the entire system. The transition relation R has one of the following forms depending on the nature of the composition (synchronous or asynchronous):

$$R = \begin{cases} R_1 \vee R_2 \vee \cdots \vee R_m & \text{asynchronous} \\ R_1 \wedge R_2 \wedge \cdots \wedge R_m & \text{synchronous} \end{cases}$$

The obvious way of applying abstraction is to distribute $\mathcal{H}$ over the cluster R_i. Using Lemma 1, we have

$$\begin{aligned} \mathcal{H}(R) &= \mathcal{H}(R_1) \vee \mathcal{H}(R_2) \vee \cdots \vee \mathcal{H}(R_m) \ \text{asynchronous} \\ \mathcal{H}(R) &\to \mathcal{H}(R_1) \wedge \mathcal{H}(R_2) \wedge \cdots \wedge \mathcal{H}(R_m) \ \text{synchronous} \end{aligned}$$

If we use partitioned transition relation, then the abstract structure $\tilde{M}$ which is constructed is not isomorphic to M_h. But, because of the equations given above, we have $M \sqsubseteq \tilde{M}$.

If we apply $\mathcal{H}$ to the innermost components of the system, we build an abstract structure $M_{app,H}$. It is easy to prove that $M_{app,H}$ is isomorphic to the structure M_{app}. Recall that M_{app} is the abstract structure built using the techniques given in [6]. The technique presented in this section builds a more refined model than $M_{app,H}$ which is isomorphic to M_{app}. We emphasize this point once again using a small example. Assume that we decide to abstract the domain of state variable x to a single value $\perp$. Intuitively, the actual value of x is irrelevant for the property we are interested in. Suppose there are two state variables y and z whose values in the next state depend on x in the following way: $y' = x$ and $z' = \neg x$. In the structure M_{app_H}, y' and z' will both become free variables. If we combine the two transitions together as $y' = x \wedge z' = \neg x$ and then abstract x, the result will be $y' \neq z'$. Clearly the structure produced by the second technique is more precise than M_{app_H}. Intuitively, in [6] the abstraction is applied to every transition of each component and therefore can produce very coarse models.

4.1 Building the abstract structure M_h

Recall that the transition relation of the abstract structure M_h is given by the following formula:

$$R_h = \exists x_1 \cdots x_n \exists x'_1 \cdots x'_n [h(x_1) = \hat{x}_1 \wedge \cdots \wedge h(x'_1) = \hat{x'}_1 \wedge \cdots \wedge R(x_1, \cdots, x'_1, \cdots)]$$

If we have BDDs encoding the transition relation R and the abstraction functions h_i, we could use standard traditional *relational product* technique to build the abstract transition relation R_h. We call this straightforward approach the traditional approach or method. Our new methodology has advantages over the traditional approach. First, in the traditional method the BDD for the abstraction functions has to be constructed before applying the method. For many abstraction functions, these BDDs are very hard to build. Second, in our experience a good variable ordering for an abstraction function might be different from a good variable ordering for the transition relation of the system. Our approach using abstract BDDs does not suffer from these problems since we never explicitly build the BDDs for the abstraction functions. Abstraction functions are employed while building the abstract BDD corresponding to the transition relation.

5 Experimental results

In order to test our ideas we modified the model-checker SMV. In our implementation, the user gives an abstraction function for each variable of interest. Once the user provides a system model and the abstraction functions, our method is completely automatic. We consider two examples in this paper: a pipelined multiplier design and the PCI local bus protocol.

5.1 Verification of a pipelined multiplier

In [6], Clarke, Grumberg, and Long propose an approach based on the *Chinese Remainder Theorem* for verifying *sequential* multipliers. The statement of the *Chinese Remainder Theorem* can be found in most texts on elementary number theory and will not be repeated here. Clarke, Grumberg, and Long use the modulus function $h(i) = i \bmod m$ for abstraction. They exploit the distributive property of the modulus function over addition, subtraction, and multiplication.

$$((i \bmod m) + (j \bmod m)) \bmod m \equiv (i + j) \bmod m$$

$$((i \bmod m) \times (j \bmod m)) \bmod m \equiv (i \times j) \bmod m$$

Let $\bullet$ represent the operation corresponding to the *implementation*. The goal is to prove that $\bullet$ is actually multiplication $\times$, or, in other words, for all x and y (within some finite range) $x \bullet y$ is equal to $x \times y$. If the actual implementation of the multiplier is composed of *shift-add* components, then the modulus function will distribute over the $\bullet$ operation. Therefore, we have the following equation:

$$(x \bullet y) \bmod m = [(x \bmod m) \bullet (y \bmod m)] \bmod m$$

Using this property and the Chinese Remainder Theorem, Clarke, Grumberg, and Long verify a sequential multiplier.

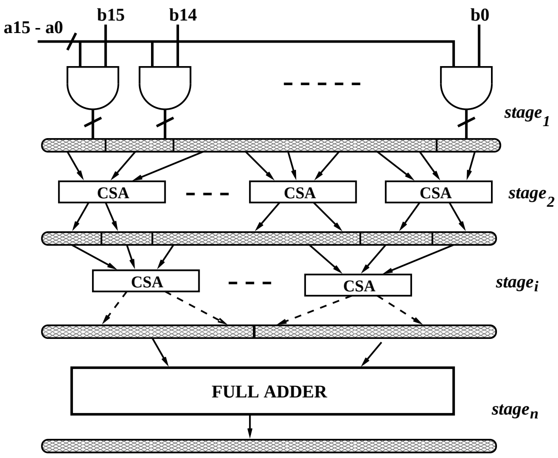

Fig. 5. Carry-save-adder pipeline multiplier

Unfortunately, this approach may not work if the multiplier is not composed of shift-add components. Suppose there is a mistake in the design of the multiplier, then there is no guarantee that the modulus operator will distribute over the operation $\bullet$ (corresponding to the actual implementation). For example, the mistake might scramble the inputs in some arbitrary way which breaks the distributive property of the $\bullet$ operation. In this case, the method proposed by Clarke, Grumberg and Long is not complete and may miss some errors. Therefore, before we apply the methodology in [6] it is necessary to check the distributive property of the modulus function with respect to the $\bullet$ operator. In other words, we must show that the following equation holds:

$$(x \bullet y) \bmod m = [(x \bmod m) \bullet (y \bmod m)] \bmod m$$

We illustrate our ideas by verifying a 16×16 pipelined multiplier which uses carry-save adders (see Figure 5). Notice that the first stage consists of *shift* operations and the last stage corresponds to the *add* operation. It easy to show that the first and the last stages satisfy the distributive property. In fact, this can be determined using classical equivalence checking methods. We will focus our attention on the intermediate stages.

Notice that the Chinese Remainder Theorem implies that it is enough to verify the multiplier by choosing $m = 5, 7, 9, 11, 13, 16, 17, 19, 23$ because of the following equation:

$$5 * 7 * 9 * 11 * 13 * 16 * 17 * 19 * 23 = 5354228880 > 2^{32} = 4294967296.$$

Our technique works as follows:

- First verify that each pipelined stage satisfies the distributive property using numbers in the set $\{5, 7, 9, 11, 13, 16, 17, 19, 23\}$). Formally, let $\bullet_i$ correspond to the operation of the i-th stage in the pipeline. We want to verify the following equation for all m in the set $\{5, 7, 9, 11, 13, 16, 17, 19, 23\}$ and $1 \leq i \leq 6$:

  $$(x \bullet_i y) \bmod m = (x \bmod m \bullet_i y \bmod m) \bmod m$$

  If the equation given above is violated, we have found a error. Notice that the equation given above can be checked by building the abstract BDD for the transition relation corresponding to the i-th stage.
- Next, assume that all the pipelined stages satisfy the distributive property. In this case, we can apply the method proposed by Clarke, Grumberg, and Long because the entire design will also satisfy the distributive property.

In Figure 6 we give our experimental results for the first step. The row for space usage corresponds to the largest amount of memory that is used during verification.

modulus	5	7	9	11	13	16	17	19	23
time(s)	99	137	199	372	636	130	1497	2648	6977
space(MB)	7.7	12.8	21.5	51.7	92.5	9.2	210	231	430

Fig. 6. Experimental Results for various modulus

5.2 The PCI local bus

This subsection briefly describes our results for the PCI local bus protocol. During verification, we found a potential error in the PCI bus protocol specification. In particular, we discovered an inconsistency between the textual specification and one of the state machines given in the PCI standard [16]. The precise nature of the error will be explained later.

During model-checking, we used following abstraction functions on various state variables:

- $h(x) = \perp$, where $\perp$ means constant;
- $h(x) =$ if $x \neq 0$ then 1 else 0;
- $h(x) =$ if $x > 1$ then 1 else 0;

Incidentally, the bug we discovered was *not* found when we applied the techniques proposed in [6].

We briefly describe the PCI local bus protocol. There are three types of devices that can be connected to the PCI local bus: masters, targets, and bridges. Masters can start transactions. Targets respond to transactions and bridges connect buses. Masters and targets are controlled by a finite-state machine. We considered a simple model which consists of one master, one target, and one bus arbiter. The model includes different timers to meet the timing specification. The master and target both include a *lock* machine to support exclusive read/write. The master also has a data counter to remember the number of data phases.

In the verification, we applied different abstractions to some of the timers, the lock machine and the data counter in the master. We also clustered the transition relations of the major state controllers in both master and the target. We checked various properties dealing with handshaking, read/write transactions, and timing in this simplified model. Next, we describe in detail the property which demonstrates the inconsistency in the design. Description of all the properties that we checked is not given here because of space restrictions.

One of the textual requirements is that *the target responds to every read/write transaction issued by the master*. This important property turns out to be false for the state machine given in the standard when the master reads or writes a single data value. The negation of this property can be expressed in ACTL as follows:

$$\mathbf{AG}(\text{m.req} \wedge \text{m.data\_cnt=1}) \rightarrow \mathbf{A}[(\text{m.req} \wedge \neg \text{t.ack})\mathbf{U}(\text{m.timeout})]) \quad (*)$$

where *m.req* corresponds to the master issuing a transaction; *m.data_cnt=1* means that the master requests one data value; *t.ack* means that the target acknowledges the master's request; and *m.timeout* means that the time the master has allowed for the transaction has expired. If this ACTL formula is true in the abstract model, it is also true in the concrete model. We verified that this formula is true, so there must be an inconsistency in the standard.

The experimental results are shown in Figure 7. the first row in Figure 7 (*Error*) corresponds to the inconsistency we discovered. The remaining properties are not described here. The second and third columns show the running time and maximum BDD nodes for the original version of SMV. The fourth and fifth columns show the results obtained using our methodology. For some cases our approach reduces the space needed for verification by a factor of 20.

Properties	SMV		SMV_ABS	
	Time(s)	# nodes	Time(s)	# nodes
Error	278	727K	65	33K
Property 1	20	164K	18	14K
Property 2	137	353K	30	66K
Property 3	99	436K	138	54K
Property 4	185	870K	40	36K
Property 5	67	352K	42	57K

Fig. 7. Experimental Results for Verifying PCI using Abstraction

6 Related work and directions for future research

In this paper, we propose a new technique for exploiting abstraction using abstract BDDs. The work of Clarke, Grumberg, and Long [6] is closest to that described here. The main advantage of our method is that we generate more accurate abstractions than existing methods. Moreover, we do not need to build BDDs for the abstraction functions.

A technique for verifying *combinational* multipliers is described in [12, 17]. Their methods use *residue BDDs* which are a special case of abstract BDDs (see [11]). However, the method proposed in [17] is not general and does not readily extend to the problem of model-checking arbitrary systems. Errors have been found in many other bus protocols by using model checking techniques [5]. By using our methodology, it should be possible to handle larger systems consisting of multiple devices connected by buses and bus bridges.

We are pursuing several directions for future research. First, we intend to try our techniques on larger designs. In fact, we are currently in the process of verifying the entire PCI local bus protocol. We also want to find ways of generating abstraction functions automatically. A methodology for refining the abstraction functions automatically would also be extremely useful.

References

1. R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Comput., Vol. C-35, No.8, pp.677-691*, Aug. 1986.
2. J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic Model Checking for Sequential Circuit Verification", *IEEE Trans. on CAD of Integrated Circuits and System, Vol.13, No.4, pp.401-424*, 1994.

3. E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent system using temporal logic", *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, January, 1983.
4. E.M. Clarke, R. Enders, T. Filkorn and S. Jha, "Exploiting Symmetry in Temporal Logic Model Checking", *Formal Methods in System Design 9(1/2):77-104*, 1996.
5. E. M. Clarke and O. Grumberg and H. Hiraishi and S. Jha and D. E. Long and K. L. McMillan and L. A. Ness, "Verification of the Futurebus+ Cache Coherence Protocol", *Formal Methods in System Design 6(2):217-232*, 1995.
6. E. M. Clarke, O. Grumberg, D. E. Long, "Model Checking and Abstraction", *ACM Transactions on Programming Languages and System (TOPLAS), Vol.16, No.5, pp.1512-1542*, Sept. 1994.
7. E.A. Emerson and A.P. Sistla, "Symmetry and Model Checking", *Formal Methods in System Design 9(1/2):105-130*, 1996.
8. P. Godefroid, D. Peled, and M. Staskauskas, "Using Partial Order Methods in the Formal Verification of Industrial Concurrent Programs", *ISSTA'96 International Symposium on Software Testing and Analysis, pp.261-269*, San Diego, California, USA, 1996. ACM Press.
9. J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, second edition, 1996. Morgan Kaufman Press.
10. C.N. Ip and D.L. Dill, "Better Verification Through Symmetry", *Formal Methods in System Design 9(1/2):41-76*, 1996.
11. S. Jha, Y. Lu, M. Minea, E. M. Clarke, "Equivalence Checking using Abstract BDDs", *Intl. Conf. on Computer Design (ICCD)*, 1997.
12. Shinji Kimura, "Residue BDD and Its Application to the Verification of Arithmetic Circuits", *32nd Design Automation Conference (DAC)*, 1995.
13. D. E. Long, "Model Checking, Abstraction and Compositional Verification", School of Computer Science, Carnegie Mellon University publication CMU-CS-93-178, July 1993.
14. K. L. McMillan, "Symbolic Model Checking: An Approach to the State Explosion Problem". Kluwer Academic Publishers, 1993.
15. D. Peled, "Combining Partial Order Reduction with on-the-fly model-checking", *Journal of Formal Methods in System Design, 8(1):39-64.*
16. PCI SGI, "PCI Local Bus Specification", Production Version Revision 2.1, June 1, 1995.
17. Kavita Ravi, Abelardo Pardo, Gary D. Hachtel, Fabio Somenzi, "Modular Verification of Multipliers", *Formal Methods in Computer-Aided Design*, pp.49-63, Nov. 1996.

A Proofs of the Lemmas

Proof of Lemma 1: Consider the binary decision trees of the functions f, p, and q. Note that these trees are exactly the same for f, p and q except the terminal nodes since all of the functions depend on the same set of variables. So if we can prove that for any node v in the trees, $f(\boldsymbol{v}) = p(\boldsymbol{v}) \vee q(\boldsymbol{v}) \rightarrow \mathcal{H}(f)(\boldsymbol{v}) = \mathcal{H}(p)(\boldsymbol{v}) \vee \mathcal{H}(q)(\boldsymbol{v})$ and $f(\boldsymbol{v}) = p(\boldsymbol{v}) \wedge q(\boldsymbol{v})) \rightarrow \mathcal{H}(f)((\boldsymbol{v}) = \mathcal{H}(p)(\boldsymbol{v}) \wedge \mathcal{H}(q)(\boldsymbol{v})$, then the formula holds for the special case of the root. This implies that the formula holds for the original function. If $\boldsymbol{v}$ is a representative, from the definition, $\mathcal{H}(f)(\boldsymbol{v}) = \bigvee_{\mathcal{H}(\boldsymbol{v'})=\boldsymbol{v}} f(\boldsymbol{v'})$; Otherwise, $\mathcal{H}(f)(\boldsymbol{v}) = 0$. The same formula holds when f is replaced by p and by q. If $f = p \vee q$, when $\boldsymbol{v}$ is non-representative, $\mathcal{H}(p)(\boldsymbol{v}) \vee \mathcal{H}(q)(\boldsymbol{v}) = 0 \vee 0 = \mathcal{H}(f)(\boldsymbol{v})$; otherwise, when $\boldsymbol{v}$ is a representative,

$$\mathcal{H}(f)(\boldsymbol{v}) = \bigvee_{\mathcal{H}(\boldsymbol{v'})=v} (p(\boldsymbol{v'}) \vee q(\boldsymbol{v'})) = (\bigvee_{\mathcal{H}(\boldsymbol{v'})=v} p(\boldsymbol{v'}) \vee (\bigvee_{\mathcal{H}(\boldsymbol{v'})=v} q(\boldsymbol{v'}))$$

In general, we have $\mathcal{H}(f) = \mathcal{H}(p) \vee \mathcal{H}(q)$. On the other hand, If $f = p \wedge q$, when $\boldsymbol{v}$ is a non-representative, it is easy to see that $\mathcal{H}(p)(\boldsymbol{v}) \wedge \mathcal{H}(q)(\boldsymbol{v}) = 0 \wedge 0 = \mathcal{H}(f)(\boldsymbol{v})$; when $\boldsymbol{v}$ is a representative, we have

$$\mathcal{H}(f) = \bigvee_{\mathcal{H}(\boldsymbol{v'})=v} f(\boldsymbol{v'}) = \bigvee_{\mathcal{H}(\boldsymbol{v'})=v} (p(\boldsymbol{v'}) \wedge q(\boldsymbol{v'}))$$

Likewise,

$$\mathcal{H}(p)(\boldsymbol{v}) \wedge \mathcal{H}(q)(\boldsymbol{v}) = \bigvee_{\mathcal{H}(\boldsymbol{v'})=v} p(\boldsymbol{v'}) \wedge \bigvee_{\mathcal{H}(\boldsymbol{v'})=v} q(\boldsymbol{v'})$$

It is easy to see that $\bigvee_{\mathcal{H}(\boldsymbol{v'})=v}(p(\boldsymbol{v'}) \wedge q(\boldsymbol{v'}))$ implies $\bigvee_{\mathcal{H}(\boldsymbol{v'})=v} p(\boldsymbol{v'}) \wedge \bigvee_{\mathcal{H}(\boldsymbol{v'})=v} q(\boldsymbol{v'})$. Consequently, $\mathcal{H}(f)(\boldsymbol{v}) \rightarrow \mathcal{H}(p)(\boldsymbol{v}) \wedge \mathcal{H}(q)(\boldsymbol{v})$. In general, $\mathcal{H}(f) \rightarrow \mathcal{H}(p) \wedge \mathcal{H}(q)$.

Proof of Lemma 3. Assume that $I : A \rightarrow img(\mathcal{H})$ is a function which is defined as $I(h(d)) = rep([d])$. First, we will show that I is well-defined and that I is a bijection. Second, using I we will build a bijection between the states of M_h and M_H.

From the definition, $h(d_1) = h(d_2)$ implies that $rep([d_1]) = rep([d_2])$ which in turn implies that $I(h(d_1)) = I(h(d_2))$. Therefore, I is a well defined function. If $d_1 \in img(\mathcal{H})$, then there exists a $d_2 \in D$, where $d_1 = rep([d_2])$. Moreover, $I(h(d_2)) = rep(d_2) = d_1$, so I is a surjection. On the other hand, if $I(h(d_1)) = I(h(d_2))$, then $rep([d_1]) = rep(d_2])$ which implies that $h(d_1) = h(d_2)$. Hence I is an injection. Since I is injective and surjective, I is a bijection.

As defined before, $S \subseteq D^n$ is the set of states of M; $S_h \subseteq A^n$ is the set of states of M_h; and $S_H \subseteq img(\mathcal{H})^n$ is the set of states of M_H. We define a mapping $u : S_h \rightarrow S_H$ as follows:

$$u(<\hat{x_1}, \cdots, \hat{x_n}>) \ = <I(\hat{x_1}), \cdots, I(\hat{x_n})>$$

where $<\hat{x_1}, \cdots, \hat{x_n}> \in S_h$ and $<I(\hat{x_1}), \cdots, I(\hat{x_n})> \in S_H$. Next we will show that $u(S_{0,h}) = S_{0,H}$ and $u(R_h) = R_H$, i.e., the bijection u preserves the initial states and the transitions. Consider an arbitrary state $<\hat{x_1}, \cdots, \hat{x_n}> \in S_{0,h}$ and an arbitrary transition $(<\hat{x_1}, \cdots, \hat{x_n}>, <\hat{x'_1}, \cdots, \hat{x'_n}>) \in R_h$ Since $<\hat{x_1}, \cdots, \hat{x_n}> \in S_{0,h}$, there exists a state $<x_1, \cdots, x_n> \in S$ such that $h(x_i) = \hat{x_i}$ and $<x_1, \cdots, x_n> \in S_0$. Since $\mathcal{H}(x_i) = rep([x_i]) = I(h(x_i)) = I(\hat{x_i})$, and $S_{0,H}$ is the existential abstraction of S_0, it follows that $<I(\hat{x_1}), \cdots, I(\hat{x_n})> S_{0,H}$.

The proof for the transition relation, is very similar. Therefore, $u(S_{0,h}) \subseteq S_{0,H}$ and $u(R_h) \subseteq R_H$. Since I is a bijection, the argument given above holds in the reverse direction. Thus, $u(S_{0,h}) = S_{0,H}$ and $u(R_h) = R_H$ This proves that $M_h \cong M_H$.