Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions [*]

Randal E. Bryant[1], Steven German[2], and Miroslav N. Velev[3]

[1] Computer Science, Carnegie Mellon University, Pittsburgh, PA
Randy.Bryant@cs.cmu.edu
[2] IBM Watson Research Center, Yorktown Hts., NY
german@watson.ibm.com
[3] Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA
mvelev@ece.cmu.edu

Abstract. In using the logic of equality with unininterpreted functions to verify hardware systems, specific characteristics of the formula describing the correctness condition can be exploited when deciding its validity. We distinguish a class of terms we call "p-terms" for which equality comparisons can appear only in monotonically positive formulas. By applying suitable abstractions to the hardware model, we can express the functionality of data values and instruction addresses flowing through an instruction pipeline with p-terms.

A decision procedure can exploit the restricted uses of p-terms by considering only "maximally diverse" interpretations of the associated function symbols, where every function application yields a different value except when constrained by functional consistency. We present a procedure that translates the original formula into one in propositional logic by interpreting the formula over a domain of fixed-length bit vectors and using vectors of propositional variables to encode domain variables. By exploiting maximal diversity, this procedure can greatly reduce the number of propositional variables that must be introduced.

We present experimental results demonstrating the efficiency of this approach when verifying pipelined processors using the method proposed by Burch and Dill. Exploiting positive equality allows us to overcome the exponential blow-up experienced previously [VB98] when verifying microprocessors with load, store, and branch instructions.

1 Introduction

For automatically reasoning about pipelined processors, Burch and Dill demonstrated the value of using propositional logic, extended with uninterpreted functions, uninterpreted predicates, and the testing of equality [BD94]. Their approach involves abstracting the data path as a collection of registers and memories storing data, units such as ALUs operating on the data, and various connections and multiplexors providing methods for data to be transferred and selected. The operation of units that transform data is

[*] This research was supported at Carnegie Mellon University by SRC Contract 98-DC-068 and by grants from Fujitsu, Motorola, and Intel. This paper will be presented at *Computer Aided Verification 1999 (CAV '99).*

abstracted as blocks computing functions with no specified properties other than functional consistency, i.e., that applications of a function to equal arguments yield equal results: $x = y$ implies $f(x) = f(y)$. The state of a register at any point in the computation can be represented by a symbolic term, an expression consisting of a combination of domain variables, function and predicate applications, and Boolean operations.

The correctness of a pipelined processor can be expressed as a formula in this logic that compares for equality the terms describing the results produced by the processor to those produced by an instruction set reference model. In their paper, Burch and Dill also describe a decision procedure for their logic based on theorem proving search methods. It uses combinatorial search coupled with algorithms for maintaining a partitioning of the terms into equivalence classes based on the equalities that hold at a given step of the search [NO80].

Burch and Dill's work has generated considerable interest in the use of uninterpreted functions to abstract data operations in processor verification. A common theme has been to adopt Boolean methods, either to allow integration of uninterpreted functions into symbolic model checkers [DPR98,BBCZ98], or to allow the use of Binary Decision Diagrams in the decision procedure [HKGB97,GSZAS98,VB98]. Boolean methods allow a more direct modeling of the control logic of hardware designs and thus can be applied to actual processor designs rather than highly abstracted models. In addition to BDD-based decision procedures, Boolean methods could use some of the recently developed satisfiability procedures for propositional logic. In principle, Boolean methods could outperform decision procedures based on theorem proving search methods, especially when verifying processors with more complex control logic.

Boolean methods can be used to decide the validity of a formula containing terms and uninterpreted functions by exploiting the property that a given formula contains a limited number of function applications and therefore can be proved to be universally valid by considering its interpretation over a sufficiently large, but finite domain [Ack54]. The formula to be verified can be translated into one in propositional logic, using vectors of propositional variables to encode the possible values generated by function applications [HKGB97]. Our implementation of such an approach [VB98] as part of a BDD-based symbolic simulation system was successful at verifying simple pipelined data paths. We found, however, that the computational resources grew exponentially as we increased the pipeline depth. Modeling the interactions between successive instructions flowing through the pipeline, as well as the functional consistency of the ALU results, precludes having an ordering of the variables encoding term values that yields compact BDDs. Similarly, we found that extending the data path to a complete processor by adding either load and store instructions or instruction fetch logic supporting jumps and conditional branches led to impossible BDD variable ordering requirements. Goel *et al* [GSZAS98] presented an alternate approach to using BDDs to decide the validity of formulas in the logic of equality with uninterpreted functions. They use Boolean variables to encode the equality relations between terms, rather than to encode term values. Their experimental results were also somewhat disappointing. To date, the possibility that Boolean methods could outperform theorem proving methods has not been realized.

In this paper, we show that the characteristics of the formulas generated when modeling processor pipelines can be exploited to greatly reduce the number of propositional variables that are introduced when translating the formula into propositional logic. We distinguish a class of terms we call *p-terms* for which equations, i.e., equality comparisons between terms, can appear only in monotonically positive formulas. Such formulas are suitable for describing the top-level correctness condition, but not for modeling any control decisions in the hardware. By applying suitable abstractions to the hardware model, we can express the functionality of data values and instruction addresses with p-terms.

A decision procedure can exploit the restricted uses of p-terms by considering only "maximally diverse" interpretations of the associated "p-function" symbols, where every function application yields a different value except when constrained by functional consistency. In translating the formula into propositional logic, we can then use vectors with fixed bit patterns rather than propositional variables to encode the possible results of function applications. This reduction in variables greatly simplifies the BDDs generated, avoiding the exponential blow-up experienced by other procedures.

Others have recognized the value of restricting the testing of equality when modeling the flow of data in pipelines. Berezin *et al* [BBCZ98] generate a model of an execution unit suitable for symbolic model checking in which the data values and operations are kept abstract. In our terminology, their functional terms are all p-terms. They use fixed bit patterns to represent the initial states of registers, much as we replace p-term domain variables by fixed bit patterns. To model the outcome of each program operation, they generate an entry in a "reference file" and refer to the result by a pointer to this file. These pointers are similar to the bit patterns we generate to denote the p-function application outcomes. Damm *et al* consider an even more restricted logic that allows them to determine the universal validity of a formula by considering only interpretations over the domain $\{0, 1\}$. Verifying an execution unit in which the data path width is reduced to a single bit then suffices to prove its correctness for all possible widths. In comparison to these other efforts, we maintain the full generality of the unrestricted functional terms of Burch and Dill while exploiting the efficiency gains possible with p-terms. In our processor model, we can abstract register identifiers as unrestricted terms, while modeling program data and instruction data as p-terms. In contrast, both [BBCZ98] and [DPR98] used bit encodings of register identifiers and were unable to scale their verifications to a realistic number of registers.

In a different paper in this proceedings, Pnueli, *et al* [PRSS99] also propose a method to exploit the polarity of the equations in a formula containing uninterpreted functions with equality. They describe an algorithm to generate small domains for each domain variable such that the universal validity of the formula can be determined by considering only interpretations in which the variables range over their restricted domains. A key difference of their work is that they examine the equation structure after replacing all function application terms with domain variables and introducing functional consistency constraints as described by Ackermann [Ack54]. These consistency constraints typically contain large numbers of equations—far more than occur in the original formula—that mask the original p-term structure. In addition, we use a new method of replacing function application terms with domain variables. Our scheme allows us to exploit maximal diversity by assigning fixed values to the domain variables generated while expanding p-function application terms.

In the remainder of the paper, we first define the syntax and semantics of our logic by extending that of Burch and Dill's. We prove our central result concerning the need to consider only maximally diverse interpretations when deciding the validity of formulas in our logic. We describe a method of translating formulas into propositional logic. We discuss the abstractions required to model processor pipelines in our logic. Finally, we present experimental results showing our ability to verify a simple, but complete pipelined processor. A more detailed presentation with complete proofs is given in [BGV99].

2 Logic of Equality with Uninterpreted Functions (EUF)

The logic of Equality with Uninterpreted Functions (EUF) presented by Burch and Dill [BD94] can be expressed by the following syntax:

$$\begin{aligned}
\textit{term} ::= {} & \textit{ITE}(\textit{formula}, \textit{term}, \textit{term}) \\
& \mid \textit{function-symbol}(\textit{term}, \ldots, \textit{term}) \\
\textit{formula} ::= {} & \mathbf{true} \mid \mathbf{false} \mid (\textit{term} = \textit{term}) \\
& \mid (\textit{formula} \wedge \textit{formula}) \mid (\textit{formula} \vee \textit{formula}) \mid \neg \textit{formula} \\
& \mid \textit{predicate-symbol}(\textit{term}, \ldots, \textit{term})
\end{aligned}$$

In this logic, *formulas* have truth values while *terms* have values from some arbitrary domain. Terms are formed by applications of uninterpreted function symbols and by applications of the *ITE* (for "if-then-else") operator. The *ITE* operator chooses between two terms based on a Boolean control value, i.e., $\textit{ITE}(\mathbf{true}, x_1, x_2)$ yields x_1 while $\textit{ITE}(\mathbf{false}, x_1, x_2)$ yields x_2. Formulas are formed by comparing two terms for equality, by applying an uninterpreted predicate symbol to a list of terms, and by combining formulas using Boolean connectives. A formula expressing equality between two terms is called an *equation*. We use *expression* to refer to either a term or a formula.

Every function symbol f has an associated *order*, denoted $ord(f)$, indicating the number of terms it takes as arguments. Function symbols of order zero are referred to as *domain variables*. We use the shortened form v rather than $v()$ to denote an instance of a domain variable. Similarly, every predicate p has an associated order $ord(p)$. Predicates of order zero are referred to as *propositional variables*.

The truth of a formula is defined relative to a nonempty domain $\mathcal{D}$ of values and an interpretation I of the function and predicate symbols. Interpretation I assigns to each function symbol of order k a function from $\mathcal{D}^k$ to $\mathcal{D}$, and to each predicate symbol of order k a function from $\mathcal{D}^k$ to $\{\mathbf{true}, \mathbf{false}\}$. Given an interpretation I of the function and predicate symbols and an expression E, we can define the *valuation* of E under I, denoted $I[E]$, according to its syntactic structure. $I[E]$ will be an element of the domain when E is a term, and a truth value when E is a formula.

A formula F is said to be *true under interpretation* I when $I[F]$ equals $\mathbf{true}$. It is said to be *valid over domain* $\mathcal{D}$ when it is true for all interpretations over domain $\mathcal{D}$. F is said to be *universally valid* when it is valid over all domains. It can be shown that if a formula is valid over some suitably large domain, then it is universally valid [Ack54]. In particular, it suffices to have a domain as large as the number of syntactically distinct function application terms occurring in F.

3 Logic of Positive Equality with Uninterpreted Functions (PEUF)

3.1 Syntax

PEUF is an extended logic based on EUF given by the following syntax:

$$
\begin{aligned}
\textit{g-term} &::= \textit{ITE}(\textit{formula}, \textit{g-term}, \textit{g-term}) \\
&\quad \mid \textit{g-function-symbol}(\textit{p-term}, \ldots, \textit{p-term}) \\
\textit{p-term} &::= \textit{g-term} \mid \textit{ITE}(\textit{formula}, \textit{p-term}, \textit{p-term}) \\
&\quad \mid \textit{p-function-symbol}(\textit{p-term}, \ldots, \textit{p-term}) \\
\textit{formula} &::= \mathbf{true} \mid \mathbf{false} \mid (\textit{term} = \textit{term}) \\
&\quad \mid (\textit{formula} \wedge \textit{formula}) \mid (\textit{formula} \vee \textit{formula}) \mid \neg \textit{formula} \\
&\quad \mid \textit{predicate-symbol}(\textit{p-term}, \ldots, \textit{p-term}) \\
\textit{p-formula} &::= \textit{formula} \mid (\textit{p-term} = \textit{p-term}) \\
&\quad \mid (\textit{p-formula} \wedge \textit{p-formula}) \mid (\textit{p-formula} \vee \textit{p-formula})
\end{aligned}
$$

This logic has two disjoint classes of function symbols giving two classes of terms. General terms, or *g-terms*, correspond to terms in EUF. Syntactically, a g-term is a g-function application or an *ITE* term in which the two result terms are hereditarily built from g-function applications and *ITE*s.

The new class of terms is called positive terms, or *p-terms*. P-terms may not appear in negative equations, i.e., equations within the scope of a logical negation. The syntax is restricted in a way that prevents p-terms from appearing in negative equations. When two p-terms are compared for equality, the result is a special, restricted kind of formula called a *p-formula*. P-formulas are built up using only the monotonically positive Boolean operations $\wedge$ and $\vee$. P-formulas may not be placed under a negation sign, and cannot be used as the control for an ITE operation.

Note that our syntax allows any g-term to be "promoted" to a p-term. Throughout the syntax definition, we require function and predicate symbols to take p-terms as arguments. However, since g-terms can be promoted, the requirement to use p-terms as arguments does not restrict the use of g-function symbols or g-terms. In essence, g-function symbols may be used as freely in our logic as in EUF, but the p-function symbols are restricted.

Observe that PEUF does not extend the expressive power of EUF—we could translate any PEUF expression into EUF by considering the g-terms and p-terms to be terms and the p-formulas to be formulas. Instead, the benefit of PEUF is that by distinguishing some portion of a formula as satisfying a restricted set of properties, we can radically reduce the number of different interpretations we must consider when proving that a p-formula is universally valid.

3.2 Diverse Interpretations

Let $\mathcal{T}$ be a set of terms, where a term may be either a g-term or a p-term. We classify terms as either p-function applications, g-function applications, or *ITE* terms, according to their top-level operation. The first two categories are collectively referred to as function application terms. For any formula or p-formula F, define $\mathcal{T}(F)$ as the set of all function application terms occurring in F.

An interpretation I partitions a term set $\mathcal{T}$ into a set of equivalence classes, where terms T_1 and T_2 are equivalent under I, written $T_1 \approx_I T_2$ when $I[T_1]$ equals $I[T_2]$. Interpretation I' is said to be a *refinement* of I for term set $\mathcal{T}$ when $T_1 \approx_{I'} T_2$ implies $T_1 \approx_I T_2$ for every pair of terms T_1 and T_2 in $\mathcal{T}$. I' is a *proper* refinement of I for $\mathcal{T}$ when it is a refinement and there is at least one pair of terms $T_1, T_2 \in \mathcal{T}$ such that $T_1 \approx_I T_2$, but $T_1 \not\approx_{I'} T_2$.

Let Σ denote a subset of the function symbols in formula F. An interpretation I is said to be *diverse* for F with respect to Σ when it provides a maximal partitioning of the function application terms in $\mathcal{T}(F)$ having a top-level function symbol from Σ relative to each other and to the other function application terms, but subject to the constraints of functional consistency. That is, for T_1 of the form $f(S_1, \ldots, S_k)$, where $f \in \Sigma$, an interpretation I is diverse with respect to Σ if I has $T_1 \approx_I T_2$ only in the case where T_2 is also a term of the form $f(U_1, \ldots, U_k)$, and $S_i \approx_I U_i$ for all i such that $1 \leq i \leq k$. If we let $\Sigma_p(F)$ denote the set of all p-function symbols in F, then interpretation I is said to be *maximally diverse* when it is diverse with respect to $\Sigma_p(F)$. Note that this property requires the p-function application terms to be in separate equivalence classes from the g-function application terms.

Theorem 1. *P-formula F is universally valid if and only if it is true in all maximally diverse interpretations.*

First, it is clear that if F is universally valid, then F is true in all maximally diverse interpretations. We prove via the following lemma that if F is true in all maximally diverse interpretations it is universally valid.

Lemma 1. *If interpretation I is not maximally diverse for p-formula F, then there is an interpretation I' that is a proper refinement of I such that $I'[F] \Rightarrow I[F]$.*

Proof Sketch: Let T_1 be a term occurring in F of the form $f_1(S_1, \ldots, S_{k_1})$, where f_1 is a p-function symbol. Let T_2 be a term occurring in F of the form $f_2(U_1, \ldots, U_{k_2})$, where f_2 may be either a p-function or a g-function symbol. Assume furthermore that $I[T_1] = I[T_2] = z$, but that either symbols f_1 and f_2 differ or $I[S_i] \neq I[U_i]$ for some value of i.

Let z' be a value not in $\mathcal{D}$, and define a new domain $\mathcal{D}' \doteq \mathcal{D} \cup \{z'\}$. Our strategy is to construct an interpretation I' over $\mathcal{D}'$ that partitions the terms in $\mathcal{T}(F)$ in the same way as I, except that it splits the class containing terms T_1 and T_2 into two parts—one containing T_1 and evaluating to z', and the other containing T_2 and evaluating to z.

Define function $h\colon \mathcal{D}' \rightarrow \mathcal{D}$ to map elements of $\mathcal{D}'$ back to their counterparts in $\mathcal{D}$, i.e., $h(z') = z$, while all other values of x give $h(x) = x$.

For p-function symbol f_1, define $I'(f_1)(x_1, \ldots, x_k)$ as z' when $h(x_i) = I[S_i]$ for all $1 \leq i \leq k_1$, and as $I(f_1)(h(x_1), \ldots, h(x_k))$ otherwise. For other function and predicate symbols, I' is defined to preserve the functionality of interpretation I, while also treating argument values of z' the same as z. That is, $I'(f)$ for function symbol f having $ord(f) = k$ is defined such that $I'(f)(x_1, \ldots, x_k) = I(f)(h(x_1), \ldots, h(x_k))$.

One can show that interpretation I' maintains the values of all formulas and g-terms as occur under interpretation I. Some of the p-terms that evaluate to z under I, including T_1, evaluate to z'. Others, including T_2, continue to evaluate to z. With respect to p-formulas, consider first an equation of the form $S_a = S_b$ where S_a and S_b are p-terms. The equation will yield the same value under both interpretations except under the condition that S_a and S_b are split into different parts of the class that originally evaluated to z, in which case the comparison will yield $\mathbf{true}$ under I, but $\mathbf{false}$ under I'. In any case, we maintain the property that $I'[S_a = S_b] \Rightarrow I[S_a = S_b]$. This implication relation is preserved by conjunctions and disjunctions of p-formulas, due to the monotonicity of these operations. By this argument we can see that I' is a proper refinement of I for $\mathcal{T}(F)$ and that $I'[F] \Rightarrow I[F]$. □

Theorem 1 is proved by repeatedly applying Lemma 1. One can show that any interpretation I of a p-formula F can be refined to a maximally diverse interpretation I^* for F such that $I^*[F]$ implies $I[F]$. It follows that the truth of F for all maximally diverse interpretations implies its truth for all possible interpretations.

4 Exploiting Positive Equality in a Decision Procedure

A decision procedure for PEUF must determine whether a given p-formula is universally valid. Theorem 1 shows that we can consider only interpretations in which the values produced by the application of any p-function symbol differ from those produced by the applications of any other p-function or g-function symbol. We can therefore consider the different p-function symbols to yield values over domains disjoint from one another and from the domain of g-function values. In addition, we can consider each application of a p-function symbol to yield a distinct value, except when its arguments match those of some other application.

We describe a decision procedure that first transforms an arbitrary EUF formula into one containing only domain and propositional variables. This restricted class of formulas can readily be translated into propositional formulas by using bit vectors as the domain of interpretation. The transformation can exploit positive equality by using fixed bit patterns rather than vectors of propositional variables to encode the domain variables representing p-function application results.

4.1 Eliminating Function and Predicate Applications in EUF

We illustrate our method by considering the formula

$$x = y \quad \Rightarrow \quad g(f(x)) = g(f(y)) \tag{1}$$

Eliminating the implication gives $\neg(x = y) \vee g(f(x)) = g(f(y))$, and hence both f and g are a p-function symbols, while x and y are g-function symbols. We introduce domain variables vf_1, vf_2 and replace term $f(x)$ with vf_1 and term $f(y)$ with the term $\mathit{ITE}(y = x, vf_1, vf_2)$. Observe that as we consider interpretations with different values for variables vf_1 and vf_2, we implicitly cover all values that an interpretation of function symbol f may yield for arguments x and y. The *ITE* structure enforces functional consistency—when $I(x) = I(y)$, we will have both terms evaluate to $I(vf_1)$.

These replacements give a formula: $\neg(x = y) \vee g(vf_1) = g(\mathit{ITE}(y = x, vf_1, vf_2))$. We then introduce domain variables vg_1 and vg_2 and replace the first application of g with vg_1, and the second with $\mathit{ITE}(\mathit{ITE}(y = x, vf_1, vf_2) = vf_1, vg_1, vg_2)$. Our final form is then:

$$\neg(x = y) \quad \vee \quad vg_1 = \mathit{ITE}(\mathit{ITE}(y = x, vf_1, vf_2) = vf_1, vg_1, vg_2) \tag{2}$$

The complete procedure generalizes that shown for the simple example. Suppose formula F contains n syntactically distinct terms $T_1, T_2, \ldots, T_n$ having the application of function symbol f as the top-level operation. We refer to these as f-application terms. We introduce domain variables $vf_1, \ldots, vf_n$ and replace each term T_i with a nested *ITE* structure U_i of the form

$$U_i \doteq \mathit{ITE}(C_{i,1}, vf_1, \mathit{ITE}(C_{i,2}, vf_2, \cdots \mathit{ITE}(C_{i,i-1}, vf_{i-1}, vf_i) \cdots))$$

where the formula $C_{i,j}$ is true iff the arguments to the top-level application of f in the terms T_i and T_j have the same values. The result of replacing every f-application term T_i in F by the new term U_i is a formula that we call $F^{(f)}$.

We remove all function symbols of nonzero order from F by repeating this process. A similar process is used to eliminate applications of predicate symbols having nonzero order, except that we introduce propositional variables $pv_1, pv_2, \ldots,$ when replacing applications of predicate symbol p. We call the final result of this process the formula F^*. Complete details are presented in [BGV99].

Theorem 2. *For EUF formula F, the transformation process yields a formula F^* containing only domain and propositional variables and such that F is universally valid if and only if F^* is universally valid.*

Proof Sketch: To prove this theorem, we first show that our procedure for replacing all instances of function symbol f in an arbitrary formula G by nested *ITE* terms to yield a formula $G^{(f)}$ preserves universal validity. (1) $G^{(f)}$ universally valid $\Rightarrow$ G universally valid. For any interpretation I of the function and predicate symbols in G, we can construct an interpretation $\hat{I}$ of the symbols in $G^{(f)}$ such that $\hat{I}[G^{(f)}] = I[G]$. Interpretation $\hat{I}$ is defined by extending I to include interpretations of the domain variables

$vf_1, \ldots, vf_n$. Each such variable vf_i is given the interpretation $\hat{I}(vf_i) \doteq I[T_i]$, i.e., the value of f-application term i under I.

(2) Conversely, G universally valid $\Rightarrow$ $G^{(f)}$ universally valid. For any interpretation $\hat{I}$ of the function and predicate symbols in $G^{(f)}$, we can define an interpretation I of the symbols in G such that $I[G] = \hat{I}[G^{(f)}]$. This interpretation is defined by introducing an interpretation of function symbol f such that the values yielded when evaluating each f-application term T_i under I matches that yielded for the nested *ITE* structure U_i under $\hat{I}$.

By a similar process we show that our procedure for replacing predicate applications preserves universal validity. The theorem is then proved by inducting on the number of function and predicate symbols. □

4.2 Using Fixed Values for P-Function Applications

We can exploit the maximal diversity property by using fixed domain values rather than domain variables when replacing p-function applications by nested *ITE* terms. First, consider the effect of replacing all instances of a function symbol f by nested *ITE* terms, as described earlier, yielding a formula $F^{(f)}$ with new domain variables $vf_1, \ldots, vf_n$.

Lemma 2. *If $f \in \Sigma$, then for any interpretation I that is diverse for F with respect to Σ, there is an interpretation $\hat{I}$ that is diverse for $F^{(f)}$ with respect to $\Sigma - \{f\} \cup \{vf_1, \ldots, vf_n\}$ such that $I[F] = \hat{I}[F^{(f)}]$.*

Proof Sketch: The proof of this lemma requires a more refined argument than that of Theorem 2. If we were to define $\hat{I}(vf_i)$ to be $I[T_i]$ for each domain variable vf_i, we may not have a diverse interpretation with respect to the newly-generated variables. Instead, we define $\hat{I}(vf_i)$ to be $I[T_i]$ only if there is no value $j < i$ such that the arguments of f-application terms T_j and T_i have equal valuations under I. Otherwise we let z' be a value not in $\mathcal{D}$, define a new domain $\mathcal{D}' \doteq \mathcal{D} \cup \{z'\}$, and let $\hat{I}(vf_i) = z'$. It can readily be seen that the value assigned to this variable will not affect the valuation of nested *ITE* structure U_i under interpretation $\hat{I}$, and hence it can be arbitrary. □

Suppose we apply the transformation process of Theorem 2 to a p-formula F to generate a formula F^*, and that in this process, we introduce a set of new domain variables V to replace the applications of the p-function symbols. Let $\Sigma_p^*(F)$ be the union of the set of domain variables in $\Sigma_p(F)$ and V. That is, $\Sigma_p^*(F)$ consists of those domain variables in the original formula F that were p-function symbols as well as the domain variables generated when replacing applications of p-function symbols. Let $\Sigma_g^*(F)$ be the domain variables in F^* that are not in $\Sigma_p^*(F)$. These variables were either g-function symbols in F or were generated when replacing g-function applications.

We first observe that we can generate all maximally diverse interpretations of F by considering only interpretations of the variables in F^* that assign distinct values to the variables in $\Sigma_p^*(F)$:

Theorem 3. *PEUF formula F is universally valid if and only if its translation F^* is true for every interpretation I^* such that if v_p is a variable in $\Sigma_p^*(F)$ and v is any other domain variable in F^*, then $I^*(v_p) \neq I^*(v)$.*

Proof Sketch: This theorem follows by inducting on the number of p-function symbols in F, using Lemma 2 to prove the induction step. $\Box$

Observe that the nested *ITE* structures we generate when replacing function applications involve many equations in the formulas controlling *ITE* operations. These can cause function symbols that appeared as p-function symbols in the original formula to be g-function symbols in F^*. In addition, many of the newly-generated variables will not be p-function symbols in F^*. For example, variables vf_1 and vf_2 are g-function symbols in Equation 2. Nonetheless, this theorem shows that we can still restrict our attention to interpretations that are diverse with respect to these variables.

Furthermore, we can choose particular domains of sufficient size and assign fixed interpretations to the variables in $\Sigma_p^*(F)$. Select disjoint domains $\mathcal{D}_p$ and $\mathcal{D}_g$ for the variables in $\Sigma_p^*(F)$ and $\Sigma_g^*(F)$, respectively, such that $|\mathcal{D}_p| \geq |\Sigma_p^*(F)|$ and $|\mathcal{D}_g| \geq |\Sigma_g^*(F)|$. Let α be any 1–1 mapping $\alpha\colon \Sigma_p^*(F) \rightarrow \mathcal{D}_p$.

Corollary 1. *PEUF formula F is universally valid if and only if its translation F^* is true for every interpretation I^* such that $I^*(v_p) = \alpha(v_p)$ for every variable v_p in $\Sigma_p^*(F)$, and $I^*(v_g)$ is in $\mathcal{D}_g$ for every variable v_g in $\Sigma_g^*(F)$.*

Proof Sketch: Any interpretation that is diverse with respect to $\Sigma_p^*(F)$ defines a 1–1 mapping from the variables in $\Sigma_p^*(F)$ to the domain. We can therefore find an isomorphic interpretation satisfying the requirements for I^* listed above. $\Box$

As an illustration, consider formula F^* given by Equation 2 resulting from the transformation of formula F given by Equation 1. We have $\Sigma_p^*(F) = \{vf_1, vf_2, vg_1, vg_2\}$ and $\Sigma_g^*(F) = \{x, y\}$. Suppose we use bit vectors of length 3 as the domain of interpretation. Then we could let $\mathcal{D}_g$ be $\{\langle 0,0,0\rangle, \langle 0,0,1\rangle\}$. We assign x the fixed interpretation $\langle 0,0,0\rangle$, and y the interpretation $\langle 0,0,a\rangle$ where a is a propositional variable. Viewing truth values $\mathbf{true}$ and $\mathbf{false}$ as representing bit values 1 and 0, respectively, the different interpretations of a will then cover both the case where x and y have equal interpretations as well as where they are distinct. For variables vf_1, vf_2, vg_1, and vg_2, we can assign fixed interpretations $\langle 1,0,0\rangle$, $\langle 1,0,1\rangle$, $\langle 1,1,0\rangle$, and $\langle 1,1,1\rangle$, respectively. Thus, we can translate our formula F into a propositional formula having just a single propositional variable.

Ackermann also describes a scheme for replacing function application terms by domain variables [Ack54]. Using his scheme, we simply replace each instance of a function application by a newly-generated domain variable and then introduce constraints expressing functional consistency. For the example formula given by Equation 1 we would get a modified formula:

$$\begin{array}{l}((x = y \Rightarrow vf_1 = vf_2) \wedge (vf_1 = vf_2 \Rightarrow vg_1 = vg_2)) \\ \quad \Rightarrow \quad (x = y \Rightarrow vg_1 = vg_2)\end{array}$$

Observe, however, that there is no clear way to exploit the maximal diversity property with this translated form. If we replace vf_1 and vf_2 by distinct values in the above case, we fail to consider any interpretations in which arguments x and y have equal values.

5 Modeling Microprocessors in PEUF

Our interest is in verifying pipelined microprocessors, proving their equivalence to an unpipelined instruction set architecture model. We use the approach pioneered by Burch and Dill [BD94] in which the abstraction function from pipeline state to architectural state is computed by symbolically simulating a flushing of the pipeline state and then projecting away the state of all but the architectural state elements, such as the register file, program counter, and data memory. Operationally, we construct two sets of p-terms describing the final values of the state elements resulting from two different symbolic simulation sequences—one from the pipeline model and one from the instruction set model. The correctness condition is represented by a p-formula expressing the equality of these two sets of p-terms.

Our approach starts with an RTL or gate-level model of the microprocessor and performs a series of abstractions to create a model of the data path using terms that satisfy the restrictions of PEUF. Examining the structure of a pipelined processor, we find that the signals we wish to abstract as terms can be classified as either program data, instruction addresses, or register identifiers. By proper construction of the data path model, both program data and instruction addresses can be represented as p-terms. Register identifiers, on the other hand, must be modeled as g-terms, because their comparisons control the stall and bypass logic. The remaining control logic is kept at the bit level.

In order to generate such a model, we must abstract the operation of some of the processor units. For example, the data path ALU is abstracted as an uninterpreted p-function, generating a data value given its data and control inputs. We model the PC incrementer and the branch target logic as uninterpreted functions generating instruction addresses. We model the branch decision logic as an uninterpreted predicate indicating whether or not to take the branch based on data and control inputs. This allows us to abstract away the data equality test used by the branch-on-equal instruction. The instruction memory can be abstracted as an uninterpreted function, since it is considered to be read-only.

To model the register file, we use the memory model described by Burch and Dill [BD94], creating a nested *ITE* structure to record the history of writes to the memory. This approach requires equations between memory addresses controlling the *ITE* operations. For the register file, such equations are allowed since g-term register identifiers serve as addresses. For the data memory, however, the memory addresses are p-term program data, and hence such equations cannot be used. Instead, we model the data memory as a generic state machine, changing state in some arbitrary way for each write operation, and returning some arbitrary value dependent on the state and the address for each read operation. Such an abstraction technique is sound, but it does not capture all of the properties of a memory. It is satisfactory for modeling processors in which there is no reordering of writes relative to each other or relative to reads.

6 Experimental Results

In [VB98], we described the implementation of a symbolic simulator for verifying pipelined systems using vectors of Boolean variables to encode domain variables, effectively treating all terms as g-terms. This simulation is performed directly on a modified gate-level representation of the processor. In this modified version, we replace all state holding elements with behavioral models we call Efficient Memory Models (EMMs). In addition all data-transformation elements (e.g., ALUs, shifters, PC incrementers) are replaced by read-only EMMs, which effectively implement the transformation of function applications into nested *ITE* expressions. Modifying this program to exploit maximal diversity simply involves having the EMMs generate expressions containing fixed bit patterns rather than vectors of Boolean variables. All performance results presented here were measured on a 125 MHz Sun Microsystems SPARC-20.

We constructed several simple pipeline processor designs based on the MIPS instruction set. We abstract register identifiers as g-terms, and hence our verification covers all possible numbers of program registers including the 32 of the MIPS instruction set. The simplest version of the pipeline implements ten different Register-Register and Register-Immediate instructions. Our program could verify this design in 48 seconds of CPU time and just 7 MB of memory using vectors of Boolean variables to encode domain variables. Using fixed bit patterns reduces the complexity of the verification to 6 seconds and 2 MB.

We then added a memory stage to implement load and store instructions. An interlock stalls the processor one cycle when a load instruction is followed by an instruction requiring the loaded result. Treating all terms as g-terms and using vectors of Boolean variables to encode domain variables, we could not verify this data path, despite running for over 2000 seconds. The fact that both addresses and data for the memory come from the register file induces a circular constraint on the ordering of BDD variables encoding the terms. On the other hand, exploiting maximal diversity by using fixed bit patterns for register values eliminates these variable ordering concerns. As a consequence, we could verify the 32-bit version of this design in just 12 CPU seconds using 1.8 MB.

Finally, we verified a complete CPU, with a 5-stage pipeline implementing 10 ALU instructions, load and store, and MIPS instructions `j` (jump with target computed from instruction word), `jr` (jump using register value as target), and `beq` (branch on equal). This design is comparable to the DLX design verified by Burch and Dill in [BD94], although our version is closer to an actual gate-level implementation. We were unable to verify this processor using the scheme of [VB98]. Having instruction addresses dependent on instruction or data values leads to exponential BDD growth when modeling the instruction memory. Modeling instruction addresses as p-terms, on the other hand, makes this verification tractable. We can verify the 32-bit version processor using 169 CPU seconds and 7.5 MB.

7 Conclusions

Eliminating Boolean variables in the encoding of terms representing program data and instruction addresses has given us a major breakthrough in our ability to verify pipelined processors. Our BDD variables now only encode control conditions and register identifiers. For classic RISC pipelines, the resulting state space is small and regular enough to be handled readily with BDDs.

We believe that there are many optimizations that will yield further improvements in the performance of Boolean methods for deciding formulas involving uninterpreted functions. We have found that relaxing functional consistency constraints to allow independent functionality of different instructions, as was done in [DPR98], can dramatically improve both memory and time performance. We have devised a variation on the scheme of [GSZAS98] for generating a propositional formula using Boolean variables to encode the relations between terms [BGV99]. Our method exploits maximal diversity to greatly reduce the number of propositional variables in the generated formula. We are also considering the use of satisfiability checkers rather than BDDs for performing our tautology checking

References

[Ack54] W. Ackermann, *Solvable Cases of the Decision Problem*, North-Holland, Amsterdam, 1954.

[BBCZ98] S. Berezin, A. Biere, E. M. Clarke, and Y. Zhu, "Combining symbolic model checking with uninterpreted functions for out of order processor verification," *Formal Methods in Computer-Aided Design FMCAD '98,* G. Gopalakrishnan and P. Windley, *eds.*, LNCS 1522, Springer-Verlag, November, 1998, pp. 187–201.

[BGV99] R. E. Bryant, S. German, and M. N. Velev, "Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic," Technical report CMU-CS-99-115, Carnegie Mellon University, 1999. Available as: `http://www.cs.cmu.edu/˜bryant/pubdir/cmu-cs-99-115.ps`.

[BD94] J. R. Burch, and D. L. Dill, "Automated verification of pipelined microprocessor control," *Computer-Aided Verification CAV '94,* D. L. Dill, *ed.*, LNCS 818, Springer-Verlag, June, 1994, pp. 68–80.

[DPR98] W. Damm, A. Pnueli, and S. Ruah, "Herbrand automata for hardware verification," *9th International Conference on Concurrency Theory CONCUR '98*, Springer-Verlag, September, 1998.

[GSZAS98] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, "BDD based procedures for a theory of equality with uninterpreted functions," *Computer-Aided Verification CAV '98*, A. J. Hu and M. Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June, 1998, pp. 244–255.

[HKGB97] R. Hojati, A. Kuehlmann, S. German, and R. K. Brayton, "Validity checking in the theory of equality with uinterpreted functions using finite instantiations," Unpublished paper presented at the *International Workshop on Logic Synthesis*, 1997.

[NO80] G. Nelson, and D. C. Oppen, "Fast decision procedures based on the congruence closure," *J. ACM*, Vol. 27, No. 2 (1980), pp. 356–364.

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel, "Deciding equality formulas by small-domain instantiations," *Computer-Aided Verification CAV '99*, this proceedings, 1999.

[VB98] M. N. Velev, and R. E. Bryant, "Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking." *Formal Methods in Computer-Aided Design FMCAD '98*, G. Gopalakrishnan and P. Windley, *eds.*, LNCS 1522, Springer-Verlag, November, 1998, pp. 18–35.