A Quantifier Elimination Algorithm for Linear Modular Equations and Disequations *

Ajith K John[1] and Supratik Chakraborty[2]

[1] Homi Bhabha National Institute, BARC, Mumbai, India
[2] CFDVS, Dept. of Computer Sc. & Engg., IIT Bombay, India

Abstract. We present a layered bit-blasting-free algorithm for existentially quantifying variables from conjunctions of linear modular (bit-vector) equations (LMEs) and disequations (LMDs). We then extend our algorithm to work with arbitrary Boolean combinations of LMEs and LMDs using two approaches – one based on decision diagrams and the other based on SMT solving. Our experiments establish conclusively that our technique significantly outperforms alternative techniques for eliminating quantifiers from systems of LMEs and LMDs in practice.

1 Introduction

Quantifier elimination (QE) is the process of converting a formula containing existential and/or universal quantifiers in a suitable logic into a semantically equivalent quantifier-free formula. Formally, let A be a quantifier-free formula over a set X of free variables in a first-order theory $\mathcal{T}$. Consider the quantified formula $Q_1 y_1\, Q_2 y_2 \ldots Q_m y_m.\, A$, where $Y = \{y_1, \ldots y_m\}$ is a subset of X, and $Q_i \in \{\exists, \forall\}$ for $i \in \{1, \ldots m\}$. QE computes a quantifier-free formula A' with free variables in $X \setminus Y$ such that $A' \equiv_{\mathcal{T}} Q_1 y_1\, Q_2 y_2 \ldots Q_m y_m.\, A$, where $\equiv_{\mathcal{T}}$ denotes semantic equivalence in theory $\mathcal{T}$. This has a number of important applications in formal verification and program analysis. Example applications include computing abstractions of symbolic transition relations, computing strongest postconditions of program statements and computing interpolants in CEGAR frameworks. Since $\forall y.\, \varphi \equiv \neg\exists y.\, \neg\varphi$ in all first-order theories, it suffices to focus on algorithms for eliminating existential quantifiers. This paper presents one such algorithm for a fragment of the theory of bit-vectors that we have found useful in verification of word-level RTL designs.

Currently, the most popular technique for eliminating quantifiers from bit-vector formulae involves *blasting* bit-vectors into individual bits (Boolean variables), followed by quantification of the blasted Boolean variables. This approach has some undesirable features. For example, blasting involves a bitwidth-dependent blow-up in the size of the problem. This can present scaling problems in the usage of Boolean reasoning tools (e.g. BDD based tools), especially when reasoning about wide words. Similarly, given an instance of the QE problem, blasting variables that are quantified may transitively require blasting other variables (that are not quantified) as well. This can cause the quantifier-eliminated

* This work was supported by a research grant from Board of Research in Nuclear Sciences, India.

formula to appear more like a propositional formula on blasted bits, instead of being a bit-vector formula. Since reasoning at the level of bit-vectors is often more efficient in practice than reasoning at the level of blasted bits, QE using bit-blasting may not be the best option if the quantifier-eliminated formula is intended to be used in further bit-vector level reasoning. This motivates us to ask if we can efficiently eliminate quantifiers in the theory of bit-vectors without resorting to bit-blasting (or model enumeration) in practice. Ideally, we would have liked to obtain such a QE procedure for the entire theory of bit-vectors. Unfortunately, we do not have this yet. We therefore focus on a fragment of the theory, namely Boolean combinations of equations and disequations of bit-vectors, that we have found useful in word-level verification of RTL designs, and present a QE procedure for this fragment.

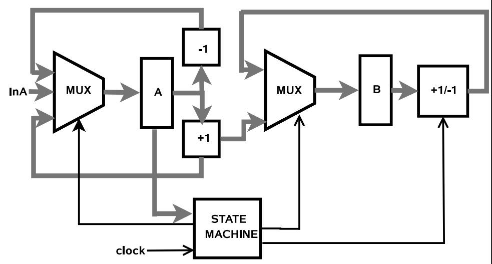
Since bit-vector arithmetic is the same as modular arithmetic on integers, our algorithm can also be viewed as one for existentially quantifying variables from a Boolean combination of linear modular integer equations and disequations.

A Linear Modular Equation (LME) is an equation of the form $c_1 \cdot x_1 + \cdots + c_n \cdot x_n = c_0 \pmod{2^p}$ where p is a positive integer constant, $x_1, \ldots, x_n$ are p-bit non-negative integer variables, and $c_0, \ldots, c_n$ are integer constants in $\{1, \ldots, 2^p - 1\}$. Similarly, a Linear Modular Disequation (LMD) is a disequation of the form $c_1 \cdot x_1 + \cdots + c_n \cdot x_n \neq c_0 \pmod{2^p}$. Conventionally, 2^p is called the modulus of the LME or LMD. For notational convenience, we will henceforth use "LMC" to refer to an LME or LMD. Since every variable in an LMC $c_1 \cdot x_1 + \cdots + c_n \cdot x_n \bowtie c_0 \pmod{2^p}$, where $\bowtie \in \{=, \neq\}$, represents a p-bit integer, it follows that a set of LMCs sharing a variable must have the same modulus. However, there are applications where we need to consider Boolean combinations of LMCs that do not share any variable, and have different moduli. In such cases, we propose to appropriately shift the moduli of LMCs, so that all LMCs have the same modulus. This can always be done since the LMCs $\lambda_1 \equiv c_1 \cdot x_1 + \cdots + c_n \cdot x_n \bowtie c_0 \pmod{2^p}$ and $\lambda_2 \equiv 2^q \cdot c_1 \cdot x_1' + \cdots + 2^q \cdot c_n \cdot x_n' \bowtie 2^q \cdot c_0 \pmod{2^{p+q}}$ are related in the following way: every solution of λ_1 can be bit-extended to give a solution for λ_2, and every solution of λ_2 can be bit-truncated to give a solution for λ_1. Hence, using λ_2 in place of λ_1 suffices for checking satisfiability and also for finding solutions of Boolean combinations of LMCs. In the remainder of this paper, we will assume without loss of generality that whenever we consider a set of LMCs, all of them have the same modulus.

Our primary motivation comes from bounded model checking (BMC) of word-level RTL designs. As an example, consider the synchronous circuit shown in Fig. 1, with the relevant part of its functionality described in VHDL with the figure. The thick shaded arrows and the thin solid arrows in the figure represent 8-bit words and 1-bit lines respectively.

The circuit comprises a controller and two 8-bit registers, A and B. The controller switches between two states, 0 and 1, depending on the value of A. In state 0, A works as a down-counter until it reaches `0x00`[3], in which case A loads itself with an input value from InA and the controller switches to state

[3] We use the `0x` prefix to denote hexadecimal values.




```
   ....
if (clock’event and clock = ’1’) then
 case state is
  when ’0’ =>
   if (A = x"00") A <= InA; state <= ’1’;
   else  A <= A-1; state <= ’0’; end if;
   if  (A = x"ff") B <= B-1;
   else B <= A+1; end if;
  when others =>
   if  (A = x"ff") A <= InA; state <= ’0’;
                   B <= B+1;
   else A <= A+1; state <= ’1’; B <= A+1;
   end if;
 end case;
end if;
   ....
```

Fig. 1. An Example Circuit

1. In state 1, A works as an up-counter until it reaches 0xff, in which case it loads the value from InA and the controller switches to state 0. Register B is always loaded with the value of $A + 1$ except when A has the value 0xff. If this happens in state 0 (down-counting state), B decrements its previously stored value; otherwise, B increments its previously stored value.

A word-level transition relation, R, for this circuit can be obtained by conjoining the following three equality relations, where all operations on A and B are assumed to be modulo 2^8.

$$\mathsf{state}' = \mathsf{ite}(\mathsf{state} = 0,\ \mathsf{ite}(\mathsf{A} = \mathsf{0x00},\ 1,\ 0),\ \mathsf{ite}(\mathsf{A} = \mathsf{0xff},\ 0,\ 1))$$
$$\mathsf{A}' = \mathsf{ite}(\mathsf{state} = 0,\ \mathsf{ite}(\mathsf{A} = \mathsf{0x00},\ \mathsf{InA},\ \mathsf{A} - 1),\ \mathsf{ite}(\mathsf{A} = \mathsf{0xff},\ \mathsf{InA},\ \mathsf{A} + 1))$$
$$\mathsf{B}' = \mathsf{ite}(\mathsf{state} = 0,\ \mathsf{ite}(\mathsf{A} = \mathsf{0xff},\ \mathsf{B} - 1,\ \mathsf{A} + 1), \mathsf{ite}(\mathsf{A} = \mathsf{0xff},\ \mathsf{B} + 1,\ \mathsf{A} + 1))$$

In the above relations, state', A' and B' refer to values of state, A and B after the next rising edge of the clock. Note also that A, A', B and B' are $8-$bit wide bit-vector variables and state and state' are propositional variables. Since R is a conjunction of equalities involving ite, and since $a = ite(b, c, d)$ represents $(b \wedge (a = c)) \vee (\neg b \wedge (a = d))$, R is essentially a Boolean combination of LMCs.

The above circuit has the property that once started in state 0, it never reaches state 1 with 0x00 in register B. Suppose we wish to use BMC to prove that this property holds for the first N cycles of operation. This can be done by unrolling the transition relation N times, conjoining the unrolled relation with the negation of the property, and then checking for satisfiability of the resulting constraint using an SMT solver that can reason about bit-vectors. Since R contains all variables (in unprimed and primed versions) that appear in the RTL description, unrolling R a large number of times gives a constraint with a large number of variables. This problem is particularly acute for circuits with a large number of internal state variables. While the number of variables in a constraint is not the only factor that affects the performance of an SMT solver, for large enough values of N, the increased variable count indeed has an adverse effect on the performance of the solver, as indicated by our experiments.

In order to alleviate the above problem, one can use an abstract transition relation R' that relates only a chosen subset of variables relevant to the property being checked, while abstracting the relation between the other variables. In our example, we can compute such an R' by existentially quantifying the bit-vector variables A and A' from R. This gives R' as:

$$((\mathsf{state}' = 1) \wedge (\mathsf{B}' = \mathsf{0x01})) \vee$$

$$((\mathsf{state}' = 0) \wedge (\mathsf{B}' = \mathsf{ite}(\mathsf{state} = 0, \mathsf{B} - 1, \mathsf{B} + 1))) \vee$$
$$((\mathsf{state}' = \mathsf{state}) \wedge (\mathsf{B}' \neq \mathsf{0x00}) \wedge (\mathsf{B}' \neq \mathsf{0x01}))$$

On careful examination, it can be seen that if we unroll R' (instead of R) during BMC, we can still prove that the circuit never reaches state 1 with B set to 0x00 if it starts in state 0. Since R' contains fewer variables than R, the constraint obtained by unrolling R' has fewer variables. In general, this can lead to significantly better performance of the back-end SMT solver, as demonstrated in our experiments.

The example presented above is representative of a more general scenario. In general, Boolean combinations of LMCs arise when building transition relations for RTL designs and/or embedded systems containing conditional statements that check for equalities of words/registers. Building an abstract transition relation in such cases requires existentially quantifying variables from Boolean combinations of LMCs. Obtaining the abstract transition relation at the word-level is particularly appealing since it allows word-level reasoning to be applied to the abstraction. This motivates us to study the problem of eliminating quantifiers from Boolean combinations of LMCs without resorting to bit-blasting (or model enumeration) in practice.

Contributions. There are two primary contributions of this paper. First, we describe a bit-blasting-free algorithm for eliminating quantifiers from conjunctions of LMCs. The algorithm is based on a layered approach, i.e., the cheaper layers are invoked first and more expensive layers are called only when required. Later, we extend this to QE algorithm for Boolean combinations of LMCs. While our algorithm uses a final layer of model enumeration for the sake of theoretical completeness, extensive experiments indicate that we never need to invoke this layer in practice. Our second contribution is an extensive set of carefully conducted experiments that not only demonstrate the effectiveness of our approach over alternative techniques, but also allows us to identify criteria for choosing the right QE technique for a given problem instance.

Related Work. Several interesting approaches have been proposed earlier for reasoning about LMEs (e.g., [6, 7]). Although our study indicates that non-trivial counts of LMDs appear in constraints arising from real verification problems, LMDs have traditionally received relatively less attention. Jain et al [7] showed that the satisfiability problem for a conjunction of LMCs is NP-hard. However, their work subsequently focused on systems of LMEs and Linear Diophantine Equations and Disequations, and discussed algorithms to compute interpolants in such systems. Bit-blasting [3] followed by bit-level QE is arguably the dominant technique used in practice for eliminating quantifiers from bit-vector constraints. As discussed earlier, this approach, though simple, destroys the word-level structure of the problem and does not scale well for LMCs with large modulus. Since LMEs and LMDs can be expressed as formulae in Presburger Arithmetic (PA) [3], QE techniques for PA (e.g.those in [5]) can also be used to eliminate quantifiers from Boolean combinations of LMCs. Similarly, automata-theoretic approaches for eliminating quantifiers from PA formulae [8] can also be used. However once converted to PA formulae, converting back to

Boolean combinations of LMCs is difficult. Also, empirical studies have shown that using PA techniques to eliminate quantifiers from Boolean combinations of LMCs often blows up in practice [3]. The work that is most closely related to our is that of Ganesh and Dill [6]. The authors of [6] present a technique for reducing LMEs to a solved form by selecting variables in a specific order. While this does not directly give us a technique to eliminate a user-specified variable from a conjunction of LMEs, their work can be extended to achieve this. More importantly, [6] does not consider the problem of eliminating variables in constraints involving LMDs. This problem is addressed in our current work.

2 Quantifier Elimination for a Conjunction of LMCs

The problem we wish to solve in this section can be formally stated as follows. Given a set of LMCs over variables $x_1, \ldots, x_n$, let A denote the conjunction of the LMCs. Without loss of generality, we wish to compute $A' \equiv \exists x_1 \cdots \exists x_t. A$, where A' is a Boolean combination of LMCs. For reasons of succinctness, we also require that A' contains no ground terms other than integer constants, and no ground (sub-)formulas other than true and false. This problem is easily seen to be NP-hard. This follows from the facts: (i) the satisfiability problem for a conjunction of LMCs is NP-hard, even when all moduli are a priori fixed to 4 (see [7]), and (ii) a conjunction of LMCs A over $x_1, \ldots, x_n$ is satisfiable iff an algorithm for computing $A' \equiv \exists x_1 \cdots \exists x_n. A$ returns true (due to the succinctness requirement of A').

Since an algorithm for computing $\exists x_i. A$ can be used in an iterative way to compute $\exists x_1 \cdots \exists x_t. A$, we will initially focus on the (seemingly simpler) problem of computing $\exists x_i. A$ in the subsequent discussion. All LMCs considered in the remainder of this section have modulus 2^p, for some positive integer p, unless stated otherwise. For notational clarity, we will therefore omit mentioning " (mod 2^p)" with LMCs in the following discussion. We have skipped the proofs of lemmas and the details of some procedures which can be found in a detailed version of this paper [13].

In the following discussion, we use names starting with "*QE1*" and "*QE*" for procedures to eliminate a single quantifier and multiple quantifiers respectively.

Lemma 1. *An LMC $c_1 \cdot x_1 + \cdots + c_n \cdot x_n \bowtie c_0$ can be equivalently expressed as $2^{k_1} \cdot x_1 \bowtie t_1$, where $\bowtie \in \{=, \neq\}$, t_1 is a term free of x_1 and k_1 is an integer such that $0 \leq k_1 \leq p-1$.*

Example: All LMCs in this example have modulus 8. Consider the LME $6x + 4y = 0$. Rearranging the terms modulo 8, we get $3 \cdot 2^1 x = 4y$. Multiplying by 3 (multiplicative inverse of 3 modulo 8) and simplifying gives, $2^1 x = 4y$.

For brevity, henceforth whenever we express LMCs as $2^{k_i} \cdot x_1 \bowtie t_i$ where $\bowtie \in \{=, \neq\}$, we will omit mentioning "t_i is a term free of x_1 and k_i is an integer such that $0 \leq k_i \leq p-1$".

Lemma 2. $\exists x_1. (2^{k_1} \cdot x_1 = t_1) \equiv (2^{p-k_1} \cdot t_1 = 0)$

Example: All LMCs in this example have modulus 8. $\exists y. (2^1.y = 5.x + 2) \equiv (2^{3-1}.(5.x + 2) = 0) \equiv (4.x = 0)$

Lemma 3. *Let A be the conjunction of m LMEs of the form $2^{k_i} \cdot x_1 = t_i$, where i ranges from 1 through m. Then $\exists x_1.\, A$ can be equivalently expressed as a conjunction of LMEs each of which is free of x_1.*

Example: All LMCs in this example have modulus 8. Consider the problem of computing $\exists y.\, ((2^1 y = 5x + 2) \wedge (2^2 y = 5x + 6z) \wedge (2^1 y = 2x + 4))$. This can be equivalently expressed as $\exists y.\, ((2y = 5x + 2) \wedge (2 \cdot (5x + 2) = 5x + 6z) \wedge (5x + 2 = 2x + 4))$. Simplifying modulo 8, we get $\exists y.\, ((2y = 5x + 2)) \wedge (5x + 2z = 4) \wedge (3x = 2)$. Using Lemma 2, we obtain the final result as $(4x = 0) \wedge (5x + 2z = 4) \wedge (3x = 2)$.

Lemma 4. *Let A be the conjunction of r LMCs of the form $2^{k_i} \cdot x_1 \bowtie t_i$, where $\bowtie \in \{=, \neq\}$ and i ranges from 1 through r. Let $2^{k_1} \cdot x_1 = t_1$ be the LME with the minimum k_i among all LMEs in A. Then $\exists x_1.\, A \equiv \psi_1 \wedge \exists x_1.\, \psi_2$, where ψ_1 is a conjunction of LMCs independent of x_1, and ψ_2 is a conjunction of LMCs with at most one LME $2^{k_1} \cdot x_1 = t_1$. In addition, ψ_2 contains only those LMDs in A in which the coefficient of x_1 is of the form 2^{k_i}, where $k_i < k_1$.*

Example: All LMCs in this example have modulus 8. Consider the problem of computing $\exists y.\, ((2^1 y = 5x + 2) \wedge (2^2 y = 5x + 6z) \wedge (2^1 y \neq 2x + 4) \wedge (2^0 y \neq 6x + 7z))$. This can be equivalently expressed as $\exists y.\, ((2y = 5x + 2) \wedge (2 \cdot (5x + 2) = 5x + 6z) \wedge (5x + 2 \neq 2x + 4) \wedge (y \neq 6x + 7z))$. Simplifying modulo 8, we get $(5x + 2z = 4) \wedge (3x \neq 2) \wedge \exists y.\, ((2y = 5x + 2) \wedge (y \neq 6x + 7z))$. Note that ψ_1 and $\exists x_1.\, \psi_2$ here are $(5x + 2z = 4) \wedge (3x \neq 2)$ and $\exists y.\, ((2y = 5x + 2) \wedge (y \neq 6x + 7z))$ respectively.

The above results immediately yield two simple algorithms: (a) *QE1_1LME* that takes an LME and a variable to quantify out, and returns the equivalent quantifier-free formula (based on Lemma 2), and (b)*QE1_Layer1* that takes a conjunction of LMCs and a variable x_1 to quantify out and returns the equivalent conjunction of ψ_1 and $\exists x_1.\psi_2$ (as given by Lemma 4).

If the k_i's of all LMDs in A are such that $k_1 \leq k_i$, then $\exists x_1.\, \psi_2$ reduces to $\exists x_1.\, (2^{k_1} \cdot x_1 = t_1)$. According to Lemma 2, this is equivalent to $2^{p-k_1} \cdot t_1 = 0$. Hence, in this case, algorithms *QE1_Layer1* and *QE1_1LME* suffice to compute $\exists x_1.\, A$. In general, however, $\exists x_1.\, \psi_2$ may contain LMDs containing x_1 that require further processing before x_1 is eliminated. We describe techniques for doing this in the following subsections.

2.1 Dropping Unconstraining LMDs

We now consider the problem of simplifying $\exists x_1.\, \psi_2$ obtained above, when $\exists x_1.\, \psi_2$ contains LMDs. Let $\psi_2 \equiv \xi \wedge \lambda$, where λ is an LMD and ξ is a conjunction of LMCs. We say that λ is *unconstraining* in $\exists x_1.\, \psi_2$ iff $\exists x_1.\, (\xi \wedge \lambda) \equiv \exists x_1.\, \xi$. Unconstraining LMDs can simply be dropped from $\exists x_1.\, \psi_2$, thereby simplifying the task of QE. Unfortunately, identifying all unconstraining LMDs from ψ_2 involves invoking an SMT solver for quantified bit-vector formulas. In this subsection, we present a sound technique for identifying a subset of unconstraining LMDs in $\exists x_1.\, \psi_2$. Our approach exploits the fact that an LMD is satisfied even if a single bit in the left-hand side of the LMD differs from the corresponding bit in the right-hand side. We therefore propose to identify LMDs in $\exists x_1.\, \psi_2$ that can be satisfied by selectively assigning values to specific bits of x_1, without causing any other LME or LMD in $\exists x_1.\, \psi_2$ to be violated. Since x_1 is existentially quantified, these LMDs are effectively unconstraining in $\exists x_1.\, \psi_2$. We illustrate this idea below through an example.

Consider the problem $\exists x.\, (\xi \wedge \lambda)$ where $\xi \equiv (4x = 6y + 2z) \wedge (2x \neq 2y + 4z) \wedge (2x \neq 6y + 6z)$ and $\lambda \equiv (x \neq y + z)$, and all LMCs have modulus 8. For clarity of exposition, we use the notation $x[i]$ to denote the i^{th} bit of a bit-vector x, and adopt the convention that $x[0]$ denotes the least significant bit of x. We claim that any solution of ξ can be "engineered" by possibly modifying the value of $x[2]$ to give a solution of $\xi \wedge \lambda$, and vice versa. In order to see why this is true, note that the LME $4x = 6y + 2z$ constrains only $x[0]$ and the LMDs $(2x \neq 2y + 4z)$, $(2x \neq 6y + 6z)$ constrain only $x[0]$ and $x[1]$. Therefore, the value of $x[2]$ does not affect satisfaction of ξ. Any solution of ξ can therefore be engineered to a solution of $\xi \wedge \lambda$ by ensuring that $x[2]$ differs from the most-significant bit of $y + z$. Hence, $\exists x.\, (\xi) \Rightarrow \exists x.\, (\xi \wedge \lambda)$. The converse, i.e. $\exists x.\, (\xi \wedge \lambda) \Rightarrow \exists x.\, (\xi)$ obviously holds. Hence in this example, $(x \neq y + z)$ is an unconstraining LMD in $\exists x.\, (\xi \wedge \lambda)$.

DropLMDSimple(E, D, x_1)

```
core ← E;
while(core ≠ E ∪ D)
  if (isExt(core, E ∪ D, x1))
    return core;
  else
    d ← getLstCnstr(D\core);
    core ← core ∪ d;
return core;
```

DropLMDWithSMT(E, D, x_1)

```
while(true)
  impl ← NULL;
  for each LMD d ∈ D
    if (E ∪ (D\d) ⇒d)
      impl ← d; break;
  if (impl = NULL)
    break;
  D ← D\impl;
return E∪D;
```

Fig. 2. Algorithms to drop unconstraining LMDs

The above idea leads to a simple algorithm, called *DropLMDSimple*, shown in Fig. 2. This algorithm takes as inputs a set of LMEs E, a set of LMDs D, and a variable x_1 to be quantified from the conjunction of all LMCs in $E \cup D$. Algorithm *DropLMDSimple* returns a subset of LMCs in $E \cup D$ such that the result of quantifying x_1 from the conjunction of LMCs in this subset is equivalent to the result of quantifying x_1 from the conjunction of LMCs in $E \cup D$.

Algorithm *DropLMDSimple* computes the desired subset in a variable *core* that is initialized to E. Subsequently, it determines if any solution to the conjunction of LMCs in *core* can be engineered by modifying specific bits of x_1 to give a solution to the conjunction of LMCs in $E \cup D$. This is achieved by invoking a function *isExt*. If such an engineering is indeed possible, then all LMDs not in *core* are unconstraining, and algorithm *DropLMDSimple* returns *core*. Otherwise we identify the LMDs in $D \setminus core$ whose truth depends on the least number of bits of x_1 using a function *getLstCnstr*. Intuitively, these LMDs are the most difficult ones to satisfy among the LMDs in $D \setminus core$. These LMDs are then included in *core* and the process repeats. Clearly, algorithm *DropLMDSimple* terminates since *core* cannot have more LMCs than those in $E \cup D$.

Since each LMD is of the form $2^{k_i} \cdot x_1 \neq t_i$, the LMD with the largest k_i is the one whose truth depends on the least number of bits of x_1. This gives a simple implementation of function *getLstCnstr*. One possible implementation of *isExt* is through the use of an SMT solver that checks if one quantified formula implies another quantified formula. However, this is inefficient in general. Instead, we propose an implementation of *isExt* based on the following Lemma.

Lemma 5. *Let k_{core} be the smallest among the k_i's of all LMCs in core. Let $D \setminus core$ be expressed as $\{(2^{k_1} \cdot x_1 \neq t_1), \ldots, (2^{k_n} \cdot x_1 \neq t_n)\}$. If $\eta = 2^{k_{core}} - \sum_{i=1}^{n} 2^{k_i} \geq 1$, any solution to the conjunction of LMCs in core can be engineered to give a solution to the conjunction of LMCs in $E \cup D$.*

We give a sketch of the proof of Lemma 5 here. Let C_1 and C_2 be the conjunction of LMCs in *core* and the conjunction of LMDs outside *core* respectively. Let π be any solution to C_1. π constrains only the bits $x[0]$ through $x[p - k_{core} - 1]$. Hence there are $2^{k_{core}}$ ways in which bits $x[p-1]$ through $x[p - k_{core}]$ can be assigned values such that π remains as a solution to C_1. It can be shown that $\eta = 2^{k_{core}} - \sum_{i=1}^{n} 2^{k_i}$ under-approximates the number of ways in which bits $x[p-1]$ through $x[p - k_{core}]$ can be assigned values such that π becomes a solution to C_2 and remains as a solution to C_1. Therefore if $\eta \geq 1$, there exists at least one assignment of values to bits $x[p-1]$ through $x[p - k_{core}]$ such that π becomes a solution to the conjunction of LMCs in $E \cup D$.

DropLMDSimple may not be able to identify all the unconstraining LMDs in $\exists x_1. \psi_2$. For example, consider the problem $\exists x. ((2x = y) \wedge (x \neq 2y) \wedge (x \neq y))$, where all LMCs have modulus 8. We have, *core* $= \{2x = y\}$, $k_{core} = 1$, $k_1 = k_2 = 0$. Therefore, $\eta = 0$ and *DropLMDSimple* identifies that it is not possible in general to engineer a solution of $(2x = y)$ to give a solution of $(2x = y) \wedge (x \neq 2y) \wedge (x \neq y)$ by assigning values to specific bits of x. Hence, *DropLMDSimple* cannot identify any LMD to drop. However, it can be seen that $(2x = y) \wedge (x \neq 2y) \Rightarrow (x \neq y)$. Hence $\exists x. ((2x = y) \wedge (x \neq 2y) \wedge (x \neq y)) \equiv \exists x. ((2x = y) \wedge (x \neq 2y))$. Once $x \neq y$ is dropped, *DropLMDSimple* can further reduce $\exists x. ((2x = y) \wedge (x \neq 2y))$ to $\exists x. (2x = y)$. Based on this idea, we present an algorithm to drop implied LMDs called *DropLMDWithSMT* (see Fig. 2). The notation used in this algorithm is the same as used in algorithm *DropLMDSimple*. The implication check in *DropLMDWithSMT* requires invoking an SMT solver, in general.

We now present an algorithm *QE1_Layer3* which drops LMDs from $\exists x_1. \psi_2$ using *DropLMDSimple* and *DropLMDWithSMT*. Given $\exists x_1. \psi_2$, *QE1_Layer3* initially employs *DropLMDSimple* to drop unconstraining LMDs. If there still exist LMDs, *DropLMDWithSMT* is applied to identify the implied LMDs and drop them. If there exist LMDs in the output of *DropLMDWithSMT*, it is given to *DropLMDSimple*. Thus finally, we are left with a conjunction of LMCs ψ_2' with possibly fewer LMDs compared to ψ_2, while guaranteeing that $\exists x_1. \psi_2 \equiv \exists x_1. \psi_2'$.

The algorithms *QE1_Layer1*, *DropLMDSimple* and *QE1_Layer3* form the first three layers of our layered QE algorithm. We present in Fig. 3, a procedure *QE1_Layers1To3* which tries to compute $\exists x_1 . A$ using these layers. Initially *QE1_Layer1* is called to reduce $\exists x_1 . A$ to $\psi_1 \wedge \exists x_1 . \psi_2$. If ψ_2 is free of LMDs, *QE1_1LME* is called to compute $\exists x_1 . \psi_2$ and hence $\exists x_1 . A$ is computed by the first layer itself. If ψ_2 is not free of LMDs, *QE1_Layers1To3* initially calls *DropLMDSimple* and later on *QE1_Layer3* (if required) to drop the LMDs. If all the LMDs in $\exists x_1 . \psi_2$ are dropped by *DropLMDSimple* (*QE1_Layer3*), $\exists x_1 . A$ gets computed in the second (third) layer. Otherwise, *QE1_Layers1To3* returns $\psi_1 \wedge \exists x_1 . \psi_2'$ such that $\psi_1 \wedge \exists x_1 . \psi_2' \equiv \exists x_1 . A$. The techniques to compute such (harder) instances of $\exists x_1 . A$ are presented in the following subsection.

2.2 Splitting and Model Enumeration

Let us have a closer look at the instances of $\exists x_1 . A$ which cannot be computed by *QE1_Layers1To3*. The difficulty in QE in such cases arises from the fact that there are no LMEs constraining some of the bits of x_1 constrained by the LMDs. For example, consider the problem of computing $\exists x. ((2x = a) \wedge (x \neq b) \wedge (x \neq c))$ where all the LMCs have modulus 8. The LME $(2x = a)$ constrains only the bits $x[1]$ and $x[0]$ whereas the LMDs constrain the bits $x[0], x[1]$ and $x[2]$. It can be observed that in this example, QE cannot be performed by the procedure *QE1_Layers1To3*. We describe two techniques to compute such instances of $\exists x_1 . A$ - *Splitting* and *Model Enumeration*[4].

Splitting is based on the observation that each LMD $2^{k_i} \cdot x_1 \neq t_i$ in A can be equivalently expressed as the disjunction of two constraints - an LMD $(2^k \cdot x_1 \neq 2^{k-k_i} \cdot t_i)$ and a conjunction $((2^k \cdot x_1 = 2^{k-k_i} \cdot t_i) \wedge (2^{k_i} \cdot x_1 \neq t_i))$ where $k_i < k$. This converts A into $A_1 \vee \ldots \vee A_n$ where each A_i is a conjunction of LMCs. $\exists x_1 . A$ is thus equivalent to $\exists x_1 . A_1 \vee \ldots \vee \exists x_1 . A_n$ where each subproblem $\exists x_1 . A_i$ is potentially simpler to compute than the original problem $\exists x_1 . A$. For example, in the previous problem, the LMD $(x \neq b)$ can be split into $(2x \neq 2b) \vee ((2x = 2b) \wedge (x \neq b))$ converting the problem into $\exists x. ((2x = a) \wedge (2x \neq 2b) \wedge (x \neq c)) \vee \exists x. ((2x = a) \wedge (2x = 2b) \wedge (x \neq b) \wedge (x \neq c))$.

Model Enumeration is based on the observation that $\exists x_1 . A$ can be equivalently expressed as $A|_{x_1 \leftarrow 0} \vee \ldots \vee A|_{x_1 \leftarrow 2^p - 1}$ (where $A|_{x_1 \leftarrow i}$ denotes A with x_1 replaced by constant i).

We call (i) the procedure which makes use of *Splitting* and *Model Enumeration* to compute $\exists x_1 . A$ as *QE1_Layer4* and (ii) the procedure which makes use of *QE1_Layer4* to compute $\exists x_1 \cdots \exists x_t . A$ as *QE_Layer4*.

We present in Fig. 3 the algorithm *QE_LMC* which computes $\exists x_1 \cdots \exists x_t . A$ using *QE1_Layers1To3* and *QE_Layer4*. *QE_LMC* initially tries to eliminate the quantified variables $x_1, \ldots, x_t$ one by one by applying the cheaper procedure *QE1_Layers1To3*. The variables which cannot be eliminated by *QE1_Layers1To3* are collected in a set Y. It can be observed that after the **for** loop in *QE_LMC*, $\exists x_1 \cdots \exists x_t . A$ can be equivalently expressed as $\varphi_1 \wedge \exists Y . \varphi_2$ where φ_1 and φ_2 are conjunctions of LMCs (using a procedure *scopeReduce* in Fig. 3). $\exists Y . \varphi_2$ is computed by *QE_Layer4* which is conjoined with φ_1 to obtain the final result.

[4] For all the benchmarks we have experimented with, *Splitting* and *Model Enumeration* were never required to eliminate quantifiers. Hence they are only briefly described here.

QE_Layer4 computes the result as a disjunction of conjunctions of LMCs. Hence the result here is, in general a Boolean combination of LMCs.

QE1_Layers1To3(A, x_1)
$\psi_1 \wedge \exists x_1.\, \psi_2 \leftarrow$ *QE1_Layer1*$(A,\ x_1)$;
if (ψ_2 is free of LMDs)
  return ($\psi_1 \wedge$ *QE1_1LME*(ψ_2, x_1));
else
  $e \leftarrow$ LME in ψ_2;
  $\psi_{D,2} \leftarrow$ set of LMDs in ψ_2;
  if(*DropLMDSimple*$(\{e\}, \psi_{D,2}, x_1) = \{e\}$)
    return ($\psi_1 \wedge$ *QE1_1LME*(e, x_1));
  else
    $\psi'_2 \leftarrow$ *QE1_Layer3*$(\{e\}, \psi_{D,2}, x_1)$;
    if$(\psi'_2 = e)$
      return ($\psi_1 \wedge$ *QE1_1LME*(e, x_1));
    else return ($\psi_1 \wedge \exists x_1.\, \psi'_2$);

QE_LMC(A, $\{x_1, \ldots, x_t\}$)
$Y \leftarrow \{\}$;
for each $x_i \in \{x_1, \ldots, x_t\}$
  $A' \leftarrow$ *QE1_Layers1To3*$(A,\ x_i)$;
  if(A' is free of x_i)
    $A \leftarrow A'$;
  else /*$A' \equiv \psi_1 \wedge \exists x_1.\, \psi'_2$*/
    $A \leftarrow \psi_1 \wedge \psi'_2$;
    $Y \leftarrow Y \cup \{x_i\}$;
$\varphi_1 \wedge \exists Y.\, \varphi_2 \leftarrow$ *scopeReduce*(A, Y);
$\varphi'_2 \leftarrow$ *QE_Layer4*$(\varphi_2,\ Y)$;
return $\varphi_1 \wedge\ \varphi'_2$;

Fig. 3. Procedures *QE1_Layers1To3* and *QE_LMC*

3 Boolean Combinations of LMCs

The QE algorithm *QE_LMC* accepts a conjunction of LMCs. In this section, we explore two approaches for extending *QE_LMC* to Boolean combinations of LMCs: *Decision Diagram (DD)* based approach and *DAG* based (*SMT solving* based) approach.

3.1 Quantifier Elimination by DD Based Approach

We introduce a data structure called Linear Modular Decision Diagram (LMDD) which represents Boolean combinations of LMCs. They are BDDs [4] with nodes labeled with LMEs. The problem we wish to solve in this subsection can be formally stated as follows. Given an LMDD f representing a Boolean combination of LMCs over a set of variables X, we wish to compute an LMDD $g \equiv \exists V.\, f$ where $V \subseteq X$.

The algorithms presented in this subsection use the following procedures. a) *createLMDD* : Creates an LMDD from a DAG representing a Boolean combination of LMCs, b) *isUnsat* : Returns true if the conjunction of LMCs in the given set is unsatisfiable and false otherwise, d) *getConjunct* : Given a set of LMCs φ, returns the conjunction of LMCs in φ, e) *AND*, *OR*, *NOT*, *ITE* : Perform the basic operations on LMDDs indicated by their names. We denote a non-terminal LMDD node f as $(P(f), H(f), L(f))$ where $P(f)$ is an LME, and $H(f), L(f)$ are the high child and low child respectively as defined in [4].

A straightforward procedure to compute $\exists V.\, f$ is to apply *QE_LMC* to each path in f similar to Black-box QE on Linear Decision Diagrams described in [1]. However, as observed in [1], this technique is not amenable to dynamic programming and the number of recursive calls to the procedure is linear in the number of paths in f (which is bad).

In the following text, we present a more efficient procedure *QuaLMoDE* to compute $\exists V.\, f$. *QuaLMoDE* makes use of a procedure *QE1_LMDD* which eliminates a single variable v from f (see Fig. 4). To compute $\exists v.\, f$, we call *QE1_LMDD* with arguments $f, \{\}, \{\}, v$. *QE1_LMDD* performs recursive traversal of f carrying along each path, the set of LMEs E and the set of LMDs D containing v, encountered on the path so far (called the context). However, it tries to simplify f using E in the following way.

When *QE1_LMDD* is called with arguments f, E, D, v, we wish to compute $\exists v.\, (f \wedge C_E \wedge C_D)$, where C_E and C_D denote the conjunctions of LMEs in E and LMDs in D respectively. Using Lemma 1, E can be expressed as $\{(2^{k_1} \cdot v = t_1), \ldots, (2^{k_n} \cdot v = t_n)\}$. Without loss of generality, let k_1 be the minimum among $k_1, \ldots, k_n$. Let g be an internal non-terminal node of f denoted as $(P(g), H(g), L(g))$ with $P(g)$ expressed as $(2^k \cdot v = t)$ such that $k \geq k_1$. It can be observed that g can be simplified to $\big((2^{k-k_1} \cdot t_1 = t), H(g), L(g)\big)$ using the LME $(2^{k_1} \cdot v = t_1)$. The procedures *selectLME* and *simplifyLMDD* (see Fig. 4) respectively perform the selection of LME with the minimum k among the LMEs in E and simplification of f using the selected LME e_1 as described above. The procedure *applyL1* in Fig. 4 returns an LME equivalent to the argument LME using Lemma 1.

It can be observed that *simplifyLMDD* does not require propagation of the context. If the same LMDD node is encountered with the same LME following two different paths, the results of the calls are the same. Hence *simplifyLMDD* can be implemented with dynamic programming.

Note that if *simplifyLMDD* is successful in eliminating all occurrences of variable v using the LME selected, *QE1_LMDD* returns without any further recursive calls. The procedure *QE1_LMDD* can be repeatedly invoked to compute $\exists V.\, f$. This is implemented in the procedure *QuaLMoDE*.

3.2 Quantifier Elimination by DAG Based Approach

The problem we wish to solve in this subsection is the following. Given a DAG f representing a Boolean combination of LMCs over a set of variables X, we wish to compute a DAG $g \equiv \exists V.\, f$ where $V \subseteq X$.

We present an algorithm *Monniaux* to compute $\exists V.\, f$ which is a simple extension of the algorithm EXISTELIM in [2]. EXISTELIM as given in [2] computes $\exists V.\, f$ where f is a Boolean combination of linear inequalities over reals. A naive way of computing this is by converting f to DNF by enumerating all satisfying assignments, and by using a QE technique for conjunctions of linear inequalities. EXISTELIM improves upon this by generalizing a satisfying assignment to obtain a cube of satisfying assignments, and by projecting the cube on the remaining variables (not in V) before its complement is conjoined with f and further satisfying assignments are found.

Our algorithm *Monniaux* is an extension of the algorithm EXISTELIM with the following changes. a) The predicates are LMCs, not linear inequalities over reals, b) the projection algorithm PROJECT [2] is replaced by *QE_LMC*, and c) the algorithm GENERALIZE2 [2] for generalization of conjunctions is replaced by an algorithm *GENERALIZE2_LMC*.

Given a formula G and a conjunction M of literals of G such that $M \Rightarrow \neg G$, the algorithm GENERALIZE2 removes unnecessary literals from M and returns

```
QE1_LMDD(f, E, D, v)
 if (f = 0 ∨ isUnsat(E ∪ D))
   return 0;
if (f = 1)
   return createLMDD(QE_LMC
   (getConjunct(E ∪ D), {v}));
if (E ≠ φ)
   e1 ← selectLME(E);
   f' ← simplifyLMDD(f, v, e1);
   if (f' is free of v)
     return AND(f', createLMDD
     (QE_LMC(getConjunct(E ∪ D), {v})));
else
   f' ← f;
e ← P(f');
if (e is free of v)
   return ITE(e, QE1_LMDD(H(f'), E, D, v),
   QE1_LMDD(L(f'), E, D, v));
else
   return OR
   (QE1_LMDD(H(f'), E ∪ {e}, D, v),
   QE1_LMDD(L(f'), E, D ∪{¬e}, v));
```

```
simplifyLMDD(f, v, e1)
 if (f = 1 or f = 0)
   return f;
e ← P(f);
if (e is free of v)
   return ITE(e,
   simplifyLMDD(H(f), v, e1),
   simplifyLMDD(L(f), v, e1));
else
   (2^k · v = t) ← applyL1(e, v);
   (2^k1 · v = t1) ← applyL1(e1, v);
   if (k ≥ k1)
     return ITE(2^(k−k1) · t1 = t,
     simplifyLMDD(H(f), v, e1),
     simplifyLMDD(L(f), v, e1));
   else
     return ITE(e,
     simplifyLMDD(H(f), v, e1),
     simplifyLMDD(L(f), v, e1));
```

Fig. 4. Algorithms *QE1_LMDD* and *simplifyLMDD*

M' such that $M \Rightarrow M'$ and $M' \Rightarrow \neg G$. However, in our experiments, we have found that GENERALIZE2 is prohibitively time consuming as it involves SMT solver calls equal to the number of literals in M.

Our algorithm *GENERALIZE2_LMC* works in the following way. Note that M assigns a Boolean value to each atomic predicate of the formula $\neg G$. We evaluate the propositional skeleton (DAG representation of the propositional structure) P of $\neg G$ using these Boolean values assigned to the atomic predicates. This assigns a Boolean value b_n to each node n in P. We now find the subset S_n of literals in M which is sufficient to evaluate n to b_n. Let S_r be the set of literals found in this way for the root r of P. Let M' be the conjunction of literals in S_r. It is easy to see that $M \Rightarrow M'$ and $M' \Rightarrow \neg G$. We illustrate this idea with the help of an example. Let $\neg G$ be the formula $ite(A, B, C) \vee ite(D, E, F)$ and let M be $A \wedge B \wedge \neg C \wedge \neg D \wedge \neg E \wedge F$ where A, B, C, D, E and F are LMCs. It is easy to see that the set of literals $\{A, B\}$ is sufficient to evaluate $ite(A, B, C)$ to true. Similarly $\{\neg D, F\}$ is sufficient to evaluate $ite(D, E, F)$ to true. Hence, it follows that $\{A, B\}$ (or $\{\neg D, F\}$) is sufficient to evaluate $\neg G$ to true. Hence *GENERALIZE2_LMC* returns $A \wedge B$ (or $\neg D \wedge F$) as M'.

4 Experimental Results

We conducted three sets of experiments a) to evaluate *QuaLMoDE*, *Monniaux* and *QE_LMC*, b) to compare the performance of *QE_LMC* with alternative QE techniques and c) to evaluate the utility of our QE algorithms in verification.

The experiments are performed on a 1.83 GHz Intel(R) Core 2 Duo machine with 2GB memory running Ubuntu 8.04. We have implemented our own LMDD package for carrying out the QE experiments by DD based approach. In our implementation, we convert LMDs with modulus 2 to equivalent LMEs as a simplification step. Hence, in this section "LMD" refers to LMD with modulus greater than 2.

Evaluation of *QuaLMoDE*, *Monniaux* and *QE_LMC*: In order to evaluate *QuaLMoDE* and *Monniaux*, we used a benchmark suite consisting of 210 *real* benchmarks and 212 *artificial* benchmarks. The *real* benchmarks are derived from real word-level VHDL designs. We created these benchmarks by obtaining the transition relations (R) of these designs and then (i) computing abstract transition relation by quantifying out the internal variables of the design from R, (ii) computing the set of states reachable in 2^i steps using iterative squaring.

We observed a significant number of LMDs in these benchmarks when expressed in Negation Normal Form (NNF) (see Fig. 5(a)). In order to generate the *artificial* benchmarks, we selected some of the *real* benchmarks and some SMTLib benchmarks from the category QF_BV/bruttomesso/simple_processor/ of the SMTLib fixed size bit-vector benchmarks [10] and used different random choices for the set of variables to be eliminated[5]. The number of variables (N), number of variables to be eliminated (E) and the number of bits to be eliminated in the benchmarks range from 3 to 175, 1 to 170 and 1 to 1265 respectively.

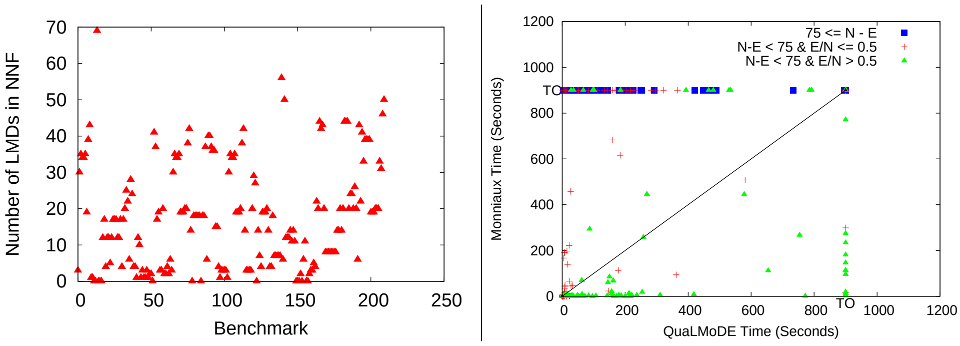


Fig. 5. Plots showing (a) significant number of LMDs in the *real* benchmarks. (b) *QuaLMoDE* Time Vs *Monniaux* Time (TO : $>$ 900 seconds)

We measured the QE time by *QuaLMoDE* and *Monniaux* for each benchmark (For *QuaLMoDE*, this includes the time taken to build the LMDD also). It is observed that (see Fig. 5(b)), for the benchmarks with $N - E$ below a certain threshold t_1 and E/N above a certain threshold t_2, *Monniaux* performs better than *QuaLMoDE* in most of the cases (For our benchmark suite, t_1 and t_2 were empirically estimated as 75 and 0.5 respectively). For the other benchmarks, *QuaLMoDE* outperforms *Monniaux*. It is also observed that, for the benchmarks with $t_1 \leq N - E$, *Monniaux* times out irrespective of E/N. We figured out that this is due to the following reasons. (i) For the benchmarks with low $N - E$ and high E/N, the interleaving of projection inside model enumeration

[5] The SMTLib benchmarks contain bit-vector operators like selection and concatenation which our work does not address. We introduced a fresh variable to denote the result of each such operator.

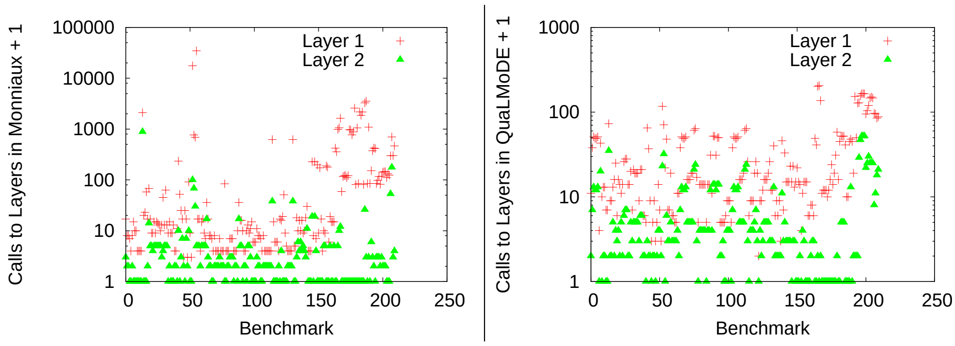


Fig. 6. Contribution of the layers in *QE_LMC*

in *Monniaux* simplifies the problem considerably whereas for the other benchmarks this simplification is not substantial. (ii) The single variable elimination strategy in *QuaLMoDE* results in more calls to *QE1_LMDD* from *QuaLMoDE* for benchmarks with low $N - E$ and high E/N.

The number of calls to *QE_LMC* from *QuaLMoDE* and *Monniaux* while performing QE from the *real* benchmarks ranges from 1 to 205 and 1 to 3842 respectively. We observed that a considerable number of these calls contain LMDs. The average number of LMDs in these *QE_LMC* calls from *QuaLMoDE* and *Monniaux* ranges from 0 to 12.2 and 0 to 18.8 respectively and the average of the ratio of the number of LMEs to the number of LMDs ranges from 0 to 1 and 0.19 to 23.4 respectively.

We evaluated the contribution of different layers of *QE_LMC* in performing QE from the *real* benchmarks. It was observed that all the quantifiers were eliminated by the first two layers - without even a single call to *QE1_Layer3* or *QE_Layer4*. A large fraction of the calls to *QE1_Layers1To3* were solved by the first layer itself and the remaining by the second layer (see Fig. 6)[6].

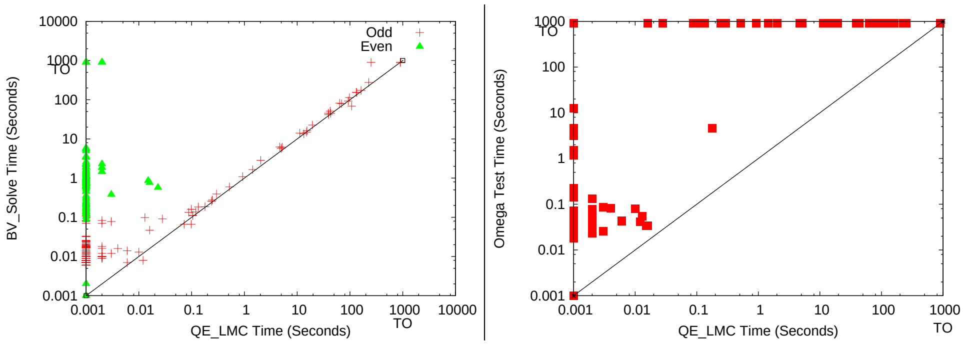


Fig. 7. Plots comparing (a) *QE_LMC* with *BV_Solve* (b) *QE_LMC* with Omega Test (TO : > 900 seconds)

Comparison of *QE_LMC* with alternative QE techniques : We compared the performance of *QE_LMC* with QE based on Presburger Arithmetic using Omega Test and QE based on bit-blasting (see Fig. 7). In the latter case, we have used a procedure *BV_Solve* which performs the elimination of quanti-

[6] Note that the y-axis of both the plots are in log-scale. One is added to the y-values to include the points with no calls to the second layer.

fied variables appearing with odd coefficients in LMEs using the ideas described in [6] followed by bit-blasting and bit-level QE using [11]. We used a set of 405 benchmarks which are instances of the QE problem at conjunction level - 371 of them arise from *QuaLMoDE/Monniaux* when QE is performed on the *real* benchmarks and 34 are randomly generated. Our results clearly demonstrate that *QE_LMC* outperforms these alternative QE techniques. In Fig. 7(a), a benchmark is labeled "Odd" if each quantified variable in it appears with odd coefficient in at least one LME and "Even" otherwise. Our results demonstrate that *BV_Solve* performs comparable to *QE_LMC* for the "Odd" benchmarks, but not for the "Even" ones. This is not surprising; since *BV_Solve* uses the technique from [6] to eliminate variables whenever possible before bit-blasting, it is able to eliminate variables without any bit-blasting for all "Odd" benchmarks. In contrast, *BV_Solve* has to bit-blast for "Even" benchmarks, thereby performing poorly.

Utility of our QE algorithms in verification : In order to evaluate the utility of our QE algorithms, we used *QuaLMoDE* to compute abstract transition relations when doing BMC of word-level VHDL designs. We derive the transition relation R of the design and then for each BMC frame i, we obtain a slice R_i of R containing only relevant part of R for this frame. Next we eliminate a chosen subset of variables (subset of internal variables) from R_i to obtain R'_i using *QuaLMoDE* as well as *QBV_Solve* (an extension of *BV_Solve* using the DD based approach). The final unrolled constraint is a conjunction of the different R'_is computed by *QuaLMoDE/QBV_Solve* which after conjoining with the negation of the property is given to an SMT solver for proving/refuting. The SMT solver used is simplifyingSTP [12][7]. Table 1 gives a summary of these results. The designs machine_1 to machine_12 are modified versions of publicly available benchmarks obtained from [9]. The remaining designs are proprietary and were obtained from safety critical applications used in nuclear reactors. They are control-oriented designs with wide data paths. Our results clearly demonstrate (i) the significant performance benefit of using abstract transition relations computed by *QuaLMoDE* in these verification exercises and (ii) the performance upper hand of *QuaLMoDE* over *QBV_Solve* in computing the abstract transition relations particularly for designs involving constant multiplications with even coefficients and large bit widths.

Our QE algorithms can be used for solving Boolean combinations of LMCs by quantifying out all the variables. However our preliminary experiments suggest that this approach is not competitive with DPLL-style SMT solvers or bit-blasting followed by QBF solving.

5 Conclusion

In this paper, we tackled the QE problem for LMCs. Our main contributions are : (i) A bit-blasting-free QE algorithm for conjunctions of LMCs which is later extended to QE algorithm for Boolean combination of LMCs, (ii) comparison of

[7] We selected simplifyingSTP because (i) it is the winner of SMT-COMP 2010 bit-vector category and (ii) it has a variable eliminator implemented as per [6].

Table 1. Experimental Results on VHDL Programs

Design	LOC	SS	TR	UNR=500		
				NA	QL	QB
machine_1	363	8	(371, 20, 547)	TO(TO)	98(4, 27)	TO(TO, -)
machine_2	373	6	(371, 19, 341)	TO(TO)	70(2, 0)	TO(TO, -)
machine_3	383	7	(395, 22, 344)	TO(TO)	75(3, 3)	TO(TO, -)
machine_4	253	4	(235, 19, 515)	1497(1418)	79(1, 0)	TO(TO, -)
machine_5	253	4	(235, 19, 387)	1527(1451)	76(1, 0)	TO(TO, -)
machine_6	363	4	(242, 15, 56)	122(80)	41(0, 0)	52(2, 3)
machine_7	379	5	(270, 20, 61)	206(152)	52(3, 1)	66(3, 5)
machine_8	251	2	(170, 13, 83)	225(195)	30(1, 1)	35(4, 1)
machine_9	251	3	(170, 13, 323)	TO(TO)	30(1, 1)	53(28, 1)
machine_10	363	5	(242, 15, 356)	TO(TO)	40(1, 0)	63(13, 3)
machine_11	363	6	(352, 22, 96)	TO(TO)	97(1, 7)	98(2, 24)
machine_12	363	5	(242, 15, 356)	TO(TO)	478(8, 427)	TO(TO, -)
board_1	404	4	(265, 13, 163)	1455(1426)	51(24, 0)	TO(TO, -)
board_2	373	3	(283, 13, 163)	TO(TO)	66(49, 0)	TO(TO, -)
board_3	503	4	(284, 13, 190)	TO(TO)	67(44, 0)	TO(TO, -)
board_4	415	3	(272, 11, 31)	362(229)	111(10, 3)	215(104, 13)

All times are in seconds. **TO** : > 1800 seconds, **LOC** : Lines of code, **SS** : Symbolic simulation time, **TR** : Transition relation details (dag size, number of variables, number of bits), **NA** : Without abstraction : total time (simplifyingSTP time), **QL** : With *QuaLMoDE* for abstraction : total time (*QuaLMoDE* time, simplifyingSTP time), **QB** : With *QBV_Solve* for abstraction : total time (*QBV_Solve* time, simplifyingSTP time) (for **NA**, **QL** and **QB** most of the remaining time is spent in slicing - we use a naive implementation of slicer), **UNR** : Number of BMC unrollings

our approach with alternative techniques and the identification of a simple-to-use criteria for choosing the right QE approach for a given problem instance. We propose to study QE from linear modular inequalities as part of future work.

Acknowledgements : We would like to thank Trevor Hansen and Vijay Ganesh for providing us with the latest version of simplifyingSTP. We also thank Mukesh Sharma, Ashutosh Kulkarni, Rajkumar Gajavelly and Nachiket Vaidya for their valuable support. We convey our special acknowledgement to Anup Bhattacharjee and S.D.Dhodapkar for their indispensable help and support.

References

1. S. Chaki, A. Gurfinkel, O. Strichman. *Decision diagrams for linear arithmetic*, In FMCAD 2009
2. D. Monniaux. *A quantifier elimination algorithm for linear real arithmetic*, In LPAR 2008
3. D. Kroening, O. Strichman. *Decision procedures : an algorithmic point of view*, Texts In Theoretical Computer Science, Springer 2008
4. R.E. Bryant. *Graph-based algorithms for boolean function manipulation.* IEEE Transactions on Computers, C-35(8):677-691, 1986
5. W. Pugh. *The Omega Test : a fast and practical integer programming algorithm for dependence analysis.* Communications of the ACM, Pages 102-114, 1992
6. V. Ganesh, D. Dill. *A decision procedure for bit-vectors and arrays*, In CAV 2007
7. H. Jain, E. M. Clarke, O. Grumberg. *Efficient Craig interpolation for linear diophantine (dis)equations and linear modular equations*, In CAV 2008
8. V. Ganesh, S. Berezin, D. Dill. *Deciding Presburger arithmetic by model checking and comparisons with other methods*, In FMCAD 2002
9. ITC'99 benchmarks, http://www.cad.polito.it/downloads/tools/itc99.html
10. SMTLib website, http://goedel.cs.uiowa.edu/smtlib/
11. CUDD release 2.4.2 website, vlsi.colorado.edu/∼fabio/CUDD
12. STP website, http://sites.google.com/site/stpfastprover/
13. http://www.cse.iitb.ac.in/∼supratik/software/qualmode