

# A Quantifier Elimination Algorithm for Linear Modular Equations and Disequations <sup>★</sup>

Ajith K John<sup>1</sup> and Supratik Chakraborty<sup>2</sup>

<sup>1</sup> Homi Bhabha National Institute, BARC, Mumbai, India

<sup>2</sup> 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 (henceforth called 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  $Q_1y_1 Q_2y_2 \dots Q_my_m. A$ , where  $Y = \{y_1, \dots, y_m\}$  is a subset of  $X$ , and  $Q_i \in \{\exists, \forall\}$  for  $i \in \{1, \dots, m\}$ . QE computes a quantifier-free formula  $A'$  with free variables in  $X \setminus Y$  such that  $A' \equiv_{\mathcal{T}} Q_1y_1 Q_2y_2 \dots Q_my_m. A$ , 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

---

<sup>★</sup> This work was supported by a research grant from Board of Research in Nuclear Sciences, India.

formula to appear 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 bits, QE using bit-blasting might 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 like 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.

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  $c_1 \cdot x_1 + \dots + c_n \cdot x_n = c_0 \pmod{2^p}$  where  $p$  is a positive integer constant,  $x_1, \dots, x_n$  are  $p$ -bit non-negative integer variables, and  $c_0, \dots, c_n$  are integer constants in  $\{0, \dots, 2^p - 1\}$ . Similarly, a Linear Modular Disequation (LMD) is a disequation of the form  $c_1 \cdot x_1 + \dots + c_n \cdot x_n \neq c_0 \pmod{2^p}$ . Conventionally,  $2^p$  is called the modulus of the LME or LMD. For notational convenience, we will henceforth use “LMC” to refer to a Linear Modular Constraint, i.e. an LME or LMD. Since every variable in an LMC  $c_1 \cdot x_1 + \dots + c_n \cdot x_n \bowtie c_0 \pmod{2^p}$ , 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  $\lambda_1 \equiv c_1 \cdot x_1 + \dots + c_n \cdot x_n \bowtie c_0 \pmod{2^p}$  and  $\lambda_2 \equiv 2^q \cdot c_1 \cdot x'_1 + \dots + 2^q \cdot c_n \cdot x'_n \bowtie 2^q \cdot c_0 \pmod{2^{p+q}}$  are related in the following way: every solution of  $\lambda_1$  can be bit-extended to give a solution of  $\lambda_2$ , and every solution of  $\lambda_2$  can be bit-truncated to give a solution of  $\lambda_1$ . Hence, using  $\lambda_2$  in place of  $\lambda_1$  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 for studying QE of LMCs 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 in the right half of the figure. The thick shaded arrows and the thin solid arrows 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  $0x00$ <sup>3</sup>, in which case  $A$  loads itself with an input value from  $InA$  and the controller switches to state 1. In state 1,  $A$  works as an up-counter until it reaches  $0xff$ , in which case it loads the value

<sup>3</sup> We use the 0x prefix to denote hexadecimal values.

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.

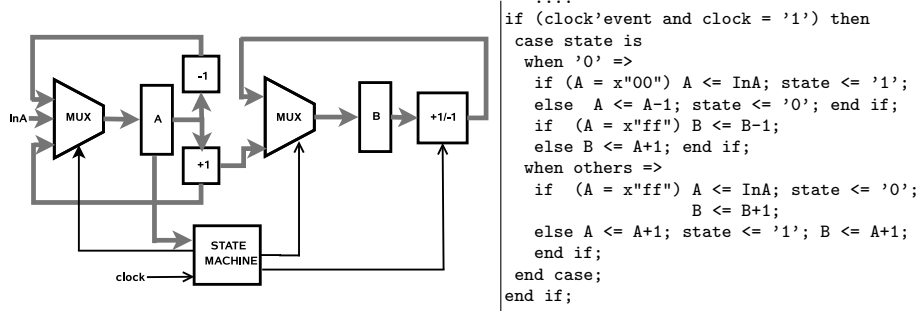


Fig. 1. An Example Circuit

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  $2^8$ .

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

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:

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

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

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 0x00 in  $\text{B}$ , 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 of 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 an algorithm for eliminating quantifiers from 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, converting the results obtained as PA formulae back

to Boolean combinations of LMCs is often 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 work 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 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  $x_1, \dots, x_n$ , 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  $x_1, \dots, x_n$  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.

**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  $0 \leq k_1 \leq p - 1$ .*

*Proof.* Consider an LME  $c_1 \cdot x_1 + \cdots + c_n \cdot x_n = c_0$ . Using the idea described in [6],  $c_1$  can be expressed as  $2^{k_1} \cdot d_1$ , where  $k_1$  is an integer such that  $0 \leq k_1 \leq p - 1$  and  $d_1$  is an odd number in  $\{1, \dots, 2^p - 1\}$ . Hence, the LME  $c_1 \cdot x_1 + \cdots + c_n \cdot x_n = c_0$  can be equivalently expressed as  $2^{k_1} \cdot d_1 \cdot x_1 + \cdots + c_n \cdot x_n = c_0$ . Rearranging terms, we get  $2^{k_1} \cdot d_1 \cdot x_1 = e_2 \cdot x_2 + \cdots + e_n \cdot x_n + c_0$ , where  $e_2, \dots, e_n$  are additive inverses modulo  $2^p$  of  $c_2, \dots, c_n$  respectively. Since  $d_1$  is odd, it has a multiplicative inverse modulo  $2^p$ , say  $d'_1$ . Multiplying both sides of the above LME by  $d'_1$ , we get the equivalent LME  $2^{k_1} \cdot x_1 = e_2 \cdot d'_1 \cdot x_2 + \cdots + e_n \cdot d'_1 \cdot x_n + c_0 \cdot d'_1$ . Putting all the transformations together, the LME  $c_1 \cdot x_1 + \cdots + c_n \cdot x_n = c_0$  can be equivalently expressed as  $2^{k_1} \cdot x_1 = t_1$ , where  $t_1$  is a term free of  $x_1$ . Since an LMD  $c_1 \cdot x_1 + \cdots + c_n \cdot x_n \neq c_0$  is equivalent to the negation of the LME  $c_1 \cdot x_1 + \cdots + c_n \cdot x_n = c_0$ , it is easy to see that it can be equivalently expressed as  $2^{k_1} \cdot x_1 \neq t_1$  where  $t_1$  is a term free of  $x_1$  and  $k_1$  is an integer such that  $0 \leq k_1 \leq p - 1$ .  $\square$

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

Henceforth whenever we express an LMC as  $2^{k_i} \cdot x_1 \bowtie t_i \pmod{2^p}$  where  $\bowtie \in \{=, \neq\}$ , it will be implicitly understood that “ $t_i$  is a term free of  $x_1$  and  $k_i$  is an integer such that  $0 \leq k_i \leq p - 1$ ”. Lemma 1 ensures that there is no loss of generality in doing this.

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

*Proof.* Let  $\varphi_1$  and  $\varphi_2$  denote the formulas  $\exists x_1. (2^{k_1} \cdot x_1 = t_1)$  and  $2^{p-k_1} \cdot t_1 = 0$  respectively. To see that  $\varphi_1 \Rightarrow \varphi_2$ , we simply multiply both sides of  $2^{k_1} \cdot x_1 = t_1$  by  $2^{p-k_1}$ , and simplify modulo  $2^p$ . To see why  $\varphi_2 \Rightarrow \varphi_1$ , recall that  $t_1$  is free of  $x_1$ . Let  $t_1$  depends on variables  $x_2, \dots, x_n$ . Given values of  $x_2, \dots, x_n$  such that the least significant  $k_1$  bits of  $t_1$  evaluate to zero, we can always find a value of  $x_1$  such that  $2^{k_1} \cdot x_1 = t_1$ . This can be done by choosing the least significant  $p - k_1$  bits of  $x_1$  to be the same as the most significant  $p - k_1$  bits of  $t_1$ . Hence,  $\varphi_2 \Rightarrow \varphi_1$ , and therefore  $\varphi_1 \equiv \varphi_2$ .  $\square$

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

**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  $x_1$ .

*Proof.* Given,  $\exists x_1. A$  is equivalent to  $\exists x_1. ((2^{k_1} \cdot x_1 = t_1) \wedge \dots \wedge (2^{k_m} \cdot x_1 = t_m))$ , where  $0 \leq k_1, \dots, k_m \leq p - 1$ . Without loss of generality, let  $k_1$  be the minimum among  $k_1, \dots, k_m$  and let  $l_i = k_i - k_1$  for  $i \in \{2, \dots, m\}$ . Then the above formula can be written as  $\exists x_1. ((2^{k_1} \cdot x_1 = t_1) \wedge (2^{l_2} \cdot 2^{k_1} \cdot x_1 = t_2) \wedge \dots \wedge (2^{l_m} \cdot 2^{k_1} \cdot x_1 = t_m))$ . This, in turn, is equivalent to  $(\exists x_1. (2^{k_1} \cdot x_1 = t_1)) \wedge (2^{l_2} \cdot t_1 = t_2) \wedge \dots \wedge (2^{l_m} \cdot t_1 = t_m)$ . Using Lemma 2, it follows that  $\exists x_1. A$  is equivalent to  $(2^{p-k_1} \cdot t_1 = 0) \wedge (2^{l_2} \cdot t_1 = t_2) \wedge \dots \wedge (2^{l_m} \cdot t_1 = t_m)$ .  $\square$

**Example:** All LMCs in this example have modulus 8. Consider the problem of computing  $\exists y. ((2^1y = 5x + 2) \wedge (2^2y = 5x + 6z) \wedge (2^1y = 2x + 4))$ . 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  $\psi_1$  is a conjunction of LMCs independent of  $x_1$ , and  $\psi_2$  is a conjunction of LMCs and at most one LME i.e.,  $2^{k_1} \cdot x_1 = t_1$ . In addition,  $\psi_2$  contains only those LMCs from  $A$  in which the coefficient of  $x_1$  is of the form  $2^{k_i}$ , where  $k_i < k_1$ .

*Proof.* Without loss of generality, let us assume that  $A$  is composed of  $m$  LMEs and  $r-m$  LMDs.  $\exists x_1. A$  is thus equivalent to  $\exists x_1. ((2^{k_1} \cdot x_1 = t_1) \wedge \dots \wedge (2^{k_m} \cdot x_1 = t_m) \wedge (2^{k_{m+1}} \cdot x_1 \neq t_{m+1}) \wedge \dots \wedge (2^{k_r} \cdot x_1 \neq t_r))$ , where  $0 \leq k_1, \dots, k_r \leq p-1$ . Without loss of generality, let  $k_1$  be the minimum of  $k_1, \dots, k_m$ . In addition, let the first  $q$  LMDs be such that their corresponding  $k_i$ 's are at least as large as  $k_1$ . In other words, let  $q \in \{0, \dots, r-m\}$  be such that  $k_1 \leq k_i$  for  $m+1 \leq i \leq m+q$  and  $k_1 > k_i$  for  $m+q+1 \leq i \leq r$ . Since  $k_1 \leq k_i$  for  $2 \leq i \leq m+q$ , the LME  $2^{k_1} \cdot x_1 = t_1$  can be used to eliminate  $x_1$  from the other LMEs and the first  $q$  LMDs following the same reasoning used in the proof of Lemma 3. Hence  $\exists x_1. A$  can be seen to be equivalent to  $\psi_1 \wedge \exists x_1. \psi_2$ , where

$$\begin{aligned} - \psi_1 &\equiv (2^{l_2} \cdot t_1 = t_2) \wedge \dots \wedge (2^{l_m} \cdot t_1 = t_m) \wedge (2^{l_{m+1}} \cdot t_1 \neq t_{m+1}) \wedge \dots \wedge \\ &\quad (2^{l_{m+q}} \cdot t_1 \neq t_{m+q}), \text{ where each } l_i = k_i - k_1. \\ - \psi_2 &\equiv (2^{k_1} \cdot x_1 = t_1) \wedge (2^{k_{m+q+1}} \cdot x_1 \neq t_{m+q+1}) \wedge \dots \wedge (2^{k_r} \cdot x_1 \neq t_r). \end{aligned}$$

□

**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  $\psi_1$  and  $\psi_2$  here are  $(5x + 2z = 4) \wedge (3x \neq 2)$  and  $(2y = 5x + 2) \wedge (y \neq 6x + 7z)$  respectively.

For the remainder of the paper, we adopt the convention that algorithms for eliminating a single variable will have names starting with “*QE1*.”, while those for eliminating multiple variables will have names starting with “*QE*”.

Lemmas 1 through 4 yield two simple algorithms: (a) *QE1-LME* 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  $\psi_1$  and  $\exists x_1. \psi_2$  (as given by Lemma 4). As we will soon see, *QE1-Layer1* forms the core of the first layer of our layered QE algorithm.

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-LME* suffice to compute  $\exists x_1. A$ . In general, however,  $\psi_2$  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  $\lambda$  is an LMD and  $\xi$  is a conjunction of LMCs. We say that  $\lambda$  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  $\psi_2$  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  $\exists x. (\xi \wedge \lambda)$ , 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  $i^{\text{th}}$  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  $\xi$  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  $\xi$ . Any solution of  $\xi$  can therefore be engineered to become 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  $\exists x. (\xi \wedge \lambda)$ .

<pre> <b>DropLMDSimple</b>(<b>E</b>, <b>D</b>, <math>x_1</math>)   core <math>\leftarrow</math> <b>E</b>;   <b>while</b>(core <math>\neq</math> <b>E</b> <math>\cup</math> <b>D</b>)     <b>if</b> (<i>isExt</i>(core, <b>E</b> <math>\cup</math> <b>D</b>, <math>x_1</math>))       <b>return</b> core;     <b>else</b>       d <math>\leftarrow</math> <i>getLstCnstr</i>(<b>D</b> \ core);       core <math>\leftarrow</math> core <math>\cup</math> d;   <b>return</b> core; </pre>	<pre> <b>DropImpliedLMD</b>(<b>E</b>, <b>D</b>, <math>x_1</math>)   <b>while</b>(true)     impl <math>\leftarrow</math> NULL;     <b>for each</b> LMD d <math>\in</math> <b>D</b>       <b>if</b> (<b>E</b> <math>\cup</math> (<b>D</b> \ d) <math>\Rightarrow</math> d)         impl <math>\leftarrow</math> d; <b>break</b>;     <b>if</b> (impl = NULL)       <b>break</b>;     <b>D</b> <math>\leftarrow</math> <b>D</b> \ impl;   <b>return</b> <b>E</b> <math>\cup</math> <b>D</b>; </pre>
---	--

**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 use the function *getLstCnstr* to identify the LMDs in  $D \setminus core$  whose truth depends on the least number of bits of  $x_1$ . 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  $2^{k_i} \cdot x \neq t_i$  in  $core$ . Let  $D \setminus core$  be expressed as  $\{(2^{k_1} \cdot x \neq t_1), \dots, (2^{k_n} \cdot x \neq t_n)\}$ . If  $2^{k_{core}} - \sum_{i=1}^n 2^{k_i} \geq 1$ , any solution to the conjunction of LMCs in  $core$  can be engineered to give a solution to the conjunction of LMCs in  $E \cup D$ .*

*Proof.* In the remainder of this proof, we use  $x[i : j]$  to denote the extraction of bits  $i$  through  $j$  of bit-vector  $x$  such that  $i \geq j$ ,  $x@y$  to denote the concatenation of bit-vectors  $x$  and  $y$ , and  $val_{[n]}$  to denote a constant bit-vector of size  $n$  which is the binary representation of the decimal value  $val$ .

It can be observed that any LMD  $(2^{k_i} \cdot x_1 \neq t_i) \in D \setminus core$  can be equivalently expressed as a disjunction of two bit-vector disequations as follows.

$$(2^{k_i} \cdot x_1 \neq t_i) \equiv \alpha_i \vee \beta_i \text{ where } \alpha_i \equiv (t_i[p-1 : k_i] \neq x_1[p-k_i-1 : 0]) \\ \text{and } \beta_i \equiv (t_i[k_i-1 : 0] \neq 0_{[k_i]})$$

Let  $C_{D \setminus core}$ ,  $C_{core}$  and  $C_{E \cup D}$  be the conjunctions of LMCs in  $D \setminus core$ ,  $core$  and  $E \cup D$  respectively. Hence,

$$C_{D \setminus core} \equiv \bigwedge_{i=1}^n (\alpha_i \vee \beta_i)$$

It is easy to see that

$$\bigwedge_{i=1}^n \alpha_i \Rightarrow C_{D \setminus core} \quad (1)$$

We claim that if  $2^{k_{core}} - \sum_{i=1}^n 2^{k_i} \geq 1$ , any solution to  $C_{core}$  can be engineered to

give a solution to  $\bigwedge_{i=1}^n \alpha_i$  keeping it as a solution to  $C_{core}$ . This can be proved as follows.

Let us have a closer look at the disequation  $\alpha_i$ .  $\alpha_i$  constrains the least significant  $p - k_i$  bits of  $x_1$ . It is easy to observe that,  $\alpha_i$  can be expressed as a conjunction of  $2^{k_i}$  disequations constraining all the bits of  $x_1$  in the following way.

$$\alpha_i \equiv ((x_1[p-1 : 0] \neq 0_{[k_i]}@t_i[p-1 : k_i]) \wedge (x_1[p-1 : 0] \neq 1_{[k_i]}@t_i[p-1 : k_i]) \\ \wedge \dots \wedge (x_1[p-1 : 0] \neq (2^{k_i} - 1)_{[k_i]}@t_i[p-1 : k_i]))$$

Hence,  $\bigwedge_{i=1}^n \alpha_i$  can be equivalently expressed as the conjunction of  $\sum_{i=1}^n 2^{k_i}$  disequations each of which constrains all the bits of  $x_1$ . Each of these disequations can be considered as a constraint which prevents us from assigning values to  $x_1[p-1:0]$ . It can be seen that  $\sum_{i=1}^n 2^{k_i}$  is an over-approximation of the number of ways of assigning values to  $x_1[p-1:0]$  prevented by  $\bigwedge_{i=1}^n \alpha_i$ .

Now, consider any solution say,  $\pi$  to  $C_{core}$ .  $\pi$  constrains only the least significant  $p - k_{core}$  bits of  $x_1$ . Hence, there are  $2^{k_{core}}$  values any of which can be assigned to  $x_1[p-1:p-k_{core}]$  keeping  $\pi$  as a solution to  $C_{core}$ . In other words, there are  $2^{k_{core}}$  ways to engineer  $\pi$  such that it constrains  $x_1[p-1:0]$  and remains as a solution to  $C_{core}$ . Recall that  $\sum_{i=1}^n 2^{k_i}$  is an over-approximation of the number of ways of assigning values to  $x_1[p-1:0]$  prevented by  $\bigwedge_{i=1}^n \alpha_i$ . Hence  $2^{k_{core}} - \sum_{i=1}^n 2^{k_i}$  is an under-approximation of the number of ways to engineer  $\pi$  such that  $\pi$  constrains  $x_1[p-1:0]$ , remains as a solution to  $C_{core}$  and becomes a solution to  $\bigwedge_{i=1}^n \alpha_i$ . Hence, if  $2^{k_{core}} - \sum_{i=1}^n 2^{k_i} \geq 1$ , there exists at least one way of assigning values to  $x_1[p-1:p-k_{core}]$  keeping  $\pi$  as a solution to  $C_{core}$  and making it a solution to  $\bigwedge_{i=1}^n \alpha_i$ . Thus, our claim that if  $2^{k_{core}} - \sum_{i=1}^n 2^{k_i} \geq 1$ , any solution to  $C_{core}$  can be engineered to give a solution to  $\bigwedge_{i=1}^n \alpha_i$  keeping it as a solution to  $C_{core}$  holds.

According to (1), any solution to  $\bigwedge_{i=1}^n \alpha_i$  is a solution to  $C_{D \setminus core}$ . Hence, if  $2^{k_{core}} - \sum_{i=1}^n 2^{k_i} \geq 1$ , any solution to  $C_{core}$  can be engineered to give a solution to  $C_{D \setminus core}$  keeping it as a solution to  $C_{core}$ . Since  $core \cup (D \setminus core) = E \cup D$  (note that  $E \subseteq core$ ), it is easy to observe that, if  $2^{k_{core}} - \sum_{i=1}^n 2^{k_i} \geq 1$ , any solution to  $C_{core}$  can be engineered to give a solution to  $C_{E \cup D}$ .  $\square$

*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. Here *core* is  $\{2x = y\}$ ,  $k_{core} = 1$ ,  $k_1 = k_2 = 0$ . Therefore,  $\eta = 0$  and *DropLMDSimple* concludes that it is not possible 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 simplify  $\exists x. ((2x = y) \wedge (x \neq 2y))$  to  $\exists x. (2x = y)$ . Based on this idea, we present an algorithm to drop implied LMDs called *DropImpliedLMD* (see Fig. 2). The notation used in this algorithm

is the same as that used in algorithm *DropLMDSimple*. The implication check in *DropImpliedLMD* 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 *DropImpliedLMD*. Given  $\exists x_1. \psi_2$ , *QE1\_Layer3* initially invokes *DropLMDSimple* to drop unconstraining LMDs. If one or more LMDs remain, *DropImpliedLMD* is invoked to identify the implied LMDs and drop them. If there exist LMDs in the output of *DropImpliedLMD*, we invoke *DropLMDSimple* once again. Thus finally, we are left with a conjunction of LMCs  $\psi'_2$  with possibly fewer LMDs vis-a-vis to  $\psi_2$ , 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. 5 a procedure *QE1\_Layers1To3* that 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  $\psi_2$  is free of LMDs, *QE1\_LME* is called to compute  $\exists x_1. \psi_2$ ; hence  $\exists x_1. A$  gets computed by the first layer itself. If  $\psi_2$  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* (or *QE1\_Layer3*),  $\exists x_1. A$  gets computed in the second (or 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 required 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 those instances of  $\exists x_1. A$  that cannot be computed by *QE1\_Layers1To3*. The difficulty in QE in such cases arises from the fact that there are some bits  $x_1$  constrained by the LMDs but not by any LME. 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 bits  $x[1]$  and  $x[0]$  of  $x$ , whereas the LMDs constrain bits  $x[0], x[1]$  and  $x[2]$ . It can be observed that in this example, QE cannot be performed by *QE1\_Layers1To3*. We now describe two techniques to compute such instances of  $\exists x_1. A$ , namely *Splitting* and *Model Enumeration*.

In the following text, we give the details of the techniques *Splitting* and *Model Enumeration*. We assume that  $A$  is expressed as  $(2^{k_1} \cdot x_1 = t_1) \wedge \dots \wedge (2^{k_m} \cdot x_1 = t_m) \wedge (2^{k_{m+1}} \cdot x_1 \neq t_{m+1}) \wedge \dots \wedge (2^{k_r} \cdot x_1 \neq t_r)$  and  $k_1$  is the minimum among  $k_1, \dots, k_m$ . The procedures presented below make use of the following Lemmas.

**Lemma 6.**  $(2^k \cdot x_1 \neq t_1) \wedge (2^{k+1} \cdot x_1 = 2t_1) \wedge (2^k \cdot x_1 \neq t_2) \wedge (2^{k+1} \cdot x_1 = 2t_2) \equiv (t_1 = t_2) \wedge (2^k \cdot x_1 \neq t_1) \wedge (2^{k+1} \cdot x_1 = 2t_1) \wedge (2^{k+1} \cdot x_1 = 2t_2)$

**Example:** Consider the conjunction of LMCs  $((x \neq b) \wedge (2x = 2b) \wedge (x \neq c) \wedge (2x = 2c))$  where all the LMCs have modulus 8. This can be equivalently expressed as  $((b = c) \wedge (x \neq b) \wedge (2x = 2b) \wedge (2x = 2c))$ .

**Lemma 7.**  $2^{k_i} \cdot x_1 \neq t_i \equiv (2^{k_i+1} \cdot x_1 \neq 2t_i) \vee ((2^{k_i+1} \cdot x_1 = 2t_i) \wedge (2^{k_i} \cdot x_1 \neq t_i))$

**Example:** The LMD  $x \neq a$  with modulus 8 can be equivalently expressed as  $(2x \neq 2a) \vee ((2x = 2a) \wedge (x \neq a))$  (here  $k_i = 0$  and  $x_1, t_i$  are  $x, a$  respectively)

**Lemma 8.**  $2^{k_i} \cdot x_1 \neq t_i \equiv (2^k \cdot x_1 \neq 2^{k-k_i} \cdot t_i) \vee ((2^k \cdot x_1 = 2^{k-k_i} \cdot t_i) \wedge (2^{k_i} \cdot x_1 \neq t_i))$  where  $k_i < k$

**Example:** The LMD  $x \neq a$  with modulus 8 can be equivalently expressed as  $(4x \neq 4a) \vee ((4x = 4a) \wedge (x \neq a))$  (here  $k_i = 0$ ,  $k = 2$  and  $x_1, t_i$  are  $x, a$  respectively).

We call the procedure which identifies pairs of constraints of the form  $(2^k \cdot x_1 \neq t_1) \wedge (2^{k+1} \cdot x_1 = 2t_1)$  and  $(2^k \cdot x_1 \neq t_2) \wedge (2^{k+1} \cdot x_1 = 2t_2)$  from the given conjunction of LMCs and replaces them by the equivalent conjunction of LMCs  $(t_1 = t_2) \wedge (2^k \cdot x_1 \neq t_1) \wedge (2^{k+1} \cdot x_1 = 2t_1) \wedge (2^{k+1} \cdot x_1 = 2t_2)$  using Lemma 6 as *SynthesizeLMEs*.

We describe here two procedures *LMEBased\_Split* and *Eager\_Split* based on Lemma 8 and Lemma 7 respectively.

The procedure *LMEBased\_Split* splits each LMD  $2^{k_i} \cdot x_1 \neq t_i$  in  $\exists x_1. A$  such that  $k_i < k_1$  one by one into disjunction of two constraints  $(2^{k_1} \cdot x_1 \neq 2^{k_1-k_i} \cdot t_i)$  and  $(2^{k_1} \cdot x_1 = 2^{k_1-k_i} \cdot t_i) \wedge (2^{k_i} \cdot x_1 \neq t_i)$  using Lemma 8.

#### **QE1\_Layer4(A, x<sub>1</sub>)**

```

A' ← LMEBased_Split(A, x1);
if(A' is free of x1)
  return A';
else
  result ← false;
  /* A' here is in general d1 ∨ ... ∨ dn where
  each di is a conjunction of LMCs */
  for each disjunct di in A'
    if(di is free of x1)
      result ← result ∨ di;
    else
      if(select_Method()=Eager_Split)
        result ← result ∨ Eager_Split(di, x1);
      else
        result ← result ∨ Model_Enumerate(di, x1);
  return result;

```

**Fig. 3.** Procedure *QE1\_Layer4*

Let us see what happens if we split the LMD  $2^{k_{m+1}} \cdot x_1 \neq t_{m+1}$  in  $\exists x_1. A$  this way into disjunction of constraints -  $(2^{k_1} \cdot x_1 \neq 2^{k_1-k_{m+1}} \cdot t_{m+1})$  and  $(2^{k_1} \cdot x_1 = 2^{k_1-k_{m+1}} \cdot t_{m+1}) \wedge (2^{k_{m+1}} \cdot x_1 \neq t_{m+1})$ . It can be observed that the first constraint  $2^{k_1} \cdot x_1 \neq 2^{k_1-k_{m+1}} \cdot t_{m+1}$  constrains the same set of bits as constrained by the LME  $2^{k_1} \cdot x_1 = t_1$  and the second constraint  $(2^{k_1} \cdot x_1 = 2^{k_1-k_{m+1}} \cdot t_{m+1}) \wedge (2^{k_{m+1}} \cdot x_1 \neq t_{m+1})$  constrains the same set of bits as constrained by the original LMD  $2^{k_{m+1}} \cdot x_1 \neq t_{m+1}$  (recall that  $k_{m+1} < k_1$ ). The splitting converts  $\exists x_1. A$  into disjunction of the following subproblems.

```

QE_Layer4(A, { $x_1, \dots, x_t$ })
   $result \leftarrow A$ ;
  for each  $x_i \in \{x_1, \dots, x_t\}$ 
     $result' \leftarrow \text{false}$ ;
    /* $result$  here is in general  $d_1 \vee \dots \vee d_n$  where
    each  $d_j$  is a conjunction of LMCs*/
    for each disjunct  $d_j$  in  $result$ 
       $d'_j \leftarrow QE1\_Layer4(d_j, x_i)$ ;
       $result' \leftarrow result' \vee d'_j$ ;
     $result \leftarrow result'$ ;
  return  $result$ ;

```

**Fig. 4.** Procedure *QE\_Layer4*

- $\exists x_1. ((2^{k_1} \cdot x_1 = t_1) \wedge \dots \wedge (2^{k_m} \cdot x_1 = t_m) \wedge (2^{k_1} \cdot x_1 \neq 2^{k_1 - k_{m+1}} \cdot t_{m+1}) \wedge \dots \wedge (2^{k_r} \cdot x_1 \neq t_r))$
- $\exists x_1. ((2^{k_1} \cdot x_1 = t_1) \wedge \dots \wedge (2^{k_m} \cdot x_1 = t_m) \wedge (2^{k_1} \cdot x_1 = 2^{k_1 - k_{m+1}} \cdot t_{m+1}) \wedge (2^{k_{m+1}} \cdot x_1 \neq t_{m+1}) \wedge \dots \wedge (2^{k_r} \cdot x_1 \neq t_r))$ .

*LMEBased.Split* initially calls *SynthesizeLMEs* and then *QE1\_Layers1To3* to compute these subproblems separately and the disjunction of the results is taken. The splitting of the LMDs is continued on the subproblems that cannot be computed by *SynthesizeLMEs* and *QE1\_Layers1To3*. This process is repeated until either  $\exists x_1. A$  is reduced into disjunction of subproblems each of which is computed by procedures *QE1\_Layers1To3* and *SynthesizeLMEs* or all the LMDs in  $\exists x_1. A$  are split.

As an example, consider the problem  $\exists x. ((2x = a) \wedge (x \neq b) \wedge (x \neq c))$ , where all the LMCs have modulus 8. *LMEBased.Split* splits the first LMD ( $x \neq b$ ) into disjunction of  $(2x \neq 2b)$  and  $((2x = 2b) \wedge (x \neq b))$ . This converts the original problem into disjunction of subproblems  $\exists x. ((2x = a) \wedge (2x \neq 2b) \wedge (x \neq c))$  and  $\exists x. ((2x = a) \wedge (2x = 2b) \wedge (x \neq b) \wedge (x \neq c))$ . The former subproblem is reduced to  $(a \neq 2b) \wedge (4a = 0)$  by *QE1\_Layers1To3*. The latter subproblem requires splitting the LMD ( $x \neq c$ ) resulting in disjunction of subproblems  $\exists x. ((2x = a) \wedge (2x = 2b) \wedge (x \neq b) \wedge (2x \neq 2c))$  and  $\exists x. ((2x = a) \wedge (2x = 2b) \wedge (x \neq b) \wedge (2x = 2c) \wedge (x \neq c))$ . The former subproblem is reduced to  $(a = 2b) \wedge (a \neq 2c) \wedge (4a = 0)$  by *QE1\_Layers1To3*. Consider the latter subproblem. *synthesizeLMEs* converts it to  $\exists x. ((2x = a) \wedge (b = c) \wedge (2x = 2b) \wedge (x \neq b) \wedge (2x = 2c))$ . This is reduced to  $(b = c) \wedge (a = 2b) \wedge (a = 2c) \wedge (4a = 0)$  by *QE1\_Layers1To3*. Hence, the procedure *LMEBased.Split* returns  $((a \neq 2b) \wedge (4a = 0)) \vee ((a = 2b) \wedge (a \neq 2c) \wedge (4a = 0)) \vee ((b = c) \wedge (a = 2b) \wedge (a = 2c) \wedge (4a = 0))$  as the result.

It can be observed that there exist QE problem instances which cannot be computed by *LMEBased.Split*. As an example, consider the problem  $\exists x. ((2x \neq a) \wedge (2x \neq b))$ , where the LMDs have modulus 4. Note that this problem cannot be computed by *LMEBased.Split* (or any of the procedures discussed so far). Our procedures *Eager.Split* and *Model.Enumerate* compute such instances of  $\exists x_1. A$ .

*Eager.Split* splits the LMDs in  $\exists x_1. A$  into a disjunction of conjunctions of LMCs using Lemma 7, until the LMDs in each conjunction either constrain the

same set of bits of  $x_1$  as constrained by the LME  $2^{k_1} \cdot x_1 = t_1$  or constrain only a single bit more than the bits of  $x_1$  constrained by the LMEs conjuncted with them. For example, consider the LMD  $x \neq b$  with modulus 8. By repeated application of Lemma 7,  $x \neq b$  can be converted to disjunction of the conjunctions of LMCs -  $(4x \neq 4b)$ ,  $(4x = 4b) \wedge (2x \neq 2b)$ ,  $(2x = 2b) \wedge (x \neq b)$ . Note that in each of these conjunctions, the LMDs constrain only a single bit more than the bits of  $x$  constrained by the LMEs conjuncted with them. This effectively converts the original problem into a disjunction of subproblems. *Eager\_Split* initially calls *SynthesizeLMEs* and then *QE1\_Layers1To3* to compute these subproblems separately and the disjunction of the results is taken.

Let us see how *Eager\_Split* computes the previous problem  $\exists x. ((2x \neq a) \wedge (2x \neq b))$ , where the LMDs have modulus 4. *Eager\_Split* splits the first LMD ( $2x \neq a$ ) into disjunction of  $(2a \neq 0)$  and  $((2a = 0) \wedge (2x \neq a))$  using Lemma 7 (here  $k_i = 1$  and  $x_1, t_i$  are  $x, a$  respectively). This converts the original problem into disjunction of subproblems  $\exists x. ((2a \neq 0) \wedge (2x \neq b))$  and  $\exists x. ((2a = 0) \wedge (2x \neq a) \wedge (2x \neq b))$ . The former subproblem is reduced to  $(2a \neq 0)$  by *QE1\_Layers1To3*. The latter subproblem requires splitting the LMD ( $2x \neq b$ ) in a similar fashion resulting in disjunction of subproblems  $\exists x. ((2a = 0) \wedge (2b \neq 0) \wedge (2x \neq a))$  and  $\exists x. ((2a = 0) \wedge (2b = 0) \wedge (2x \neq a) \wedge (2x \neq b))$ . The former subproblem is reduced to  $(2a = 0) \wedge (2b \neq 0)$  by *QE1\_Layers1To3*. The latter problem is converted to  $\exists x. ((2a = 0) \wedge (2b = 0) \wedge (a = b) \wedge (2x \neq b))$  by *SynthesizeLMEs*. This is reduced to  $(2a = 0) \wedge (2b = 0) \wedge (a = b)$  by *QE1\_Layers1To3*. Hence, the procedure *Eager\_Split* returns  $(2a \neq 0) \vee ((2a = 0) \wedge (2b \neq 0)) \vee ((2a = 0) \wedge (2b = 0) \wedge (a = b))$  as the result.

*Model Enumeration* is based on the observation that  $\exists x_1. A$  can be equivalently expressed as  $A|_{x_1 \leftarrow 0} \vee \dots \vee A|_{x_1 \leftarrow 2^p - 1}$  (where  $A|_{x_1 \leftarrow i}$  denotes  $A$  with  $x_1$  replaced by constant  $i$ ). Although the number of disjuncts can be reduced by limiting the enumeration to disjuncts that are satisfiable (with the help of an SMT solver), in the worst case it is  $2^p$ . The procedure *Model\_Enumerate* is an implementation of this technique.

It can be observed that *Eager\_Split* and *Model\_Enumerate* are complete. In other words, the QE problem  $\exists x_1. A$  can be solved completely by *Eager\_Split/Model\_Enumerate*. However they are expensive in terms of the number of disjuncts generated. The number of disjuncts *Eager\_Split* generates ( $\eta$ ) can be a-priori determined. We select between *Eager\_Split* and *Model\_Enumerate* depending on the value of  $\eta$  using a function *select\_Method*. *select\_Method* chooses *Eager\_Split* and *Model\_Enumerate* for lower and higher values of  $\eta$  respectively.

We present (see Fig. 3) the procedure *QE1\_Layer4* to compute  $\exists x_1. A$  using the procedures *LMEBased\_Split*, *Eager\_Split* and *Model\_Enumerate*. Given  $\exists x_1. A$ , *QE1\_Layer4* tries to compute it using *LMEBased\_Split*. If it cannot be computed using *LMEBased\_Split*, the problem is computed by calling either *Eager\_Split* or *Model\_Enumerate*.

We present (see Fig. 4) the procedure *QE\_Layer4* which computes  $\exists x_1 \dots \exists x_t. A$  using *QE1\_Layer4*. *QE\_Layer4* eliminates the quantifiers  $\exists x_1, \dots, \exists x_t$  one by one

by making use of the procedure *QE1\_Layer4*. Note that *result* here is in general a disjunction  $d_1 \vee \dots \vee d_n$  where each  $d_i$  is a conjunction of LMCs.

We present in Fig. 5 the algorithm *QE\_LMC* that computes  $\exists x_1 \dots \exists x_t. A$  using *QE1\_Layers1To3* and *QE\_Layer4*. *QE\_LMC* initially tries to eliminate the quantified variables  $x_1, \dots, x_t$  one by one by invoking *QE1\_Layers1To3*. Variables that cannot be eliminated by *QE1\_Layers1To3* are collected in a set  $Y$ . It can be observed that after the **for** loop in *QE\_LMC*,  $\exists x_1 \dots \exists x_t. A$  can be equivalently expressed as  $\varphi_1 \wedge \exists Y. \varphi_2$  where  $\varphi_1$  and  $\varphi_2$  are conjunctions of LMCs. This is achieved using the function *scopeReduce* in Fig. 5. Finally  $\exists Y. \varphi_2$  is computed by *QE\_Layer4*. The result is conjoined with  $\varphi_1$  to obtain the final result. *QE\_Layer4* computes  $\exists Y. \varphi_2$  as a disjunction of conjunctions of LMCs. Hence the final result is, in general, a Boolean combination of LMCs.

<pre> <b>QE1_Layers1To3</b>(<b>A</b>, <math>x_1</math>) <math>\psi_1 \wedge \exists x_1. \psi_2 \leftarrow QE1\_Layer1(A, x_1);</math> <b>if</b> (<math>\psi_2</math> is free of LMDs)   <b>return</b> (<math>\psi_1 \wedge QE1\_LME(\psi_2, x_1)</math>); <b>else</b>   <math>e \leftarrow</math> LME in <math>\psi_2</math>;   <math>\psi_{D,2} \leftarrow</math> set of LMDs in <math>\psi_2</math>;   <b>if</b> (<math>DropLMDSimple(\{e\}, \psi_{D,2}, x_1) = \{e\}</math>)     <b>return</b> (<math>\psi_1 \wedge QE1\_LME(e, x_1)</math>);   <b>else</b>     <math>\psi'_2 \leftarrow QE1\_Layer3(\{e\}, \psi_{D,2}, x_1);</math>     <b>if</b> (<math>\psi'_2 = e</math>)       <b>return</b> (<math>\psi_1 \wedge QE1\_LME(e, x_1)</math>);     <b>else return</b> (<math>\psi_1 \wedge \exists x_1. \psi'_2</math>); </pre>	<pre> <b>QE_LMC</b>(<b>A</b>, <math>\{x_1, \dots, x_t\}</math>) <math>Y \leftarrow \{\};</math> <b>for each</b> <math>x_i \in \{x_1, \dots, x_t\}</math>   <math>A' \leftarrow QE1\_Layers1To3(A, x_i);</math>   <b>if</b> (<math>A'</math> is free of <math>x_i</math>)     <math>A \leftarrow A'</math>;   <b>else</b> /* <math>A' \equiv \psi_1 \wedge \exists x_i. \psi'_2</math> */     <math>A \leftarrow \psi_1 \wedge \psi'_2</math>;     <math>Y \leftarrow Y \cup \{x_i\}</math>; <math>\varphi_1 \wedge \exists Y. \varphi_2 \leftarrow scopeReduce(A, Y);</math> <math>\varphi'_2 \leftarrow QE\_Layer4(\varphi_2, Y);</math> <b>return</b> <math>\varphi_1 \wedge \varphi'_2</math>; </pre>
---	--

Fig. 5. Procedures *QE1\_Layers1To3* and *QE\_LMC*

### 3 Boolean Combinations of LMCs

The QE algorithm *QE\_LMC* accepts a conjunction of LMCs. Extending *QE\_LMC* to formulae that are Boolean combinations of LMCs requires the formulae in Disjunctive Normal Form (DNF). However DNF is a space-inefficient representation. Hence, the core issue here is finding a space-efficient representation for DNF. This issue is also encountered in other first order theories such as linear arithmetic over reals. Following the ideas in [1] and [2], we explore two approaches for extending *QE\_LMC* to Boolean combinations of LMCs - *Decision Diagram (DD)* based approach and *Directed Acyclic Graph (DAG)* based approach.

#### 3.1 DD Based Approach

We introduce a data structure called Linear Modular Decision Diagram (LMDD) that represents Boolean combinations of LMCs. LMDDs are like BDDs [4], but with nodes labeled by 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 helper functions: a) *createLMDD* for creating an LMDD from a DAG representing a Boolean

combination of LMCs, b) *isUnsat* to determine if the conjunction of LMCs in the given set is unsatisfiable, d) *getConjunct* to compute the conjunction of LMCs in a given set  $\varphi$ , e) *AND*, *OR*, *NOT*, *ITE* to 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 the LME labeling the node and  $H(f)$  and  $L(f)$  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 originating at the node  $f$  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 can be exponential in the number of nodes).

In the following discussion we present a more efficient procedure *QuaLMoDE* to compute  $\exists V. f$ . *QuaLMoDE* makes use of a procedure called *QE1\_LMDD* that eliminates a single variable  $v$  from  $f$  (see Fig. 6). To compute  $\exists v. f$ , we call *QE1\_LMDD* with arguments  $f$ ,  $\{\}$ ,  $\{\}$  and  $v$ . *QE1\_LMDD* performs a recursive traversal of the LMDD rooted at  $f$  collecting the set of LMEs  $E$ , and the set of LMDs  $D$ , containing  $v$  that it encountered along the path from  $f$ .

In general, if  $E$  denotes the set of LMEs and  $D$  denotes the set of LMDs, *QE1\_LMDD*( $f, E, D, v$ ) computes an LMDD for  $\exists v. (f \wedge C_E \wedge C_D)$ , 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  $\{(2^{k_1} \cdot v = t_1), \dots, (2^{k_n} \cdot v = t_n)\}$ . Without loss of generality, let  $k_1$  be the smallest among  $k_1, \dots, k_n$ . Let  $g$  be an internal non-terminal node of  $f$ . Thus  $g$  can be represented as  $(P(g), H(g), L(g))$ . Suppose  $P(g)$  is  $(2^k \cdot v = t)$  where  $k \geq k_1$ . It can be observed that  $g$  can then be simplified to  $((2^{k-k_1} \cdot t_1 = t), H(g), L(g))$  using the LME  $(2^{k_1} \cdot v = t_1)$ . Procedures *selectLME* and *simplifyLMDD* (see Fig. 6) respectively perform the selection of LME with the minimum  $k$  among the LMEs in  $E$  and simplification of  $f$  using the selected LME as described above. The procedure *applyL1* in Fig. 6 returns an LME equivalent to the argument LME using Lemma 1.

It can be observed if the same LMDD node is encountered with the same LME following two different paths, the results of the calls to *simplifyLMDD* must be the same. Hence *simplifyLMDD* can be implemented with dynamic programming.

If *simplifyLMDD* is successful in eliminating all occurrences of variable  $v$  using the selected LME, it returns  $f'$  where  $f' \equiv \exists v. f$ . The problem  $\exists v. (f \wedge C_E \wedge C_D)$  is now reduced to  $f' \wedge \exists v. (C_E \wedge C_D)$ . It can be observed that  $\exists v. (C_E \wedge C_D)$  can be computed by *QE\_LMC*. In this case, *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 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  $\exists V. f$ , that is a simple extension of the algorithm EXISTELIM in [2]. EXISTELIM as given in [2] computes

<pre> <b>QE1_LMDD(f, E, D, v)</b>   if (f = 0 <math>\vee</math> isUnsat(E <math>\cup</math> D))     return 0;   if (f = 1)     return createLMDD(QE_LMC       (getConjunct(E <math>\cup</math> D), {v}));   if (E <math>\neq</math> <math>\phi</math>)     e<sub>1</sub> <math>\leftarrow</math> selectLME(E);     f' <math>\leftarrow</math> simplifyLMDD(f, v, e<sub>1</sub>);     if (f' is free of v)       return AND(f', createLMDD         (QE_LMC(getConjunct(E <math>\cup</math> D), {v})));   else     f' <math>\leftarrow</math> f;     e <math>\leftarrow</math> 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 <math>\cup</math> {e}, D, v),       QE1_LMDD(L(f'), E, D <math>\cup</math> {<math>\neg</math>e}, v)); </pre>	<pre> <b>simplifyLMDD(f, v, e<sub>1</sub>)</b>   if (f = 1 or f = 0)     return f;   e <math>\leftarrow</math> P(f);   if (e is free of v)     return ITE(e,       simplifyLMDD(H(f), v, e<sub>1</sub>),       simplifyLMDD(L(f), v, e<sub>1</sub>));   else     applyL1(e, v);     /* gives (2<sup>k</sup> · v = t) */     applyL1(e<sub>1</sub>, v);     /* gives (2<sup>k<sub>1</sub></sup> · v = t<sub>1</sub>) */     if (k <math>\geq</math> k<sub>1</sub>)       return ITE(2<sup>k-k<sub>1</sub></sup> · t<sub>1</sub> = t,         simplifyLMDD(H(f), v, e<sub>1</sub>),         simplifyLMDD(L(f), v, e<sub>1</sub>));   else     return ITE(e,       simplifyLMDD(H(f), v, e<sub>1</sub>),       simplifyLMDD(L(f), v, e<sub>1</sub>)); </pre>
---	---

**Fig. 6.** Algorithms *QE1\_LMDD* and *simplifyLMDD*

$\exists V. f$  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.

The algorithm *Monniaux* designed by us 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 (see [2]) is replaced by *QE\_LMC*, and c) the algorithm GENERALIZE2 (see [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 described in [2] removes unnecessary literals from  $M$  and returns  $M'$  such that  $M \Rightarrow M'$  and  $M' \Rightarrow \neg G$ . However, in our experiments with LMCs, we have found that GENERALIZE2 is prohibitively time consuming as it involves a large number of SMT solver calls. We therefore designed algorithm *GENERALIZE2\_LMC* that works in the following way. Given a conjunction of literals  $M$ , we effectively have an assignment of Boolean value to each atomic predicate in 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  $M$  that 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 a simple 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 cause  $ite(A, B, C)$  to evaluate to **true**. Similarly  $\{\neg D, F\}$  is sufficient to cause  $ite(D, E, F)$  to evaluate to **true**. Hence,  $\{A, B\}$  (or  $\{\neg D, F\}$ ) is sufficient to cause  $\neg G$  to evaluate to **true**. Hence *GENERALIZE2\_LMC* would therefore return  $A \wedge B$  (or  $\neg D \wedge F$ ) as  $M'$ .

## 4 Experimental Results

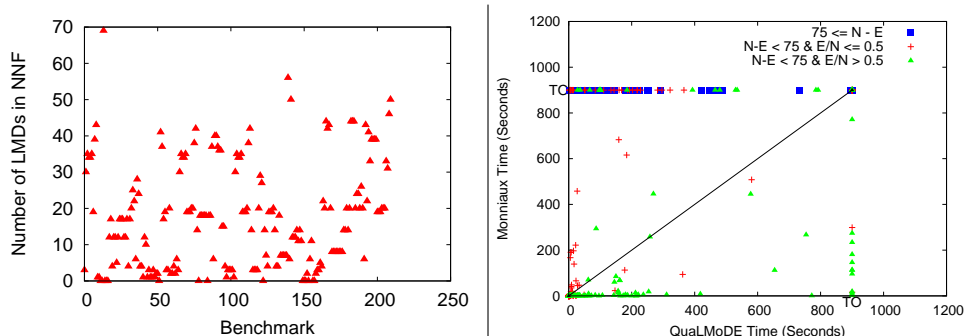
We performed three sets of experiments to achieve the following goals: a) evaluate the performance of *QualMoDE*, *Monniaux* and *QE\_LMC*, b) compare the performance of *QE\_LMC* with alternative QE techniques and c) evaluate the utility of our QE algorithms in word-level RTL verification.

The experiments were performed on a 1.83 GHz Intel(R) Core 2 Duo machine with 2GB memory running Ubuntu 8.04. We implemented our own LMDD package for carrying out QE experiments using the DD based approach.

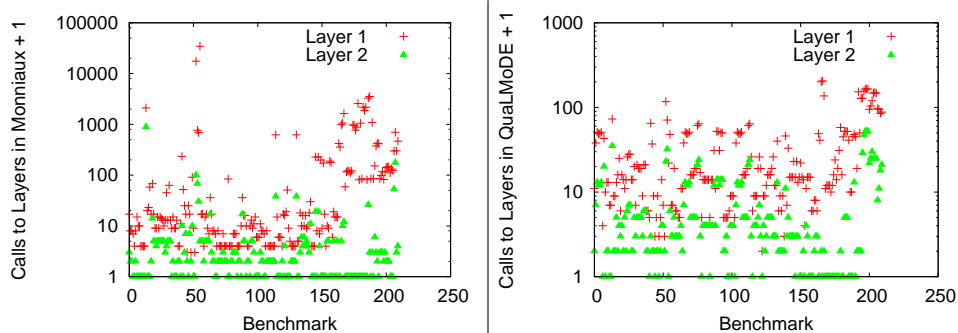
**Simplification heuristics used in our implementation :** We use the following simplification heuristics in our implementation.

1. We convert LMDs with modulus 2 to equivalent LMEs. For example consider the LMD  $x + y \neq 1 \pmod{2}$ . It is easy to see that this LMD can be equivalently expressed as  $x + y = 0 \pmod{2}$ . We have seen that this helps in easy elimination of existentially quantified variables involved in LMCs with modulus 2..
2. In a non-terminal LMDD node  $f$  expressed as  $(P(f), H(f), L(f))$ , we keep the LME  $P(f)$  in a normal form. Normalization of the LME  $c_1 \cdot x_1 + \dots + c_n \cdot x_n = c_0 \pmod{2^p}$  is done as follows. Initially, the terms on the left hand side of the LME are rearranged based on the lexicographical ordering between the variable names. Without loss of generality, let  $c_1 \cdot x_1$  be the first term in the rearranged LME. We apply Lemma 1 to convert this LME to the equivalent form  $2^{k_1} \cdot x_1 = t_1 \pmod{2^p}$ . We have seen that this helps in identification of equivalent LMEs during the LMDD creation and hence more compact LMDDs.
3. Our procedure for construction of LMDDs is same as the procedure for construction of BDDs [4] with the LMEs considered as independent Boolean variables. This results in inconsistent paths in the LMDDs. We make use of the following technique to eliminate inconsistent paths in an LMDD. We perform a recursive traversal of the LMDD carrying the context (LMEs and LMDs encountered on the path so far) along each path. Let  $f$  be an LMDD node denoted as  $(P(f), H(f), L(f))$  which is encountered with context  $\varphi$  as we are traversing the LMDD. If  $\varphi \Rightarrow P(f)$ , we identify the edge  $f \rightarrow L(f)$  as unneeded. Similarly, if  $\varphi \Rightarrow \neg P(f)$ , we identify the edge  $f \rightarrow H(f)$  as unneeded. We can make the technique amenable to dynamic programming by carefully dropping the LMCs in the context which do not affect the implication checks. We have found that this technique helps in significant reduction in LMDD sizes in some benchmarks.

**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 were obtained in the following manner. We took a set of real word-level VHDL designs and derived their symbolic transition relations. One set of *real* benchmarks was obtained by quantifying out all the internal variables (i.e. neither input nor output of the top-level module) from these symbolic transition relations. Effectively this gives abstract transition relations of the designs. The second set of *real* benchmarks were obtained by applying iterative squaring to the symbolic transition relations for 3-5 steps. Each step of iterative squaring involves quantifying out one copy of all state variables in the symbolic transition relations. We observed a significant number of LMDs in these benchmarks when expressed in Negation Normal Form (NNF) (see Fig. 7(a)). In order to generate the *artificial* benchmarks, we selected some of our *real* benchmarks and some SMTLib benchmarks from the category QF\_BV/bruttomesso/simple\_processor/ [10] and used random choices for the set of variables to be eliminated<sup>4</sup>. The total number of variables ( $N$ ), number of variables to be eliminated ( $E$ ) and the number of bits to be eliminated in the entire set of benchmarks range from 3 to 175, 1 to 170 and 1 to 1265 respectively.



**Fig. 7.** Plots showing (a) significant number of LMDs in the *real* benchmarks. (b) *QualMoDE* Time Vs *Monniaux* Time (TO : > 900 seconds)



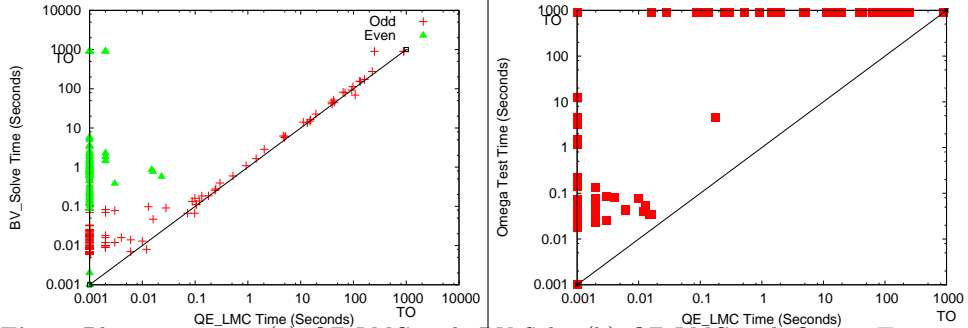
**Fig. 8.** Contribution of the layers in *QE-LMC*

<sup>4</sup> 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.

We measured the QE time by *QuaLMoDE* and *Monniaux* for each benchmark (for *QuaLMoDE*, this includes the time taken to build LMDDs). It was observed that (see Fig. 7(b)) for benchmarks with  $N - E$  below a certain threshold  $t_1$  and  $E/N$  above a certain threshold  $t_2$ , *Monniaux* outperformed *QuaLMoDE* in most cases. For our benchmark suite,  $t_1$  and  $t_2$  were empirically estimated as 75 and 0.5 respectively. For the other benchmarks, we observed that *QuaLMoDE* outperformed *Monniaux*. It was also observed that, for benchmarks with  $t_1 \leq N - E$ , *Monniaux* timed out irrespective of  $E/N$ . We figured out that the different behaviours of *Monniaux* and *QuaLMoDE* were due to the following reasons. (i) For benchmarks with low  $N - E$  and high  $E/N$ , the interleaving of projection inside model enumeration in *Monniaux* simplified the problem considerably whereas for the other benchmarks this simplification was not substantial. (ii) The single variable elimination strategy in *QuaLMoDE* resulted in a large number of calls to *QE1\_LMDD* for benchmarks with low  $N - E$  and high  $E/N$ .

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

We evaluated the roles of different layers of *QE\_LMC* in performing QE for the *real* benchmarks. It was observed that all 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 were solved by the second layer (see Fig. 8)<sup>5</sup>.



**Fig. 9.** 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 have compared the performance of *QE\_LMC* with QE based on Presburger Arithmetic using Omega Test and with QE based on bit-blasting (see Fig. 9). In the latter case, we implemented a procedure *BV\_Solve* that first quantifies out variables appearing with odd coefficients in LMEs using the ideas described in [6] and

<sup>5</sup> Note that the y-axis of both plots are in log-scale. One is added to the y-values to include the points with no calls to the second layer.

then uses bit-blasting and BDD based bit-level QE [11] for the remaining variables. We used a set of 405 benchmarks that are instances of the QE problem for conjunction of LMCs; 371 of these arise from calls from *QuaLMoDE/Monniaux* when QE is performed on the *real* benchmarks and the remaining 34 are randomly generated. Our results clearly demonstrate that *QE\_LMC* outperforms both alternative QE techniques. In Fig. 9(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. Hence 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 checking safety properties of a set of word-level VHDL designs using BMC. We first derived the symbolic transition relation  $R$  of each design. For each BMC frame  $i$ , we then used slicing to obtain a slice  $R_i$  of  $R$  containing only the 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 to handle Boolean combinations of LMCs). The final unrolled constraint is a conjunction of the different  $R'_i$ s computed by *QuaLMoDE/QBV\_Solve*. This is conjoined with the negation of the safety property being checked and given to an SMT solver for checking satisfiability. The SMT solver used is simplifyingSTP [12]<sup>6</sup>. Table 1 gives a summary of these results. The designs in our experiments 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 benefits 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 in principle for checking the satisfiability of Boolean combinations of LMCs. This can be done by quantifying out all variables. However preliminary experiments suggest that this approach is not competitive with DPLL-style SMT solvers or with bit-blasting followed by QBF solving. Thus, the intended usage of our algorithms is for eliminating some (but not all) variables from Boolean combinations of LMCs.

**Significance of *GENERALIZE2\_LMC* in *Monniaux* :** In order to evaluate the significance of *GENERALIZE2\_LMC* in *Monniaux*, we compared

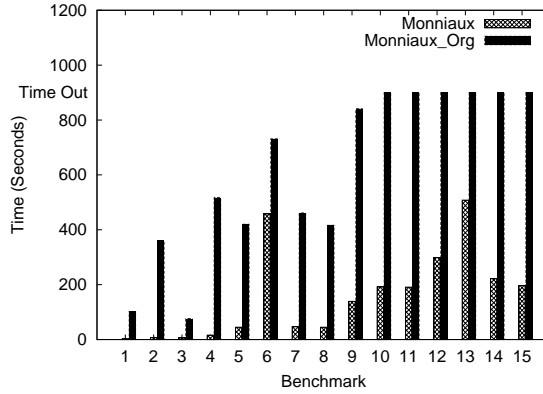
<sup>6</sup> We selected simplifyingSTP because (i) it is the winner of SMT-COMP 2010 bit-vector category and (ii) it has a variable eliminator implemented as per [6].

**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

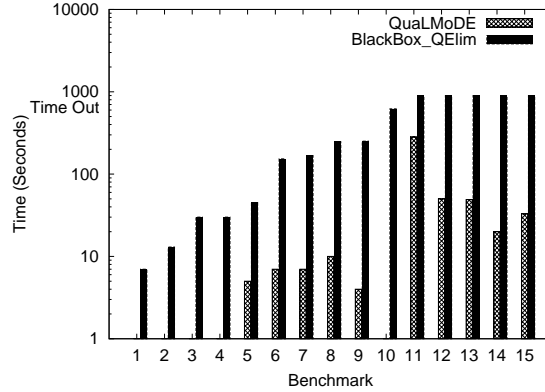
the performance of *Monniaux* with a procedure *Monniaux\_Org* which is same as *Monniaux* except that it uses the algorithm GENERALIZE2 in [2] in place of *GENERALIZE2\_LMC* for generalization of conjunctions. Fig. 10 plots the time spent by *Monniaux/Monniaux\_Org* for QE from fifteen *real* benchmarks. This clearly demonstrates that *GENERALIZE2\_LMC* is crucial in effective extension of *QE\_LMC* to Boolean combinations of LMCs in the DAG based approach.



**Fig. 10.** Plot showing significance of *GENERALIZE2\_LMC* in *Monniaux* (Time Out : > 900 seconds)

**Significance of *simplifyLMDD* and single variable elimination strategy in *QualMoDE*** : In order to evaluate the significance of *simplifyLMDD* and the single variable elimination strategy used in *QualMoDE*, we compared the performance of *QualMoDE* with a procedure *BlackBox\_QElim* which is *QualMoDE* without *simplifyLMDD* and the single variable elimination strategy. *BlackBox\_QElim* applies *QE\_LMC* to each path of the LMDD similar to

the Black-box QE on Linear Decision Diagrams described in [1] (with *QE\_LMC* as the procedure for QE from conjunctions of predicates). Fig. 11 plots the time spent by *QuaLMoDE/BlackBox\_QElim* for QE from fifteen *real* benchmarks (Note that the y-axis is in log-scale). This clearly demonstrates the crucial role of *simplifyLMDD* and the single variable elimination strategy in *QuaLMoDE*.



**Fig. 11.** Plot showing significance of *simplifyLMDD* and single variable elimination strategy in *QuaLMoDE* (Time Out : > 900 seconds)

#### Details of procedures *BV\_Solve* and *QBV\_Solve* :

***BV\_Solve*** : It can be observed that given  $\exists x_1. A$ , (as per the notation in Section 2), the work in [6] can be used directly to eliminate the quantifier if there exists an LME with odd coefficient for  $x_1$ . For example, consider the problem of computing  $\exists x. ((2y = 5x + 2) \wedge (4y \neq 5x + 6z))$  where all the LMCs have modulus 8. The work in [6] can be used to convert this to  $\exists x. ((x = 2y + 6) \wedge (4y \neq 5x + 6z))$ . Substituting  $2y + 6$  for  $x$  in  $(4y \neq 5x + 6z)$ , directly gives  $(2y \neq 6z + 6)$  as the result.

However, if there is no LME with odd coefficient for the variable to be eliminated, the work in [6] cannot be directly used to perform the QE.  $\exists y. ((2y = 5x + 2) \wedge (4y \neq 5x + 6z))$  is an example to this.

Given  $\exists x_1 \dots \exists x_t. A$ , the procedure *BV\_Solve* eliminates the set of variables  $Y \subseteq \{x_1, \dots, x_t\}$  which can be eliminated using the work in [6] as mentioned above. This converts  $\exists x_1 \dots \exists x_t. A$  into  $\exists x_1 \dots \exists x_l. \varphi_1 \wedge \varphi_2$  where  $\{x_1, \dots, x_l\} \equiv \{x_1, \dots, x_t\} \setminus Y$  and  $\varphi_1, \varphi_2$  are conjunctions of LMCs.  $\exists x_1 \dots \exists x_l. \varphi_1$  is computed by bit-blasting and bit-level QE using [11]. The resulting BDD for  $\exists x_1 \dots \exists x_l. \varphi_1$  is converted to a bit-vector formula (using extraction operation on bit-vectors) and is conjuncted with  $\varphi_2$  to obtain the final result. Note that this approach does not give back the result as a Boolean combination of LMCs.

We have observed that *BV\_Solve* scales poorly for QE instances with (i) no LME with odd coefficient for the variable to be eliminated and (ii) large modulus (See Fig. 9).

***QBV\_Solve*** : The procedure *QBV\_Solve* (see Fig. 12) extends the procedure *BV\_Solve* to Boolean combinations of LMCs using the DD based approach. The notation used here is the same as the one used in section 3.1. In order to compute  $\exists V. f$ , we call *QBV\_Solve* with arguments  $f, \{\}, \{\}, V, \text{false}$ . *QBV\_Solve* performs recursive traversal of  $f$  carrying the context along each path. Effectively *QBV\_Solve* applies *BV\_Solve* to each path of the LMDD  $f$  and returns the result as a dag  $\varphi$  which is the disjunction of *BV\_Solve* applied to each path of  $f$ .

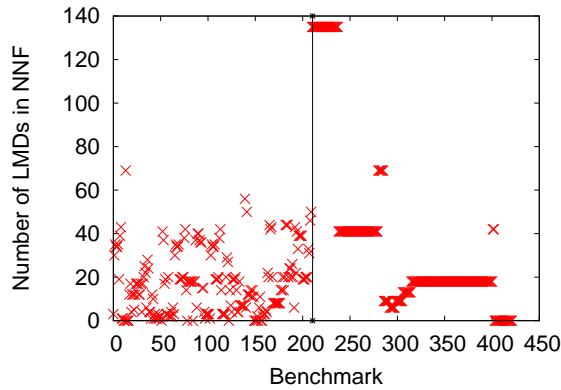
```

QBV_Solve( $f, E, D, V, \varphi$ )
  if ( $f = 1$  or  $f = 0$ )
    if ( $f = 1$  and !isUnsat( $E \cup D$ ))
       $\varphi \leftarrow \varphi \vee \text{BV\_Solve}(\text{getConjunct}(E \cup D), V)$ ;
    else
      QBV_Solve( $H(f), E \cup \{e\}, D, V, \varphi$ );
      QBV_Solve( $L(f), E, D \cup \{\neg e\}, V, \varphi$ );

```

**Fig. 12.** Algorithm *QBV\_Solve*

**Plots with *artificial* benchmarks :** All the plots presented previously except the plot in Fig. 7(b) include only the *real* benchmarks. We present here the plots in Fig. 7(a) and Fig. 8 with the *artificial* benchmarks added. In each the plot, there is a vertical line separating the *real* and *artificial* benchmarks. The first 210 benchmarks on the left hand side of the vertical line are *real* benchmarks and the 212 benchmarks on the right hand side of the vertical line are *artificial* benchmarks. Note that in the plots in Fig. 14 and Fig. 15, the number of LMDs in *QE\_LMC* calls from *QuaLMoDE/Monniaux* is plotted as yerrorbars to show the maximum, minimum and average number of LMDs in the calls.



**Fig. 13.** Plot showing the number of LMDs in the benchmarks

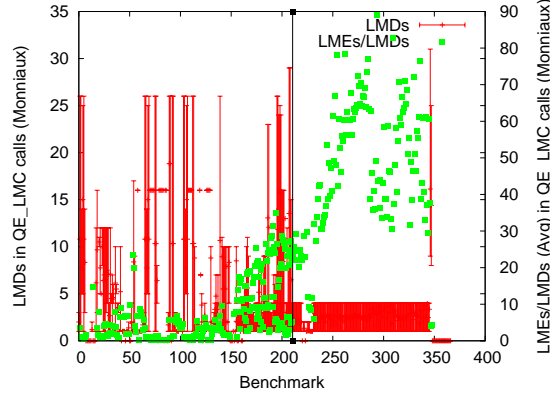


Fig. 14. Benchmark-wise details of  $QE\_LMC$  calls for *Monniaux*

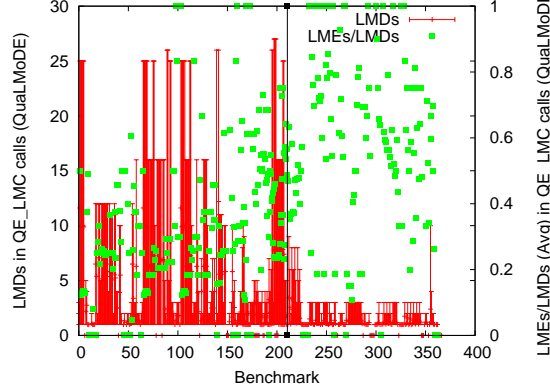


Fig. 15. Benchmark-wise details of  $QE\_LMC$  calls for *QuaLMoDE*

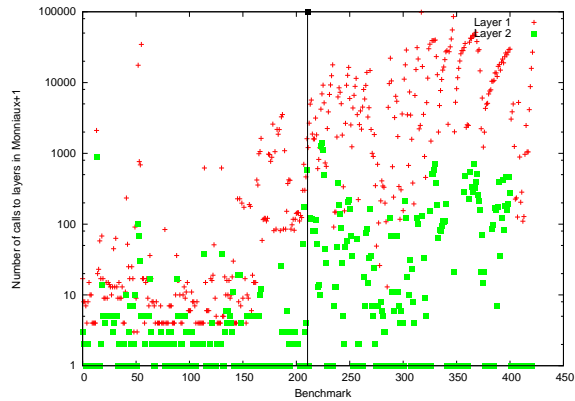
## 5 Conclusion

In this paper, we addressed the QE problem for LMCs. Our main contributions are: (i) a bit-blasting-free QE algorithm for conjunctions of LMCs that is later extended to a QE algorithm for Boolean combinations of LMCs, and (ii) comparison of 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 for linear modular inequalities and non-linear modular equalities as part of future work.

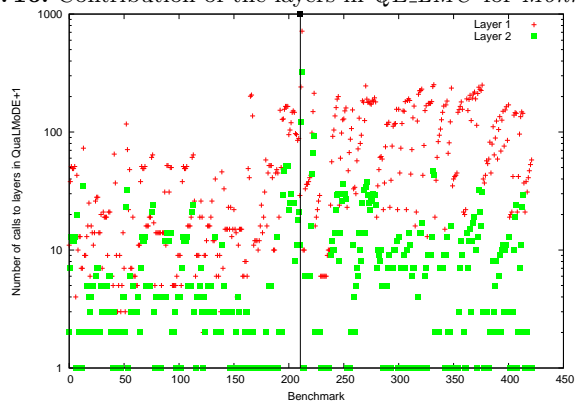
**Acknowledgements :** We would like to thank Trevor Hansen and Vijay Ganesh for providing us with the latest version of simplifyingSTP. We also thank Mukesh Sharma, Ashutosh Kulkarni, Rajkumar Gajavelly and Nachiket Vaidya for their valuable support. We convey our special acknowledgement to Anup Bhattacharjee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



**Fig. 16.** Contribution of the layers in *QE\_LMC* for *Monniaux*



**Fig. 17.** Contribution of the layers in *QE\_LMC* for *QualMoDE*

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](http://vlsi.colorado.edu/~fabio/CUDD)
12. STP website, <http://sites.google.com/site/stpfastprover/>
13. A. John, S. Chakraborty. *A quantifier elimination algorithm for linear modular equations and disequations*, Technical Report TR-11-33, CFDVS, IIT Bombay, [http://www.cfdvs.iitb.ac.in/reports/reports/ajith\\_report.pdf](http://www.cfdvs.iitb.ac.in/reports/reports/ajith_report.pdf)