On the Satisfiability of Modular Arithmetic Formulae

Bow-Yaw Wang *

Institute of Information Science
Academia Sinica
Taiwan

Abstract. Modular arithmetic is the underlying integral computation model in conventional programming languages. In this paper, we discuss the satisfiability problem of propositional formulae in modular arithmetic over the finite ring $\mathbf{Z}_{2^\omega}$. Although an upper bound of $2^{2^{O(n^4)}}$ can be obtained by solving alternation-free Presburger arithmetic, it is easy to see that the problem is in fact **NP**-complete. Further, we give an efficient reduction to integer programming with the number of constraints and variables linear in the length of the given linear modular arithmetic formula. For non-linear modular arithmetic formulae, an additional factor of ω is needed. With the advent of efficient integer programming packages, our algorithm could be useful to software verification in practice.

1 Introduction

Modular arithmetic is widely used in the design of cryptosystems and pseudo random number generators [22, 12]. Since integers use a finite binary representation in conventional programming languages such as C, modular arithmetic is often required in software verification as well. Indeed, many algorithms are designed to avoid overflow in modular arithmetic explicitly. Verification tools therefore need to support modular arithmetic to check these algorithms.

In this paper, we discuss the satisfiability problem of propositional formulae in modular arithmetic. All arithmetic computation in the formulae is over the finite ring $\mathbf{Z}_{2^\omega}$ for some fixed ω. In addition to linear terms, non-linear terms such as multiplications and modulo operations of arbitrary terms are allowed. We show that the satisfiability problem is **NP**-complete for formulae of linear modular arithmetic. The problem is still in **NP** for full modular arithmetic.

We give an efficient reduction to integer programming to have a practical decision procedure for modular arithmetic. Several issues have to be addressed in our construction. Firstly, modular computation must be simulated by linear constraints, as well as all logical operations. Furthermore, non-linear multiplications and modulo operations need to be expressed in the form of linear constraints. Most importantly, we would not like our reduction to increase the size of the

* This work was partly supported by NSC under grants NSC 94-2213-E-001-003- and NSC 95-2221-E-001-024-MY3.

problem significantly. Our construction should not use more than a linear number of constraints and variables in the length of the modular arithmetic formula.

It is well-known that the first-order non-linear arithmetic theory is undecidable [8]. Presburger arithmetic is a decidable first-order linear arithmetic theory [6, 15, 19]. In [15], Oppen shows an upper bound of $2^{2^{2^{O(n \lg n)}}}$ for determining the truth of Presburger arithmetic formula of length n. If the number of quantifier alternation is m, the problem can be solved in time $2^{2^{O(n^{m+4})}}$ and space $2^{O(n^{m+4})}$ [19]. Although Presburger arithmetic can express first-order linear arithmetic properties, it does not allow modular arithmetic nor non-linear operations.

Integer programming optimizes a given linear objective function subject to a set of linear constraints [16]. The problem is known to be **NP**-complete. Unlike Presburger arithmetic, it does not allow arbitrary logical combinations of constraints but their conjunction. It does not allow modular arithmetic either.

Other decision procedures for linear arithmetic are available. In [4], a survey of the automata-theoretic approach is given. For a special class of quantifier-free Presburger arithmetic, [21] gives an efficient reduction to Boolean satisfiability. The tool CVC Lite [2] contains a decision procedure to check validity of linear arithmetic formula. Similar to [6, 15, 19], none of them considers modular arithmetic nor non-linear operations. In [1], a decision procedure for systems of modular arithmetic inequalities is proposed. Although the authors use an algebraic approach to check the satisfiability of (in)equalities in a system. It is unclear whether the logical and modulo operations can be added within their framework.

We note that our reduction may serve as a reduction to Presburger arithmetic. Since Presburger arithmetic does not allow modular arithmetic, encoding it in linear constraints allows us to solve the problem by various decision procedures for Presburger arithmetic. However, solving the corresponding Presburger arithmetic formula requires $2^{2^{O(n^4)}}$ in the length of the modular arithmetic formula. Our reduction is more efficient asymptotically.

The remaining of paper is organized as follows. Section 2 contains the background. It is followed by the syntax and semantics of linear modular arithmetic in Section 3. The algorithm for the satisfiability of linear modular arithmetic is presented in Section 4. The syntax and semantics of modular arithmetic formulae are defined in Section 5. Section 6 discusses the satisfiability problem for non-linear modular arithmetic. Applications of our algorithm are discussed in Section 7. We report our preliminary experimental results in Section 8. Finally, Section 9 concludes the paper.

2 Preliminaries

Let $\mathbf{Z}$ be the set of integers, $\mathbf{Z}^+$ the set of positive integers, and $\mathbf{Z}^\times$ the set of non-zero integers. In the following exposition, we will fix the set X of integer variables and $m = 2^\omega$ where $\omega \in \mathbf{Z}^+$.

Definition 1. *([11], for example) For any $a \in \boldsymbol{Z}, b \in \boldsymbol{Z}^{\times}$, there are $q, r \in \boldsymbol{Z}$ such that $a = bq + r$ and $0 \leq r < |b|$.*

The numbers q and r are called *a quotient b* ($a \operatorname{quo} b$) and *a modulo b* ($a \bmod b$) respectively. We also define *signed quotient* and *signed modulo* as follows.

$$a \operatorname{smod} b \triangleq \begin{cases} a \bmod b & \text{if } 0 \leq a \bmod b < \lfloor \frac{|b|}{2} \rfloor \\ a \bmod b - |b| & \text{if } \lfloor \frac{|b|}{2} \rfloor \leq a \bmod b < |b| \end{cases}$$

$$a \operatorname{squo} b \triangleq \frac{a - (a \operatorname{smod} b)}{b}$$

For example, -7 quo $-3 = 3$ and $-7 \bmod -3 = 2$, but -7 squo $-3 = 2$ and -7 smod $-3 = -1$ for $-7 = -3 \times 3 + 2$. We say a is *congruent to b modulo* m, $a \equiv b \pmod{m}$, if $(a - b) \bmod m = 0$. For any $a \in \mathbf{Z}$, the *residue class of a modulo* m is the set $[a] \triangleq \{x | x \equiv a \pmod{m}\}$. It is easy to verify that the *residue class system* $\mathbf{Z}_m = (\{[0], [1], \ldots, [m-1]\}, +, [0], \cdot, [1])$ is a commutative ring with identity [11].

Since $\mathbf{Z}_m$ consists of residue classes of integers modulo m, several representations of the equivalence classes are possible. Particularly, we call $\{-\frac{m}{2}, \ldots, -1, 0, 1, \frac{m}{2} - 1\}$ the *signed representation* and $\{0, 1, \ldots, m-1\}$ the *unsigned representation*. To emulate integral computation in conventional languages, we use the signed representation if not mentioned otherwise. If $c \in \mathbf{Z}$, the notation $c \in \mathbf{Z}_m$ denotes that c is an element in the signed representation of $\mathbf{Z}_m$.

Let $c, a_{i,j} \in \mathbf{Z}$ and $x_j \in X$ for $0 \leq i < M$, $0 \leq j < N$. Given a set of M *linear constraints* $\sum_{j=0}^{N-1} a_{i,j} x_j \sim_i c_i$ where $\sim_i \in \{\leq, <, =, >, \geq\}$, and a linear objective function $\sum_{j=0}^{N-1} b_j x_j$, the *integer programming problem* is to find a *valuation* $\rho : X \to \mathbf{Z}$ such that ρ satisfies all linear constraints and attains the maximum value of the objective function. We denote an instance of integer programming problem as follows.

$$\begin{array}{lll} \text{maximize} & \sum_{j=0}^{N-1} b_j x_j & \\ & \sum_{j=0}^{N-1} a_{0,j} x_j & \sim_0 \quad c_0 \\ & \sum_{j=0}^{N-1} a_{1,j} x_j & \sim_1 \quad c_1 \\ \text{subject to} & \vdots & \\ & \sum_{j=0}^{N-1} a_{M-1,j} x_j & \sim_{M-1} c_{M-1} \end{array}$$

It is known that the integer programming problem is **NP**-complete [16].

3 Linear Modular Arithmetic

For any $c \in \mathbf{Z}_m$ and $x \in X$, the syntax of the Linear Modular Arithmetic Formula over $\mathbf{Z}_m$ is defined in Figure 1. We use the symbols $\%$ and $\div$ for the modulo and quotient operators respectively in our object language to avoid confusion. Also, we do not use syntactic translation for equality nor any of the logical

$$\begin{aligned}
\text{Term } t &\triangleq c \mid c \cdot t \mid t \,\%\, c \mid t \div c \mid t + t' \\
\text{Atomic Proposition } l &\triangleq \mathsf{ff} \mid t \leq t' \mid t = t' \\
\text{Formula } f &\triangleq l \mid \neg f \mid f \wedge f' \mid f \vee f'
\end{aligned}$$

Fig. 1. Syntax of Linear Modular Arithmetic Formula over $\mathbf{Z}_m$

connectives. A more efficient reduction can be attained by treating each operator separately, although it does not improve the performance asymptotically. Finally, only constants in $\mathbf{Z}_m$ are allowed. Overflowed constants cause compilers to generate warnings; they can be identified rather easily.[1]

$$\begin{aligned}
[\![c]\!]_\rho &\triangleq c & [\![t \,\%\, c]\!]_\rho &\triangleq [\![t]\!]_\rho \bmod c \\
[\![c \cdot t]\!]_\rho &\triangleq c[\![t]\!]_\rho \operatorname{smod} m & [\![t \div c]\!]_\rho &\triangleq [\![t]\!]_\rho \operatorname{quo} c \\
[\![t + t']\!]_\rho &\triangleq [\![t]\!]_\rho + [\![t']\!]_\rho \operatorname{smod} m
\end{aligned}$$

$$\begin{aligned}
[\![\mathsf{ff}]\!]_\rho &\triangleq \mathsf{false} & [\![\neg f]\!]_\rho &\triangleq \neg [\![f]\!]_\rho \\
[\![t \leq t']\!]_\rho &\triangleq [\![t]\!]_\rho \leq [\![t']\!]_\rho & [\![f \wedge f']\!]_\rho &\triangleq [\![f]\!]_\rho \wedge [\![f']\!]_\rho \\
[\![t = t']\!]_\rho &\triangleq [\![t]\!]_\rho = [\![t']\!]_\rho & [\![f \vee f']\!]_\rho &\triangleq [\![f]\!]_\rho \vee [\![f']\!]_\rho
\end{aligned}$$

Fig. 2. Semantics of Linear Modular Arithmetic Formula over $\mathbf{Z}_m$

For any valuation ρ, the semantic function $[\![\bullet]\!]_\rho$ for linear modular arithmetic formulae over $\mathbf{Z}_m$ is defined in Figure 2. Since $c \in \mathbf{Z}_m$, it is unnecessary to compute the signed representations for constants, modulo and quotient operations. For the others, their semantic values are obtained by the signed modulo m.

Assume each integral and logical computation in conventional languages takes $O(1)$ time. We can now phrase the *satisfiability problem* as follows.

Problem 1. (Satisfiability) Given a linear modular arithmetic formula f over $\mathbf{Z}_m$ with variables $\bar{x}$, determine whether there is a valuation ρ such that $[\![f]\!]_\rho = \mathsf{true}$.

Since the evaluation of any linear modular arithmetic formula is in $\mathbf{P}$, we immediately have the following upper bound for the satisfiability problem.

Proposition 1. *The satisfiability problem for any linear modular arithmetic formula f can be decided in $\boldsymbol{NP}$.*[2]

[1] In gcc 4.0.2, the warning message "integer constant is too large for its type" is shown.
[2] Please refer to [23] for all proofs of the propositions and theorems in this paper.

The lower bound of the problem can be obtained by reduction from **3CNF**. Although Boolean variables are not allowed in linear modular arithmetic, they can be simulated by the parity of integer variables fairly easily.

Proposition 2. *The satisfiability problem for any linear modular arithmetic formula f is **NP**-hard.*

Corollary 1. *The satisfiability problem for linear modular arithmetic formula is **NP**-complete.*

4 Solving the Satisfiability Problem for Linear Modular Arithmetic

Since modular arithmetic is the default integral computation in conventional languages, deciding the satisfiability of linear modular arithmetic formula could be useful in software verification. One may, of course, use binary encoding and solve the problem in the Boolean domain. But it would disregard the nature of the problem. We are therefore looking for alternatives capable of exploiting the underlying mathematical structure of the problem.

Given an instance of any syntactic class (terms, atomic propositions, or formulae), we translate it to an integer variable and a set of constraints. Intuitively, the integer variable has the semantic value of the given instance for any valuation subject to the set of constraints. For terms, the integer variable has a value in $[-\frac{m}{2}, \frac{m}{2} - 1]$. For atomic propositions and formulae, it has values 0 or 1.

$$\sigma(c) \triangleq (p, p = c) \qquad \sigma(c \cdot t) \triangleq \begin{pmatrix} \alpha \\ p, \ -\frac{m}{2} \le p < \frac{m}{2} \\ cp' - mq = p \end{pmatrix} \text{ where } (p', \alpha) = \sigma(t)$$

$$\sigma(t_0 + t_1) \triangleq \begin{pmatrix} \alpha_0 \\ p, \quad \alpha_1 \\ -\frac{m}{2} \le p < \frac{m}{2} \\ p_0 + p_1 - mq = p \end{pmatrix} \text{ where } \begin{matrix} (p_0, \alpha_0) = \sigma(t_0) \\ (p_1, \alpha_1) = \sigma(t_1) \end{matrix} \qquad \sigma(t \,\%\, c) \triangleq \begin{pmatrix} \alpha \\ p, \ 0 \le p < |c| \\ p' - cq = p \end{pmatrix} \text{ where } (p', \alpha) = \sigma(t)$$

$$\sigma(t \div c) \triangleq \begin{pmatrix} \alpha \\ p, \ 0 \le p' - cp < |c| \end{pmatrix} \text{ where } (p', \alpha) = \sigma(t)$$

Fig. 3. Linear Constraints for Terms

Consider, for example, the following translation of $t \,\%\, c$ (Figure 3).

$$\begin{pmatrix} \alpha \\ p, \ 0 \le p < |c| \\ p' - cq = p \end{pmatrix} \text{ where } (p', \alpha) = \sigma(t)$$

The semantic value p' and constraints α of t are obtained by $\sigma(t)$ recursively. Since the semantic value p of $t \,\%\, c$ is equal to $p' \,\%\, c$, we add the constraints $0 \leq p < |c|$ and $p' - cq = p$. The following proposition shows that the semantics of terms is still retained in spite of the constraints in Figure 3.

Proposition 3. *Let t be a term in linear modular arithmetic. Then, there is a valuation ρ such that $[\![t]\!]_\rho = d \Leftrightarrow$ there is a valuation η such that η satisfies α and $\eta(p) = d$ where $(p, \alpha) = \sigma(t)$.*

$$\lambda(\mathsf{ff}) \triangleq (p, p = 0) \qquad \lambda(t_0 \leq t_1) \triangleq \begin{pmatrix} & \alpha_0 & \\ & \alpha_1 & \\ p, & 0 \leq p \leq 1 \\ & p_0 - p_1 - (m-1)(1-p) \leq 0 \\ & p_0 - p_1 + mp > 0 \end{pmatrix} \text{ where } \begin{array}{l} (p_0, \alpha_0) = \sigma(t_0) \\ (p_1, \alpha_1) = \sigma(t_1) \end{array}$$

$$\lambda(t_0 = t_1) \triangleq \begin{pmatrix} & \alpha_0 & \\ & \alpha_1 & \\ & 0 \leq q_0 + q_1 \leq 1 & \\ p, & p_0 - p_1 + m(1-q_0) - q_0 \geq 0 & \\ & p_0 - p_1 - m(1-q_1) + q_1 \leq 0 & \\ & p_0 - p_1 - m(q_0 + q_1) \leq 0 & \\ & p_0 - p_1 + m(q_0 + q_1) \geq 0 & \\ & 1 - q_0 - q_1 = p & \end{pmatrix} \text{ where } \begin{array}{l} (p_0, \alpha_0) = \sigma(t_0) \\ (p_1, \alpha_1) = \sigma(t_1) \end{array}$$

Fig. 4. Linear Constraints for Atomic Propositions

For atomic propositions, observe

$$-m < -m+1 = -\frac{m}{2} - (\frac{m}{2} - 1) \leq [\![t_0]\!]_\rho - [\![t_1]\!]_\rho \leq (\frac{m}{2} - 1) - (-\frac{m}{2}) = m - 1 < m.$$

Consider the atomic proposition $t_0 \leq t_1$. From Figure 4, we have

$$\begin{pmatrix} & \alpha_0 & \\ & \alpha_1 & \\ p, & 0 \leq p \leq 1 \\ & p_0 - p_1 - (m-1)(1-p) \leq 0 \\ & p_0 - p_1 + mp > 0 \end{pmatrix} \text{ where } \begin{array}{l} (p_0, \alpha_0) = \sigma(t_0) \\ (p_1, \alpha_1) = \sigma(t_1) \end{array}.$$

Since the variables p_0 and p_1 have the semantic values of the terms t_0 and t_1 respectively, we have $-m < p_0 - p_1 \leq m - 1$. If $p_0 \leq p_1$, it is easy to verify that the constraints are satisfied if the semantic value p is 1. Conversely, if the variable p has the value 1, $p_0 - p_1 - (m-1)(1-p) = p_0 - p_1 \leq 0$ is enforced by the constraints. Thus $p_0 \leq p_1$.

For equality, one could use a less efficient construction by conjunction and comparison. But we have a slightly better translation in Figure 4. Intuitively, the variables q_0 and q_1 denote $p_0 > p_1$ and $p_0 < p_1$ respectively. Note that q_0 and q_1 must be 0 or 1 by the constraint $0 \leq q_0 + q_1 \leq 1$. If $q_0 = 1$, then $p_0 - p_1 - 1 \geq 0$ by the constraint $p_0 - p_1 + m(1 - q_0) - q_0 \geq 0$. Hence $p_0 > p_1$. Conversely, suppose $p_0 > p_1$ but $q_0 = 0$. There are two cases. If $q_1 = 0$, we have $p_0 - p_1 \leq 0$ by the constraint $p_0 - p_1 + m(q_0 + q_1) \leq 0$, a contradiction. If $q_1 = 1$, $p_0 - p_1 + 1 \leq 0$ by the constraint $p_0 - p_1 - m(1 - q_1) + q_1 \leq 0$, also a contradiction. Hence, $q_0 = 1$ if and only if $p_0 > p_1$ for any valuation satisfying the constraints. And the semantic value of $t_0 = t_1$ is 1 if and only if $q_0 = q_1 = 0$, namely, $1 - q_0 - q_1$.

The following proposition shows that we can replace the semantics values of atomic propositions by 0 or 1.

Proposition 4. *Let l be an atomic proposition in linear modular arithmetic. Then, (1) there is a valuation ρ such that $[\![l]\!]_\rho = \mathsf{true} \Leftrightarrow$ there is a valuation η such that η satisfies α and $\eta(p) = 1$ where $(p, \alpha) = \lambda(l)$; (2) there is a valuation ρ such that $[\![l]\!]_\rho = \mathsf{false} \Leftrightarrow$ there is a valuation η such that η satisfies α and $\eta(p) = 0$ where $(p, \alpha) = \lambda(l)$.*

Let p_0 and p_1 be the semantic values of the subformulae f_0 and f_1 respectively. Consider the constraints $p_0 + p_1 \geq p$ and $p_0 + p_1 \leq 2p$ in the translation of their disjunction (Figure 5). We would like the semantic value p of their disjunction to be 0 when both p_0 and p_1 are 0. It is achieved by the constraint $p_0 + p_1 \geq p$. On the other hand, the constraint $p_0 + p_1 \leq 2p$ is added to enforce $p = 1$ when any of the disjuncts is true.

$$\phi(l) \triangleq \lambda(l) \qquad \phi(\neg f) \triangleq \begin{pmatrix} \alpha \\ p,\ 1 - p' = p \end{pmatrix} \text{ where } (p', \alpha) = \phi(f)$$

$$\phi(f_0 \wedge f_1) \triangleq \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ p, \quad 0 \leq p \leq 1 \\ p_0 + p_1 \geq 2p \\ p_0 + p_1 \leq 1 + p \end{pmatrix} \text{ where } \begin{matrix} (p_0, \alpha_0) = \phi(f_0) \\ (p_1, \alpha_1) = \phi(f_1) \end{matrix}$$

$$\phi(f_0 \vee f_1) \triangleq \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ p, \quad 0 \leq p \leq 1 \\ p_0 + p_1 \geq p \\ p_0 + p_1 \leq 2p \end{pmatrix} \text{ where } \begin{matrix} (p_0, \alpha_0) = \phi(f_0) \\ (p_1, \alpha_1) = \phi(f_1) \end{matrix}$$

Fig. 5. Linear Constraints for Formulae

Note that we do not rearrange the input formula to canonical forms. Since the rearrangement could increase the length of the formula significantly, it would not be efficient. In order to have linear number of constraints and variables, it is crucial not to transform the input formula to canonical forms.

Given a formula in linear modular arithmetic, there is a set of constraints such that the semantic value of the formula is denoted by the designated variable in our construction. Our progress is summarized in the following proposition.

Proposition 5. *Let f be a formula in linear modular arithmetic. Then, (1) there is a valuation ρ such that $[\![f]\!]_\rho = \mathsf{true} \Leftrightarrow$ there is a valuation η such that η satisfies α and $\eta(p) = 1$ where $(p, \alpha) = \phi(f)$; and (2) there is a valuation ρ such that $[\![f]\!]_\rho = \mathsf{false} \Leftrightarrow$ there is a valuation η such that η satisfies α and $\eta(p) = 0$ where $(p, \alpha) = \phi(f)$.*

To solve the satisfiability problem of a linear modular arithmetic formula f, we first obtain an integer variable p and a set of constraints α from the translation $\phi(f)$. It is not difficult to see that the satisfiability problem can be solved by optimizing the objective function p with respect to α.

Our translation is constructed recursively. A recursive call is invoked for each subformula in the input formula. Further, a constant number of constraints and variables are added in each recursion. Since the number of subformulae is linear in the length of the input formula, the corresponding integer programming problem has the number of variables and constraints linear in the length of the input formula. The following theorem summarizes our result on the satisfiability problem for linear modular arithmetic formulae.

Theorem 1. *Given a formula f in linear modular arithmetic, the satisfiability problem can be solved by an instance of the integer programming problem with the number of constraints and variables linear in $|f|$.*

5 Modular Arithmetic

$$\text{Term } t \triangleq \ldots \mid t \cdot t' \mid t \,\%\, t'$$

$$[\![t \cdot t']\!]_\rho \triangleq [\![t]\!]_\rho [\![t']\!]_\rho \operatorname{smod} m$$
$$[\![t \,\%\, t']\!]_\rho \triangleq [\![t]\!]_\rho \bmod [\![t']\!]_\rho$$

Fig. 6. Syntax and Semantics of Modular Arithmetic over $\mathbf{Z}_m$

The syntax and semantics of modular arithmetic extend those of linear modular arithmetic by multiplication, $t \cdot t'$, and modulo operation, $t \,\%\, t'$, of terms (Figure 6). Similar to linear terms, the semantic value of term multiplication uses the signed modulo to reflect the semantics of conventional programming languages. On the other hand, it is unnecessary to compute the signed representation for modulo operations of terms since overflow could not occur.

The lower bound of the satisfiability problem for modular arithmetic formula follows from Proposition 2. Additionally, the evaluation of any modular arithmetic formula can also be done in polynomial time, we immediately have the following theorem.

Theorem 2. *The satisfiability problem for modular arithmetic formula is* ***NP****-complete.*

6 Solving the Satisfiability Problem for Modular Arithmetic

Based on the translation of linear modular arithmetic formulae, multiplications and modulo operations of arbitrary terms can be emulated in integer programming. Of course, one could use the binary representation and encode a multiplier circuit in linear modular arithmetic. But it would introduce too many temporary variables. Besides, the mathematical nature of the problem would not be preserved by Boolean circuits. We hereby propose a more efficient translation.

In order to compute non-linear terms, we will use the binary representations of operands' semantic values. But it becomes complicated for negative numbers. However, it is safe to use the unsigned representation in this context. Observe

$$ab \equiv (a+m)b \equiv a(b+m) \equiv (a+m)(b+m) \pmod{m}.$$

We therefore assume the unsigned representation, compute the result, then convert it back to the signed representation for multiplications of terms. Thus, only the linear constraints of the unsigned multiplication is needed.

$$\chi(p_0, p_1) \triangleq \left(c, \begin{array}{r} p_1 < m \\ 0 \le b_i \le 1 \text{ for } 0 \le i < \omega \\ \sum_{i=0}^{\omega-1} 2^i b_i = p_0 \\ 0 \le c_i \le 2^i p_1 \text{ for } 0 \le i < \omega \\ 2^i p_1 - 2^i m(1-b_i) \le c_i \text{ for } 0 \le i < \omega \\ 2^i m b_i \ge c_i \text{ for } 0 \le i < \omega \\ \sum_{i=0}^{\omega-1} c_i = c \end{array} \right)$$

Fig. 7. Linear Constraints for Unsigned Multiplication

More concretely, suppose $0 \le p_0 < m$. The constraints $0 \le b_0, \ldots, b_{\omega-1} \le 1$ and $\sum_{i=0}^{\omega-1} 2^i b_i = p_0$ compute the unsigned representation of p_0 (Figure 7). Intuitively, the bit string $b_{\omega-1} b_{\omega-2} \cdots b_1 b_0$ is the binary representation for p_0.

To compute the partial result $c_i = 2^i b_i p_1$, we use the constraints $0 \le c_i \le 2^i p_1$, $2^i p_1 - 2^i m(1-b_i) \le c_i$, and $2^i m b_i \ge c_i$. If $b_i = 0$, we have $2^i m b_i = 0 \ge c_i \ge 0$. On the other hand, we have $2^i p_1 - 2^i m(1-b_i) = 2^i p_1 \le c_i \le 2^i p_1$ when $b_i = 1$. Thus, $c_i = 2^i b_i p_1$.

Proposition 6. *Let p_0, p_1 be variables. Then, there is a valuation ρ such that $0 \leq \eta(p_0) = d_0, \rho(p_1) = d_1 < m$, and $\rho(p_0)\rho(p_1) = d \Leftrightarrow$ there is a valuation η such that η satisfies α, $\eta(p_0) = d_0$, $\eta(p_1) = d_1$, and $\eta(p) = d$ where $(p, \alpha) = \chi(p_0, p_1)$.*

$$\zeta(p') \triangleq \left(p, \begin{array}{c} 0 \leq a \leq 1 \\ \frac{m}{2}(a-1) \leq p' \leq \frac{m}{2}a - 1 \\ -ma \leq p + p' \leq ma \\ -m(1-a) \leq p - p' \leq m(1-a) \end{array} \right)$$

Fig. 8. Linear Constraints for Absolute Value

For modulo operations of terms, note

$$a \bmod b = a \bmod |b| = \begin{cases} |a| \bmod |b| & \text{if } a \geq 0 \\ (-|a|) \bmod |b| = |b| - (|a| \bmod |b|) & \text{if } a < 0. \end{cases}$$

We can therefore perform the modulo operations of terms by their absolute values. Consider the constraints $0 \leq a \leq 1$ and $\frac{m}{2}(a-1) \leq p' \leq \frac{m}{2}a - 1$ in Figure 8, where p' has the semantic value of any term. Intuitively, p' is non-negative if and only if $a = 1$. Suppose $p' \geq 0$ and $a = 0$. We would have $-\frac{m}{2} \leq p' \leq -1$, a contradiction. Conversely, $a = 1$ implies $0 \leq p' \leq \frac{m}{2} - 1$. Hence $p' \geq 0$.

Proposition 7. *Let p' be a variable. Then, there is a valuation ρ such that $-m \leq \rho(p') = d' \leq m$ and $|\rho(p')| = d \Leftrightarrow$ there is a valuation η such that η satisfies α, $\eta(p') = d'$, and $\eta(p) = d$ where $(p, \alpha) = \zeta(p')$.*

We can now describe the linear constraints for non-linear terms. For multiplication $t_0 \cdot t_1$, we first get the unsigned representation p'_0 and p'_1 of the semantic values of t_0 and t_1 respectively. This is done by the constraints $p'_0 = p_0 + ma$, $0 \leq p'_0 < m$, $p'_1 = p_1 + mb$, and $0 \leq p'_1 < m$ where $(p_0, \alpha_0) = \sigma(t_0)$ and $(p_1, \alpha_1) = \sigma(t_1)$ respectively. Then we compute the unsigned result p' by $\chi(p'_0, p'_1)$. Finally, the result is converted to the signed representation p by $p' - md = p$ and $-\frac{m}{2} \leq p < \frac{m}{2}$ (Figure 9).

To compute the semantic value of $t_0 \,\%\, t_1$, we first get the absolute values p'_0 and p'_1 of the semantic values of t_0 and t_1 by $\zeta(p_0)$ and $\zeta(p_1)$ respectively. The constraints $p'_0 - r = p'$ and $0 \leq p' < p'_1$ give $p' = |p_0| \bmod |p_1|$ where r is a multiple of $|p_1|$. Suppose $p_0 \geq 0$. Then $a = 1$ by the constraint $\frac{m}{2}(a-1) \leq p_0 \leq \frac{m}{2}a - 1$. Hence $p = p' = |p_0| \bmod |p_1|$ by the constraint $-2m(1-a) \leq p - p' \leq m(1-a)$. On the other hand, $p_0 < 0$ implies $a = 0$. Hence $p = p'_1 - p' = |p_1| - (|p_0| \bmod |p_1|)$ by the constraint $-ma \leq p - p'_1 + p' \leq 2ma$ (Figure 9).

$$\sigma(t_0 \cdot t_1) \triangleq \left(\begin{array}{rl} & \alpha_0 \\ & \alpha_1 \\ & \alpha_2 \\ & p'_0 = p_0 + ma \\ p, \quad & 0 \leq p'_0 < m \\ & p'_1 = p_1 + mb \\ & 0 \leq p'_1 < m \\ & p' - md = p \\ & -\frac{m}{2} \leq p < \frac{m}{2} \end{array} \right) \quad \begin{array}{rl} & (p_0, \alpha_0) = \sigma(t_0) \\ \text{where} & (p_1, \alpha_1) = \sigma(t_1) \\ & (p', \alpha_2) = \chi(p'_0, p'_1) \end{array}$$

$$\sigma(t_0 \,\%\, t_1) \triangleq \left(\begin{array}{rl} & \alpha_0 \\ & \alpha_1 \\ & \alpha_2 \\ & \alpha_3 \\ & \alpha_4 \\ p, \quad & p'_0 - r = p' \\ & 0 \leq p' < p'_1 \\ & 0 \leq a \leq 1 \\ & \frac{m}{2}(a-1) \leq p_0 \leq \frac{m}{2}a - 1 \\ & -2m(1-a) \leq p - p' \leq m(1-a) \\ & -ma \leq p - p'_1 + p' \leq 2ma \end{array} \right) \quad \begin{array}{rl} & (p_0, \alpha_0) = \sigma(t_0) \\ & (p_1, \alpha_1) = \sigma(t_1) \\ \text{where} & (p'_0, \alpha_2) = \zeta(p_0) \\ & (p'_1, \alpha_3) = \zeta(p_1) \\ & (r, \alpha_4) = \chi(p'_1, p'') \end{array}$$

Fig. 9. Linear Constraints for Non-linear Terms

Proposition 8. *Let t be a non-linear term in modular arithmetic. Then, there is a valuation ρ such that $[\![t]\!]_\rho = d \Leftrightarrow$ there is a valuation η such that η satisfies α and $\eta(p) = d$ where $(p, \alpha) = \sigma(t)$.*

Since the number of constraints and variables in the unsigned multiplication is $O(\omega)$, our translation requires $O(\omega)$ constraints and variables for non-linear terms. In summary, the satisfiability problem for modular arithmetic formula can be reduced to an instance of integer programming with $O(\omega|f|)$ constraints and variables.

Theorem 3. *Given a formula f in modular arithmetic over $\boldsymbol{Z}_m$ where $m = 2^\omega$, the satisfiability problem can be solved by an instance of the integer programming problem with the number of constraints and variables linear in $\omega|f|$.*

7 Applications

Our decision procedure may be useful in software verification, especially for programs in conventional programming languages. For hardware verification, our reduction may work as a non-linear constraint solver which accepts control signals from other decision procedures. Particularly, we find that the following areas may benefit from our algorithm.

Modern proof assistants allow external decision procedures to discharge proof obligations [13, 17, 10]. Although modular arithmetic is essential to many number theoretic and cryptographic algorithms, there is no proof assistant which provides decision procedures for modular arithmetic to the best of our knowledge. Since it is rather tedious to deal with modular arithmetic in each integral computation, verifiers simply assume the infinite-precision integer model in software verification. Subsequently, algorithms certified by proof assistants are not exactly the same as their implementations. Our procedure may help verifiers work in a more realistic computational model.

If a proof assistant is used to determine the truth values of predicates, abstract models constructed in predicate abstraction [7, 20] may be inadequate for the same reason. In the presence of non-linear modular arithmetic, our integer programming-based procedure may also be more efficient than, say, SAT-based technique used in predicate abstraction [5] (see Section 8). The new technique refines the abstraction and may perform better in such circumstances.

Another possible application of our algorithm is SAT-based model checking ([3], for instance). Our word-level decision procedure may be better for models with modular arithmetic, but it does not seem to fare well on Boolean satisfiability. However, modern integer programming packages support distributed computation [18]. Our approach gives a parallel SAT solver indirectly.

8 Experimental Results

We have implemented the algorithm to solve the satisfiability problem of modular arithmetic formulae. Our implementation generates instances of integer programming problems in the MPS format [14]. These files are then sent to the SYMPHONY package [18] as inputs. SYMPHONY is an open-sourced mixed integer programming solver. In addition to the conventional execution model, the SYMPHONY package also supports Parallel Virtual Machine [9]. We therefore conduct our experiments with both the uni- and multi-process versions. The uni-process version runs on an Intel Pentium 4 2.8GHz Linux 2.6.17 workstation with 2GB memory. The multi-process version runs on a PC cluster consisting of fifteen AMD Athlon MP 2000+ Linux 2.4.22 workstations with 1GB memory. For comparison, we repeat the experiments by the SAT solver zchaff on the workstation of the same configuration as the uni-process version.[3] We are interested in solving the following problems in $\mathbf{Z}_{256}$ (that is, $\omega = 8$).

i. $(x \cdot y = 143) \wedge (x \leq 143) \wedge (y \leq 143) \wedge ((x \neq 1 \wedge y \neq 1))$
ii. $x \cdot y \cdot z + y \cdot z + 2 \cdot x \cdot z + 2 \cdot z + 3 \cdot x \cdot y + 3 \cdot y + 6 \cdot x + 6 = 0$
iii. $x \cdot y \cdot z - y \cdot z - 2 \cdot x \cdot z + 2 \cdot z - 3 \cdot x \cdot y + 3 \cdot y + 6 \cdot x - 6 = 0$
iv. $((x \neq 0) \vee (y \neq 0) \vee (z \neq 0)) \wedge x \cdot y \cdot z + y \cdot z + 2 \cdot x \cdot z + 2 \cdot z + 3 \cdot x \cdot y + 3 \cdot y + 6 \cdot x = 0$
v. $((x \neq 0) \vee (y \neq 0) \vee (z \neq 0)) \wedge x \cdot y \cdot z - y \cdot z - 2 \cdot x \cdot z + 2 \cdot z - 3 \cdot x \cdot y + 3 \cdot y + 6 \cdot x = 0$

[3] Unfortunately, we have not conducted all the experiments in the same platform at the time of writing. Each workstation in our PC cluster is a bit outdated than the workstation used in the uni-version version.

Our first experiment is to factorize 143. Although it is easy to see that $11 \times 14 = 143$ is a solution, other solutions may be possible in $\mathbf{Z}_{256}$. Other experiments find roots to three-variable polynomials of degree three. In Experiment (ii) and (iii), the polynomials have constant terms. Hence their roots are always non-trivial. For polynomials without constant terms, trivial solutions can easily be found. We therefore look for non-trivial solutions in Experiment (iv) and (v).

	Uni-process		Multi-process	
Experiment	solution	time	solution	time
(i)	$x = 53, y = 51$	183.97	$x = 13, y = 11$	1.90
(ii)	-	> 600	$x = y = 0, z = 253$	0.39
(iii)	-	> 600	$x = y = 0, z = 3$	0.92
(iv)	$x = y = 0, z = 128$	0.84	$x = 42, y = 0, z = 6$	0.61
(v)	$x = y = 0, z = 128$	1.12	$x = y = 0, z = 128$	1.60

(a) with Integer Programming Package

Experiment	solution	time
(i)	x = 15, y = 129	0.19
(ii)	x = 64, y = 254, z = 255	2.08
(iii)	x = y = 0, z = 3	1.53
(iv)	x = y = 0, z = 128	1.52
(v)	x = y = 0, z = 128	1.52

(b) with SAT Solver

Fig. 10. Experimental Results

Figure 10 shows the solution and the user time (in seconds) for each experiment. The multi-process solver does improve the performance significantly. For example, the factorization is done in less than two seconds by the multi-process solver. But it takes more than three minutes with the uni-process solver. Another interesting observation is that the solutions are not necessarily obvious. The factorization found by the uni-process solver is somewhat unexpected. Instead of the unique factorization in $\mathbf{Z}$, we have $53 \times 51 \equiv 143$ in $\mathbf{Z}_{256}$. Similarly, the solution found by the multi-process solver in Experiment (iv) is correct only in $\mathbf{Z}_{256}$. These unexpected solutions are precisely the reasons why bugs may occur. On the other hand, the SAT solver performs rather stably. Although it may not always outperform the uni-process integer programming package, it does solve all problems in seconds. The multi-process integer programming package is able to finish and outperform the SAT solver in three of the five problems. More thorough experiments are still needed to compare both techniques.

9 Conclusion

Deciding the satisfiability of modular arithmetic formula is essential in software verification. We have characterized the complexity of its satisfiability problem and provided an efficient reduction to the integer programming problem. Our result shows that it is more efficient to develop specialized algorithms than apply more general algorithms for Presburger arithmetic. Additionally, the number of constraints and variables is linear in the length of the input formula in our reduction. With heuristics like relaxation and rounding, the satisfiability problem could be solved efficiently by modern integer programming packages in practice.

It would be interesting to compare our algorithm with other techniques [4, 21, 1], especially those with the binary encoding scheme. Since the satisfiability problem of modular arithmetic formula is **NP**-complete, one could also build a decision procedure based on SAT solvers. But the binary encoding would eliminate the mathematical nature of the problem. Although our preliminary experimental results suggest that our approach may be useful in finding solutions to multi-variant low-degree polynomials, it is unclear which approach will prevail in practice.

There are still a few missing pieces in our construction. Our translation of the unsigned multiplication is not satisfactory. It would be more useful if our construction used only $O(\lg \omega)$ variables and constraints. Additionally, the quotient and remainder operations of arbitrary terms are not allowed. Although it is possible to encode them in modular arithmetic formula, an efficient construction similar to non-linear terms is certainly welcome.

Acknowledgement. The author would like to thank anonymous referees for their constructive comments in improving the paper.

References

1. Babić, D., Musuvathi, M.: Modular arithmetic decision procedure. Technical Report MSR-TR-2005-114, Microsoft Research (2005)
2. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In Alur, R., Peled, D.A., eds.: Computer Aided Verification. Volume 3114 of LNCS., Springer-Verlag (2004) 515–518
3. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: Design Automation Conference, ACM Press (1999) 317–320
4. Boigelot, B., Wolper, P.: Representing arithmetic constraints with finite automata: An overview. In Stuckey, P.J., ed.: International Conference on Logic Programming. Volume 2401 of LNCS., Springer-Verlag (2002) 1–19
5. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. Formal Methods in System Design **25**(2–3) (2004) 105–127
6. Cooper, D.C.: Theorem proving in arithmetic without multiplication. Machine Intelligence **7** (1972)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM Symposium on Principles of Programming Languages. (1977) 238–252

8. Enderton, H.: A Mathematical Introduction to Logic. Academic Press (1972)
9. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.: PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing. The MIT Press (1994)
10. Huet, G., Kahn, G., Paulin-Mohring: The Coq proof assistant: a tutorial: version 6.1. Technical Report 204, Institut National de Recherche en Informatique et en Automatique (1997)
11. Hungerford, T.W.: Algebra. Volume 73 of Graduate Texts in Mathematics. Springer-Verlag (1980)
12. Knuth, D.E.: The Art of Computer Programming. Volume II, Seminumerical Algorithms. Addison-Wesley (1997)
13. Melham, T.F.: Introduction to the HOL theorem prover. University of Cambridge, Computer Laboratory. (1990)
14. Murtagh, B.A.: Advanced Linear Programming: Computation and Practice. McGrawHill (1981)
15. Oppen, D.C.: Elementary bounds for presburger arithmetic. In: ACM Symposium on Theory of Computing, ACM (1973) 34–37
16. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1994)
17. Paulson, L.C., Nipkow, T.: Isabelle tutorial and user's manual. Technical Report TR-189, Computer Laboratory, University of Cambridge (1990)
18. Ralphs, T.K., Guzelsoy, M.: The SYMPHONY callable library for mixed integer programming. In: INFORMS Computing Society. (2005)
19. Reddy, C.R., Loveland, D.W.: Presburger arithmetic with bounded quantifier alternation. In: ACM Symposium on Theory of Computing, ACM (1978) 320–325
20. Saídi, H., Graf, S.: Construction of abstract state graphs with PVS. In Grumberg, ed.: Computer Aided Verification. Volume 1254 of LNCS., Springer Verlag (1997) 72–83
21. Seshia, S.A., Bryant, R.E.: Deciding quantifier-free presburger formulas using parameterized solution bounds. In: Logic in Computer Science, IEEE Computer Society (2004) 100–109
22. Stinson, D.R.: Cryptography: Theory and Practice. CRC Press, Inc (1995)
23. Wang, B.Y.: On the satisfiability of modular arithmetic formula. Technical Report TR-IIS-06-001, Institute of Information Science, Academia Sinica (2006) http://www.iis.sinica.edu.tw/LIB/TechReport/tr2006/tr06.html.