Modular Arithmetic Decision Procedure

Id: madp.tex,v 1.13 2005/07/18 22:38:51 root Exp

Domagoj Babić
Computer Science Department, University of British Columbia, Vancouver, Canada
babic@cs.ubc.ca

Madanlal Musuvathi
Microsoft Research, Redmond, WA, USA
madanm@microsoft.com

Abstract

All integer data types in programs (such as int, short, byte) have an underlying finite representation in hardware. This finiteness can result in subtle integer-overflow errors that are hard to reason about both for humans and analysis tools alike. As a first step towards finding such errors automatically, we will describe two modular arithmetic decision procedures for reasoning about *bounded* integers.

We show how to deal with modular arithmetic operations and inequalities for both linear and non-linear problems. Both procedures are suitable for integration with Nelson-Oppen framework [1, 2, 3]. The linear solver is composed of Müller-Seidl algorithm [4] and an arbitrary integer solver for solving preprocessed congruences and inequalities.

For the non-linear problems we use Newton's p-adic iteration algorithm to progressively reason about the satisfiability of the input constraints modulo 2^k, for increasing k. We use a SAT solver only for the base case when $k = 1$. According to our knowledge, this is the first Nelson-Oppen decision procedure capable of reasoning about multiplication over bounded integers without converting the entire problem to a SAT instance.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; I.2.3 [*Artificial Intelligence*]: Deduction and Theorem Proving

General Terms Algorithms, Verification

Keywords Modular arithmetic, Automated decision procedures, Software verification

1. Introduction

Integer data types in programs are represented using a finite number of bits in the underlying hardware and thus, have a finite range. Their finite representation can result in *integer-overflows* when the result of arithmetic operations on these data types lies outside their range. While it is convenient to neglect these overflows for most programming tasks and treat the data types as essentially unbounded, unexpected integer-overflows are frequent in many programs. Given that most programmers easily forget to reason about finite nature of basic data types, it is absolutely imperative for program analysis tools *not* to neglect such behavior. When undetected, these overflows can result in serious security vulnerabilities.

As a motivating example of such behavior, Listing 1 shows a snippet of code for the JPEG image processing routine. The function `read_jpeg` takes as argument a pointer to a JPEG image. The macro INPUT_2BYTES decodes the length of the image and stores it in the variable `length`. The function then allocates a buffer of size `length+2` at line `6`, and calls a function at line `12` that copies `length-2` bytes from the image to the buffer. Before the call to this function, there is an (implicit) assert that checks if the buffer has sufficient space. At first sight, it appears that this assert can never fail. However, when `length` is 0, a buffer of size 2 is allocated at line `6`. The subtraction at line `9` causes an underflow, so 0xFFFE (65534) bytes are copied, overflowing the buffer. This fails the assertion.

```
 1  BOOL read_jpeg(IN ptr cinfo) {
 2      VOID *pBuffer;
 3      UINT16 length;
 4
 5      INPUT_2BYTES(cinfo, length);
 6      pBuffer = GpMalloc(length + 2);
 7      // assume(pBuffer.length == length+2);
 8      ...
 9      INT size = length - 2;
10      ...
11      // assert(size < pBuffer.length);
12      GpMemcpy(pBuffer, cinfo->nxt, size);
13  }
```

Listing 1. JPEG integer overflow bug

When undetected, the overflow in the function in Listing 1 causes a buffer overrun, which is a serious security vulnerability. This particular vulnerability was recently exploited [1] and required a costly corrective action. A number of other critical bugs has been related to integer underflows/overflows. Just to mention a few: OpenSSH buffer overflow[2], Microsoft JScript bug[3], FreeBSD system call buffer overflow[4], Snort TCP packet reassembly integer overflow[5], Apache chunked-encoding vulnerability[6]

As exploits based on integer-overflows are becoming more prevalent, it is very important to build program-analysis tools that

[1] http://www.microsoft.com/technet/security/Bulletin/MS04-028.mspx
[2] http://www.securityfocus.com/bid/5093
[3] http://www.microsoft.com/technet/security/bulletin/MS03-008.mspx
[4] http://www.securityfocus.com/bid/5493
[5] http://www.securityfocus.com/bid/7178
[6] http://www.securityfocus.com/bid/5033

can detect overflow related errors automatically. However, the current state-of-the-art analysis tools (such as [5, 6]) fall short of doing so. This is primarily because the theorem provers and symbolic reasoning engines that underly these tools can only reason about (unbounded) integers and thus are unable to detect overflow behavior. For instance, when analyzing the function in Listing 1 these tools incorrectly claim that the assertion is always true,[7] resulting in unsound analysis.

As a first step towards finding integer-overflow based errors automatically, this paper proposes two decision procedures for *bounded* integers. The first one is for linear and the second for non-linear modular arithmetic. Using linear decision procedure, we later show that a simple verification-condition generating tool [7] can detect the error in Listing 1.

An obvious way to detect integer-overflow based errors is of course is to detect *all* integer-overflows in the program and flag each as an error. This can be done, for instance, by using a value-range analysis [8] to obtain conservative upper and lower bounds for each program variable and then checking if each arithmetic operation produces a result within the range of the integer data type. However, such an analysis produces too many false warnings as not all overflows result in an error. By reasoning about modular arithmetic explicitly, our decision procedures can be used to check only for those overflows that lead to an assertion failure.

Moreover, many low-level systems programs *rely* on the overflow behavior of the integer data types for their correct execution. A very good example is the Transport Control Protocol (TCP), which uses 32-bit sequence numbers for reliable data transfer. The protocol specification [9, 10] allows these sequence numbers to wrap around, primarily to support the transfer of very large files. Accordingly, any protocol implementation has to explicitly reason about integer-overflows. The decision procedures described in this paper can be used to build program analysis tools that can detect errors in such programs.

While there are other decision procedures for reasoning about bounded integers in the research literature, this paper significantly improves upon previous work in the following novel ways. First, our linear decision procedure is the first to support the symbolic reasoning of *inequalities*. This is crucial as many of the program invariants, including array bounds checking, involve inequalities between program expressions.

Second, we also propose a decision procedure that supports multiplication. To reason about the resulting non-linear expressions, the decision procedure uses Newton's p-adic iteration to progressively reason about the satisfiability of the input constraints modulo 2^k, for increasing k. For the base case ($k = 1$), all variables are restricted to a single bit and the decision procedure uses a SAT solver check satisfiability. Note that reasoning about multiplication over unbounded integers is undecidable [11], so many verification tools conservatively treat multiplication as an uninterpreted function.

Finally, we show how our decision procedures can be used in a Nelson-Oppen combination framework [2, 3]. This allows one to combine our decision procedures with other procedures for reasoning about uninterpreted functions, arrays, and lists. The resulting combination can support a wide variety of program analysis tasks.

1.1 Related Work

Our work is most related to that of Müller-Olm and Seidl [4], in which the authors provide a method to infer linear congruences invariant at a program point. However, their algorithm cannot be used for solving inequalities and non-linear congruences.

[7] This is because `length-2` $<$ `length+2` holds when `length` $\in \mathcal{Z}$.

Many have studied the theory of fixed-width bit-vectors (such as [12, 13]), which can be used to model arithmetic operations on integer data types. However, this body of work has focussed on hardware verification and does not support reasoning about inequalities. As an extreme, CBMC [14] is a tool that converts an ANSI-C program into a Boolean circuit and then use a SAT solver to check for violation of assertions in the program. These methods lose the structure of the program by splitting each integer variable in the program to a sequence of bits and as a result, do not scale. Furthermore, even the problems that can be solved in polynomial time by Gaussian elimination might take exponential time with the number of variables when a DPLL SAT solver is used.

2. Basic Definitions

This section defines the basic notions that will be used in further exposition. *Bit-vectors* are defined as arrays of bits. Slightly abusing the terminology, by *bounded integers* we will assume the elements of the ring of integers. *Unbounded integers* are elements of the set of integers $\mathbb{Z}$. By bit-vector operators we mean logic operators, subvector extraction, concatenation, left/right shift, and one's complement (bitwise complement). All operators are defined in little-endian arithmetic – higher value bits are stored at a higher address. We will distinguish signed and unsigned bounded integers. If the type is not stated explicitly, unsigned bounded integers will be assumed.

If two integers a, b have the same remainder r upon division by the natural number m, where $0 \leq r < m$, then a and b are said to be *congruent* modulo m, written as $a \equiv b \mod m$. Congruence naturally extends to polynomials. The following are equivalent

$$\begin{gathered} a \equiv b \mod m \\ a = b + mt,\ \forall t \in \mathbb{Z} \qquad (1) \\ m | a - b \end{gathered}$$

Equation of the form $f(x_0, \ldots, x_n) \equiv 0 \mod m$ is called a *congruence*. A system of congruences is a system of such equations. A system is said to be *satisfiable* if there exists an assignment to all variables such that all congruences mod m in the system evaluate to 0, otherwise it is said to be *unsatisfiable*. *Complete system* of equations is a system of m linearly independent equations over n variables such that $n = m$. If $n > m$ the system is said to be *incomplete*. Solutions of a system of equations $A \cdot \vec{x} = \vec{b}$ are composed of *particular* solutions and solutions of the system $A \cdot \vec{x} = \vec{0}$, called *homogeneous* solutions. For more details see any introductory book on linear algebra.

We say that an integer a *divides* an integer b if there exists an integer c such that $a \cdot c = b$, usually denoted as $a \mid b$. *P-adic expansion* was introduced by Hensel. Given any prime $p \in \mathbb{Z}$, each integer $a \in Z$ has a unique (up to the first leading zero term) finite p-adic expansion:

$$a = \sum_{i}^{k} a_i p^i, \quad 0 \leq a_i < p$$

A *ring* is a set $\mathcal{R}$ with two binary operations $\cdot, +$ (commonly interpreted as addition and multiplication) satisfying additive and multiplicative associativity, additive commutativity, left and right distributivity, and existence of additive identity and inverse. A *commutative ring* also satisfies multiplicative commutativity. Ring modulo m is denoted as $\mathbb{Z}_m$. For more details see [15]. Modular inverse of $a \in \mathbb{Z}_m$, if it exists, is $b \in \mathbb{Z}_m$ such that $a \cdot b \equiv 1 \mod m$. A zero-divisor is $a \in \mathbb{Z}_m$, such that:

- there exists $b \in \mathbb{Z}_m$

- neither of the elements is zero
- $a \cdot b \equiv 0 \mod m$

Difference Constraints are constraints of the form $x - y \bowtie c$ and $x \bowtie c$, where x, y are variables, c a constant, and $\bowtie \in \{=, \neq, <, >, \leq, \geq\}$. *UTVPI* constraints are slightly more general $a \cdot x + b \cdot y \bowtie c$, such that $a, b \in \{0, 1\}$ [16].

A *singular matrix* is a square matrix that does not have a matrix inverse over a commutative ring. Let $f_0, \ldots, f_{m-1}$ be polynomials in n variables $x_0, \ldots, x_{n-1}$. The *Jacobian* J is a square matrix of partial derivatives, defined as

$$J = \begin{bmatrix} \frac{\partial f_0}{\partial x_0} & \cdots & \frac{\partial f_0}{\partial x_{n-1}} \\ \vdots & & \vdots \\ \frac{\partial f_{n-1}}{\partial x_0} & \cdots & \frac{\partial f_{n-1}}{\partial x_{n-1}} \end{bmatrix}$$

Given an incomplete system of congruences, the Jacobian will not be a square matrix, but can be extended to one appending rows of zeros. Such a matrix is singular and has no inverse.

Signed and unsigned integers used in programming are bounded. 32-bit unsigned integers have the range $\{0, .., 2^{32} - 1\}$, while signed integers have the range $\{-2^{31}, .., 0, .., 2^{31} - 1\}$. Contemporary software formal verification tools reason about multiplication by translating each single bit to a boolean variable and encoding multipliers directly. The problem is then given to a SAT solver. A disadvantage of this approach is the loss of the natural problem structure. Alternative approach is to represent multiplication as an uninterpreted function, loosing precision on the way.

Recently, Müller-Olm and Seidl [4] have presented interesting results on software analysis based on modular arithmetic. Proposed algorithm deals only with linear congruences. The authors claim that their approach is efficient as expensive computation of inverses is avoided. Although the advantage of their approach has been somewhat decreased by the discovery of efficient algorithms for the computation of inverses modulo 2^k [17], we will still consider the Müller-Seidl algorithm as the basic algorithm for solving linear congruences.

3. Decision Procedure for Linear Congruences

In this section we present a decision procedure for linear modular arithmetic with inequalities that makes it possible to reason about overflows in a mathematically clean way. Linear congruences can be solved with Müller-Seidl algorithm [4]. The congruence solver alone does not suffice because the ordering of elements (inequality) is not defined on the rings of integers. Dumping everything on an integer solver wouldn't work either because there is no notion of overflow in Presburger arithmetic.

In our approach we simulate modular arithmetic with arithmetic over unbounded integers with an additional set of range constraints that bound the range of each variable. The congruence solver is used to preprocess the congruences in polynomial time. Results can be either parametric or concrete solutions over the set of integers. As the solutions are expressed over $\mathbb{Z}$, an integer solver can be used to solve the set of constraints consisting of preprocessed congruences and inequalities. Although none of the decision procedures on its own can reason about both overflows and inequalities, the two together can.

This approach has several advantages. First, a linear system of congruences is solved in polynomial time, while integer solvers might take exponential time in the worst case. Second, this combination yields a mathematically clean way to reason about overflows.

Müller-Seidl algorithm is based on a modified Gaussian elimination. The goal is to bring the coefficient matrix to a triangular form through a series of elementary and invertible transformations. Carefully choosing the pivot, the algorithm avoids computing modular inverses. Given an $n \times n$ linear system of congruences $A \cdot \vec{x} \equiv \vec{b} \mod 2^k$, the algorithm computes all solutions in $\mathcal{O}\left(k \cdot n + \log(k) \cdot n^3\right)$. Elements of the ring $\mathbb{Z}_{2^k}$ can have zero divisors, thus homogeneous system $A \cdot \vec{x} \equiv 0 \mod 2^k$ can have non-trivial solutions. All homogeneous solutions can be found in time $\mathcal{O}\left(\log(k) \cdot n^3\right)$.

Given a satisfiable and linearly independent set of n equations with n variables from $\mathbb{Z}_m$, solutions can be expressed as a linear combination of particular solutions p_i and homogeneous solutions h_i (Eq. 2). Solutions of an incomplete system can be expressed in parametric form. If the system of congruences is complete, congruence solver will return a set of concrete solutions out of which each consists of a particular part (p_i) and a homogeneous part (h_i). By introduction of a decision variable $\delta = \{0, 1\}$, each concrete solution can be represented as a linear equality:

$$x_i = p_i + \delta_i \cdot h_i \tag{2}$$

Solving incomplete systems yields a set of parametric solutions E_s of the form:

$$\begin{array}{rcl} y_1 & = & \sum_i c_{1,i} \cdot x_i + t_1 \cdot m \\ \cdots & \cdots & \cdots \\ y_n & = & \sum_i c_{n,i} \cdot x_i + t_n \cdot m \end{array}$$

Dependant variables y_i are expressed as a linear combination of independent variables x_i. As explained previously (Eq. 1), congruences modulo m can be represented as equalities with an additional factor $t_i \cdot m$. The bounds of slack variables t_i can be limited without loss of generality. Essentially, the slack variables represent the number of wraparounds and can be computed in linear time with the number of additive terms in the parametric solutions. For example, given $y, x_1, x_2 \in \mathsf{uint}_{32}$ and a parametric solution $y = 11 \cdot x_1 + 5 \cdot x_2 + t \cdot 2^{32}$, it follows that $-15 \leq t \leq 0$. In general, the bounds of slack variables can be larger than the ranges of regular variables.

All dependant and independent variables are restricted according to their type with a range constraint given as a predicate. We will represent the set of range predicates with P. For example, for an unsigned 32-bit integer $x : \mathsf{uint}_{32}$ corresponding range constraint would be $0 \leq x < 2^{32}$.

The Müller-Seidl algorithm takes a set of congruences E and returns UNSAT if the system is unfeasible, a set of parametric solutions E_s if the system is incomplete, or a set of concrete solutions otherwise. We extend the basic algorithm with a postprocessor that computes bounds of slack variables for incomplete systems. The set of range predicates and t_i bounds is denoted as P_r.

The algorithm is given in Listing 1. Unless the Müller-Seidl algorithm returns UNSAT, the set $E_\mathrm{s} \cup P$ is satisfiable, but $E_\mathrm{s} \cup P_\mathrm{r} \cup D$, where D is a set of inequalities over dependant and independent variables, might not be. If the number of concrete solutions is small (ie. if there is a small number of non-trivial homogeneous solutions), all concrete points are evaluated against the set of inequalities $D \cup P$ in linear time $\mathcal{O}\left(|D \cup P|\right)$. The value of that "small number" is determined heuristically. Although all solutions of a complete system can be found in the polynomial time, in the worst case, the number of concrete solutions can be exponential. If the number of non-trivial homogeneous solutions is large or if the result is a set of parametric solutions, a new set of constraints $EQ = E_\mathrm{s} \cup P_\mathrm{r} \cup D$ is constructed and passed to an integer solver.

Algorithm 1 Linear modular arithmetic decision procedure

```
 1: procedure SOLVE($E$,$P$,$D$)
 2:     $E_\mathrm{s} = $ MÜLLER-SEIDL($E$)
 3:     if $E_\mathrm{s} \neq$ UNSAT then
 4:         $P_\mathrm{r} = P \cup \{$ bounds of $t_i\}$
 5:         if The number of concrete solutions is small then
 6:             while $E_\mathrm{s} \neq \emptyset$ do
 7:                 pick $s \in E_\mathrm{s}$
 8:                 $E_\mathrm{s} \leftarrow E_\mathrm{s} \backslash \{s\}$
 9:                 if $s$ satisfies $D \cup P_\mathrm{r}$ then
10:                     return $s$
11:                 end if
12:             end while
13:         else
14:             $EQ = E_\mathrm{s} \cup P_\mathrm{r} \cup D$
15:             return INTSOLVE($EQ$)
16:         end if
17:     end if
18:     return UNSAT
19: end procedure
```

EXAMPLE 1. *Let us consider the formalization of the JPEG integer overflow bug in listing 1. Variable* `size` *is denoted as s, and* `length` *as l. Although s is a 32-bit signed integer variable, the right value of the assignment expression can be only a 16-bit unsigned integer. The range of s is adjusted accordingly. These adjustments might not be straightforward in all cases. Later we present a more general way to deal with multiple types – congruences with multiple moduli.*

$$\begin{aligned} s &\equiv l-2 \mod 2^{16} \\ s &\geq l+2 \\ 0 &\leq l, s < 2^{16} \end{aligned}$$

A parametric solution is $s = l - 2 + t \cdot 2^{16}, \forall t \in \mathbb{Z}$. Then we compute the range of t to restrict the number of cases the integer solver needs to consider. In this case, the range of t is $0 \leq t \leq 1$. There are two solutions that satisfy all constraints, $s = 0xFFFF(65535), l = 1, t = 1$ and $s = 0xFFFE(65534), l = 0, t = 1$. Finding one of them proves the existence of the counterexample.

It might not be obvious that a set of congruences and a set of equalities with slack variables, resulting from converting congruences into equalities, are equisatisfiable. The equisatisfiability property means that both set of constraints have exactly the same set of solutions, or no solution at all. This property is crucial for the soundness of our method and can be easily proven. Although it suffices to prove equisatisfiability for the set of parametric solutions with dependant and independent variables, we present a more general proof that holds for any set of constraints.

THEOREM 1 (Equisatisfiability). *Given $\vec{f} = [f_0, \ldots, f_{n-1}]$, $\vec{x} = [x_1, \ldots, x_{n-1}]$, and $\vec{t} = [t_1, \ldots, t_{n-1}]$ the two sets of constraints $\vec{f}(\vec{x}) \equiv 0 \mod m$ and $\vec{f}(\vec{x}) = \vec{t} \cdot m$ are equisatisfiable.*

Proof 1. *Assume that $\vec{x_0}$ is a solution of $\vec{f}(\vec{x}) \equiv 0 \mod 0$, then each $f_j(\vec{x_0}), 1 \leq j < n$ evaluates to some multiple of m, which is congruent to 0 mod m. Hence, there exists a $\vec{t_0}$ such that $\vec{f}(\vec{x_o}) = \vec{t_0} \cdot m$, so for each solution of the set of congruences there is a solution of the corresponding system of equalities with slack terms. In the other direction, if $\vec{x_0}$ is a solution of $\vec{f}(\vec{x}) = \vec{t} \cdot m$, obviously it satisfies the set of congruences $\vec{f}(\vec{x_0}) \equiv 0 \mod m$. It follows that if one set of constraints has a solution, so does the other. By contrapositive it follows that if one is unsatisfiable, so is the other.* □

3.1 Optimizations and Extensions of the Basic Algorithm

The decision procedure (Listing 1) can be optimized by using incrementally more expensive decision procedures before calling the integer solver in order to try to prove unsatisfiability as early as possible. This approach corresponds to layered decision procedures [18]. After line 14, a call to a solver for difference constraints could be added. Difference constraints can be solved in polynomial time by detection of negative cycles. If the set of difference constraints is unsatisfiable, the algorithm can return UNSAT. If the unsatisfiability cannot be proven by solving difference constraints, we still might want to try replacing all integer variables with rational ones and call a solver (Simplex) for linear constraints over rationals. This can still be cheaper in practice than immediately calling an integer solver.

Logic operators are often used in software, although not as often as in hardware design. A decision procedure for modular arithmetic and inequalities must also support logic operators to be practical. Logic operators and conditionals can be encoded as linear integer programming problems (see pg. 232, [19]). Bit vector operators can be encoded by introduction of a fresh variable for each single bit. When it is necessary to reason about bits, this seems unavoidable.

4. Decision Procedure for Non-linear Congruences

In this section we will present a decision procedure for non-linear congruences which does not depend on the integer solver. A SAT solver is used only for solving the base case. The procedure returns concrete solutions that need to be checked against inequalities – and that can be done in linear time with the number of inequalities. It is essentially a depth-first search procedure based on Newton's p-adic iteration. In its elementary form, *Newton's iteration* is given as

$$x_k = x_{k-1} - f'(x_{k-1})^{-1} \cdot f(x_{k-1})$$

Newton's iteration formula on the space of p-adic expansion has been successfully applied to factorization of polynomials, computation of the greatest common divisor of polynomials, polynomial division, and partial fraction decomposition [20]. When Newton's iteration is applied to factorization of polynomials it is called Hensel's lifting [15]. The foundations of the work presented in this paper are based on its application to interpolation of univariate and multivariate polynomials [21].

To derive the polynomial iteration formula we need to start with Taylor's expansion:

$$\begin{aligned} f(x) &= \frac{f^{(n)}(r)}{n!} \cdot (x-r)^n + \cdots + \frac{f''(r)}{2!} \cdot (x-r)^2 + \\ &\quad + f'(r) \cdot (x-r) + f(r) = \\ &= f_n \cdot (x-r)^n + \cdots + f_2 \cdot (x-r)^2 + \\ &\quad + f_1 \cdot (x-r) + f_0 \end{aligned}$$

If $\mathcal{R}$ is an arbitrary (commutative with unity) ring and $f \in \mathcal{R}[x]$ is a polynomial of degree at most n, *Taylor's expansion* of f around $r \in \mathcal{R}$ is given below. For $f \in \mathbb{Z}[x]$ and $n \in \mathbb{Z}$ all factors $\frac{f^{(i)}(n)}{i!}$ are integers. The proof is available in the literature [22]. Polynomials have a unique p-adic representation:

$$\begin{aligned} f(x) &= \sum_i f_i x^i = \sum_i \left(\sum_j f_{ji} p^j \right) x^i = \\ &= \sum_j \left(\sum_i f_{ji} x^i \right) p^j = \sum_j F_j p^j \end{aligned}$$

and can be approximated with lower order terms as follows

$$f(x) \equiv F_k \cdot p^k + \cdots + F_2 \cdot p^2 + F_1 \cdot p + F_0 \mod p^{k+1}$$

Intuitively, Newton's p-adic iteration constructs more and more precise approximations from the initial solution in $x_1 \in \mathbb{Z}_p$. In programming, polynomial coefficients, as well as the variables themselves, are most often in $\mathbb{Z}_{16}$, $\mathbb{Z}_{32}$, or $\mathbb{Z}_{64}$. Hence, only a small finite number of iterations is needed to compute the exact solution.

Newton's iteration formula on the space of p-adic expansion can be used for solving non-linear congruences. As we are interested in congruences modulo 2^k, we will use 2-adic expansion. Solving a system of equations over 32-bit integral variables is equivalent to solving the corresponding system of congruences mod 2^{32}. First, the system is solved mod 2, meaning that the least significant bit is solved first. If there is no solution in $\mathbb{Z}_2$, the entire system is unsatisfiable. Otherwise, the algorithm lifts solutions one by one in a depth-first search manner.

It is well known that solving multivariate non-linear congruences over the ring $\mathbb{Z}_2$ is $\mathcal{NP}$-complete [23]. We will assume that a SAT solver is used for solving the base case. Each bit will correspond to a boolean variable with the usual interpretation $\sigma(1) = \mathsf{TRUE}$ and $\sigma(0) = \mathsf{FALSE}$. Base case solutions are iteratively lifted. Let us consider solving a univariate non-linear congruence $f(x) \equiv 0 \mod p^K$ to illustrate the technique. Assume that a solution $r_{k-1} \in \mathbb{Z}_{p^k}, k < K$ is known. Obviously, $f(x)\big|_{x=r_{k-1}} \equiv 0 \mod p^k$. Then it also follows that $f(x)\big|_{x=r_{k-1}} \equiv 0 \mod p^{k-1}$, so solutions are given as $r_{k-1} + t \cdot p^{k-1}$. Now the p-adic expansion of the polynomial can be rewritten as

$$\begin{aligned} f(r_{k-1} + t \cdot p^{k-1}) &\equiv f(r_{k-1}) + f'(r_{k-1}) \cdot t \cdot p^{k-1} + \\ &\quad + f''(r_{k-1}) \cdot (t \cdot p^{k-1})^2 + \ldots \\ &\quad + f^{(k-1)}(r_{k-1}) \cdot (t \cdot p^{k-1})^n \text{ mod } p^k \\ f(r_{k-1} + t \cdot p^{k-1}) &\equiv 0 \\ f(r_{k-1}) + f'(r_{k-1}) \cdot t \cdot p^{k-1} &\equiv 0 \mod p^k \end{aligned}$$

$$t \equiv -f'(r_{k-1})^{-1} \cdot \frac{f(r_{k-1})}{p^{k-1}} \mod p \tag{3}$$

Higher order terms are congruent to 0 because $p^k \mid p^{i \cdot (k-1)}, \forall i : 2 \le i \le n$. Final result (Eq. 3) follows from a well known rule for solving congruences saying that both sides of the congruence and the modulo can be divided by the same number. Solution r_k is therefore given by the recurrence

$$r_k \equiv r_{k-1} + \left[-f'(r_{k-1})^{-1} \cdot \left(\frac{f(r_{k-1})}{p^{k-1}} \right) \text{ mod } p \right] \cdot p^{k-1} \text{ mod } p^k \tag{4}$$

The lifting is repeated until the target ring $\mathbb{Z}_{p^K}$ is reached or the system becomes unsatisfiable. Eq. 4 can be naturally extended to multiple multivariate polynomials. Define $\vec{f}$, $\vec{x}$, and $\vec{t}$ as transposed $\vec{f} = [f_0, \ldots, f_{m-1}]$, $\vec{x} = [x_1, \ldots, x_{n-1}]$, and $\vec{t} = [t_1, \ldots, t_{n-1}]$, then the recurrence can be written as

$$\vec{x}_k \equiv \vec{x}_{k-1} + \left[-J(\vec{x}_{k-1})^{-1} \cdot \left(\frac{\vec{f}(\vec{x}_{k-1})}{p^{k-1}} \right) \text{ mod } p \right] \cdot p^{k-1} \text{ mod } p^k$$

Every solution in $\mathbb{Z}_{p^k}$ is also a solution in $\mathbb{Z}_{p^{k-1}}$. This observation can be used to optimize the linear iteration. Instead of computing the inverse of Jacobian for every point, it is sufficient to compute it only for the initial points:

$$\begin{aligned} x_k &\equiv x_1 \mod p \\ &\Rightarrow \\ J(\vec{x}_{k-1}) &\equiv J(\vec{x}_1) \mod p \end{aligned}$$

$$\vec{x}_k \equiv \vec{x}_{k-1} + \left[-J(\vec{x}_1)^{-1} \cdot \left(\frac{\vec{f}(\vec{x_{k-1}})}{p^{k-1}} \right) \text{ mod } p \right] \cdot p^{k-1} \text{ mod } p^k$$

Three cases need to be distinguished. *(i)* If the inverse of Jacobian is computable $\mathrm{mod}\ p$, the system has a unique solution $\vec{x}_{k-1} + \vec{t} \cdot p^{k-1}, 0 \le t_i < p$. *(ii)* If there is no inverse and the second factor evaluates to zero, solutions are given as $\vec{x}_{k-1} + \vec{t} \cdot 2^{k-1}, \forall t_i : 0 \le t_i < p$. All solutions are candidates for the starting points for the next iteration. *(iii)* Otherwise, the starting point $\vec{x}_{k-1}$ was a bad choice and there is no solution. In general, n and m do not need to be equal. If $n \neq m$, the Jacobian will always be singular and it suffices to compute $\vec{f}(\vec{x}_{k-1})$.

EXAMPLE 2. *The following example will illustrate how to solve a simple non-linear congruence using Newton's iteration. Solutions mod 2 are computed by some other means, for example with a SAT solver. Solution candidates in $\mathbb{Z}_{2^{k+1}}$, resulting from lifting solutions from previous iteration, is denoted as X_k.*

$$\begin{aligned} 3yx^2 + 7x &\equiv 0 \mod 16 \\ 2xy + 13y^2 + 3 &\equiv 0 \mod 16 \end{aligned}$$

It follows that $x_1 = \{[0,1],[1,1]\}$. Jacobian is

$$\begin{bmatrix} 6xy + 7 & 3x^2 \\ 2y & 2x + 26y \end{bmatrix}$$

and has no inverse mod 2 in the starting points. Both points need to be lifted to $\mathbb{Z}_4$, yielding $X_1 = \{[0,1],[0,3],[2,1],[2,3],[1,1],[1,3],[3,1],[3,3]\}$. Out of these, only two are solutions in $\mathbb{Z}_4$, namely $x_2 = \{[0,1],[0,3]\}$. Again, both solutions can be lifted resulting in $X_2 = \{[0,1],[0,5],[4,1],[4,5],[0,3],[0,7],[4,3],[4,7]\}$. Checking X_2 for solutions in $\mathbb{Z}_8$ gives $x_3 = \{[0,1],[0,3],[0,5],[0,7]\}$. Repeating the iteration for one more step finds the solutions of the system $x_4 = \{[0,1],[0,7],[0,9],[0,15]\}$.

When lifting an unsigned solution x_k to the ring $\mathbb{Z}_{2^{k+1}}$ it suffices to consider only solutions $0 \le x_k < 2^{k+1}$. Lifting of signed bounded integer solutions differs only in the range that needs to be considered $-2^k \le x_k < 2^k$. For linear iteration there are only two possible values for each variable in x_k that need to be considered at each step. If higher order iteration schemes are used, the number of cases grows exponentially.

Now, let us consider solving a base case of a typical non-linear congruence

$$\begin{aligned} 3x^2y + 7x &\equiv 0 \mod 2 \\ 2xy + 13y^2 + 3 &\equiv 0 \mod 2 \end{aligned}$$

According to Fermat's little theorem

$$a^p \equiv a \mod p$$

any exponent can be iteratively reduced to 1. All polynomial coefficients can be reduced too. Hence, we obtain

$$\begin{aligned} xy + x &\equiv 0 \mod 2 \\ y + 1 &\equiv 0 \mod 2 \end{aligned}$$

These congruences can be reduced to a boolean satisfiability problem of the form $\neg((x \wedge y) \oplus x) \wedge (y)$, where $\oplus$ is logical XOR. Replacing the literal $x \wedge y$ with z we get $\neg(z \oplus x) \wedge (y) \wedge (z \Leftrightarrow (x \wedge y))$, which equals to $\neg(z \oplus x) \wedge (y) \wedge (z \vee \bar{x} \vee \bar{y}) \wedge (\bar{z} \vee x) \wedge (\bar{z} \vee y)$. XOR clauses can be represented as positive and negative equivalences, while literal conjunctions of N literals can be converted into $N + 1$ new CNF clauses. Odd number of literals in an XOR clause results in positive equivalences and even number in negative equivalences. In general, a positive equivalence evaluates to TRUE if an even number of literals are FALSE. Negative equivalence evaluates to TRUE when odd number of literals evaluate to TRUE [24]. Equivalence clauses are solvable in polynomial time by Gaussian elimination or by applying special preprocessing [25, 24], but Davis-Putnam based SAT algorithms can still take exponential time on random XOR-SAT formulae [26].

In order to get the initial set of points one can either convert the congruences into CNF and use any incremental SAT solver to discover all the solutions. Transformation of N-literal equivalence clause into CNF yields 2^{N-1} clauses, and can cause a significant memory blow out. One way to remedy this is to use solvers that can reason about equivalence clauses [27] without converting them to CNF.

For finding an inverse of Jacobian, our decision procedure relies on Müller-Seidl algorithm for solving linear congruences. The inverse of Jacobian needs to be computed only for the base case. Furthermore, systems of equations resulting from software verification are often incomplete, so in that case it can be immediately concluded that Jacobian will be singular and have no inverse.

The second term in recurrence in Eq. 4 can be easily computed by substituting the variables with concrete values. Even if the system of equations is incomplete, all the concrete solutions mod 2 will be discovered in the base step.

4.1 Algorithm

The algorithm for solving non-linear congruences is presented in Listing 2. It finds one solution if a system of non-linear congruences with multiple moduli is satisfiable, or returns UNSAT if there is no solution. According to our experience, many problems in software verification form incomplete systems of congruences and therefore are, most often, easily satisfiable. That insight motivated us to organize the search as a depth first search (DFS).

Parameters to the algorithm are $\mathcal{S}$ – FIFO queue for storage of temporal solutions, $\vec{f}$ – initial set of congruences, $\vec{f}_{BOOL}$ – simplified (according to Fermat's little theorem and coefficient reduction) congruences mod 2, and $\vec{m}$ – a column vector of moduli. Procedure EXTEND performs zero/sign extension of concrete assignments to variables. Only concrete values that participate in expressions with mixed types need to be extended.

Although there exist efficient algorithms for computing sparse Jacobian matrices [28], we assume only a simple polynomial derivation for computing Jacobian.

THEOREM 2 (Soundness and completeness). *Assuming a sound and complete SAT solver, the algorithm is sound and complete.*

Proof 2. Soundness. *It follows straightforwardly form the check at line 11. Reaching the target ring $M = 2^k$, points that do not satisfy $\vec{f}(\vec{x}_{k-1}) \equiv 0 \mod 2^k$ are discarded. Hence, the algorithm cannot return a spurious solution.*

Completeness. *The completeness of the lifting process follows from the mathematical derivation of the Newton's iteration given before. The only part that remains to be proven is the completeness of solving the base case. Any satisfiable system of congruences mod 2 has a finite number of solutions. From the assumption of a sound and complete SAT solver it follows that the SAT solver will discover all and only valid solutions. This concludes the proof.* □

THEOREM 3 (Termination). *Assuming that the SAT solver terminates, the above algorithm terminates too.*

Proof 3. *There is a finite number of solutions of the base case. A new blocking clause is added in line 7 for each solution. That prevents the rediscovery of the same solution. Termination of the base case follows. Lifting process itself terminates because at each iteration only a finite number of solutions can be generated, and there is a final number of iterations.* □

THEOREM 4 (Optimality). *Given an unsatisfiable system of congruences, the algorithm will prove unsatisfiability examining a minimal number of bits.*

Proof 4. *A solution $\vec{x}_k$ mod 2^i is also a solution mod 2^j : $\forall i, j \mid i > j > 0$. By contrapositive argument it follows that if there is no solution mod 2^j, then there is no solution mod $2^i, i > j > 0$. As the algorithm starts with a base case and iteratively lifts the solutions in a DFS manner, it follows that for each initial point it will prove unsatisfiability mod 2^k, where k is the smallest k for which the system becomes unsatisfiable.* □

It is important to note that because the algorithm enumerates concrete solutions (if there are any), slack variables are not needed. The lifting algorithm needs to be aware of the types of variables, but range constraints need not be expressed explicitly. Evaluating inequalities is trivial as each concrete point is evaluated against the set of inequalities D.

4.2 Extensions and Optimizations

In this section we propose an extension and suggest several possible optimizations of the basic approach. So far, we presented the mathematical tools for solving systems of non-linear congruences and therefore also how to reason about multiplication of bounded integers. Handling logic operators with a modular arithmetic decision procedure is much harder. Strictly speaking, Newton's iteration is defined only on multiplication and addition, so logic operators cannot be directly supported. There are several alternatives. One possibility would be to solve the system of equations, and reason about bit-vector operators later. The bit-vector operators represent additional constraints that should be used to prune the search space during iteration. Using them only after the final solutions are found would result in a performance penalty. The second option is to encode bit-vector operators as additional linear congruences and inequalities (section 3). A disadvantage of this approach are large constants that can appear, which in general slow down Simplex and integer solvers. Additionally, all the inequalities are solved only after concrete solutions of non-linear congruences are available. The approach we take is to represent the bit-vector operators as additional non-linear congruence constraints. These additional constraints prune the search space as early as possible.

A system of non-linear congruences with multiple moduli is a system such that $m_i \leq m_j$ for $i < j$. It should be obvious that generality is not lost. A set of congruences with multiple moduli can be solved as a system of congruences with a single modulo $M = \mathsf{LCM}_i(m_i)$, where LCM is the least common multiple:

Algorithm 2 DFS decision algorithm for solving non-linear congruences with multiple moduli

```
1:  procedure SOLVE($\mathcal{S}, \vec{f}, \vec{f}_{BOOL}, \vec{m}$)
2:      $\mathcal{J} = \left[\frac{\partial f_i}{\partial x_j}\right], 0 \le i < m, 0 \le j < n$                    ▷ Compute Jacobian
3:      $M \leftarrow \mathsf{LCM}_i\left(m_i\right) \in \vec{m}$
4:      while $\left(\vec{x}_1 \leftarrow \mathsf{SAT}\left(\vec{f}_{BOOL}\right)\right) \ne \mathsf{UNSAT}$ do
5:          $\mathcal{S} \leftarrow \{\langle \vec{x}_1, 2 \rangle\}$
6:          $J = \mathcal{J}\left(\vec{x}_1\right)$                    ▷ Evaluate Jacobian in the initial point
7:          $\vec{f}_{BOOL} \leftarrow \vec{f}_{BOOL} \wedge \left(\sum_{l_i \in \mathsf{supp}(\vec{x}_1)} \neg l_i\right)$                    ▷ Add a blocking clause
8:          while $\mathcal{S} \ne \emptyset$ do
9:              pick $\langle \vec{x}_{k-1}, k \rangle \in \mathcal{S}$                    ▷ Take next initial point
10:             $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\langle \vec{x}_{k-1}, k \rangle\}$
11:             if $\vec{f}\left(\vec{x}_{k-1}\right) \equiv 0 \mod 2^k$ then
12:                 if $k = M$ then
13:                     return $\vec{x}_{k-1}$
14:                 end if
15:                 if $\exists J^{-1}$ then
16:                     $\mathcal{S} \leftarrow \mathcal{S} \cup \left\{\langle \vec{x}_{k-1} + \left[-J^{-1} \cdot \left(\frac{\vec{f}\left(\vec{x}_{k-1}\right)}{2^{k-1}}\right) \mod 2\right] \cdot 2^{k-1} \mod 2^k, k+1 \rangle\right\}$
17:                 else if $\left(\frac{\vec{f}(\vec{x}_{k-1}}{2^{k-1}}\right) \equiv 0 \mod 2$ then
18:                     $\mathcal{S} \leftarrow \mathcal{S} \cup \left\{\langle \vec{x}_{k-1} + \vec{t} \cdot 2^{k-1}, k+1 \rangle\right\}, 0 \le t_i \le 1, |x_i| \ge k$
19:                 else
20:                     break                    ▷ Discard bad starting point
21:                 end if
22:                 for all $x_i \in \vec{x}_{k-1} : |x_i| < k$ do
23:                     EXTEND($x_i$)                    ▷ Zero/sign extension
24:                 end for
25:             end if
26:         end while
27:     end while
28:     return UNSAT
29: end procedure
```

$$
\begin{array}{rcll}
f_1\left(x_1, \ldots, x_n\right) & \equiv & 0 & \mod m_1 \\
\ldots & \ldots & \ldots & \\
f_r\left(x_1, \ldots, x_n\right) & \equiv & 0 & \mod m_r \\
& \Longrightarrow & & \\
\frac{M}{m_1} \cdot f_1\left(x_1, \ldots, x_n\right) & \equiv & 0 & \mod M \\
\ldots & \ldots & \ldots & \\
\frac{M}{m_r} \cdot f_r\left(x_1, \ldots, x_n\right) & \equiv & 0 & \mod M
\end{array}
$$

Each congruence can be seen as a single constraint determining the k_i least significant bits of variables $x_1, \ldots, x_n$.

Given a set of base case solutions, they need to be lifted to $\mathbb{Z}_M$. Congruences with smaller moduli restrict fewer bits. Thus, when a ring $\mathbb{Z}_{m_t}$ is reached through the iterative lifting process, all congruences with moduli $m_i < m_t$ can be eliminated from further consideration. The variables that appear only in congruences modulo m_i can be of size at most i, and therefore those variables need not to be lifted further. It is important to realize that because the system with multiple moduli $m_1 \le \cdots \le m_r$ is less constrained than the system of congruences modulo m_r, it can have a model, while the second may not.

To support bit-vector operators, we need one more definition – a definition of variable types. Types will be used only to tell the lifting algorithm when a variable has been computed to its full precision. At that point, the algorithm will consider the computed value to be the exact solution and will not lift it further. A *type of a variable* is denoted by int_k for signed bit-vectors and uint_k for unsigned. In both cases k denotes the ring $\mathbb{Z}_{2^k}$. Subscript represents the size of the variable in bits $k = |a|$. If two variables of different types appear within the same expression, sign (resp. zero) extension is performed for signed (resp. unsigned) during the lifting of concrete values.

Logic and bit-vector operators can be encoded as non-linear congruences with multiple moduli (see Table 1). Extension to bit-wise operators is straight-forward. A new variable is introduced for each bit and the bit vector is assembled by multiplying each variable with corresponding power of 2. Instead of introducing all variables at once, variables can be introduced in a lazy manner as needed. If the set of constraints can be proven to be unsatisfiable by inspecting only a few least significant bits, the overhead of encoding will be largely avoided.

Logic operators (XOR $\oplus$, OR $\vee$, AND $\wedge$, and equivalence $\Leftrightarrow$) are defined on bits. Left (resp. right) shift of bit-vector b by x positions is denoted as $\mathsf{shl}\left(b, x\right)$ (resp. $\mathsf{shr}\left(b, x\right)$), where x is of an arbitrary unsigned integer type. Concatenation is appending one bit-vector to another and it is written as $a@b$. In general, the types of a and b can differ. The resulting bit-vector must be large enough to accommodate the result, more precisely $|d| = |a| + |b|$ for $d = a@b$. All the operators, except shr, are defined equally for signed bounded integers. Signed right shift $a = \mathsf{shr}_s\left(b, x\right)$ can be computed through unsigned shifts as shown.

$a, b : \mathsf{uint}_k$	$a = \mathsf{shl}\,(b, x)$	$a \equiv 2^x \cdot b \mod 2^k$
$a, b : \mathsf{uint}_k; c : \mathsf{uint}_{k-x}$	$a = \mathsf{shr}\,(b, x)$	$c \equiv b \mod 2^x$ $a \cdot 2^x \equiv b - c \mod 2^k$
$a, b, c : \mathsf{int}_k; d : \mathsf{uint}_1$	$a = \mathsf{shr}_s\,(b, x)$	$d \equiv \mathsf{shr}\,(b, k-1) \mod 2$ $c \equiv \mathsf{shl}\,(\mathsf{shr}\,(-1, k-x), k-x) \mod 2^k$ $a \equiv c \cdot d + \mathsf{shr}\,(b, x) \mod 2^k$
$a : \mathsf{uint}_{i-j+1}; b, c : \mathsf{uint}_k$	$a = b\,[i : j]$	$c \equiv \mathsf{shl}\,(b, k-i-1) \mod 2^k$ $a \equiv \mathsf{shr}\,(c, k-i+j-1) \mod 2^{i-j+1}$
$a : \mathsf{uint}_{k_a}; b : \mathsf{uint}_{k_b}; d : \mathsf{uint}_{k_a+k_b}$	$d = a@b$	$d \equiv \mathsf{shl}\,(a, k_b) + b \mod 2^{k_a+k_b}$
$a, b : \mathsf{uint}_1$	$a = \neg b$	$a \equiv b + 1 \mod 2$
$a, b : \mathsf{uint}_1$	$a = b \vee c$	$a \equiv b \cdot c + b + c \mod 2$
$a, b : \mathsf{uint}_1$	$a = b \wedge c$	$a \equiv b \cdot c \mod 2$
$a, b : \mathsf{uint}_1$	$a = b \oplus c$	$a \equiv b + c \mod 2$

Table 1. Bit-vector operations on unsigned bounded integers

EXAMPLE 3. *Let us consider encoding a simple disjunct of linear into a conjunct of non-linear constraints. The given constraints are: $a, b, c : \mathsf{uint}_{32}$ and $a == b \vee a < c$. Modulo is 2^{32} and will not be explicitly written. The second term of the disjunct can be rewritten as $a == c + x, x > 0, x : \mathsf{uint}_{32}$. The decision procedure interprets equality as congruence, hence we get a system of constraints $(a \equiv b \vee a \equiv c + x) \wedge (0 \leq a, b, c, x < 2^{32}) \wedge (x > 0)$. Variable x is a slack variable. Now, we introduce two additional slack variables y and z, such that $y \cdot z \equiv 0$ and $y, z : \mathsf{uint}_{32}$. It follows that the conjunction of constraints $(a \equiv b + y) \wedge (a \equiv c + x + z) \wedge (y \cdot z \equiv 0) \wedge (x > 0) \wedge (0 \leq a, b, c, x, y, z < 2^{32}) \wedge (y \cdot z < 2^{32})$ is equisatisfiable to the original disjunct. The product $y \cdot z$ must not overflow, as one slack variable might be a zero divisor of the other, hence the last constraint.*

The proposed algorithm has two performance bottlenecks. If the system of congruences is easily satisfiable and has a large number of solutions, but the inequalities make it unsatisfiable, the algorithm would enumerate all the concrete solutions. Running a negative cycle detection as a preprocessing step decreases the likelihood of this happening. The second bottleneck is the line 18. For N variables there might be 2^N bits that need to be considered at each lifting step in this case. It might be possible to use Horner's scheme to decrease the number of cases. Another approach that could be effective is to translate a system of multivariate polynomial congruences into an isomorphic system of univariate congruences [29], do the lifting on the system of univariate congruences and reconstruct the solutions.

4.3 Comparison with SAT Encodings

The logic of bit-vectors for Nelson-Oppen setting has been thoroughly analyzed by Barrett et al. and Ganesh at al. [30, 31]. Their decision procedure is based on an eager conversion to boolean satisfiability instance which is given to a SAT solver. UCLID is another tool that is based on eager conversion to SAT [32]. Each bit of each bit-vector is converted into a boolean variable. Operators are encoded explicitly, even multiplication, which is known to be a hard problem for both SAT solvers and model checkers alike [33].

In the linear case, polynomial preprocessing algorithm assures that the subset of the set of constraints that can be solved in linear time is solved in linear time. No SAT solver can guarantee that, as explained on the example of XOR clauses.

According to the proof of optimality, it follows that our decision procedure will examine minimal number of bits for unsatisfiable sets of congruences. SAT solvers are likely to get lost in the problem choosing wrong variables for splitting and cannot provide such guarantees. Our approach is insensitive to the exponents or the number of multiplications, while each additional multiplier makes the problem much harder for SAT solvers. Finally, logical constraints are encoded as congruences and can therefore be used early during the search.

5. Nelson-Oppen Integration

Automated theorem provers are the engines of many formal verification tools. Decision procedures for theories that meet certain requirements can be integrated in a modular way by using Nelson-Oppen framework for combining theories [2, 3]. Nelson-Oppen method is based on three basic assumptions:

1. the formula to be tested for satisfiability is quantifier free
2. the signatures of the theories are *mutually disjoint*
3. the theories must be *stably infinite*

Reasoning about quantified theories is out of the scope of this paper, so we will not address this issue. Although the second assumption can be somewhat relaxed [34], we will assume that the signatures are mutually disjoint.

An excellent introduction to combining theories and related notions, like stable-infiniteness, was given in [35]. A theory T is stably infinite if for every T-satisfiable quantifier free formula ϕ there exists a T-interpretation satisfying ϕ whose domain is infinite.

DEFINITION 1. *A signature Σ consists of a set of constants Σ^C, function symbols Σ^F, and a set of predicate symbols Σ^P. A Σ-theory is any set of Σ-sentences. Σ-formulae are defined in the usual way. The theory of modular arithmetic M with signature Σ is defined as:*

- $\Sigma^F = \{+, *_c\} \mid c \in \mathbb{Z}_m$*, for the linear case*
- $\Sigma^F = \{+, *\} \mid c \in \mathbb{Z}_m$*, for the non-linear case*
- *+, -, $*$, and $*_c$ are interpreted as standard addition, subtraction, multiplication, and multiplication with a constant c over $\mathbb{Z}_m$*
- $\Sigma^C = \mathbb{Z}$
- $\Sigma^P = \{\equiv, <, \leq, >, \geq\}$
- $x \equiv y \mod m_k$ *iff* $x = y + t \cdot m_k, \forall t \in \mathbb{Z}$
- $\Sigma^P \setminus \{\equiv\}$ *defined over* $\mathbb{Z}$.

In the following discussion it will be demonstrated that the theory of linear modular arithmetic can be simulated with the theory of unbounded integers with a set of bounds over each variable. Inequality is well defined over the set of integers. The same reasoning applies to the non-linear case. The only distinction is that range constraints are implicit and the decision procedure produces concrete solutions, so an integer solver is not called. To show that the theory M is stably infinite, we will start with a theory of unbounded integers, which is known to be stably infinite. We introduce a set of equations over fresh variables $V = \{v_i = i \mid 0 \leq i < \max_k m_k\}$. Integer constants are interpreted only by M, but variables v_i can be shared. The theory remains stably infinite.

When Müller-Seidl algorithm returns a set of parametric solutions, we're only interested in a subset of all possible solutions, namely those allowed by variable types. Range constraints can be represented as disjunctions of equalities, for ex. $P = \left\{x = v_0 \vee x = v_1 \vee \cdots \vee x = v_{2^k-1}\right\}$, and can be passed to M for each variable $x : \mathsf{uint}_{2^k}$. As range predicates can be seen as a part of normal process of exchanging equalities, rather than domain restrictions, the theory of modular arithmetic is stably infinite. The notion of range predicates was first mentioned but not elaborated by Barret et al. [12][8]. Instead of passing huge disjunctions of equalities over the interface, range predicates can be implicitly inferred from variable types.

Congruence operator is M-specific. Thus, the decision procedure for M cannot propagate congruences. This does not represent a problem as the theory of linear modular arithmetic is simulated with a theory of unbounded integers and the decision procedure infers equalities over $\mathbb{Z}$, which can be shared. The theory of non-linear modular arithmetic is not directly simulated by the theory of unbounded integers as such theory would be undecidable. Rather, the decision procedure returns concrete points from $\mathbb{Z}_m^N$, where N is the number of variables. Without loss of generality, such points can be seen as points from $\mathbb{Z}^N$. Again, both equality and inequality is well defined on any subset of $\mathbb{Z}$.

Modulo operation partitions the set of integers into equivalence classes. For modulo m there are exactly m equivalence classes. The set of minimal elements of all equivalence classes covers exactly the range $\{0, \ldots, m-1\}$, assuming unsigned bounded integers. It follows that the inequalities used in programming are actually defined over the set of minimal elements of equivalence classes.

DEFINITION 2 (Bounded inequalities). *For a ring $\mathbb{Z}_m$, bounded inequalities $\bowtie_m, \bowtie \in \{<, \leq, >, \geq\}$ are defined over the minimal elements of equivalence classes in which operation mod m partitions the set of integers $\mathbb{Z}$.*

THEOREM 5 (Relational Equivalence). *Given $a, b \in \mathbb{Z}_m$ such that $0 \leq a, b < m$, $a \bowtie_m b$ iff $a' \bowtie b'$ for $a', b' \in \mathbb{Z}$ and $a' = a, b' = b, 0 \leq a', b' < m$.*

The proof trivially follows from the properties of inequality relations. Thus, we can reason about bounded inequalities using inequalities over integers. Due to range constraints, bounded inequalities cannot be used to detect overflows (unless a larger data type is used, as shown before). If an overflow happens, the variable will no longer be a minimal element of its equivalence class. For that reason, our decision procedure interprets inequalities over the set of unbounded integers, making detection of overflows possible, while respecting the exact semantics of bounded inequalities.

Equality generation is the last issue that needs to be satisfied. The theory of integers (and therefore of modular arithmetic) is *non-convex*. Simply put, the theory might imply a disjunction of equalities, without implying a single equality. Recently it has been proven that generation of disjunctions of equalities is $\mathcal{NP}$-complete [16] for UTVPI constraints (the same result holds for difference constraints). As general ILP constraints are more expressive, it follows that generating disjunctions of equalities for the theory of modular arithmetic is at least $\mathcal{NP}$-complete . All equalities and disjunctions of equalities over shared variables can be generated using a brute-force algorithm in $2 \cdot \binom{n}{2}$ calls to an $\mathcal{NP}$-complete decision procedure, where n is the number of shared variables. This is a trivial corollary following from the decidability requirement on Nelson-Oppen theories.

[8] Originally called type predicates.

EXAMPLE 4. *In this example we will illustrate how to generate all equalities and disjuncts of equalities with a brute-force algorithm in $2 \cdot \binom{n}{2}$ calls to the decision procedure for (non)-linear modular arithmetic. Let $S = s_1, \ldots, s_n$ be a set of shared variables between theory M and other theories. Let e_{ij} denote $s_i = s_j$ and $\neg e_{ij}$ its negation. There can be $\binom{n}{2}$ possible equalities between shared variables. First, it needs to be checked whether M implies any equalities by finding all pairs (i, j) such that $M \Rightarrow e_{ij}$ and $i \neq j$. This can be done by checking the unsatisfiability of $\binom{n}{2}$ formulas $M \wedge \neg e_{ij}$. The set of found equalities is denoted as E. In the next step we check the unsatisfiability of $M \bigwedge_{i,j} \neg e_{ij} \mid i \neq j \wedge e_{ij} \notin E$. If satisfiable, M doesn't imply a disjunction of equalities and we are done. Otherwise, we examine the proof of unsatisfiability. The set of inequalities involved in the proof of unsatisfiability represents a single disjunct. We eliminate one of the equalities, and repeat the procedure. As there can be at most $\binom{n}{2}$ equalities, generation of disjuncts of equalities requires at most $\binom{n}{2}$ calls. Thus, all equalities and a disjunction of remaining equalities can be inferred in $2 \cdot \binom{n}{2}$ calls in the worst case.*

In practice, it suffices to infer only one equality or one disjunct at the time to enable the theorem prover to make further progress. Incremental decision procedure can be used for the first step (inferring equalities). Generation of disjuncts of equalities can be implemented by a resettable decision procedure that avoids redoing all the work when a single constraint is removed.

An important question is whether generation of equalities can be circumvented. Nelson-Oppen method is based on the exchange of equalities. A promising approach for circumventing the generation of equalities are layered theorem provers [18] that sacrifice the deduction power and modularity for avoiding the equality generation. The layered approach tries to solve the problem by a series of incrementally more expensive decision procedures. The last decision procedure that MathSAT uses at the end is an incomplete integer solver based on doubly exponential Fourier-Motzkin algorithm. Hence, it seems to be at the disadvantage for integer problems. Furthermore, it can't handle multiplication and integer overflows/underflows.

6. Conclusions and Open Problems

This paper presents a novel way to reason about bounded integers, where each integer variable has predefined upper and lower bounds. Bounded integers correspond to finite representations of program variables in the hardware. Instead of treating these bounded-integer variables as bit-vectors [36], the modular arithmetic decision procedures described in this paper, interprets arithmetic operations on these variables as operations performed modulo 2^k, for an appropriate k. We treat linear and non-linear case separately. For each case we propose a distinct decision procedure and analyse advantages and disadvantages.

The authors believe that this paper makes an important step in building efficient linear and non-linear decision procedures for bounded integers that support all integer operations used in programming, including multiplication and logical operations.

This paper also presents some interesting open problems. First, the lifting scheme described in this paper applies to concrete solutions, forcing the decision procedure to enumerate all solutions in the worst case. This can be improved by designing a lifting scheme which works on *symbolic* solutions, or equivalently on sets of concrete solutions. Another important question is whether equality generation is a too expensive requirement. Giving away some deductive power and modularity in exchange for being able to avoid generation of equalities might be a valuable tradeoff for software model checking.

Acknowledgments

We would like to thank Shuvendu Lahiri, Peter Montgomery, Cesare Tinelli, and Calogero Zarba for constructive comments and interesting discussions.

References

[1] Detlefs, D., Nelson, G., Saxe, J.S.: Simplify: A Theorem Prover for Program Checking. Technical report, HP Laboratories Palo Alto, Technical Report HPL-2003-148 (2003)

[2] Nelson, G.: Techniques for program verification. PhD thesis, Stanford University (1979)

[3] Nelson, G., Oppen, D.C.: Simplification by. ACM Transactions on Programming Languages and Systems **1** (1979) 245–257

[4] Müller-Olm, M., Seidl, H.: Analysis of Modular Arithmetic. In: To appear in ESOP 2005. (2005)

[5] Ball, T., Rajamani, S.K.: The SLAM Project: Debugging System Software via Static Analysis. In: POPL 2002. (2002) 1–3

[6] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with Blast. In: Proceedings of the 10th SPIN Workshop, Springer (2003)

[7] Flanagan, C., Leino, K., Lillibridge, M., Nelson, G., Saxe, J., Stata, R.: (Extended static checking for Java)

[8] Harrison, W.: Compiler analysis for the value ranges of variables. IEEE Transactions on Software Engineering **3** (1977) 243–250

[9] Postel, J.: Transmission Control Protocol. RFC 793, USC/Information Sciences Institute (1981)

[10] Braden, R.: Requirements for Internet hosts – Communication layers. RFC 1122, USC/Information Sciences Institute (1989)

[11] Matiyashevich, Y.: Hilbert's 10^{th} Problem: What can we do with Diophantine equations? In: A talk given at a seminar of IREM in Paris. (2001)

[12] Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for bit-vector arithmetic. In: Proceedings of the 35th Design Automation Conference. (1998) San Francisco, CA.

[13] Cyrluk, D., Möller, O., Rueß, H.: An efficient decision procedure for the theory of fixed-sized bit-vectors. In Grumberg, O., ed.: 9th International Conference on Computer-Aided Verification (CAV'97). Volume 1254 of Lecture Notes in Computer Science., Springer-Verlag (1997) 60–71

[14] Clarke, E., Kroening, D.: Hardware verification using ANSI-C programs as a reference. In: Proceedings of ASP-DAC 2003, IEEE Computer Society Press (2003) 308–311

[15] von zur Gathen, J., Gerhard, J.: Modern Computer Algebra. Cambridge University Press, 2^{nd} edition (2003)

[16] Lahiri, S.K., Musuvathi, M.: An Efficient Decision Procedure for UTVPI Constraints. Technical report, Microsoft Research Technical Report MSR-TR-2005-67 (2005)

[17] Matula, D.W., Fit-Florea, A., Thornton, M.A.: Table Lookup Structures for Multiplicative Inverses Modulo 2^k. In: Proceedings of 17^{th} IEEE Symposium on Computer Arithmetic ARITH-17. (2005)

[18] Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Schulz, S., Sebastiani, R.: An Incremental and Layered Procedure for the Satisfiability of Linear Arithmetic Logic. In Halbwachs, N., Zuck, L.D., eds.: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005). Volume 3440 of Lecture Notes in Computer Science., Springer (2005) 317–333

[19] Sierksma, G.: Linear and integer programming: theory and practice. Marcel Dekker (2002)

[20] Yun, D.Y.Y.: Algebraic algorithms using p-adic constructions. In: Proceedings of the third ACM symposium on symbolic and algebraic computation. (1976) 248–259

[21] Zippel, R.: Effective Polynomial Computation. Kluwer Academic Publishers (1993)

[22] Rosen, K.H.: Elementary Number Theory and Its Applications, 4^{th} edition. Addison Wesley (2000)

[23] Patarin, J., Goubin, L.: Trapdoor One-Way Permutations and Multivariate Polynomials. In: Proceedings of 1^{st} International Information and Communications Security Conference. (1997)

[24] Warners, J.P., van Maaren, H.: Recognition of tractable satisfiability problems through balanced polynomial representations. In: Proceedings of the 5th Twente workshop on on Graphs and combinatorial optimization, Elsevier Science Publishers B. V. (2000) 229–244

[25] Warners, J.P., Van-Maaren, H.: A two phase algorithm for solving a class of hard satisfiability problems. Operations Research letters **23** (1998) 81–88

[26] Jia, H., Moore, C., Selman, B.: From spin glasses to hard satisfiable formulas. In: Proceedings of SAT 2004. (2004)

[27] Babić, D., Hu, A.J.: Hypersat. In: To appear in SAT 2005 competition proceedings. (2005)

[28] Hossain, A.K.M.S.: On The Computation of Sparse Jacobian Matrices and Newton Steps. PhD thesis, Department of Informatics, University of Bergen, Norway (1998)

[29] Zippel, R.: Newton's iteration and the sparse Hensel algorithm. In: Proceedings of the fourth ACM symposium on symbolic and algebraic computation. (1981) 68–72

[30] Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for bit-vector arithmetic. In: Proceedings of the 35th Design Automation Conference. (1998) 522–527

[31] Ganesh, V., Berezin, S., Dill, D.L.: Technical Report: A Decision Procedure for Fixed-Width Bit-Vectors. Technical report, Stanford University (9^{th} April 2005)

[32] Lahiri, S.K., Seshia, S.A.: The UCLID Decision Procedure. In: Proceedings of Computer-Aided Verification conference. (2004)

[33] Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded Model Checking Using Satisfiability Solving. Formal Methods in System Design (2001) 7–34

[34] Zarba, C.G.: Combining non-disjoint theories. In Gor'e, R., Leitsch, A., Nipkow, T., eds.: International Joint Conference on Automated Reasoning (Short Papers). Technical Report DII 11/01, Universit'a of Siena, Italy (2001) 180–189

[35] Manna, Z., Zarba, C.G.: Combining decision procedures. In: Formal Methods at the Cross Roads: From Panacea to Foundational Support. Volume 2757 of Lecture Notes in Computer Science., Springer (2003) 381–422

[36] Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In Alur, R., Peled, D.A., eds.: CAV. Lecture Notes in Computer Science, Springer (2004)