Constraint solving on modular integers

Arnaud Gotlieb[1], Michel Leconte[2], and Bruno Marre[3]

[1] INRIA Rennes Bretagne Atlantique, Campus Beaulieu, 35042 Rennes, France
arnaud.gotlieb@inria.fr
[2] ILOG Lab, IBM France, Gentilly, France
leconte@ibm.fr
[3] CEA, LIST, Gif-sur-Yvette, F-91191, France
marre@cea.fr

Abstract. Constraint solving over finite-sized integers involves the definition of propagators able to capture modular (a.k.a. wrap-around) integer computations. In this paper, we propose efficient propagators for a fragment of modular integer constraints including adders, multipliers and comparators. Our approach is based on the original notion of *Clockwise Interval* for which we define a complete arithmetic. We also present three distinct implementations of modular integer constraint solving in the context of software verification[4].

1 Introduction

Using constraint solving to automatically generate program inputs is an emerging trend in software verification. In the last decade, several tools based on Finite Domains (FD) constraint solving were proposed that perform test inputs generation for C programs (e.g., InKa [7], PathCrawler [9]), test case generation for reactive programs (e.g., GATEL [10]), or property-oriented software verification (e.g., CPBPV [4], Euclide [6]). In these tools, automated verification of inten-

```
unsigned long len = 2147483648;     % Equal to 2^31
void f(unsigned long buf) {
1.        if (buf + len < buf) {
2.                                  ...
```

Fig. 1. Program taking care of integer overflow

sive integer computations involves solving constraints over finite-sized integers. As an example, consider the problem of reaching[5] statement 2 in the program of Fig.1 that requires solving the decision `buf+len < buf` over unsigned 32-bits integers. A naive translation of the decision of statement 1 as constraint

[4] This work is supported by ANR-07-SESUR-003 CAVERN Project
[5] Reachability is a fundamental problem in software program verification.

$buf + len < buf$ where buf belongs to $0..2^{32} - 1$ and $len = 2^{31}$, yields an incorrect result saying that statement 2 is unreacheable. In fact, it is trivial to see that this constraint is unsatisfiable when it is interpreted over Finite Domains (FD). However, statement 2 can be reached by selecting a test value such as $buf = 2^{31}$, as $2^{31} + 2^{31} = 2^{32}$ corresponds to value 0 in unsigned 32-bits integer arithmetic. Note also that simplifying $buf + len < buf$ in $len < 0$ is forbidden in this arithmetic. The overall reason is that decision 2 should be rather interpreted as $buf + len < buf \ mod(2^{32})$. In fact, this problem is sometimes reported to as the "wrapping effect" and it turned out that programmers who take care of possible integer overflows routinely write programs that use this effect. In Fig.1, statement 2 can only be reached by a wrapping behaviour. Unfortunately, all the previously mentionned tools that exploits constraint techniques for automated test data generation or property-oriented verification simply ignore this wrapping effect. In fact, as soon as finite domains are specified for each input and intermediate variable, these tools consider that programs with integer overflows are necessarily incorrect and should be rejected. This is obviously abusive and often conducts to report false negatives.

This paper adresses this problem by providing efficient constraint solving over modular integer computations. We propose bound-consistency propagators for a linear fragment of these constraints that includes adders, multipliers and comparators. Our approach is based on the original notion of *Clockwise Interval* that captures the wrapping effect by considering intervals with modular integer bounds. An example of such *Clockwise Interval* is the interval $[7, 2]_8$ that represents all the integers x such that $x \ mod \ 8 = 7, x \ mod \ 8 = 0, x \ mod \ 8 = 1, x \ mod \ 8 = 2$. For these Clockwise Intervals, we give a complete arithmetic that has not been published elsewhere.

In the context of the ANR CAVERN project[6] we independently built three distinct implementations of modular integer constraint solving that are variations of clockwise interval arithmetics. These implementations of modular integer constraint solving are used in three software verification tools. Our first implementation called MAXC is used in the context of automatic test input generation for C programs. It implements bound-consistency filtering for a linear fragment of modular integer constraints. For example, for constraint $buf \oplus 2^{31} < buf$ where $\oplus$ denotes modular addition over 32-bit integers, MAXC automatically prunes the domain of buf to the Clockwise Interval $[2^{31}, 2^{32} - 1]_{2^{32}}$, removing half of the variation domain of buf. Our second implementation called JSOLVER [8] is intented to perform automatic analysis of rule-based programs. JSOLVER is based on classic intervals but it takes into account modular integer computations. A comparison with the Clockwise Interval arithmetics shows that JSOLVER is efficient but not optimal when computing local-consistencies over these constraints. Finally, the third implementation is called COLIBRI and it enables automatic test data generation for reactive programs [10]. COLIBRI implements modular integer constraints on domains represented as union of classic intervals.

[6] `cavern.inria.fr`

The rest of the paper is organized as follows: Sec.2 introduces the notations and the formal definitions used in the rest of the paper. Sec.3 presents bound-consistency filtering on modular integer constraints. Sec.4 describes our three distinct implementations and discusses their relations with Clockwise Intervals. Finally, Sec.5 concludes and draws several perspectives to this work.

2 Preliminaries

2.1 Notations

Let $\mathbb{Z}$ denote the set of integers and $\mathbb{Z}_b$ denote the finite set of integers modulo b. For any $x \in \mathbb{Z}$ and $y \in \mathbb{Z}^*$, $x \bmod y$ denotes the integer r such that $\exists q \in \mathbb{Z}$ $r = x - y * q$ and $0 \leq r < y$, while x quo y denotes q the quotient. In the following, we will fix $b = 2^n$ where n is any non-negative integer. Since $\mathbb{Z}_b$ consists of residue classes, several representations are possible. In this paper, we will consider two representations that can be used to emulate integral computations in imperative languages such C or Java: the unsigned representation $\{0, 1, \ldots, b-1\}$ and the signed representation $\{-\frac{b}{2}, \ldots, -1, 0, 1, \ldots, \frac{b}{2}-1\}$. For the sake of simplicity, we will use the unsigned representation (unless it is mentionned otherwise).

In the context of integer-based manipulations, a classic interval noted $x..y$ where $x, y \in \mathbb{Z}$ and $x \leq y$ denotes the finite ordered set $\{x, x+1, \ldots, y-1, y\}$.

Definition 1 (Width). *The width of an interval $x..y$ is an integer, defined as follows: $wid(x..y) \triangleq y - x$.*

2.2 Clockwise Interval

Definition 2 (Clockwise Interval). *Let x and y be two integers modulo b, a Clockwise Interval (CI) is noted $[x, y]_b$ and denotes the set $\{x, x+1 \bmod b, \ldots, y-1 \bmod b, y\}$.*

It differs from classic interval in that any of its element is a residue class of integer modulo b. Furthermore, the bound y is not required to be greater than x as the set $\{x, x+1 \bmod b, \ldots, y-1 \bmod b, y\}$ is unordered. By convention, we consider that $[0, b-1]_b$ is the canonical representation of $\mathbb{Z}_b$ itself. Note that other representations exist: $[1, 0]_b$, $[2, 1]_b$,...,$[b-1, 0]_b$. Clockwise Intervals that have a positive or null width are called *proper* CIs, while others are called *improper* CIs. The width of a CI is defined by extending the definition of width over classic intervals, by using the canonical representation: $wid([x, y]_b) = wid(x..y)$. Note that width can then becomes negative in this case. The set of clockwise intervals over $\mathbb{Z}_b$ is finite. It is composed of $\{[]_b, [0, 0]_b, \ldots, [b-1, b-1]_b, [0, 1]_b, [1, 0]_b, \ldots, [b-2, b-1]_b, [b-1, b-2]_b, \ldots, [0, b-1]_b\}$, where $[]_b$ denotes the empty clockwise interval.

Definition 3 (Cardinality). *Let $[x, y]_b$ be a CI, then its cardinality is an integer modulo b defined as: $card([x, y]_b) \triangleq (y - x + 1) \bmod b$.*

By convention, $card([0, b-1]_b) = b$ and $0 < card([x, y]_b) \leq b$. For example, $card([7, 0]_8) = 2$ while $wid([7, 0]_8) = -7$. The following property immediatly holds:

Proposition 1. *A CI $[x, y]_b$ contains exactly $card([x, y]_b)$ elements, if represented over $[1, b]_b$.*

Proof. If $y \geq x$, then the set $[x, y]_b = \{x, x+1, \ldots, y-1, y\}$ is ordered and contains $y - x + 1$ elements. The special case where $y - x + 1 = b$ corresponds to the CI $[0, b-1]_b$ and then $card([0, b-1]_b) = b$.
If $y < x$, then $[x, y]_b = \{x, x+1, \ldots, b-1\} \cup \{0, 1, \ldots, y\}$ and so, it contains $(b - x) + (y + 1)$ elements. In this case, $b - x + y + 1 \equiv y - x + 1 \bmod b$ that gives the expected result.

2.3 Building clockwise intervals

A classic interval can be converted into a CI by using the following formula:

$$x..y \bmod b \triangleq \begin{cases} [0, b-1]_b & \text{if } wid(x..y) \geq b \\ [x \bmod b, y \bmod b]_b & \text{otherwise} \end{cases}$$

We define the *hull* of a set of modular integers as being the smallest Clockwise Interval w.r.t. cardinality, that contains all the elements of the set. By convention, proper clockwise intervals are considered smaller than improper ones when they have same cardinality. Formally,

Definition 4 (Hull). *Let $S = \{x_1, \ldots, x_p\}$ be a subset of $\mathbb{Z}_b$, the hull of S is a CI noted $\square S$, defined as:*

$$\square S \triangleq Inf_{card}(\{[x_i, x_j]_b | \{x_1, \ldots, x_p\} \subseteq [x_i, x_j]_b)$$

Building an algorithm from this definition yields an untractable procedure as it would require considering $p!$ possible combinations of the bounds. Fortunately, we have the following proposition:

Proposition 2. *Let $S = \{x_0, \ldots, x_{p-1}\}$ be an* **ordered** *subset of $\mathbb{Z}_b$, and let x_{-1} denotes x_{p-1}, then*

$$\square S = [x_i, x_{i-1}]_b \text{ where } i \in 0..p-1 \text{ such that } card([x_i, x_{i-1}]_b) \text{ is minimized}$$

Therefore, when S is ordered, $\square S$ can be computed in linear time w.r.t. size of S.

Proof. The case where $[x_i, x_{i-1}]_b$ is proper, i.e., $x_i = x_0$ and $x_{i-1} = x_p$, is trivial. Let suppose that $[x_i, x_{i-1}]_b$ is an improper CI. Firstly, it is clear that $S \subseteq [x_i, x_{i-1}]_b$ as S is ordered ($\forall i \in 1..p-2, x_0 \leq x_{i-1} \leq x_i \leq x_{p-1}$). Secondly, as $card([x_i, x_{i-1}]_b)$ is minimized, it remains to show that there does not exist a CI $[k, l]_b$ where $j \neq i - 1$ that contains S and that is tighter than $[x_i, x_{i-1}]_b$). If $l > x_{i-1}$ then $x_{i-1} \notin [k, l]_b$ and if $k < x_i$ then $x_i \notin [k, l]_b$, meanning that $l \leq x_{i-1}$ and $k \geq x_i$. By this, we get $card([k, l]_b) \geq card([x_i, x_{i-1}]_b)$ which contradicts the hypothesis.

2.4 Clockwise Interval Arithmetic

Having defined Cl, we now turn on the definition of Clockwise Interval Arithmetic that allows us to perform computations over intervals.

Definition 5 (Addition). *Let $[i,j]_b$ and $[k,l]_b$ be two CI, then the addition operation, noted $\oplus$, is defined as:*

$$[i,j]_b \oplus [k,l]_b \triangleq \begin{cases} [0,b-1]_b & \text{if } card([i,j]_b) = b \text{ or } card([k,l]_b) = b \\ & \text{or } card([i,j]_b) + card([k,l]_b \geq b \\ [(i+k) \mod b, (j+l) \mod b]_b & \text{otherwise} \end{cases}$$

Correction property: $\forall x \in [i,j]_b, \forall y \in [k,l]_b, (x+y) \mod b \in [i,j]_b \oplus [k,l]_b$.

For example, $[2,3]_8 \oplus [3,2]_8 = [0,7]_8$ while $[2,2]_8 \oplus [3,3]_8 = [5,5]_8$.

Definition 6 (Substraction). *Let $[i,j]_b$ and $[k,l]_b$ be two CI, then the substraction operation, noted $\ominus$, is defined as:*

$$[i,j]_b \ominus [k,l]_b \triangleq \begin{cases} [0,b-1]_b & \text{if } card([i,j]_b) = b \text{ or } card([k,l]_b) = b \\ & \text{or } card([i,j]_b) + card([k,l]_b \geq b \\ [(i-l) \mod b, (j-k) \mod b]_b & \text{otherwise} \end{cases}$$

Correction property: $\forall x \in [i,j]_b, \forall y \in [k,l]_b, (x-y) \mod b \in [i,j]_b \ominus [k,l]_b$.

For example, we have $[0,1]_8 \ominus [0,1]_8 = [7,1]_8$. Note that $[0,b-1]_b$ is absorbing for $\oplus$ and $\ominus$. For those two operations, similarily to the situation in classic Interval Arithmetic, the computations can be performed on the bounds of Clockwise Intervals. This is no longer the case for multiplication and division, as the tightest CI that encloses all the solutions cannot be computed by using only bounds of its operands in those cases. Let us first define precisely the considered operations:

Definition 7 (Multiplication by a constant k). *Let k be a constant modulo b and $[i,j]_b$ a CI, then the multiplication by k is defined as follows:*

$$k * [i,j]_b \triangleq \square(\{k*i \mod b, k*(i+1) \mod b, ..., k*j \mod b\})$$

Definition 8 (Multiplication). *Let $[i,j]_b$ and $[k,l]_b$ be two CI, then the multiplication operation, noted $\otimes$, is defined as:*

$$[i,j]_b \otimes [k,l]_b \triangleq \square(\{i*k \mod b, i*(k+1) \mod b, ..., (i+1)*k \mod b, ..., j*l \mod b\})$$

Definition 9 (Division). *Let $[i,j]_b$ and $[k,l]_b$ be two CI, then the division operation, noted $\oslash$, is defined as:*

$$[i,j]_b \oslash [k,l]_b \triangleq \square(\{i/k \mod b, i/(k+1) \mod b, ..., (i+1)/k \mod b, ..., j/l \mod b\})$$

As an example, consider the *multiplication by a constant* operation $4 \otimes [2,4]_8$. With the formula, we get $\square(\{4*2 \bmod 8, 4*3 \bmod 8, 4*4 \bmod 8\}) = \square(\{0,4\}) = [0,4]_8$. Unfortunately, the bounds of the resulting CI $[0,4]_8$ cannot be computed by using only the bounds of CI operands as $4*2 \equiv 4*4 \equiv 0 \bmod 8$. Computing the resulting CI by enumerating all the elements of its operands seems unreasonnable in the context of large-sized machine integers. The following subsection describes a method that permits to compute the resulting optimal CI in the case of multiplication by a constant k, without requiring a ful enumeration of the domain of possible values.

2.5 An efficient method for computing optimal CI in the presence of multiplication operators

The method is based on the following notes:

- the structure of $\mathbb{Z}_{2^n}$ is well known: the divisors of 0 are powers of 2 ;
- thanks to proposition 2, $\square(\{x_1, ..., x_p\})$ can be computed efficiently when the set $\{x_1, ..., x_p\}$ is ordered.

Let k be a constant modulo $b = 2^n$, let $[i,j]_b$ be a CI, we describe a method that allows to compute the minimum and the maximum values of $k * [i,j]_b = \square(\{k*i \bmod b, k*(i+1) \bmod b, ..., k*j \bmod b\})$.

We start by eliminating some trivial cases: If $k = 0$, then $k*[i,j]_b = \square(\{0\}) = [0,0]_b$. If $k = 1$, then $k * [i,j]_b = [i,j]_b$. If $i \leq j$ and $k * j < b$, then $k * [i,j]_b = [k*i, k*j]_b$. Let now suppose that k is a constant greater or equal to 2 and $k * j \geq b$ or $i > j$. We have the following proposition:

Proposition 3. *Let $k \neq 2^w$, $q_1 = k*i$ quo b and $q_2 = k*j$ quo b, then:*
$Max(k * [i,j]_b) = b - d$ where $d = Min_{q_1 < q \leq q_2}(q * b \bmod k)$ and
$Min(k * [i,j]_b) = d'$ where $d' = Min_{q_1 < q \leq q_2}(-q * b \bmod k)$.

Proof. (sketch of, partial) Let p be the element of $[i,j]_b$ for which $k*p \bmod b$ is maximized in $\mathbb{Z}_b$, and let q be the smallest value such that $k*p < q*b$, then we consider $d = q*b - k*p$. We claim that $d = q*b \bmod k$ as $0 < p < k$. It remains to find the value of q that minimizes $q * b \bmod k$. As $p \in [i,j]_b$, we know that $q_1 < q \leq q_2$ by definition of q. Therefore we can explore the possible values of q from $q_1 + 1$ to q_2, up to $k - 1$ values.

For example, consider $k * [i,j]_b$ where $k = 5$ and $[i,j]_b = [2,7]_8$. Applying Prop.3, we get $q1 = 5*2$ div $8 = 1$ and $q2 = 5*7$ div $8 = 4$. For $q = 2,3,4$, computing $r_q = q*b \bmod k$ and $r_{-q} = -q*b \bmod k$ leads to:
$r_2 = 16 \bmod 5 = 1$ and $r_{-2} = -16 \bmod 5 = 4$,
$r_3 = 24 \bmod 5 = 4$ and $r_{-3} = -24 \bmod 5 = 1$,
$r_4 = 32 \bmod 5 = 2$ and $r_{-4} = -32 \bmod 5 = 3$.
The minimum over the r_i is obtained when $q = 2$ and then $Max(5 * [2,7]_8) = 8 - r_2 = 1$. For the r_{-i}, it is obtained when $q = 3$ leading to $Min(5 * [2,7]_8) = r_{-3} = 1$. Hence, $5 * [2,7]_8 = [1,7]_8$ has been computed by exploring only the

divisors of b in $k * i..k * j$, instead of looking at all the double products $k * l$ within the same range.

Finding similar propositions for generalized multiplication and division may be possible, but one can also use Prop.3 to compute over-approximations of the resulting CIs. It suffices to use the bounds of each operand interval as a constant, to apply Prop.3 on each of the four double products, and keep the smallest intersection of results. But note that, optimality is usually lost with this approach.

3 Constraint propagation over Clockwise Intervals

In this section, we define projection functions that allow to perform constraint propagation over CI. As usual in Finite Domains constraint solving, each variable X is associated a finite domain $dom(X)$ of possible values. We consider here that domain are (over-)approximated by CI: $CI(X) \triangleq \square(dom(X))$.

3.1 Set-based operations over CI

Inclusion, union and intersection of Clockwise Intervals are defined by using their set-theoretic definition counterpart. For example, *inclusion* over CI is defined as follows:

$$[i,j]_b \subseteq [k,l]_b \iff \{i, i+1, \ldots, j\} \subseteq \{k, k+1, \ldots, l\}$$

Note however that union and more surprisingly intersection are not closed over CI. For example, $[5,2]_8 \cap [1,6]_8 = \{1,2,5,6\}$. Hence, we define the *meet operation* as taking the smallest CI that contains all the elements of the intersection:

$$[i,j]_b \bigwedge [k,l]_b \triangleq \square(\{i, i+1, \ldots, j\} \cap \{k, k+1, \ldots, l\})$$

For example, we got: $[5,2]_8 \bigwedge [1,6]_8 = [1,6]_8$ and $[5,1]_8 \bigwedge [0,6]_8 = [5,1]_8$. The main question is whether these operations can be computed efficiently. The following property helps answering this question:
Let x be an integer modulo b, then $x \in [i,j]_b$ is true iff $x \geq i \wedge x \leq j$ when $[i,j]_b$ is proper and $x \geq i \vee x \leq j$ when $[i,j]_b$ is improper. This property comes directly from definition of CI.

The meet operator $\bigwedge$ As the computations of *meet* is at the core of constraint propagation engine, finding an efficent algorithm is of great importance. The definition given above requires to explore each element of both domains at least once. This can be costly when large domains are involved during constraint propagation. The following proposition offers ways to compute the meet operation more efficently:

Proposition 4. *Let $X = [i,j]_b$ and $Y = [k,l]_b$ be two CI, then $X \bigwedge Y$ is defined as:*

if $wid(X) * wid(Y) = 0$ *(suppose for example that* $X = [i, i]_b$*)*

$$X \bigwedge Y = \begin{cases} [i,i] & \text{if } X = [i,i]_b \wedge i \in Y \\ [k,k] & \text{if } Y = [k,k]_b \wedge k \in X \\ []_b & \text{otherwise} \end{cases}$$

if $wid(X) * wid(Y) > 0$ *then*

$$X \bigwedge Y = \begin{cases} []_b & \text{if } wid(X) > 0 \wedge wid(Y) > 0 \wedge max\{i,k\} > min\{j,l\} \\ [max\{i,k\}, min\{j,l\}]_b & \text{otherwise} \end{cases}$$

if $wid(X) * wid(Y) < 0$ *then*

$$X \bigwedge Y = \begin{cases} []_b & \text{if } j < k \wedge l < i \\ [k,j]_b & \text{if } j \geq k \wedge l < i \\ [i,l]_b & \text{if } j < k \wedge l \geq i \\ Y & \text{if } j \geq k \wedge l \geq i \\ & \wedge\ card(Y) \leq card(X) \\ X & \text{if } j \geq k, l \geq i \\ & \wedge\ card(X) < card(Y) \end{cases}$$

In these cases, proving that $CI(X) \bigwedge CI(Y) = \square(\{i, i+1, \ldots, j\} \cap \{k, k+1, \ldots, l\})$ *is not difficult.*

Note that the situation differs from classic Interval Arithmetic where the intersection of two intervals is always an interval enclosed within its two operands. Here, $[i,j]_b \bigwedge [k,l]_b$ is sometimes not included in both $[i,j]_b$ or $[k,l]_b$. This could be problematic w.r.t. the monotony of projection functions. Fortunately, the meet operation requires to minimize the cardinality of the resulting clockwise interval. Hence, each time a projection function is called on variable X, the cardinality of $CI(X)$ decrases. This ensures the computations progress towards a fixpoint.

$\bigvee$: the join operator The join operation is defined accordingly:

$$[i,j]_b \bigvee [k,l]_b \triangleq \square(\{i, i+1, \ldots, j\} \cup \{k, k+1, \ldots, l\})$$

Proposition 5. *Let* $CI(X) = [i,j]_b$ *and* $CI(Y) = [k,l]_b$*, then* $CI(X) \bigvee CI(Y)$ *can be defined as follows:*
if $wid(CI(X)) \geq 0 \wedge wid(CI(Y)) \geq 0$ *then*

$$CI(X) \bigvee CI(Y) = \begin{cases} [i,l]_b & \text{if } card([i,l]_b) \leq card([k,j]_b) \\ [k,j]_b & \text{otherwise} \end{cases}$$

Note that these two operations ($\bigwedge$,$\bigvee$) give the CI set a structure of a finite lattice.

3.2 Relations over CI

Let X, Y be two variables over $\mathbb{Z}_b$, the relation $X = Y$ leads to prune $CI(X)$ and $CI(Y)$ with the following rule: $CI(X), CI(Y) \leftarrow CI(X) \bigwedge CI(Y)$. In CI Arithmetic, the relation $X \leq Y$ leads to prune $CI(X) = [i, j]_b$ and $CI(Y) = [k, l]_b$ with the rule $CI(X) \leftarrow CI(X) \bigwedge [0, max(CI(Y))]$. Other relations can easily be derived from these ones.

3.3 Bound-consistency for modular integer constraints

From the formula given above, one can derive practical algorithms to perform bound-consistency on modular integer constraints. The simplest approach is to implement propagators on Clockwise Intervals within an AC-3 propagation algorithm. Once a CI becomes empty, then the constraint system is shown as being unsatisfiable. If none CI become void, then the resulting CIs encompass all the solutions of modular integer constraints.

For the linear fragment of modular integer constraints (i.e., addition, substraction, multiplication by a constant) this approach maintains optimal CIs at the cost of bounds computations. However, as soon as variable multiplication is encountered, optimality requires time-quadratic exploration of CIs. This is prohibitive in the context of 32-bits or 64-bits integer arithmetic. This problem is similar to the situation in bit-vector arithmetic [2] where variable multiplication requires time-quadratic computations on the number of bits. For these non-linear constraints, as said previously, one can gave up optimality by computing Clockwise Intervals that over-approximate optimal clockwise intervals. In the implementations described below, several propositions are made in this direction.

4 Implementations

In the context of the ANR CAVERN project, three distinct implementations of modular integer constraint solving were done. During this work, it appears that Clockwise Interval may be a unifying notion capturing the essence of modular integer interval computations.

4.1 MAXC

At INRIA Rennes, the Clockwise Interval Arithmetic shown above was directly implemented in MAXC, a solver dedicated to modular constraint solving. In a near future, this solver should be integrated within EUCLIDE [6], an automatic test data generator for critical C programs. The constraint system that is derived from EUCLIDE includes modular constraints based on arithmetic operators (+,-,*,div,mod) and high-level operators such as reification and global constraints dedicated to program verification. We do not detail these operators here as our paper is focussed on modular constraint solving. Propagators in MAXC are

implemented in C for efficiency reasons while the general propagation queue is implemented in Prolog. Each variable is associated to a CI and contracting propagators aim at pruning CIs of their inconsistent values. The size of variables that can be represented in MAXC ranges from 1 bit to 64 bits as these are the sizes typically found in primitive types in C. The data structure for encoding CIs maintains cardinality and width:

```
typedef struct {
  USH       empty     ;  /* is an empty domain ?                        */
  USH       sign      ;  /* is a signed domain ?                        */
  USH       size      ;  /* allowed size = 1,2,3,4,8,16,32 or 64 bits   */
  UL        min       ;  /* min_value of domain                         */
  UL        max       ;  /* max_value of domain                         */
  UL        wid       ;  /* absolute value of width of domain           */
  SSH       sign_wid  ;  /* sign of width: SINGLE is 0 (eg [3,3]),
                            PROPER +1 (eg [3,6]),IMPROPER -1 (eg [6,3]) */
  UL        card      ;  /* cardinality of domain. 0 is the whole domain*/
  ULL       basis     ;  /* basis of modular calculus. 0 denotes 2^64   */
} TYPE_LFD ;
```

In this data structure, USH stands for *unsigned short integer* which corresponds to 16-bits integers while UL stands for *unsigned long*, i.e. 32-bits integers. Other keywords can easily be understood as variations of these two. Note that encoding 64-bits integer Clockwise Interval arithmetics is still possible but greater formats cannot be encoded. The solver applies bound-consistency propagators on this data structure for $\oplus, \ominus, \otimes, ...$ It maintains optimal CIs for the linear fragment of these constraints. The input format of constraints is an intermediate one, where complex constraints have already been decomposed in simpler ones. Typical requests are of the form:

```
test1 :-                                      % In 3-bits integer arith.,
   solveur:init_env(E),          %  X = 5, Y in 2..7, Z in 5..0, Z = X*Y
   lfd:news([X,Y,Z],int(8),['X','Y','Z'],E),            % should produce
   lfd:equal(const('5'),X),                        % Y in 3..6, Z in 6..7
   lfd:equal(in('2','7'),Y),
   lfd:equal(in('5','0'),Z),
   lfd:equal('*',X,Y,Z),
   solveur:solve(E),
   lfd:affiche([X,Y,Z]).
```

Many operators still have to be implemented in order to capture modular integer constraints coming from C programs, including bit-to-bit operators (e.g., `&, |, ~`) , logical operators (e.g., `&&, ||`), nonlinear operators coming from destrucive assignment (i.e., `i *= i++` that correspond to constraint $i_2 = (i_1+1)^2$), and so on.

4.2 JSOLVER

JSolver is a IBM-ILOG Constraint-Based Programming library in (pure) Java. It is derived from the C++ library IBM-ILOG Solver and has been tailored for

the static and dynamic analyses of rule-based programs [3, 8]. Currently, these analyses are performed using an idealized integer arithmetic where modular computations are ignored. Consequently overflows on integers are reported as errors and the corresponding rule-based programs are rejected which is the expected behaviour, as these programs are exploited by end-users and not by developers.

We recently investigated the use of CP to perform static analysis of rules in order to optimize their compilation in a discrimination network [5]. Unlike the above usage, this requires using the program execution semantics where integer overflows are silently done (such as in Java). We report here on our first implementation of bound consistency for integer modular constraints by using classic intervals as defined in mathbooks: a classic interval $a..b$ with integer bounds a and b is the set of integers $\{x|a \leq x \leq b\}$. Let us consider two positive 32-bits integers and suppose we want to determine the range of the sum of these (signed) integers ranging from 1 to $2^{31} - 1$. By using an idealized semantics for integer computations, we get that the sum is ranging from 2 to $2(2^{31} - 1)$. Of course, this range could be exactly represented by using unbounded values such as bigInteger in Java or approximated by $2.. + infinity$. But, taking into account modular integer arithmetic, we found that the sum is actually ranging on $-2^{31}..-2$ union $2..2^{31}-1$. The classic interval which covers all these values is the set of all representable signed values on 32 bits $MIN\_INT..MAX\_INT$. Note that such classic intervals usually over-approximate the results that could be computed using Clockwise Intervals as, for example, the CI $[2, -2]_{2^{32}}$ on signed 32-bits integers corresponds precisely to $-2^{31}.. - 2$ union $2..2^{31} - 1$ that is over-approximated by the classic interval $-2^{31}..2^{31} - 1$. To give a flavor of inferences which could be made on classic interval for modular integer arithmetic, let us continue our example by constraining the sum to be greater than -2. Let x, y and z be three signed 32-bits integers such that z, the sum of x and y, is greater than -2. x and y are ranging on $1..MAX\_INT$ and z is ranging on $-1..MAX\_INT$. As the transformation $x = z - y$ (resp. $y = z - x$) is correct in modular integer arithmetic, we actually found that x (resp. y) is ranging on $1..MAX\_INT - 1$. As $z = x + y$, we deduce that z is ranging on $2..MAX\_INT$, then discovering that z is positive.

To formally define what our computations are, let us assume that we are dealing with modular integer with a machine representation ranging from a smaller integer denoted by m and a larger integer here denoted by M. let $x..y$ be a classic interval. u, v represent x and y in a (m, M) computer integer arithmetics if $m \leq u \leq M$, $m \leq v \leq M$ and $x = u + k_u(M - m + 1)$, $y = k_v(M - m + 1)$ for two integers k_u and k_v. We may then introduce a $cast_{m,M}$ function from classic intervals to (m, M) intervals with the following definition:

$$cast_{m,M}(x..y) = \begin{cases} u..v & \text{if } k_u = k_v \\ m..M & \text{if } k_u ! = k_v \end{cases}$$

This cast function provides concise definition for modular arithmetic. For example the (best) forward operator for the sum z of x and y is z in $cast_{m,M}(x_{min} + y_{min}..x_{max} + y_{max})$. As $z = x + y$ is equivalent to $x = y - z$ in modular arithmetic,

the (best) backward operator is defined by x *in* $cast_{m,M}(z_{min} - y_{max}..z_{max} - y_{min})$ and y *in* $cast_{m,M}(z_{min} - x_{max}..z_{max} - x_{min})$.

The multiplication by a scalar is not so straightforward. On 32-bits signed integer, the powers of 2 are divisors of 0 and the congruence domains [8] are not preserved. For example, $2 * MIN\_INT = 0$ and $3 * MIN\_INT$ which is equal to MIN_INT is not even divisible by 3. However, solving $ax = b$ on 32-bits signed integers is not so difficult. First we note that if a power of 2 divides a, it should also divide b for the equation to have a solution. By simplifying by this power of 2, say 2^p, we obtain $a'x = b' \ mod \ 2^{(32-p)}$. We find then the inverse u of $a' \ mod \ 2^{(32-p)}$. We infer that $x = (ua')x = u(a'x) = ub' \ mod \ 2^{(32-p)}$. Finally, we obtained a range for x in 32-signed integers and a congruence domain to be propagated. We can apply this method to the solving of ax in $m..M$ on 32-bits integers in a similar way. First, if 2^p divides a, we keep only the multiples of 2^p from $m..Max$. Then we simplify m and M to solve ax in $m'..M' \ mod \ 2^{(32-p)}$ by finding an inverse u of a', leading to x in $u*m'..u*M' \ mod \ 2^{(32-p)}$. Here again, we infer a range for $x \ mod \ 2^{32}$ and a linear congruence to be propagated. To end this short report on our preliminary implementation, we should say that the general multiplication of two variables is propagated by using the cast function. This leads very often to the top approximation of the full integers, being not complete but at least correct. For future implementations, we are thinking of making use of the k_u integer indicators in the cast function definition, as proposed in the tool COLIBRI described below. We also think to switch our implementation from classic intervals to clockwise intervals as the performance should be similar whilst the precision is improved.

4.3 COLIBRI

COLIBRI is a constraint library developped at CEA LIST for its test generation tools: GATeL for the functional testing of LUSTRE/SCADE models [10], PathCrawler for the structural testing of C code [11] and Osmose for the structural testing of binary code [1]. This library provides domains and constraints for integer, real and floating point interval arithmetics. Furthermore, a congruence domain is combined with the integer domain as described in [8].

The integer domain is implemented by union of intervals with finite bounds. These bounds can be any integer since we use big integers provided by the Gnu Multi-Precision library. This representation of integer domains allows to precisely represent improper clockwize intervals. For example, using clockwise intervals we have $[2, 4]_8 \oplus [4, 7]_8 = [6, 3]_8$. This interval corresponds to the following union of classic intervals $0..3 \cup 6..7$ which is denoted by $[0..3, 6..7]$ in COLIBRI.

In order to handle the signed and unsigned integer arithmetics used by computer languages, COLIBRI provides signed and unsigned modular arithmetics operations when the modulo is a positive power of two (i.e., when $b = 2^n$ with $n > 0$). For each operation op in $+, -, \times, div, rem, power$ we provide the operations $op_{s,n}$ and $op_{u,n}$ which correspond to the modular signed and unsigned versions of op. The implementation of these operations uses the following definition of modular operations.

$$\forall (A, B, C) \in {\mathbb{Z}_{2^n}}^3, A\ op_{su,n}\ B = C \equiv (\exists K, AopB = C + K \times 2^n)$$

The range of K can be easily characterized for each $op_{su,n}$ according to 2^n. For example, for the $+_{u,n}$ operation $-1 \leq K \leq 1$, while for the $\times_{u,n}$ operation $0 \leq K \leq 2^n - 1$.

Thus, according to the previous equivalence, the constraint propagators of any modular operation can be implemented with those of non modular operations. This is exactly the way modular operations are implemented in COLIBRI. For example, the constraint $A +_{u,n} B = C$ where variables A, B and C belong to $[0, 2^n - 1]$ is handled by the following conjunction of constraints: $A+B = X \wedge K \times 2^n = Y \wedge X+Y = C$ where the initial domain of K is $[0, 2^n-1]$, Notice that the congruence domain knows that variable Y is a factor of 2^n and as soon as C (resp. Y) handles a congruence one can infer a congruence for Y (resp. C).

For each modular operation, the variable K is a precise indication of underfow/overflow: when $K < 0$ this means that there is an underflow, when $K > 0$ this means that there is an overflow while when $K = 0$ there is no underfow/overflow. Such an indicator could be very helpfull for verification tools when checking computation w.r.t. underfow/overflow. This is why COLIBRI modular constraints handle a supplementary argument UO which abstracts the sign of K: UO belongs to $[-1, 1]$ and has the same sign as K. Any assignment of this variable UO can be used to force underflow ($UO = -1$), overflow ($UO = 1$) of normal computation ($UO = 0$). Moreover, one can force non normal behaviour by stating that $UO <> 0$.

To conclude this short presentation of modular operations in COLIBRI, let us remark that the accuracy of this implementation relies on the use of union of intervals with big integer bounds. This could be considered very expensive for constraints systems involving heavy computations. However, as shown by a recent experiment [2] using SMT-LIB benchmarks our implementation of modular arithmetics is competitive with powerfull SMT solvers.

5 Conclusions and perspectives

In this paper, we introduced Clockwise Intervals as a way to capture modular interger interval computations. We described three distinct implementations of modular integer constraint solving that have applications in program testing and analysis. We have also seen that finding optimal bounds in bound-consistency filtering of modular integer computations is not trivial and often requires approximations. For general multiplication and division, efficient ways to compute optimal bounds still need to be found. On the foundations of the approach, Clockwise Interval appears as a good tool to describe bound-consistency on modular integer computations but its relations with other Interval Arithmetics still need to be studied. On the applications of modular integer constraint solving, experimental evaluation is required in the diverse contexts presented earlier

in the paper, namely, automatic test inputs generation for C programs, test case generation for reactive programs and rule-based program analysis. Another perspective of this work concerns the way other dedicated constraint solver could be married with Clockwise Intervals. For example, the recent work of Bardin et al. in [2] showed that dedicated bitvectors operators could be efficiently coupled with classic intervals. It remains to find ways to integrate such arithmetics within Clockwise Intervals.

Acknowledgments

We would like to thank the members of the ANR CAVERN project who participated to our initial discussions on this topic, namely Bruno Berstel, Bernard Botella, Claude Michel, Michel Rueher, and Nicky Williams.

References

1. S. Bardin and P. Herrmann. Structural testing of executables. In *1th Int. Conf. on Soft. Testing, Verif. and Valid. (ICST'08)*, pages 22–31, 2008.
2. S. Bardin, P. Herrmann, and F. Perroud. An alternative to sat-based approaches for bit-vectors. In *Tools and Algorithms for the Construction and Analysis (TACAS'10)*, pages 84–98, 2010.
3. B. Berstel and M. Leconte. Using constraints to verify properties of programs. In *2nd Workshop on Constraints in Software Testing, Verification and Analysis, CSTVA'10*, 2010. Co-located with ICST'10 in Paris, April.
4. H. Collavizza, M. Rueher, and P. Van Hentenryck. Cpbpv: A constraint-programming framework for bounded program verification. In *Proc. of CP2008*, LNCS 5202, pages 327–341, 2008.
5. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
6. A. Gotlieb. Euclide: A constraint-based testing platform for critical c programs. In *2th IEEE International Conference on Software Testing, Validation and Verification (ICST'09)*, Denver, CO, Apr. 2009.
7. A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Proceedings of Computational Logic (CL'2000)*, LNAI 1891, pages 399–413, London, UK, July 2000.
8. M. Leconte and B. Berstel. Extending a cp solver with congruences as domains for software verification. In *1st Workshop on Constraints in Software Testing, Verification and Analysis, CSTVA'06*, 2006. Co-located with CP'06 in Nantes, September.
9. B. Marre, P. Mouy, and N. Williams. On-the-fly generation of k-path tests for c functions. In *Proceedings of the 19th IEEE Int. Conf. on Automated Software Engineering (ASE'04)*, Linz, Austria, September 2004.
10. Bruno Marre and Benjamin Blanc. Test selection strategies for lustre descriptions in gatel. *Electronic Notes in Theoretical Computer Science*, 111:93 – 111, 2005.
11. N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Proc. Dependable Computing - EDCC'05*, 2005.