Constraint solving on modular integers

* Arnaud Gotlieb*, Michel Leconte**, Bruno Marre***

* INRIA Research center of Bretagne - Rennes Atlantique
** ILOG Lab, IBM France
*** CEA List

ModRef’10 Workshop, 6/09/10
Software Verification with CP

- Automatic verification of programs (e.g., a C function or a Java method) requires the generation of test input that reach given locations.

```c
f( int i, int j )
{
    int tmp = i+j ;
    ...
    if( tmp > i*j) ...
}
```

Values of (i, j) to reach ... ?

requires to solve

\[ i + j > i \times j \]

- **Constraint-Based Testing** tools include techniques that address this problem with:
  - CP over Finite Domains techniques
  - Abstract domains computations (Intervals, Polyhedra, Congruences, ...)
Wrap-around integer computations

• Most architectures implement wrap-around arithmetic (modular integers):

  char (-128..127, 1 byte),
  unsigned char (0..255, 1 byte),
  short (-32768..32767, 2 byte),
  unsigned short (0..65535, 2 byte),
  long (-2147483648..2147483647, 4 byte),
  unsigned long (0..4294967295, 4 byte),
  ...

• Problem in the previously mentionned tools:

  Expressions are interpreted using ideal integer arithmetic rather than
  wrap-around integer arithmetic

• Example:

  the C expression
  short a,b,c; c=a+b
  should be interpreted as
  c=a+b mod(2^{16}) in -32768..32767
  rather than just
  c=a+b in inf .. sup
Programs that suppose wrap-around integer computations

• Good programming practices suggest taking care of integer overflows:

```c
unsigned long len = 2^{31};

int f(unsigned long buf) {
    if (buf + len < buf)
        ...
```

Value of buf to reach ... ?

• Typical analysis tools would incorrectly declare ... as being unreacheable!

NB: Simplifying $buf + len < buf$ in $len < 0$
is forbidden in wrap-around integer arithmetic!
Bound-consistency for integer computations

Let $a, b$ be unsigned over 4 bits
$a$ in $0..15$, $b$ in $0..15$

$b = 2 \times a$;  

// Ideal Arithmetic
// $a$ in $0..7$  $b$ in $0..14$

// Wrap-around arithmetic
// $a$ in $0..15$  $b$ in $0..14$
Bound-consistency for integer computations

Let $a, b$ be unsigned over 4 bits

$a$ in 8..9, $b$ in 0..15

$b = 2 \times a$;  \hspace{1cm} \text{// Ideal Arithmetic}

\hspace{1cm} \text{// fail}

\hspace{1cm} \text{// Wrap-around arithmetic}

\hspace{1cm} \text{// $a$ in 8..9, $b$ in 0..2}
Can we implement wrap-around interval ideal arithmetic with modulo?

- Yes, but results wouldn’t be optimal

\[ A = 8, \; B \text{ in } 2..4, \; C \neq A \times B \mod(16) \] (in SICStus clpfd)

gives \( C \text{ in } 0..15 \) although \( C=9, C=10, \ldots C=15 \) have no support

- \( 8 \times 2..4 = \{ 8 \times 2=0_{16}, \; 8 \times 3=8_{16}, \; 8 \times 4=0_{16} \} \)

\[ \subset 0..8 \]

smallest interval that contains all the double products!
Our approach: to build an Interval Constraint Solver using *Clockwise Intervals*

**Def 1: Clockwise Interval (CI)**

Let \( b = 2^\omega, x \) and \( y \) be two integers modulo \( b \),

a CI \([x,y]_b\) denotes the set \( \{x, x+1 \mod b, \ldots, y-1 \mod b, y\} \)

**Ex:** \([6,1]_8\) denotes the unordered set of integers modulo 8: \( \{6,7,0,1\} \)

By convention: \([0, b-1]_b\) is the canonical representation of \( \mathbb{Z}_b \)
Cardinality

Def 2: *Cardinality*
Let \([x,y]_b\) be a CI, then \(\text{card}([x,y]_b)\) is an integer such that:

\[
\begin{align*}
\text{card}([x,y]_b) &= b & \text{if } [x,y]_b &= [0, b-1]_b \\
&= (y - x + 1) \mod b & \text{otherwise}
\end{align*}
\]

Prop 1: A CI \([x,y]_b\) contains exactly \(\text{card}([x,y]_b)\) elements
Hull

- The hull of a set of modular integers $S$ is the smallest CI w.r.t. cardinality, that contains all the elements of $S$.

Def 3: **(Hull)** Let $S = \{x_1, \ldots, x_p\}$ be a subset of $\mathbb{Z}_b$, the hull of $S$ is a CI, noted $\square S$, $\square S = \operatorname{Inf}_{\operatorname{card}}( \{[x_i, x_j]_b \mid \{x_1, \ldots, x_p\} \subseteq [x_i, x_j]_b \})$

Prop 2: Let $S = \{x_1, \ldots, x_p\}$ be an ordered subset of $\mathbb{Z}_b$, and let $x_{-1}$ denotes $x_{p-1}$, then

$\square S = [x_i, x_{i-1}]$ where $i$ such that $\operatorname{card}([x_i, x_{i-1}])$ is minimized

Corollary: $\square S$ can be computed in linear time w.r.t. the size of $S$
Clockwise interval arithmetic

\[
[i,j]_b \oplus [k,l]_b = \{ (i \oplus k) \mod b, (i \oplus k+1) \mod b, \ldots, (j \oplus l) \mod b \}
\]

for any \( \oplus \) in \{\oplus, \ominus, \otimes, \ldots\}

(Addition)

\[
[i,j]_b \oplus [k,l]_b = [0, b-1]_b \quad \text{if card}([i,j]_b) = b \quad \text{or card}([k,l]_b) = b \\
= [(i+k) \mod b, (j+l) \mod b]_b \quad \text{otherwise}
\]

(Substraction)

\[
[i,j]_b \ominus [k,l]_b = [0, b-1]_b \quad \text{if card}([i,j]_b) = b \quad \text{or card}([k,l]_b) = b \\
= [(i-l) \mod b, (j-k) \mod b]_b \quad \text{otherwise}
\]
Where the things become more complicated!

- Multiplication by a constant: $k \otimes [i,j]_b$

- Unlike in classical Interval Arithmetic, results cannot be computed using only the bounds

  $5 \otimes [2,7]_8 = □\{10 \mod 8, \ldots, 35 \mod 8\} = [1, 7]_8$

- but, 1) in $\mathbb{Z}_2^w$, divisors of 0 are well-known

  2) Thanks to prop2, $□\{x_1, \ldots, x_p\}$ can be computed efficiently when $\{x_1, \ldots, x_p\}$ is ordered
• **Prop3:** Let $k \neq 2^n$, $q_1 = k_i \div b$, $q_2 = k_j \div b$, then

$$\text{Max}(k \otimes [i,j]_b) = b - d \quad \text{where} \quad d = \text{Min}_{q_1 < q \leq q_2} (q \times b \mod k)$$

and

$$\text{Min}(k \otimes [i,j]_b) = d' \quad \text{where} \quad d' = \text{Min}_{q_1 < q \leq q_2} (-q \times b \mod k)$$
For $k \cdot [i, j]_b$

computing the upper bound can be done modulo $k$ instead of modulo $b$!

$q \cdot b = q \cdot 2^w = 0 \mod b$

Then, $d = \text{Min}_{q_1 \leq q \leq q_2} (q \cdot b - k \cdot p)$

$\text{Max}(k \cdot [i, j]_b)$

Then, $d = \text{Min}_{q_1 \leq q \leq q_2} (q \cdot b \mod k)$ and
$\text{Max}(k \otimes [i, j]_b) = b - d$
\[ k = 5, \ i = 2, \ j = 7, \ b = 8 \]
\[ 5 \times [2,7]_8 = [1,7]_8 \]

**Prop3:** Let \( k \neq 2^n \), \( q_1 = k \cdot i \div b \), \( q_2 = k \cdot j \div b \), then

\[
\begin{align*}
\text{Max}(k \cdot [i,j]_b) &= b - d \\
\text{Min}(k \cdot [i,j]_b) &= d' \\
\end{align*}
\]

where \( d = \text{Min}_{q_1 \leq q_2} (q \cdot b \mod k) \)
and \( d' = \text{Min}_{q_1 \leq q_2} (-q \cdot b \mod k) \)

- \( q = 2 \), \( 16 \mod 5 = 1 \), \( -16 \mod 5 = 4 \)
- \( q = 3 \), \( 24 \mod 5 = 4 \), \( -24 \mod 5 = 1 \)
- \( q = 4 \), \( 32 \mod 5 = 2 \), \( -32 \mod 5 = 3 \)
Relations over Clockwise Intervals

• Inclusion, union and intersection of CIs are defined with their set-theoretic counterparts

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

• However, union and more surprisingly intersection are not closed over CIs, e.g.,

\[ [5, 2]_8 \cap [1, 6]_8 = \{1, 2, 5, 6\} \]

Hence, we define the meet and join operations using the hull operator

\[ [5, 2]_8 \text{ meet } [1, 6]_8 = \bigdiamond\{1, 2, 5, 6\} = [1, 6]_8 \]

• \( X = Y \) leads to prune both \( \text{CI}(X) \) and \( \text{CI}(Y) \) using \( \text{CI}(X) \) meet \( \text{CI}(Y) \)
Three implementations of constraint solving over modular integers (in progress)

• **MAXC (INRIA):**
  - Developed for EUCLIDE, a platform for verifying critical C programs
  - In SICStus Prolog (700loc) + C (300loc)
  - Direct implementation of Clockwise Intervals over 1, 2, 3, 4, 8, 16, 32 bits only
  - unsigned only, no conversions, few arithmetic and relations

• **JSOLVER (ILOG):**
  - Static analysis of rule-based programs (ILOG Rules)
  - Domain and Bound-consistencies on ideal integer arithmetic and
    - use of a cast function to map the results on wrap-around

• **COLIBRI (CEA):**
  - Constraints library used by CEA test generation tools (GATeL for LUSTRE models, PathCrawler for C code, Osmose for binary code)
  - Integer/Real/Floating points interval arithmetics (union of disjoint intervals), Congruences, Difference constraints
  - signed and unsigned cases
COLIBRI (CEA): 2 extra ideas

- For each \( op \) in \{+, -, *, \text{div}, \text{rem}\}, COLIBRI provides a modular version \( op_2^n \), modular constraint propagators are handled by non modular operations:
  \[ A \, op_2^n \, B = C \iff A \, op \, B = C + K \times 2^n \]
  The range of \( K \) varies according to signed/unsigned, \( n \) and \( op \).
  
  **Example:** \( A +_2^n B = C \)
  - Signed: \([A,B,C] :: [-2^{n-1}..2^{n-1}-1], K :: [-1..1]\)
  - Unsigned: \([A,B,C] :: [0..2^{n-1}], K :: [0..1]\)

- For each \( op_2^n \), an extra argument \( UO :: [-1..1] \) allows to read / provoke an underflow (\( UO = -1 \)), overflow (\( UO = 1 \)) or a nominal behavior (\( UO = 0 \))

  An extra constraint maintains the invariant \( \text{sign}(UO) = \text{sign}(K) \)
  When \( UO = K = 0 \), \( A \, op_2^n \, B = A \, op \, B \)

  **Example:** \( n = 3, A,B,C \) unsigned, \( A :: [2..4], B :: [5..7], C :: [0..7], UO :: [0..1] \)
  \( A +_2^3 B =_UO C \Rightarrow A + B = C + K \times 8 \) with \( K :: [0..1] \) and \( \text{sign}(K) = \text{sign}(UO) \)
  \( \Rightarrow C :: [0..3, 7] \)
For any arithmetic operator, compute intervals of $\mathbb{Z}$ and then project them on computer intervals using a cast function

- Let $[a, b]$ be an interval of $\mathbb{Z}$ and $u, v$ represent $a, b$ in a $(m, M)$ computer integer
  \[ a = u + k_u(M - m + 1), \quad m \leq u \leq M \]
  \[ b = v + k_v(M - m + 1), \quad m \leq v \leq M \]
- $\text{cast}_{m,M}([a, b]) = \begin{cases} 
[u, v] & \text{if } k_u = k_v \\
[m, M] & \text{otherwise}
\end{cases}$
Further work

- Finding optimal bounds for non-linear constraints is hard

→ practical solution: relaxing optimality using over-approximations,
  e.g., \( X \) in \( [a..b] \), \( Y \) in \( [c..d] \) then \( Z = X \times Y \) in \( \min(aY, Xc)..\max(bY,Xd) \)

- Finishing our three implementations and performing a serious experimental evaluation is indispensable → next step

- Deal with constraints where distinct basis are considered,
  e.g.,
  ```java
  short x;
  long y;
  x = (short) y;
  ```