Extending a CP Solver with Congruences as Domains for Program Verification

Michel Leconte[1] and Bruno Berstel[1,2]

[1] ILOG
9, rue de Verdun – 93250 Gentilly – France
[2] Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85 – 66123 Saarbrücken – Germany
{leconte,berstel}@ilog.fr

Abstract. Constraints generated for Program Verification tasks very often involve integer variables ranging on all the machine-representable integer values. Thus, if the propagation takes a time that is linear in the size of the domains, it will not reach a fix point in practical time. Indeed, the propagation time needed to reduce the interval domains for as simple equations as $x = 2y + 1$ and $x = 2z$ is proportional to the size of the initial domains of the variables. To avoid this *slow convergence* phenomenon, we propose to enrich a Constraint Programming Solver (CP Solver) with *congruence domains*. This idea has been introduced by [1] in the abstract interpretation community and we show how a CP Solver can benefit from it, for example in discovering immediately that $12x + |y| = 3$ and $4z + 7y = 0$ have no integer solution.

1 Introduction

Programs made of production (or condition-action) rules [2–5], which are at the basis of Business Rules Management Systems (BRMS) [6, 7], are gaining more and more interest in the industry, as a way to externalize the rapidly changing behaviors of applications, due to frequent regulation updates or to competitive pressure. However, in order to deliver the expected agility and robustness, the business users expects from the BRMS that they assist them in mastering their rapidly growing rule bases. This concern includes the verification of properties on the rule programs, as well as code navigation tools that are aware of the rule program semantics.

Program verification problems, that is, the question of whether a program satisfies a given property, can often be formulated as satisfiability and non-satisfiability problems. Using a CP Solver to solve such problems has the advantage of being able to address a large class of formulas. This comes at the price of completeness, but practical experience shows that it is most of the time effective [8]. The solutions that are found correspond to answers (witnesses or counterexamples) to the program verification questions.

However, the constraint problems that derive from program verification questions carry specificities that challenge the efficiency of a "plain" CP Solver. In

particular, and in constrast with combinatorial optimization problems, the domains of the input variables are very large, being typically only bounded by the machine representation of numbers. Also, because program verification often works by refutation, verification problems tend to produce unsatisfiable constraint problems. Since the time taken by CP Solvers to conclude to unsatisfiability may be proportional to the size of the domains of the variables, they result in being inefficient in some cases, and especially on bug-free programs, for which the constraint problems are unsatisfiable. This is illustrated by Example 1 in Section 2.

In this paper we propose to incorporate *congruences as domains* into CP Solvers, in order to prove the unsatisfiability of equations on integer variables more efficiently, that is, in a time independent from the size of the domains of the variables. Congruences are about the division of an integer by another, and the remainder in this division. Congruence analysis comes from the static analysis community. It has been introduced by Granger in [1]. As shown in that paper [1], congruence analysis is not restricted to linear equations, but can handle also general multiplication expressions. We extend its scope to other non-linear expressions such as absolute value, or minimum.

In Section 2 we describe the slow convergence issue, and how it relates to Program Verification. Then in Section 3 we study the existing related work. The technique of congruences as domains, as well as the scope we address, are introduced in Section 4. Section 5 provides the formulas for propagation, and Section 6 studies the cooperation between congruence and interval domains. Finally Section 7 illustrates the slow convergence phenomenon with experimental data, Section 8 presents the usage that was made of this technique in ILOG's commercial products, and Section 9 concludes.

2 Slow Convergence in Program Verification

In this paper we will use the following notations (all variables below are elements of $\mathbb{Z}$, the set of the integers).

- We will note $a\mathbb{Z} + b$ the set $\{az + b \mid z \in \mathbb{Z}\}$.
- For $x \in a\mathbb{Z} + b$, we will also note $x \equiv b\,[a]$.
- We will use $a \wedge b$ for the greatest common divisor of a and b.
- We will use $a \vee b$ for the least common multiplier of a and b.

As mentioned in the introduction, a particularity of constraint problems related to program verification, is that the input variables behave as if not bounded. For example, integer input variables are often supposed to take any value from machine integers according to their type. This is completely different from what happens when using a constraint solver for solving combinatorial problems, where we always try to reduce the initial domain of variables as much as it can be with respect to the problem.

Consider a simple constraint such as, say, $2x + 2y = 1$ where x and y are integer variables with value ranging from $-d$ to d, for some integer d. Obviously,

there is no solution to this constraint, and the "usual" interval reduction will find it, by reducing the domains of x and y down to empty sets. But to achieve this, the interval reduction will have to step through all the domains $[-d, d]$, then $[-d+1, d-1]$, etc. up to empty ones.

We stress that this *slow convergence* phenomenon occurs during the propagation of constraints: the time taken to reach a fix point is asymptotically proportional to the width of the domains. Propagation occurs both initially and during labelling; as a result, slow convergence may happen when searching for a solution. For example, the equation $2x + 2y + z = 1$ leads to the (slow) propagation of the previous constraint $2x + 2y = 1$ if z has been assigned 0 during the search. It is to avoid both cases that we have implemented congruences as domains.

The slow convergence issue over real numbers has motivated the development of special interval narrowing techniques in [9]. Unfortunately, they do not apply to the integer issue.

From the point of view of program verification, such problems can occur in various situations. Consider for instance a program containing the following loop: `while (x is even) increment x`. This program always terminates. As mentioned in [10], a way to prove it is to prove that the conjunction of the loop test expressed on the program states before and after one loop step, is unsatisfiable. Here the program state is the value of x; after one loop step it equals $x + 1$. To conclude to the termination of the program, a prover may thus want to show that the constraint $\texttt{even}(x) \wedge \texttt{even}(x+1)$ is unsatisfiable. This boils down to the constraint problem exposed above, with the same slow convergence problem.

As other examples of program verification problems that lead to proving that a conjunction of integer equalities is unsatisfiable, consider the problem of overlapping conditions in a guarded integer program. The original motivation for the work presented in this paper comes from the verification of rule programs, but the overlapping conditions problem may arise in any context that involves guarded commands.

Example 1. Consider the following program, which implements in some fictious guarded command language a simple version of the function that returns the number of days in a Gregorian calendar year.

```
function nbOfDays (y : int) : int is
   y === 0 mod 4  ->  366  |
   y === 1 mod 4  ->  365  |
   y === 2 mod 4  ->  365  |
   y === 3 mod 4  ->  365
end
```

The question is whether the guards in this program overlap or not.

The answer is 'no', that is, the program is bug-free (with respect to the question). To prove it, we shall first translate the guards into constraints, which

gives the four constraints $y = 4x_i + i$, for $i = 0, 1, 2, 3$. In these constraints y and x_i are integer variables lying in $[-d-1, d]$ for some integer d (potentially $2^{31}-1$). Then we shall prove that for any two distinct i and j between 0 and 3, the conjunction $y = 4x_i + i \wedge y = 4x_j + j$ is unsatisfiable. As seen previously in this section, interval reduction can achieve this, but will need d steps. The rest of this paper shows that using congruences as domains allows a CP Solver to prove the unsatisfiability in a fixed number of steps.

3 Related Work

Congruence analysis has been introduced by Granger [1, 11] with applications to automatic vectorization. Today congruence analysis is an important technique, especially to verify pointer alignment properties [12, 13]. In this paper we extend its scope beyond linear equations and multiplication, to other non-linear expressions such as absolute value, or the minimum operator.

Congruence domains have also been extended to constraints of the form $x - y \equiv b\,[c]$ [14, 15]. Toman *et al.* proposed in [16] a $O(n^4)$ normalization procedure for conjunctions of such constraints, Miné improved it to $O(n^3)$ in [14]. Grids [17, 18] are another extension which addresses *relational* congruence domains, in the presence of equalities of the form $\sum a_i x_i \equiv b\,[c]$, while we address the more specific non-relational domains for $x \equiv b\,[c]$. In [19], Granger proposes an extension of the congruence analysis by considering sets of rationals of the form $a\mathbb{Z} + b$, where $a, b \in \mathbb{Q}$.

In this paper, we extend a solver based on the finite domain CP Solver ILOG JSolver [20] with congruence as domains for variables. Very few CP Solvers reason with congruence. The Alice system [21] and its successor Rabbit [22] implement some congruence reasoning capabilities as part of formal constraint handling. Let us look at an example, which was taken from [22].

Example 2. Find all integer *positive* solutions of $x^3 + 119 = 66x$.

From $x^3 < 66x$, Rabbit finds that $x^2 < 66$, and then $x < 8$. Rabbit then performs two factorizations, namely $119 = 7 \times 17$ and $66 = (3 \times 17) + 15$. It then uses a congruence reasoning to deduce from these factorizations that $x^3 \equiv 15x\,[17]$, which can also be written $x(x^2 - 15) \equiv 0\,[17]$. Rabbit then applies the deduction rule

$$\text{if } a \times b \equiv 0\,[k] \text{ and } k \text{ is prime, then } a \equiv 0\,[k] \text{ or } b \equiv 0\,[k]$$

to deduce that $x \equiv 0\,[17]$ or $x^2 \equiv 15\,[17]$. Since x is positive and $x < 8$, the constraint $x \equiv 0\,[17]$ is always false. Finally $x^2 \equiv 15\,[17]$ leads to $x = 7$ as this is the only admissible value for x among $[1, 7]$.

As we can see from Example 2, the congruence capabilities of Rabbit are pretty important, and they are based on redundant modular equations generation. In this example, it involves the factorization of 119 as 7×17.

To solve this example, congruence domains of the form $a\mathbb{Z}+b$ as we propose are not enough. However, here pure interval reduction is enough to find the solution. When the example is given to ILOG JSolver, the domain of x starts at $[1, 2^{31}-1]$, and is reduced to $[2, 1290]$, then to $[3, 43]$, to $[5, 13]$, to $[6, 9]$, to end with $x = 7$.

As we will see, interval and congruence domains interact smoothly [1, 12, 13]. This is an example of the so-called reduced product operation of the theory of abstract interpretation [1, 23, 24]. Numeric domains such as intervals and congruences are available in Static Analysis Systems such as Astrée [25] or the Parma Polyhedra Library [26, 27].

Finally, another approach to avoiding the slow convergence problem on constraints such as $x = 2y + 1$ and $x = 2z$, is to use an integer linear solver inside the CP Solver. Note that this would handle only linear equations.

4 Congruences as Domains

4.1 Scope

The scope of constraints that we consider here extends to any equality constraint over integer variables and expressions. The integer expressions are built using the usual $+$, $-$, $\times$, $\div$ arithmetic operators, as well as the power, absolute value, minimum, and maximum ones.

We also consider *element expressions* in the form $\mathbf{t}[i]$, where $\mathbf{t}$ is an array and i is an integer variable. The element expression denotes the i-th element of the array. In simple element expression, this element is an integer; in generalized ones, the element is an integer variable. An *element constraint* is a constraint of the form $z \in \{\mathbf{t}[i]\}$, which amounts to the disjunction $\bigvee_i z = \mathbf{t}[i]$.

Finally, we also consider *if-then-else expressions* in the form $\mathsf{if}(c, e_1, e_2)$, where c is a constraint, and the e_i are integer expressions. The if-then-else expression denotes the e_1 expression if the constraint c is true, and the e_2 expression if the constraint c is false.

Although the whole range of integer constraints is covered, congruence analysis is of course not a decision procedure. That is, congruence analysis alone will not always detect the unsatisfiability of a set of integer constraints. And this does not harm, since it is simply meant to strengthen the constraint propagation.

4.2 Using Congruences on Interval Domains

Example 3. Find all integer solutions of $2x + 4y + 6z = 1$.

On the example above, the interval-based constraint propagation will perform no bound reduction at all. In particular, the unsatisfiability of the constraint will not be detected by constraint propagation.

A congruence reasoning shows that the expression $2x + 4y + 6z$ is even, and thus cannot be made equal to 1. At least, this illustrates a missing propagation.

Remember that a constraint $\sum_i a_i x_i = c$ has no solution if the greatest common divisor $\bigwedge_i a_i$ does not divide the constant c. We can use this property in the propagators of integer linear constraints: we compute the greatest common divisor of the coefficients of uninstantiated variables, and check if it divides the constant minus the sum of the $a_i x_i$ for instantiated variables.

This **passive** use of a congruence constraint, where congruences are used to update the bounds of interval domains, may already be useful. In addition it has little overhead on propagation time since this check has to be done up-front, and then only when a variable becomes instantiated.

4.3 Storing Congruence Information Separately

Example 4. Find all integer solutions of the problem made of the constraints $2x + 4y + 3z = 1$ and $z = 2t + 12$.

The passive use of congruence information just described will not detect that z cannot be even in $2x + 4y + 3z = 1$. Thus the unsatisfiability of the two constraints will be not detected. However, it would be a bad idea to use such a congruence constraint in an **active** way without caution.

Imagine for example that the congruence constraint not only checks for the constants dividing the greatest common divisor, but also *adjusts* the bounds of the domains of variables accordingly. Let us say that propagating $2x+4y+3z = 1$ would lead to adjust the bounds of z in such a way that these bounds are not even. Coming back to Example 4, an empty domain will be found for z, as the constraints will eventually lead to a domain with both odd and even bounds. Unfortunately, this would exhibit a slow convergence behavior since the bounds would change by one unit at a time.

The way to solve this last problem is to share the congruence information between constraints, that is to say to equip variables with congruence information, as opposed to hide it in the actual values of their bounds. Consequently, in the very same way we associate a point wise finite domain to each integer variable, we associate to each of them a congruence domain in the form of a pair (a, b) that represents the set $a\mathbb{Z} + b$. Then for each expression, the congruence domain of the expression can be computed from the congruence domains of the sub-expressions, using the formulas detailled in next section. Similarly the computed congruence domains are propagated by equalities constraints to reduce the congruence domains of the variables.

This way, the unsatisfiability of the two constraints $x = 2y$ and $x = 2z + 1$ is found by a congruence reasoning deducing that x should be both even and odd. This reasoning requires a number of steps which is independent from the size of the domains of the variables involved.

This active use of congruence information, where domains are reduced, subsumes the passive use described previously, which only performs divisibility checks.

5 Propagation of Congruences as Domains

5.1 Propagation Through Operations

Each integer variable has a congruence domain, noted $a\mathbb{Z} + b$, which represents all possible values for this variable. We use $0\mathbb{Z} + b$ to represent the constant b, and $1\mathbb{Z} + 0$ as the domain of a variable with all integers as possible values.

Now we have to define how to compute the congruence domain for expressions. We only give here the formulas for the addition and multiplication operations. These formulas, and those for substraction and division, can be found in [1]. Given $x \in a\mathbb{Z} + b$ and $y \in a'\mathbb{Z} + b'$:

$$x + y \in (a \wedge a')\mathbb{Z} + (b + b') \tag{1}$$

$$x \times y \in (aa' \wedge a'b \wedge ab')\mathbb{Z} + bb' \tag{2}$$

One can note that the square expression has a more precise characterization than the one derived from the general multiplication case. Given $x \in a\mathbb{Z} + b$:

$$x^2 \in (a^2 \wedge 2ab)\mathbb{Z} + b^2 \tag{3}$$

Let us look now at the union expressions, which result from constraints of the form $z \in \{x, y\}$. Given $x \in a\mathbb{Z} + b$ and $y \in a'\mathbb{Z} + b'$:

$$\text{if } z \in \{x, y\} \text{ then } z \in (a \wedge a' \wedge |b - b'|)\mathbb{Z} + b \tag{4}$$

The dissymmetry between b and b' in this formula is only apparent. Indeed, let α denote the greatest common divisor of a, a', and $|b - b'|$ appearing in (4): in particular it divides $|b - b'|$. That is, $\exists k \in \mathbb{Z}, b - b' = \alpha k$. In other words, we have $b \equiv b'\ [\alpha]$.

This formula for the union gives the formula for *if-then-else* expressions. Remember that $\mathsf{if}(c, e_1, e_2)$ is an expression taking the value e_1 when c is true and e_2 when c is false. If the constraint c is known to be true (resp. false), then the congruence domain for if-then-else is the congruence domain of e_1 (resp. e_2). However if the truth value of the constraint is unknown, then the expression has a congruence domain which is the union of the congruence domains of the two expressions. (This makes union an over-approximation of *if-then-else* expressions.) Given $x \in a\mathbb{Z} + b$, $y \in a'\mathbb{Z} + b'$, and a constraint c:

$$\mathsf{if}(c, x, y) \in \begin{cases} a\mathbb{Z} + b & \text{if } c \text{ is known to be true} \\ a'\mathbb{Z} + b' & \text{if } c \text{ is known to be false} \\ (a \wedge a' \wedge |b - b'|)\mathbb{Z} + b & \text{otherwise} \end{cases} \tag{5}$$

This also leads to the formula for a min expression, as $\min(x, y)$ is equivalent to $\mathsf{if}(x < y, x, y)$. Formulas for max and absolute value may be easily found if we remark that $\max(x, y) = \mathsf{if}(x < y, y, x)$ and $|x| = \mathsf{if}(x < 0, -x, x)$.

Finally, for an array $\mathbf{t}$ of integer variables and an integer variable i, the expression $z = \mathbf{t}[i]$ is equivalent to $z \in \{\mathbf{t}[j]\}$ for all values j which are non-negative, and less than the length of the array $\mathbf{t}$.

5.2 Propagation Through Equality

We now indicate how to deal with equality constraints. As usual when propagating through equality, we just have to compute the intersection of the domains. Given $x \in a\mathbb{Z} + b$ and $y \in a'\mathbb{Z} + b'$:

$$\text{if } x = y \text{ then } x \in \begin{cases} (a \vee a')\mathbb{Z} + b'' & \text{if } (a \wedge a') \text{ divides } (b - b') \\ \emptyset & \text{otherwise} \end{cases} \tag{6}$$

The number b'' can be computed as follows. Let $x = au + b$ and $y = a'v + b'$, the equality $x = y$ gives $au + b = a'v + b'$, that is, $au - a'v = b' - b$. Since $a \wedge a'$ divides $b - b'$, this can be simplified by $a \wedge a'$ into $\alpha u - \alpha' v = \beta$. Since α and α' are relatively prime, Bezout's theorem ensures that there exist u_0 and v_0 such that $\alpha u_0 + \alpha' v_0 = 1$. These numbers can be computed using a generalized Euclid's algorithm [28]. Combining the last two equations gives $\alpha(u - \beta u_0) = \alpha'(v + \beta v_0)$. Since α and α' are relatively prime, $u - \beta u_0$ is a multiple of α'. From $x = au + b$, we have $x \in \alpha' a\mathbb{Z} + b''$, where $b'' = b + \beta u_0$.

If the equality constraint involves expressions instead of variables, then the congruence domains of the expressions are used to compute the intersection. This resulting domain is then downward propagated to the sub-expressions of the expressions until it falls back to the variables.

Example 5. Find all integer solutions to $4x = 3|y| + 2$.

Let us use Example 5 to illustrate how the propagation of congruence domains proceeds. In the absence of further information, we have $x, y \in 1\mathbb{Z} + 0$. The formulas for addition (1), multiplication (2), and absolute value (5) give that $4x \in 4\mathbb{Z} + 0$ and $3|y| + 2 \in 3\mathbb{Z} + 2$.

The formula (6) for the equality constraint gives that both expressions belong to $12\mathbb{Z}+8$. Since $4x \in 12\mathbb{Z}+8, x \in 3\mathbb{Z}+2$. Since $3|y|+2 \in 12\mathbb{Z}+8, |y| \in 4\mathbb{Z}+2$. The absolute value can be decomposed into the case where $y \in 4\mathbb{Z} + 2$, and the case where $-y \in 4\mathbb{Z} + 2$. This latter case gives $y \in 4\mathbb{Z} - 2$, which is the same as $4\mathbb{Z} + 2$. Eventually $y \in 4\mathbb{Z} + 2$. The domains cannot be further reduced: a fix point is reached.

6 Cooperation of Congruences and Intervals

The idea here is to merge the two notions and to consider domains of the form $a\mathbb{Z}+b \cap [min, max]$. In the Abstract Interpretation framework, this corresponds to the reduced cardinal product of congruence domains and interval domains. It is called Reduced Interval Congruence (RIC) in [12, 13]. By combining the two domains, information coming from interval domains will be used by the congruence domain and vice-versa.

Let us first examine how to communicate information from interval domains to congruence domains.

- When a variable is bound, as for instance in $x = b$, this can be formulated in congruences as $x \in 0\mathbb{Z} + b$.

– When it is found that $x \in \{b_i\}$ for some constants b_i, this implies that $x \in (\bigwedge_{i>0} |b_i - b_0|)\mathbb{Z} + b_0$.
– For an element constraint $z \in \{\mathbf{t}[i]\}$, the range of the variable i restricts the elements of $\mathbf{t}$ that are to be taken into account to compute the congruence domain of z.

To communicate information from congruence domains to interval domains, one will use the fact that the bounds of a variable must lie in the same congruence domain as the variable itself. That is, if $x \in [min, max]$ and $x \in a\mathbb{Z} + b$, then min and max must be adjusted in order to belong to $a\mathbb{Z} + b$. When $a \neq 0$, the adjusted min is $a\lceil (min - b)/a \rceil + b$ and the adjusted max is $a\lfloor (max - b)/a \rfloor + b$.

If the diameter $max - min$ is less than a, the variable will have a singleton or empty domain. For instance if the interval domain had been reduced to $[0, a-1]$, then the variable can be instantiated to b, which is the only element of $a\mathbb{Z} + b \cap [0, a-1]$. Similarly, if the interval domain had been reduced to $[0, |b| - 1]$, then the solver fails, as $a\mathbb{Z} + b \cap [0, |b| - 1] = \emptyset$ for any a.

Also, for an element constraint $z \in \{\mathbf{t}[i]\}$, the congruence domain of z is to be taken into account to remove from the index variable domain the values i_0 for which $z = \mathbf{t}[i_0]$ cannot be satisfied.

Let us close this section with a example showing non-trivial reductions.

Example 6. Consider the two constraints $4x = 3y + 2$ and $|x| - 12z = 2$.

We have already seen that the first constraint leads to $x \in 3\mathbb{Z} + 2$ and $y \in 4\mathbb{Z}+2$. Now, looking at the second constraint, we deduce that $|x| \in 12\mathbb{Z}+2$. Since $|x| = \mathsf{if}(x < 0, -x, x)$, we deduce that $x \in 12\mathbb{Z} + 10$ (if $x < 0$) or $x \in 12\mathbb{Z} + 2$ (if $x \geq 0$). Because $12\mathbb{Z} + 10 \cap 3\mathbb{Z} + 2 = \emptyset$, we are left with $x \in 12\mathbb{Z} + 2$ and $x \geq 0$.

7 Experimental Illustration

In this section we describe two examples that suffer from the slow convergence problem, and we provide experimental data that exhibits the phenomenon.

Example 7. Solve $2x + 3y + 6z = 2$, with $x, y, z \in [-10^d, 10^d]$.

In Table 1, the times mentioned are the time taken to generate the first solution. The variable x is instantiated first to -10^d, then to $-10^d + 1$ and finally to $-10^d + 2$ which leads to a solution. The numbers show that without congruence domains this time is proportional to the size of the domains, while it is not when using congruence domains. The result is then found in a time lower than what can be measured; this is interesting per se, but the table shows that this time does not depend on the size of the domains.

Example 8. Prove that $2x + 2y = 1$, with $x, y \in [-10^d, 10^d]$, has no solution.

In Table 2, the times mentioned are the time taken during the propagation, which concludes to the unsatisfiability of the constraint. Here also, the numbers show that without congruence domains this time is proportional to the size of the domains, while it is not when using congruence domains.

Value of d	Time without C.D.	Time with C.D.
4	0.04 s	0 s
5	0.19 s	0 s
6	1.88 s	0 s
7	18.85 s	0 s
8	241.82 s	0 s
9	946.57 s	0 s

Table 1. Times taken for solving Example 7.

Value of d	Time without C.D.	Time with C.D.
4	0.12 s	0 s
5	0.06 s	0 s
6	0.62 s	0 s
7	6.16 s	0 s
8	61.96 s	0 s
9	871.05 s	0 s

Table 2. Times taken in propagation for Example 8.

8 Industrial Usage

From a marketing perspective, Program Verification is an important feature for BRMS [29]. As mentioned in the introduction, it aims at helping the non-technical users to master the rule programs their author using the system. This takes both the form of verification of properties on the program, and of semantics-aware code navigation tools. For commercial products such as ILOG JRules, this is a key differentiator.

As an illustration, one of the verification tasks we perform is to detect when a rule is never applicable, that is, when the tests in its condition part are always unsatisfiable. When the user creates a rule within the Eclipse IDE [30], a rule that is never applicable will be immediately signaled, with the tests causing the unsatisfiability highlighted.

We have embedded in ILOG JRules a new verification solver, as a Java library built on a Constraint-based Programming Solver derived from ILOG JSolver [20]. This CP Solver is part of ILOG JRules since release 4.5 which was delivered in 2003, and has kept evolving since. The passive use of congruence as presented in section 4 is present in ILOG JRules since release 6.0. We are now implementing congruence as domains for the next release of ILOG JRules.

9 Conclusion

Integer constraint propagation exhibits a *slow convergence* phenomenon when the time to reach a fix point or to fail is proportional to the size of the domains of the variables.

To avoid this phenomenon for some integer equality constraints, we added to a CP Solver some congruence reasoning capabilities. We have taken the idea of equiping the variables with congruence domains from the abstract interpretation community [1], as it leads to efficient and scalable implementations. We have shown how a CP Solver can benefit from these congruence domains with several examples, concluding with illustrations on the interaction of interval and congruence domains.

This work is part of the already distributed ILOG JRules product, and will be completely integrated in the next ILOG JRules release.

References

1. Granger, P.: Static analysis of arithmetic congruences. International Journal of Computer Math (1989) 165–199
2. Allen Newell, H.A.S.: Human problem solving. Prentice Hall, Englewood Cliffs, NJ, USA (1972)
3. Davis, R., Buchanan, B.G., Shortliffe, E.H.: Production rules as a representation for a knowledge-based consultation program. Artif. Intell. **8**(1) (1977) 15–45
4. Forgy, C.: Rete: A fast algorithm for the many patterns/many objects match problem. Artif. Intell. **19**(1) (1982) 17–37
5. Baralis, E., Widom, J.: An algebraic approach to static analysis of active database rules. ACM Trans. Database Syst. **25**(3) (2000) 269–332
6. ILOG: ILOG JRules. (2006) `http://www.ilog.com`.
7. JBoss: Drools. (2006) `http://www.drools.org`.
8. Collavizza, H., Rueher, M.: Exploration of the capabilities of constraint programming for software verification. In Hermanns, H., Palsberg, J., eds.: TACAS. Volume 3920 of Lecture Notes in Computer Science., Springer (2006) 182–196
9. Lhomme, O., Gotlieb, A., Rueher, M.: Dynamic optimization of interval narrowing algorithms. J. Log. Program. **37**(1-3) (1998) 165–183
10. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In Steffen, B., Levi, G., eds.: VMCAI. Volume 2937 of Lecture Notes in Computer Science., Springer (2004) 239–251
11. Granger, P.: Static analysis of linear congruence equalities among variables of a program. In Abramsky, S., Maibaum, T.S.E., eds.: TAPSOFT, Vol.1. Volume 493 of Lecture Notes in Computer Science., Springer (1991) 169–192
12. Balakrishnan, G., Reps, T.W.: Analyzing memory accesses in x86 executables. In Duesterwald, E., ed.: CC. Volume 2985 of Lecture Notes in Computer Science., Springer (2004) 5–23
13. Venable, M., Chouchane, M.R., Karim, M.E., Lakhotia, A.: Analyzing memory accesses in obfuscated x86 executables. In Julisch, K., Krügel, C., eds.: DIMVA. Volume 3548 of Lecture Notes in Computer Science., Springer (2005) 1–18
14. Miné, A.: A few graph-based relational numerical abstract domains. In Hermenegildo, M.V., Puebla, G., eds.: SAS. Volume 2477 of Lecture Notes in Computer Science., Springer (2002) 117–132
15. Bagnara, R.: Data-Flow Analysis for Constraint Logic-Based Languages. PhD thesis, Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56125 Pisa, Italy (1997) Printed as Report TD-1/97.
16. Toman, D., Chomicki, J., Rogers, D.S.: Datalog with integer periodicity constraints. In: SLP. (1994) 189–203

17. Bagnara, R., Dobson, K., Hill, P.M., Mundell, M., Zafanella, E.: Grids: A domain for analyzing the distribution of numerical values. In: LOPSTR. (2006)
18. Müller-Olm, M., Seidl, H.: A generic framework for interprocedural analysis of numerical properties. In Hankin, C., Siveroni, I., eds.: SAS. Volume 3672 of Lecture Notes in Computer Science., Springer (2005) 235–250
19. Granger, P.: Static analyses of congruence properties on rational numbers (extended abstract). In Hentenryck, P.V., ed.: SAS. Volume 1302 of Lecture Notes in Computer Science., Springer (1997) 278–292
20. ILOG: ILOG JSolver. (2000) http://www.ilog.com.
21. Laurière, J.L.: A language and a program for stating and solving combinatorial problems. Artif. Intell. **10**(1) (1978) 29–127
22. Laurière, J.L.: Programmation de contraintes ou programmation automatique. Technical report, L.I.T.P. (1996) http://www.lri.fr/~sebag/Slides/Lauriere/Rabbit.pdf.
23. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. (1977) 238–252
24. Cousot, P., Cousot, R.: Static determination of dynamic properties of generalized type unions. In: Language Design for Reliable Software. (1977) 77–94
25. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The Astrée analyzer. In: ESOP'05. (2005)
26. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.M.: Possibly not closed convex polyhedra and the Parma Polyhedra Library. In Hermenegildo, M.V., Puebla, G., eds.: Static Analysis: Proceedings of the 9th International Symposium. Volume 2477 of Lecture Notes in Computer Science., Madrid, Spain, Springer-Verlag, Berlin (2002) 213–229
27. Parma Polyhedra Library: PPL. (2006) http://www.cs.unipr.it/ppl.
28. Knuth, D.E.: Seminumerical Algorithms. Second edn. Volume 2 of The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts (1981)
29. Hendrick, S.D.: Business Rule Management Systems: Addressing Referential Rule Integrity. IDC. (2006) http://www.idc.com/getdoc.jsp?containerId=201262.
30. The Eclipse Consortium: Eclipse 3.0. (2005) http://www.eclipse.org.