Constraint Logic Programming over Infinite Domains with an Application to Proof

Sebastian Krings Michael Leuschel
Institut für Informatik, Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{krings,leuschel}@cs.uni-duesseldorf.de

We present a CLP(FD)-based constraint solver able to deal with unbounded domains. It is based on constraint propagation, resorting to enumeration if all other methods fail. An important aspect is detecting when enumeration was complete and if this has an impact on the soundness of the result. We present a technique which guarantees soundness in the following way: if the constraint solver finds a solution it is guaranteed to be correct; if the constraint solver fails to find a solution it can either return the result "definitely false" in case it knows enumeration was exhaustive, or "unknown" in case it was aborted. The technique can deal with nested universal and existential quantifiers. It can easily be extended to set comprehensions and other operators introducing new quantified variables. We show applications in data validation and proof.

1 Introduction

Tool support is vital for the success of formal methods in general, and state-based formal methods in particular. Some key technologies which enable effective validation of formal models are model checking, proof but *constraint solving* as well. Indeed, constraint solving enables animation and model checking of high-level formal models, it enables a large range of validation tasks ranging from bounded model checking to constraint-based deadlock detection. It is also crucial for test-case generation from formal models and can be used for finding counter examples to proof obligations.

Various techniques can be used to solve constraints expressed in specification languages like B, Z, TLA, Alloy, or VDM. The main techniques are SAT solving, SMT solving and constraint programming. SAT solving in this area has been made popular and practical via Alloy [16].

Thus far, these techniques were often limited to first-order constraints and unable to deal with unbounded data values. Let us examine the simple constraint $x = 10 * 10$ over the integer variable x. To solve this constraint using SAT solving the integer x has to be represented by propositions, i.e., as a bit-vector. For this, it is important to know how many bits are needed, both for x itself and for intermediate values that can occur while computing x. If one chooses too few bits, then the SAT solver may fail to find a solution where one exists, or report a solution where none exists (in case overflows are not detected).[1] Also, the encoding of values as bit vectors reaches its limits for more involved types like relations over large domains or higher-order values. Take for example a relation r which takes a set of elements over a domain D and another set of elements over D. If D is of size 10, we need $2^{20} =$ bits to represent a possible value of r. If D is of size 20, we need $2^{40} = 1,099,511,627,776$ bits, i.e., already 128 Gigabyte to store one variable.

[1] Alloy recently has added overflow detection; but in case no model was found we do not know whether an overflow may have prevented finding a solution.

To appear in EPTCS.

2 Constraint Solving

The key challenges when solving constraints in high-level languages such as B are universal and existential quantifications as well as set comprehensions and lambdas. Each can be arbitrarily nested and are not limited to finite values. So far we have tried different approaches to constraint solving as extensions to the CLP(FD)-based kernel of our model checker ProB in order to enable it to handle infinite domains: We developed a translation to SAT via Kodkod [27] and we integrated Z3 [9, 20].

The different techniques each have their own strengths and weaknesses: While there are highly efficient algorithms for SAT solving, encoding of higher-order constraints is often not feasible. Usually, the domain of integers has to be limited in order to allow bitlevel encoding of arithmetic. This is even more problematic if higher-order logic or set theory come into play. Due to a combinatorial blowup, resulting SAT constraints contain too many variables and become unsolvable.

SMT solvers on the other hand rely on decision procedures for different underlying logics. Following the DPLL(T) algorithm, these solvers enumerate predicate values and try to infer logical consequences. Hence, it is easy to extract a proof from an unsatisfiable query while it is difficult to extract a model out of a satisfiable one.

In contrast, CLP(FD) systems use constraint propagation and are more focused on data rather than predicates. Indeed, they show the opposite behavior: a satisfiable query always returns a model. At the same time it is difficult to extract proof of unsatisfiability.

3 Technique

In the following sections we describe how we extended CLP(FD) to enable handling of infinite domains and quantifiers. In Section 3.1, we will explain how we track enumeration of CLP(FD) variables. Afterwards, we introduce a way to randomize the enumeration of large intervals in Section 3.2. For simplicity, we will discuss our techniques on a small interpreter supporting only integer variables with arbitrary and possibly infinite domains together with arithmetic expression on them, negation and nested existential and universal quantification.

3.1 Detection and Categorization of Enumeration

It is common for constraint satisfaction problems, that domain propagation alone is not enough to infer values for participating variables. This might be due to an underspecified problem or limitations in constraint solvers, e.g., global constraints that are too expensive to check. Usually, constraint solvers rely on enumeration of possible values if all other methods fail. However, there are some key difficulties:

1. Enumerating all values is only possible for reasonably sized domains. Even for finite but large domains, exhaustive enumeration can be impossible due to computational limitations. Obviously the same holds for infinite domains.
2. Depending on the scope of negations, quantifiers and other (nested) constructs finding a solution for a variable might imply both satisfiability and unsatisfiability of a (sub-)constraint.
3. Hence, if a solution is found it is not immediately clear if enumeration can be stopped.

We intend to overcome these limitations by tracking the scope of enumerations, i.e., tracking in which contexts enumeration occur. We distinguish the following types of enumerations, based on the effect on the overall solver result:

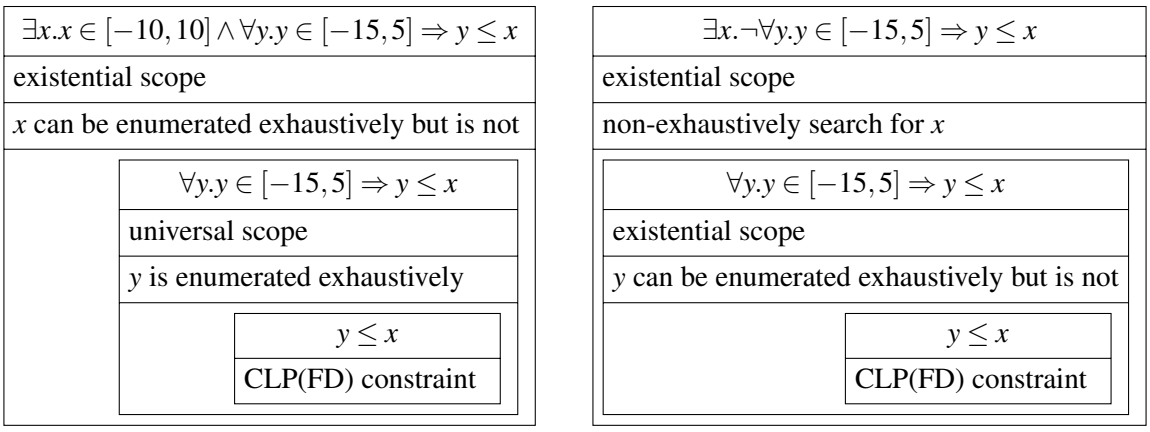


Figure 1: Nested Enumeration Scopes

- Enumeration does not occur. The result is not influenced, e.g., when no valuation is found the formula is unsatisfiable.
- Enumeration is exhaustive. In this case, all possible values for a variable were considered. If no valuation is found, the formula in question is unsatisfiable.
- Enumeration occurs and is not exhaustive. In this case, we cannot directly infer if the formula is satisfiable or not and have to examine the context (aka scope) in which the enumeration occurred.

Figure 1 shows the nesting of enumeration scopes for two simple predicates. The outer variable x is quantified existentially in both cases. In the first one, we can enumerate all possible values exhaustively, while in the second case only non-exhaustive search is possible. However, we only need to find one solution anyway, partial enumeration is not a problem. The inner variable y can be enumerated exhaustively. In the first example, we have to do so in order to validate the universal quantification. In the second example, exhaustive enumeration is possible but not necessary.

For further examples, take a look at the following constraints:

- `x*x = 10000` is true, a solution (`x = -100`) can be computed without resorting to enumeration.
- Using backtracking, we can find all solutions. Again, no enumeration is needed to compute `{x|x*x=10000} = {-100,100}`.
- In contrast, enumeration is necessary to find a solution to `x>10000 & x mod 1234 = 1` as propagation of `mod` does not render the domain of `x` finite. However, the solution found is sound, enumeration did not influence the result.
- As a consequence, we cannot compute all solutions. Our approach is unable to solve `{x|x>10000 & x mod 1234 = 1}`.
- For certain unsatisfiable constraints, such as `x*x = 10001`, CLP(FD) detects unsatisfiability by domain propagation. No enumeration occurs, the predicate is guaranteed to be false.
- In contrast, no solution is found for `x>10000 & x mod 1234 = 1 & x*x = 10*x`. Common propagation rules such as the ones used in SWI Prolog's CLP(FD) [29, 2] are too weak to conclude $x = 0 \vee x = 10$ from $x * x = 10 * x$ as long as there is no upper bound attached to x. In consequence, we cannot detect unsatisfiability as we have to (partially) enumerate the infinite domain of `x`. Since no solution has been found, enumeration cannot be ignored; Indeed, the predicate might still be true.

```
solve_constraint(Constraint, TopLevelVars) :-
    retractall(enum_warning),
    solve(Constraint, ExistsWF, AllWF),
    ground_vars(TopLevelVars, ExistsWF, AllWF).

ground_vars(TopLevelVars, ExistsWF, AllWF) :-
    maplist(enumerate_exists_aux, TopLevelVars),
    ground_wfs(ExistsWF, AllWF).

solve(A & B,EWF,AWF) :- solve(A,EWF,AWF), solve(B,EWF,AWF).
solve(A or B,EWF,AWF) :- solve(A,EWF,AWF) ; solve(B,EWF,AWF).
...
solve(not(A),EWF,AWF) :- solve_not(A,EWF,AWF).
solve(V in D,_,_) :- V in D.
solve(A = B,_,_) :-
    compute_exprs(A,B,AE,BE),
    AE #= BE.
...
solve(forall(X,LHS => RHS),_EWF,AWF) :-
    when(ground(AWF), enumerate_forall(X,LHS,RHS)).
solve(exists(X,RHS),EWF,_AWF) :-
    when(ground(EWF), enumerate_exists(X,RHS)).

solve_not(A & B,EWF,AWF) :-
    solve_not(A,EWF,AWF) ; solve_not(B,EWF,AWF).
solve_not(A or B,EWF,AWF) :-
    solve_not(A,EWF,AWF), solve_not(B,EWF,AWF).
...
```

Listing 1: Core of interpreter

3.1.1 Setting up Constraints

Constraints are set up using two Prolog predicates, `solve` for positive and `solve_not` for negative constraints. CLP(FD) constraints are immediately set up. Part of the interpreter is shown in Listing 1, the complete source code and usage instructions can be obtained from `https://github.com/wysiib/infinite_domain_solver`.

To control enumeration, we pass around two "wait flags". The first is used to trigger setup of constraints and enumeration of variables that are existentially quantified, i.e., they are introduced by $\exists$ or $\neg\forall$. The other does the same for universal quantification.

Listing 2 shows the implementation in our simple solver. Once a wait flag is grounded, the code in Listings 3 and 4 is called: We set up the inner constraints of the quantifier using fresh wait flags, that will later be grounded as well. This accounts for a hierarchy of scopes, where each may need to find a single solution or inspect all possible solutions for a variable. Again, see Figure 1 for an example.

```
solve(forall(X,LHS => RHS),_EWF,AWF) :-
    when(ground(AWF),enumerate_forall(X,LHS,RHS)).
solve(exists(X,RHS),EWF,_AWF) :-
    when(ground(EWF),enumerate_exists(X,RHS)).
```

Listing 2: Setup of Quantifiers

```
enumerate_exists(Var,RHS) :-
    % setup inner constraints
    solve(RHS,NewEWF,NewAWF), !,
    ground_wfs(NewEWF,NewAWF),
    enumerate_exists_aux(Var).
enumerate_exists_aux(Var) :-
    fd_size(Var,sup), !,
    % non-exhaustively enumerate infinite domain
    % need to find just one element!
    assert(enum_warning),
    fd_inf(Var,Min), fd_sup(Var,Max),
    enumerate_infinite(Var,0,Min,Max).
enumerate_exists_aux(Var) :-
    indomain(Var).
```

Listing 3: Enumerate Existentially Quantified Variable

3.1.2 Enumeration

Enumeration occurs in two phases. First, we ground the wait flag triggering enumeration of existentially quantified variables. This leads to the coroutines set up in Listing 2 being resumed. The key difference between enumerating an existentially quantified variable (Listing 3) and a universally quantified one (Listing 4) lies within the solver's reaction to infinite domains. Existentially quantified ones are enumerated as shown in Listing 3:

1. The inner constraint of the quantifier is set up as usual,
2. Inner variables are enumerated,
3. We begin enumerating the quantified variable:
    - If the domain is infinite, we store that enumeration cannot be exhaustive. Afterwards, we start enumerating.
    - Otherwise, we use regular CLP(FD) labeling.

In the second step, we cannot enumerate all possible values of the variable in question. We thus have to decide on a value selection strategy. Our simple example interpreter enumerates as follows: Starting from zero we alternate between the positive and negative value of an increasing counter, skipping values not in the corresponding domains. As soon as both upper and lower bounds are passed, the domain has been enumerated exhaustively.

```
enumerate_forall(Var,LHS,RHS) :-
    LHS = (_ in Min .. Max),
    % setup of inner constraints: there is a choicepoint here
    % to allow for different solutions to inner variables
    solve(LHS & RHS,NewEWF,NewAWF), !,
    ground_wfs(NewEWF,NewAWF),
    enumerate_forall_aux(Min,Max,Var).
% need to exhaustively enumerate infinite domain -> exception
enumerate_forall_aux(_,sup,_) :- throw(enum_infinite).
enumerate_forall_aux(inf,_,_) :- throw(enum_infinite).
% domain is finite, try all elements
enumerate_forall_aux(Current,Max,Var) :-
    Current =< Max, !,
    try_forall_value(Current,Var), % does not bind variable
    Current2 is Current + 1,
    enumerate_forall_aux(Current2,Max,Var).
enumerate_forall_aux(_,_,_).
```

Listing 4: Enumerate Universally Quantified Variable

This simple enumeration pattern is not sophisticated enough for complicated constraints. To improve, one could rely on techniques like the level diagonalization suggested in [7].

In ProB, we combine the simple enumerator with other (non-exclusive) strategies like case splits. Another alternative is to use random enumeration as we will show in Section 3.2. After all existentially quantified variables have been enumerated, we ground the wait flag that triggers universal quantifiers. We enumerate as outlined in Listing 4:

1. The inner constraint of the quantifier is set up as usual,
2. We begin enumerating the variable:
   - If the domain is infinite, we throw an enumeration exception. We would need to fully enumerate an infinite domain in order to solve the constraint. This futile attempt is dropped. Keep in mind, that the occurrence of infinite domains is usually influenced by the order of variables. By enumerating variables with finite domains first, we might shrink other domains.
   - Otherwise, we try all values in its domain to check the universal quantification.

With this extension, our CLP(FD)-based solver is able to handle both existential and universal quantification. In several cases, for instance in $y = 2 \wedge \forall x.(x \in [0,10] \Rightarrow x > y)$ the solver can recognize exhaustiveness of labelings. As a result, satisfiability and unsatisfiability can be deduced and reported to the user. In case of infinite or large domains the solver can tell if a result is still valid, despite the fact that a domain has not been enumerated completely. To increase coverage of large domains in the face of timeouts, the following section will introduce random enumeration.

3.2 Randomized Enumeration of Large Intervals

Another limitation of most CLP(FD) systems lies in how the next value of a variable is selected upon labeling. Within SICStus, the user can select between the following strategies [5] together with the

Data: List a
Result: Random permutation of a
for $i \in [0, length(a) - 1]$ **do**
  chose j randomly such that $0 \leq j \leq i$
  if $j \neq i$ **then**
    $perm[i] := perm[j]$
  end
  $perm[j] := a[i]$
end
return $perm$

Algorithm 1: Fisher-Yates / Knuth shuffle

options *up*, to use ascending order and *down*, to use descending order:

- *step*, i.e., a binary choice between $X = B$ and $X \neq B$, where B is the lower or upper bound of X.
- *enum*, i.e., multiple choice for X corresponding to the values in its domain.
- *bisect*, i.e., binary choice between $X \leq M$ and $X > M$, where $M = \lfloor \frac{\min(X)+\max(X)}{2} \rfloor$.
- *median*, i.e., binary choice between $X = M$ and $X \neq M$, where M is the median of the domain.
- *middle*, i.e., binary choice between $X = M$ and $X \neq M$, where $M = \lfloor \frac{\min(X)+\max(X)}{2} \rfloor$.

Summarizing, CLP(FD) variables can be set to domain values in various ways using domain splitting or simple enumeration. One property is common to all strategies. The domains are traversed deterministically. In case the domains are small enough, i.e., they can be enumerated exhaustively, there is no need for a different strategy. However, for large domains this might not be sufficient. Instead of traversing the domain linearly, it could be beneficial to use a random permutation:

- For large domains, values of different sizes will be tried out before a timeout occurs.
- It is less likely to get stuck in some part of the search space where there is no solution. If we fear search is stuck we could restart as described in [14]. This is common in SAT and SMT solvers.
- For applications like test case generation it is desirable to compute test inputs that fulfill some coverage criterion, e.g., that certain intervals or sets of parameters or values have been used. With linear enumeration and backtracking, generated test cases might only differ in the variable set last.

Note that we want to compute a random permutation of the domain. To avoid duplicates we do not want to randomly draw elements from the domain. Furthermore, we have to keep track of the exhaustiveness of our enumeration in order to detect unsatisfiability.

A classic algorithm to compute random permutations for given intervals is the Fisher-Yates shuffle [11] or Knuth shuffle [18]. Its pseudo code can be found in Algorithm 1. The algorithm has a weakness: the list to be shuffled has to be in memory completely. This is not feasible for intervals too large to be stored. We hence need an algorithm allowing us to compute a random permutation on the fly.

One such algorithm can be constructed using cryptographic techniques as outlined in [25]. The key idea is to construct an encryption function encrypting the elements to be permuted onto themselves. In order to allow for later decryption, an encryption function has to be unambiguously, i.e., given a fixed key there has to be a one-to-one mapping between plaintext and ciphertext. This will ensure, that we do in fact compute a permutation, i.e., we do not add or remove elements. Before we can present our implementation, we introduce a few definitions regarding ciphers. The definitions are following [26].

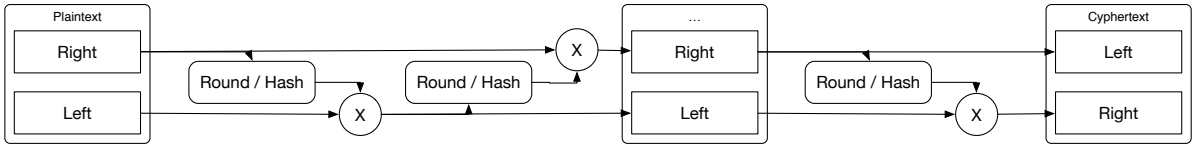


Figure 2: Feistel Network

Definition 3.1 (Block Cipher)**.** *An n-bit* block cipher *is a function* $E : [0,1]^n \times K \to [0,1]^n$*, such that for each key* $K \in \mathscr{K}$*,* $E(P,K)$ *is an invertible mapping from* $[0,1]^n$ *to* $[0,1]^n$*, written* $E_K(P)$*.* $E_K(P)$ *is called the* encryption function *for K. The inverse mapping is the* decryption function*, denoted* $D_K(C)$*.* $C = E_K(P)$ *denotes that ciphertext C results from encrypting plaintext P under K.*

Definition 3.2 (Random Cipher)**.** *A (true)* random cipher *is an n-bit block cipher implementing all* $2n!$ *bijections on* $2n$ *elements. Each of the* $2n!$ *keys specifies one such permutation. Obviously, the key space for a true random cipher is too large to be used in practice.*

Definition 3.3 (Iterated Block Cipher)**.** *An* iterated block cipher *is a block cipher involving the sequential repetition of an internal function called a round function. Parameters include the number of rounds r, the block bit size n, and the bit size k of the input key K from which r subkeys* K_i*, one for each round, are derived. For invertibility, for each* K_i *the round function is a bijection on the round input.*

Definition 3.4 (Feistel Cipher)**.** *A* Feistel cipher *is an iterated block cipher mapping a* $2t$*-bit plaintext* (L_0, R_0)*, for t-bit blocks* L_0 *and* R_0*, to a ciphertext* (R_r, L_r)*, through an r-round process where* $r \geq 1$*. For* $1 \leq i \leq r$*, round i maps* (L_{i-1}, R_{i-1}) *to* (L_i, R_i) *as follows:*

$$L_i = R_{i-1}, R_i = L_{i-1} \oplus f(R_{i-1}, K_1).$$

For each round, the subkey K_i *is derived from the given cipher key K.* f *is the round function mentioned in Definition 3.3. Often other (product) ciphers are used as* f*. However, F does not need to be invertible for the Feistel cipher to be. Figure 2 shows how a Feistel cipher operates on the input blocks.*

Now, we can follow one of the approaches suggested by [4] to construct a cipher that does not operate on $[0,1]^n$ but rather on $[0,k]$. The authors describe the construction of a generalized Feistel cipher:

1. Choose $a, b \in \mathbb{N}$ with $ab \leq k$ to decompose m taken from $[0,k]$ into $L = m \mod a$ and $R = \lfloor \frac{m}{a} \rfloor$.
2. Use (L,R) as inputs to a Feistel cipher as defined in Definition 3.4.
3. Perform a given number of iterations using random round functions whose ranges contain $[0,k]$.

Observe that Definition 3.4 assumes that L and R have the same length. Hence, the overall bit-length has to be divisible by 2[2]. With the construction above, we can thus only create ciphers for certain intervals, i.e., we can construct a cipher $[0,15] \to [0,15]$ but not $[0,5] \to [0,5]$. However, the cipher $[0,15] \to [0,15]$ can be used to permute $[0,5]$ by iterating to the next index if a number ≤ 6 is drawn. Intervals that do not start with 0 are shifted. The shift is later re-added to the drawn number.

The implementation of random permutations is outlined in Algorithm 2 and Algorithm 3. We can use a simplified version of the construction in [4] because we do not rely on strong cryptographic properties. On the interval $[1,6]$ it proceeds as follows:

1. Compute the interval length $l = 6 - 1 + 1$.

[2]For technical reasons the actual implementation uses a bit-length divisible by 4.

Data: Interval I to draw from
Result: Bitmasks BL to extract L, BR to extract R, number of bits n
$length = \max(I) - \min(I) + 1$
$n = \ln(length)$
if *n is odd* **then**
    $n = n + 1$
end
$BR = 2^{\frac{n}{2}-1}$
$BL = 2^n - 1 - BR$

Algorithm 2: Random Permutation: Setup

Data: Interval I, current index c, max index $maxIdx$, left/right mask BL, BR, number of bits n
Result: Next random element rnd and next index to draw $next$
do
    $left = c \;\&\; BL >> \lfloor \frac{n}{2} \rfloor$
    $left = c \;\&\; BR$
    $left, right = feistel\_rounds(left, right)$
    $rnd = (left << \lfloor \frac{n}{2} \rfloor) | right$
    $c = c + 1$
while $rnd > \max(I) - \min(I) \wedge c < \max$
if $c > maxIdx$ **then**
    return failure, permutation enumerated exhaustively
else
    return rnd, $next = c + 1$
end

Algorithm 3: Random Permutation: Next Element

2. Find the number of bits need to store l. In this case $l = 6$ means we need at least 3 bits.

3. Round up to the next even number to allow symmetric split into L and R. We can split an interval of 4 bits. Hence, we need to take into account all inputs from $[0, 15]$.

4. Compute the bit masks $BL = 1100$ and $BR = 0011$ used to extract the first 2 and the last 2 bits.

5. Set the current index to 0 and select a random key to choose a permutation.

6. To draw the next element of the random permutation:

   (a) Use a Feistel cipher to encrypt the current index using the key.

   (b) Increment the index.

   (c) Repeat if the cipher value is larger than 5, else return $(5 + 1)$ to stay within range.

For the rounding function one can use a hash function such as Prolog's term_hash. We use a given random seed as the encryption key. All subkeys are identical to the main key.

4 High-Level Reasoning using CHR

So far, ProB was mostly used for animation, model checking and data validation [21]. Hence, it used to be tailored towards finding solutions to satisfiable formulas, which is where constraint programming has its strengths. Therefore, choosing CLP(FD) as a basis for ProB has been a reasonable decision:

- It can deal with large and only partially known data.
- Even though B is a higher-order language including sets, relations and functions, everything could be expressed by or in conjunction with CLP(FD) variables and constraints.
- Support of reification made it considerably easier to propagate information between the different solvers, i.e., from integer constraints to set constraints and vice versa.
- Although satisfiability of predicates written in B is in general undecidable, ProB is able to solve a useful class of subproblems.

New applications like symbolic model checking however made shortcomings of CLP(FD) obvious: Even for simple constraints like $x < y \wedge y < x$ contradictions often can only be detected if variables have finite domains. Again, this is due to propagation relying on finite upper and lower bounds [29, 2].

To improve, we implemented a set of rules working on top of the CLP(FD) variables. Our initial idea was to perform some kind of high-level propagation, where new CLP(FD) constraints are discovered from the existing constraints. That is, we would implement propagation rules like the transitivity of $<$ stating that $x < y \wedge y < z \Rightarrow x < z$. These rules are implemented in CHR [12, 13], a committed choice language that can be embedded in Prolog. CHR supports three different kinds of rules to modify a *constraint store* holding the current state of constraints:

- Simplification rules of the form $h_1, \ldots, h_n \,|\, g_1, \ldots, g_m \iff b_1, \ldots, b_o$. Here, $h_1, \ldots, h_n$ are the so called "head", i.e., constraints that have to be found in the constraint store for the rule to act upon. $g_1, \ldots, g_m$ are called "guards". These are predicates that have to hold for the rule to be allowed to fire. If the rule can be executed, the heads are rewritten into the "bodies" $b_1, \ldots, b_o$, i.e., $h_1, \ldots, h_n$ are removed from the constraint store while $b_1, \ldots, b_o$ are added.
- Propagation rules of the form $h_1, \ldots, h_n \,|\, g_1, \ldots, g_m \implies b_1, \ldots, b_o$. Here, the bodies are added to the constraint store without removing the heads.
- Simpagation rules such as $h_1, \ldots, h_l \setminus h_{l+1}, \ldots, h_n \,|\, g_1, \ldots, g_m \iff b_1, \ldots, b_o$ combine the former. The constraints $h_1, \ldots, h_l$ are kept in the constraint store while $h_{l+1}, \ldots, h_n$ are removed.

We augmented the constraint solver with CHR rules handling integer arithmetic, focusing on detection of contradictions involving linear inequalities. The rules are comparable to those introduced in the finite domain solver of [13, Ch. 8]. However, we do not handle domains inside CHR, but rather integrate with CLP(FD). Regarding the implementation of infinite domain solvers in CHR see [13, Ch. 9].

Listing 5 includes an extract of the CHR rules encoding properties of $<$ and $\leq$. As can be seen, we introduced rules for (anti-)reflexivity, antisymmetry, idempotence and transitivity. For instance, transitivity of $\leq$ is given by the CHR rule `leq(X,Y), leq(Y,Z) ==> leq(X,Z)`, stating that from $X \leq Y \wedge Y \leq Z$ we can infer $X \leq Z$. Newly inferred constraints are submitted to the underlying CLP(FD) system.

In addition to inferring new constraints, CHR rules can be used to infer unsatisfiability. As an example, look at the rule encoding the antireflexivity of $<$. It states that if we have $X < X$, `fail` has to be executed. Due to CHR being a committed-choice language, the whole CHR run is discarded. The Prolog rule that added the offending constraint by calling `leq(X,X)` fails. In consequence, further propagation is stopped and the failure has to handled by the solver.

```
reflexivity  @ leq(X,X) <=> true.
antisymmetry @ leq(X,Y), leq(Y,X) <=> X = Y.
idempotence  @ leq(X,Y) \ leq(X,Y) <=> true.
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).

antireflexivity @ lt(X,X) <=> fail.
idempotence     @ lt(X,Y) \ lt(X,Y) <=> true.
transitivity    @ lt(X,Y), leq(Y,Z) ==> lt(X,Z).
transitivity    @ leq(X,Y), lt(Y,Z) ==> lt(X,Z).
transitivity    @ lt(X,Y), lt(Y,Z)  ==> lt(X,Z).
```

Listing 5: CHR rules for integer (in-)equalities

predicate	with CHR	without CHR
$x > 3$	1.2	1.08
$x > y \wedge y > x$	0.92	timeout
$x = 3 \wedge x > y \wedge y = 4$	0.98	1.07
$x = 3 \wedge x > y$	0.86	1.0
$x = 3 \wedge x < y$	1.1	1.11
$w > x \wedge x > y \wedge y > z \wedge w = 1 \wedge z = 1$	0.98	0.95
$w > x \wedge x > y \wedge y > z \wedge z > w$	0.88	timeout
$x+2 > y+1 \wedge y > x$	timeout	timeout
$x > y \wedge y > x+1$	0.9	timeout

Table 1: Small Benchmarks with / without CHR (in s)

Of course a fully fledged solver needs to include several other CHR rules dealing with common cases of integer constraints. As can be seen in the examples in Table 1, the example set of CHR rules is far from being complete: It is able to handle simple cases like $x > y \wedge y > x$ and transitive cases like $w > x \wedge x > y \wedge y > z \wedge z > w$. However, it is so far unable to do simple arithmetic as in $x+2 > y+1 \wedge y > x$. In ProB we have added both arithmetic as well as further high-level rules.

5 Applications

Within our model checker and animator ProB we have several applications for an infinite domain constraint solver. We already mentioned animation, model checking and constraint-based validation in the introduction. In this section, we will look at other applications.

We start with an overview of ProB's kernel in Section 5.1. In Section 5.2 we explain how our constraint solving technique can be used for proof and disproof of B proof obligations. Furthermore, we used it to solve SMT problems. As a second application, we present our effort to use ProB for data validation tasks. Section 5.3 will describe the main challenges.

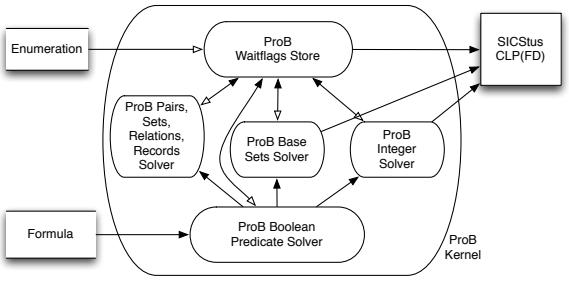


Figure 3: A view of the ProB kernel

5.1 The ProB Constraint Solving Kernel

The ProB kernel can be viewed as a constraint-solver for the basic datatypes of B and the various operators on it. It heavily relies on the SICStus Prolog CLP(FD) system [6] which follows the general implementation scheme of [17]. It supports booleans, integers, user-defined base types, pairs, records and inductively: sets, relations, functions and sequences. These datatypes and operations are embedded inside B predicates, which can make use of the usual logical connectives ($\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg$) and typed universal ($\forall x.P \Rightarrow Q$) and existential ($\exists x.P \wedge Q$) quantification. An overview of the various solvers residing within the ProB kernel can be seen in Figure 3.

Different solvers are linked via reification variables. Enumeration is controlled by setting up coroutines at various choice points. For more fine-grained enumeration, coroutines might be set up together with a wait flag that allows the solver to trigger delayed constraints. The techniques presented in this paper are fully integrated with the solving kernel and its features.

5.2 Prove and Disprove

As mentioned above, one of our key motivations is to move ProB from being guaranteed sound for finite domains to infinite domains. This is particularly important if ProB is to be used as a prover.

In [24, 19], we embedded ProB into Rodin [1], an IDE for Event-B, in order to generate counter-examples for proof obligations. Given a sequent with goal $G(x_1,\ldots,x_k)$ and hypotheses $H_i(x_1,\ldots,x_k)$ we build the predicate

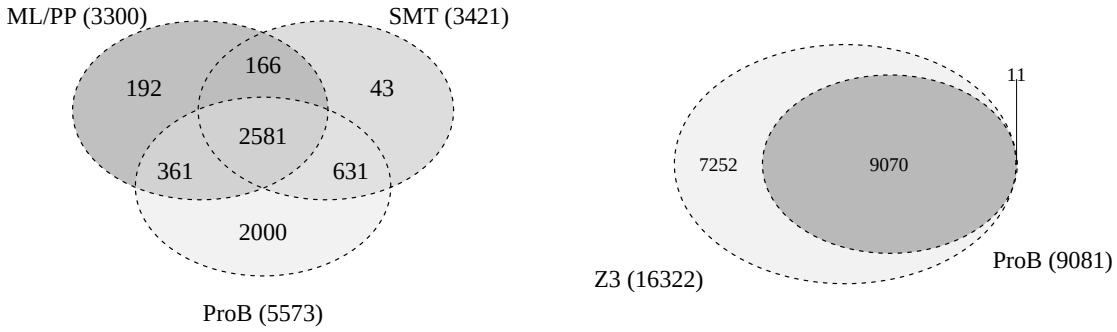
$$\exists x_1,\ldots,x_k : (H_1(x_1,\ldots,x_k) \wedge \ldots \wedge H_n(x_1,\ldots,x_k)) \Rightarrow \neg G(x_1,\ldots,x_k)$$

and feed it to our constraint solver. If the predicate does hold, ProB returns a valuation for $x_1,\ldots,x_k$, representing a counter-example to the sequent.

In general, checking the satisfiability of propositional formulas is NP complete. Beyond that, typical Event-B proof obligations consist of first order logic formulas, for which the problem becomes undecidable. Previously [22, 23, 24], we overcame this limitation by limiting domains to be finite. However, this prevents drawing any conclusions from the absence of a counter-example.

The techniques presented in this work have been used to extend the disprover to a fully fledged prover. By observing the state of enumerations as explained in Section 3.1, ProB is able to tell if the search for a counter-example was exhaustive. If this is the case, we can report a proof to the user.

We performed several benchmarks comparing our prover to ML and PP [8], two specialized provers for B and Event-B. Additionally, we compared the performance of SMT solvers on the proof obligations.



(a) (Dis-)Proof (b) SMT solving

Figure 4: Benchmarks

An overview is given in Figure 4a, where we benchmarked on a diverse selection of different B and Event-B machines. As you can see, ProB outperforms the other proves. Details can be found in [19].

5.2.1 SMT Solving

The techniques presented can also be used to solve SMT problems, such as the ones collected by the SMT-LIB project. In particular, we used all benchmarks that involve (non-)linear integer arithmetic and quantification but no other constructs like arrays or bit vectors. The results can be seen in Figure 4b. ProB augmented with enumeration tracking cannot compete with Z3 [9]. Yet, a considerable number of both satisfiable and unsatisfiable benchmarks can be solved using our technique.

5.3 Data Validation

Model checking and constraint-based validation aside, ProB and B are often used for data validation tasks [15]. The challenge here is to efficiently handle large relations and sets, e.g., representing track topologies, and at the same time effectively solving constraints and dealing with certain infinite functions which are used to manipulate data. Notable applications come from the railway industry [10] or university timetabling [28].

6 Related Work

SMT solvers [3] such as Z3 [9] are able to handle infinite domains and quantifiers. As outlined in the introduction, a major difference lies within the handling of data. While SMT solvers are more focused on predicates, CLP(FD) systems are more oriented towards data. As a result, SMT solvers can handle infinite domains more efficiently and detect unsatisfiability in more cases. However, model generation (in particular in the presence of quantifiers) is often easier using CLP(FD).

In [20] we investigated a different approach to add high-level reasoning to a CLP(FD)-based solver. Instead of implementing rules in CHR we connected the SMT solver Z3 to SICStus Prolog. We transferred each predicate asserted in CLP(FD) to Z3 and queried both solvers for a solution. Furthermore, intermediate assignments and domains could be communicated back and forth.

While the approach was more powerful when it comes to reasoning, using CHR rules has the advantage of an immediate integration into Prolog. Hence, there is no communication overhead and no translation between different representations is needed.

7 Conclusion and Future Work

Summarizing, we have presented an approach to lift a CLP(FD)-based solver to infinite domains. Our approach tracks enumerations occurring during search and interprets how they affect the overall result. Two extensions, high-level reasoning and random enumeration were used to increase applicability.

Techniques have been added to the animator and model checker ProB. Here, they allowed us to use ProB as a prover and SMT solver. Further applications in data validation show that CLP(FD) together with our extensions makes for a useful general purpose solver.

While we were pleasantly surprised by the overall performance, the extended CLP(FD) solver is still too weak to cope with the requirements a tool like ProB has. Yet, it plays its part in combined solvers such as the one we outlined in [20] or in solver portfolios.

References

[1] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta & Laurent Voisin (2010): *Rodin: an open toolset for modelling and reasoning in Event-B*. International Journal on Software Tools for Technology Transfer 12(6), pp. 447–466, doi:10.1007/s10009-010-0145-y.

[2] Krzysztof R. Apt & Peter Zoeteweij (2007): *An Analysis of Arithmetic Constraints on Integer Intervals*. Constraints 12(4), pp. 429–468, doi:10.1007/s10601-007-9017-9.

[3] Clark Barrett, Roberto Sebastiani, Sanjit Seshia & Cesare Tinelli (2009): *Satisfiability Modulo Theories*. In: Handbook of Satisfiability, chapter 26, IOS Press, pp. 825–885, doi:10.3233/978-1-58603-929-5-825.

[4] John Black & Phillip Rogaway (2002): *Ciphers with Arbitrary Finite Domains*. In: Topics in Cryptology — CT-RSA 2002, LNCS 2271, Springer, pp. 114–130, doi:10.1007/3-540-45760-7_9.

[5] Mats Carlsson & Thom Fruehwirth (2014): *Sicstus PROLOG User's Manual 4.3*. Books On Demand - Proquest.

[6] Mats Carlsson, Greger Ottosson & Björn Carlson (1997): *An open-ended finite domain constraint solver*. In: Programming Languages: Implementations, Logics, and Programs, LNCS 1292, Springer, pp. 191–206, doi:10.1007/BFb0033845.

[7] Jan Christiansen & Sebastian Fischer (2008): *EasyCheck — Test Data for Free*. In Jacques Garrigue & Manuel V. Hermenegildo, editors: Functional and Logic Programming: 9th International Symposium, FLOPS 2008, Ise, Japan, April 14-16, 2008. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 322–336, doi:10.1007/978-3-540-78969-7_23. Available at `http://dx.doi.org/10.1007/978-3-540-78969-7_23`.

[8] ClearSy (2016): *Atelier B, User and Reference Manuals*. Aix-en-Provence, France. Available at `http://www.atelierb.eu/`.

[9] Leonardo De Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: Proceedings TACAS, LNCS 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[10] Jerome Falampin, Hung Le-Dang, Michael Leuschel, Mikael Mokrani & Daniel Plagge (2013): *Improving Railway Data Validation with ProB*. In: Industrial Deployment of System Engineering Methods, Springer, pp. 27–44, doi:10.1007/978-3-642-33170-1_4.

[11] Ronald Aylmer Fisher & Frank Yates (1953): *Statistical tables for biological, agricultural and medical research*, 3 edition. Oliver and Boyd.

[12] Thom Frühwirth (1998): *Theory and practice of Constraint Handling Rules*. The Journal of Logic Programming 37(1–3), pp. 95 – 138, doi:10.1016/S0743-1066(98)10005-5.

[13] Thom Frühwirth (2009): *Constraint Handling Rules*. Cambridge University Press, doi:10.1017/CBO9780511609886.

[14] Carla P. Gomes, Bart Selman & Henry Kautz (1998): *Boosting Combinatorial Search Through Randomization*. In: Proceedings AAAI, AAAI/IAAI, American Association for Artificial Intelligence, pp. 431–437.

[15] Dominik Hansen, David Schneider & Michael Leuschel (2016): *Using B and ProB for Data Validation Projects*. In: Proceedings ABZ, LNCS 9675, Springer, doi:10.1007/978-3-319-33600-8_10.

[16] Daniel Jackson (2006): *Software Abstractions: Logic, Language and Analysis*. MIT Press.

[17] Joxan Jaffar & Spiro Michaylov (1987): *Methodology and Implementation of a CLP System*. In: Proceedings ICLP, pp. 196–218.

[18] Donald E. Knuth (1997): *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3 edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, doi:10.1137/1012065.

[19] Sebastian Krings, Jens Bendisposto & Michael Leuschel (2015): *From Failure to Proof: The ProB Disprover for B and Event-B*. In: Proceedings SEFM, LNCS 9276, Springer, doi:10.1007/978-3-319-22969-0_15.

[20] Sebastian Krings & Michael Leuschel (2016): *SMT Solvers for Validation of B and Event-B models*. In: Proceedings iFM, LNCS 9681, Springer, doi:10.1007/978-3-319-33693-0_23.

[21] Michael Leuschel, Jens Bendisposto, Ivaylo Dobrikov, Sebastian Krings & Daniel Plagge (2014): *From Animation to Data Validation: The ProB Constraint Solver 10 Years On*. In Jean-Louis Boulanger, editor: Formal Methods Applied to Complex Systems: Implementation of the B Method, chapter 14, Wiley ISTE, Hoboken, NJ, pp. 427–446, doi:10.1002/9781119002727.ch14.

[22] Michael Leuschel & Michael Butler (2003): *ProB: A Model Checker for B*. In: Proceedings FME, LNCS 2805, Springer, pp. 855–874, doi:10.1007/978-3-540-45236-2_46.

[23] Michael Leuschel & Michael Butler (2008): *ProB: An Automated Analysis Toolset for the B Method*. Software Tools for Technology Transfer (STTT) 10(2), pp. 185–203, doi:10.1007/s10009-007-0063-9.

[24] Olivier Ligot, Jens Bendisposto & Michael Leuschel (2007): *Debugging Event-B Models using the ProB Disprover Plug-in*. Proceedings AFADL.

[25] M. Luby & C. Rackoff (1988): *How to Construct Pseudorandom Permutations from Pseudorandom Functions*. SIAM Journal on Computing 17(2), pp. 373–386, doi:10.1137/0217022.

[26] Alfred J. Menezes, Scott A. Vanstone & Paul C. Van Oorschot (1996): *Handbook of Applied Cryptography*, 1st edition. CRC Press, Inc., Boca Raton, FL, USA, doi:10.1201/9781439821916.

[27] Daniel Plagge & Michael Leuschel (2012): *Validating B, Z and TLA+ using ProB and Kodkod*. In: Proceedings FM, LNCS 7436, Springer, pp. 372–386, doi:10.1007/978-3-642-32759-9_31.

[28] David Schneider, Michael Leuschel & Tobias Witt (2015): *Model-Based Problem Solving for University Timetable Validation and Improvement*. In: Proceedings FM, LNCS 9109, Springer, pp. 487–495, doi:10.1007/978-3-319-19249-9_30.

[29] Markus Triska (2014): *Correctness Considerations in CLP(FD) Systems*. Ph.D. thesis, Vienna University of Technology.