Mutual B refinements as a justification for constraints model reformulations

Jean-Louis Dufour
R&T Department, Safran Electronics & Defense, France

Abstract

Model reformulation is a mandatory activity to get the most out of the constraint solvers used. When it is a manual process, the problem of correctness arises, especially when the model is used in a critical system. The B-method, used for more than 30 years to prove critical software, offers a notion of refinement to prove the correctness of an implementation with respect to a specification. We propose that this notion of refinement be used to justify the correctness of a reformulation of a constraints model. The new notion of equivalence of models seems less restrictive than the existing notions, but still adequate in the context of constraint satisfaction. This not the case in the context of constrained optimization, and the notion will have to be refined.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Abstraction; Theory of computation → Program verification

Keywords and phrases constraints reformulation, formal refinement

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Acknowledgements I thank Jean-Marie Courteille for helpful suggestions on the presentation of the B method.

1 Context and motivation

When the eight queens problem (place them on a chessboard so that no two queens threaten each other) is modeled as a Constraint Satisfaction Problem (CSP), solutions may be represented in many ways (names come from [15]):

queen-based a set of eight positions (pairs of coordinates),

square-based an 8x8 boolean array,

column-based (resp. row-based) an array of eight row (resp. column) indexes,

...

Maybe the first representation will be the most readable in a high-level language like Minizinc [16] or Essence [6] , maybe the 2nd will be the most efficient in an SMT solver like Z3 [13], and maybe the 3rd will be the most efficient in a CP solver like CP-SAT [8].

A change of representation between two '*equivalent*' models is called a *reformulation*, and it is either manual of automated: when Minizinc generates a CP-SAT model, it reformulates because CP-SAT doesn't know the *set* concept. This automated reformulation is generally correct (but each new version of MiniZinc contains bug fixes), but the same cannot be said of a manual translation from Minizinc to Z3 or between CP-SAT and Z3.

Constraint solvers will soon be used in safety-critical functions, and at that point the problem of correctness will arise. In a future certification process, a Minizinc or Essence model will be considered as a specification, and one of the questions will be: is this CP-SAT or Z3 model *equivalent*, in a formal sense, to its specification?

This paper owes its existence to [14], which recognizes this problem, recognizes that it is not addressed today, explains why it is not obvious, and outlines a solution based on logic and model theory. Technically this approach is the most natural, and promising. But we would like to share another point of view, which is to say that instead of seeing constraints as a logic problem, we can see them as the *specification of a dedicated solver*; and in this case, software formal methods give a natural definition of what a correct reformulation is, and how to verify it.

Our contribution consists in proposing a non-standard use of the B-method [1, 19]: to justify the 'equivalence' of two constraint models, we *virtually* translate them into two B models, and we ask that they *refine* each other (in the software sense). On the other hand, we do not claim to have a practical method: even simple models, like the n-queens, give rise to difficult proofs, but we will outline a way forward in the conclusion.

The next section reviews two concepts related to model reformulation: the *reducibility* of [18] and the *channeling* of [4]. These two concepts are the technical inspiration for this paper, as they have almost identical counterparts in the field of software refinement. Then, we provide a crash course on the refinement concept [21] of the B-method, with a focus on the particular use we make of it. And finally, we present the translation of constraint models into B models and explain what the mutual refinement means in the context of constraint satisfaction. This leads us to identify a limitation of our approach in the constrained optimization context.

2 Reducibility and channeling

The connections between model reformulation and software refinement were suggested by two CP concepts: the *reducibility* of [18] and the *channeling* of [4].

2.1 Equivalence via Reducibility

Usually, two constraint models over the same variables are considered 'equivalent' when they have the same solutions. Logically speaking, a solution is a model (of the constraints; warning: the word 'model' is overloaded in this paragraph) and so we can also say that the underlying concept is the logical equivalence of the constraints in first-order logic. This is consistent with the notion of "redundant constraints", of which the literature is full, always understood as logical implication.

But when the variables differ, no standard mathematical or computer science notion comes close to it, and the problem is absolutely not obvious. In 1989, Rossi, Petrie and Dhar [18] make a fundamental contribution to fill this gap with the concept of '*reducibility*'. Given two CSPs P_1 and P_2, the definition of 'P_1 is reducible to P_2' is complex, but the idea is that every solution of P_1 must be obtained by a "syntactic transformation" of a solution of P_2. In this reducibility concept, two properties are worth noting, because our proposal does not satisfy them:

1. every solution of P_1 is obtained, none is lost;
2. the transformation is "syntactic", the goal is to avoid to simply solve P_1, with the solutions of P_2 playing not part.

2.2 Redundant modeling via channeling constraints

It has long been known that adding redundant constraints is worth trying to speed up constraint propagation. In 1996, Cheng, Lee and Wu [4] go further and propose the concepts of *redundant modeling* via *channeling constraints*:

"Intuitively speaking, our method amounts to implementing more than one model of a given problem and somehow connect the model implementations. ... In order for the different models to cooperate during constraint-solving, the network must be connected so that pruning can be propagated among the networks in a multi-directional manner. *One* *way to achieve a multi-directional connection is to use constraints to express the relationship among variables in different models.* We call such constraints, which are not part of the original problem specification and exist only as an artifact of network connection, *channeling constraints*."

In [3] (revised and extended version of [4]), the n-queens problem is used as a case study, with two representations: the row-based one ($x_i = j$ denotes that the queen of row i is in column j) and the column-based one ($y_i = j$ denotes that the queen of column i is in row j). With the channeling constraints $x_i = j \iff y_j = i$ they observe that the channeling is so strong that

" Under this formulation, constraints in model one (or two) can be regarded as redundant constraints of the channeling constraints plus constraints in model two (or one). Such a relationship does not necessarily hold true in general as programmers may choose to connect only certain subsets of variables or values. "

Perhaps inspired by this remark, [10] give a new use of channeling constraints: model *induction*. First, they give a formal definition of a representation (they use the term '*viewpoint*', first proposed by [7]). If the channeling constraints between two viewpoints V_1 and V_2 are strong enough to define a *total and injective* function from V_1 to V_2, constraints on V_1 can be transformed into constraints on V_2. They illustrate this with the transformation of the row-based model of the 4-queens into a square-based model (4x4 array of booleans) via the set of channeling constraints $x_i = j \iff z_{ij} = 1$.

High-level constraint languages like Essence and MiniZinc allow representations using abstract datatypes like sets. So these models must be refined in order to be solved by CP or SMT solvers. Usually, there are several ways to refine an abstract datatype, and [6] propose to automatically do *redundant modelling* from an Essence specification: the generation of a low-level model gives rise to 'representations annotations' which describe how the refinements have transformed the variables. When you generate redundant models, these annotations allow channeling constraints to be automatically generated. The exact algorithm is given in [12].

3 Refinement in the B-method

When a safety-critical function is based on software, this software must be compliant with safety requirements, which take the form of a natural language specification, from which test sets are manually derived. But in some cases (typically in the railway industry), compliance is not based on tests, but on proofs. To do this, the natural language specification is manually translated into a formal specification, and when a corresponding design is (usually manually) completed, *Proof Obligations (POs)* are automatically generated. We illustrate this with a toy example, and we take the opportunity to present just what is needed of the B syntax.

We will proceed in three steps: we begin with a basic version, which corresponds to what was achievable in the 70s, called 'Hoare logic' [20]. This basic version was unable to prove real software, and the necessary evolutions were finalized in the 80s. It turns out that two of these necessary evolutions are just what is needed in our constraint modelling context, they will be the subject of the two last subsections.

3.1 Hoare logic (with B syntax)

```
MACHINE toy
VARIABLES        xx
INVARIANT        xx : NATURAL1
```

```
INITIALISATION xx:(xx=1 or xx=2 or xx=3)
OPERATIONS
    xx_ <-- read =
        xx_ := xx
END
```

The listing above is named a *machine* (similar to a class in an Object Oriented language), but this name is misleading, because there is almost nothing executable about it: it is just a specification. It is made of

1. a list of *variables*, called the *state* of the machine; here it is just the single variable *xx* ('xx' because B requires that variable names have at least 2 characters);
2. an *invariant* property on these variables; the minimal property is the type of each variable, here we are a bit more demanding: `NATURAL1` is the subset of the `INTEGER`s greater than or equal to 1;
3. an *initialisation* which must establish the invariant; we comment on this just after;
4. a list of *operations* with read/write access to these variables, which must preserve the invariant; here we can only `read` the state *xx* into the output parameter *xx_*.

The interesting part is the initialisation (with an 's', because the development of the B method started in the Oxford Programming Research Group in the second half of the 80s):

```
xx :(xx=1 or xx=2 or xx=3)
```

It literally means:

write in xx a value such that the property 'xx=1 or xx=2 or xx=3' is true.

This substitution (the B word for 'instruction' or 'state update') is *non-deterministic*, which means that after 'execution', several states are possible. Be careful, it has nothing to do with randomness (if we look for an analogy, it is more like execution on a Non-Deterministic Turing Machine), it's just that at this level of specification, we don't want to choose, and we leave the choice to a future refinement. It is called the *such that* substitution, and more generally, the syntax

```
variable_list :( constraint_on_this_variable_list )
```

specifies that the variables in `variable_list` have to be substituted with values *such that* the Boolean formula `constraint_..._list` is satisfied.

But in a software context, the goal is to obtain software, and this machine is not software: it is too *abstract* and it needs to be *refined* with only deterministic substitutions. A correct and deterministic *refinement* is

```
REFINEMENT toy_i REFINES toy
VARIABLES         xx
INITIALISATION    xx := 1   /** deterministic substitution **/
/** operation 'read' is implicitely repeated as such;  **/
/** it was already deterministic                       **/
END
```

The interesting part is again the initialisation, which now is deterministic (only 1 possible execution):

```
xx := 1
```

asks that the new value of *xx* be 1. This refinement is correct because the corresponding PO (Proof Obligation) is true:

$$\boxed{\forall x \in \mathbb{Z}, x = 1 \Rightarrow x \in \{1, 2, 3\}}$$

This PO has been generated according to *Hoare logic*, the great ancestor of the B method. In our context, Hoare logic has two problematic limitations, but, fortunately, they have since been overcome:

1. the specification and the associated code talk about the same variable xx, it is not possible to change the representation of the objects,
2. the refinement is deterministic: it is not possible to have intermediate refinements that would retain some of the non-determinism.

3.2 Non-deterministic refinements

Sometimes, the algorithmic gap between a machine and a refinement is significant and leads to complex proofs. Then it is interesting to design an intermediate refinement. In our toy example, an intermediate refinement may be:

```
xx :(xx=1 or xx=2)
```

It is non-deterministic, but less than the former specification and more than the former code. It is written:

`xx :(xx=1 or xx=2 or xx=3)` $\sqsubseteq$ `xx :(xx=1 or xx=2)` $\sqsubseteq$ `xx := 1`

where $S \sqsubseteq T$ reads "the substitution S *is refined by* the substitution T".

Refinement is a partial order on substitutions, so we can immediately define *equivalence* of substitutions. It makes no sense in the software context, but it will be a key point in our context of constraint reformulation. For example, we have both

`xx :(xx=1 or xx=2 or xx=3)` $\sqsubseteq$ `xx :(1<=xx & xx<=3)`

and

`xx :(1<=xx & xx<=3)` $\sqsubseteq$ `xx :(xx=1 or xx=2 or xx=3)`,

which means that these two substitutions are *equivalent* (in the software sense): they *reformulate* each other (again, in the software sense).

A little more syntax: when the set of legal new values for a variable is explicitly known (i.e. *in extension*), instead of asking $x : (x = k_1 \text{ or } x = k_2 \text{ or } \ldots \text{ or } x = k_n)$ we can ask $x :: \{k_1, k_2, \ldots, k_n\}$. This is the '*belongs to*' substitution. To be complete (without adding to the confusion, I hope), I add that we must be careful not to confuse the *substitution* $x :: \{1, 2, 3\}$ (a modification of x) with the *predicate* $x : \{1, 2, 3\}$ (a property of x).

3.3 Data refinement

Often, expressing a high-level specification using low-level data structures makes that specification difficult to understand. Databases are typical examples: if we want to manage people's ages, at the highest level it is best to manipulate a partial function from names to ages, at a medium level dictionaries, at a lower level hash-tables and at the lowest level arrays.

Data refinement in B depends on the observability of this data (via the operations, because states are private), and one of the best examples is given in the B-Book [1] §11.2.5:

```
MACHINE Little_Example_1
VARIABLES      yy
INVARIANT      yy : FIN(NATURAL1)
INITIALISATION yy := {}
OPERATIONS
  enter(nn) =
    PRE nn : NATURAL1 THEN
      yy := yy \/ {nn}
    END
;
  mm <-- maximum =
    PRE yy /= {} THEN
      mm := max (yy)
    END
END
```

```
MACHINE Little_Example_2
VARIABLES      zz
INVARIANT      zz : NATURAL
INITIALISATION zz := 0
OPERATIONS
  enter(nn) =
    PRE nn : NATURAL1 THEN
      zz := max( {zz,nn} )
    END
;
  mm <-- maximum =
    PRE zz /= 0 THEN
      mm := zz
    END
END
```

These two machines are nano-databases, where you can `enter` positive numbers, but once they are entered you can no more access them individually: you can just ask for their `maximum`. Their states are very different: the left one uses a (FINite) set, whereas the right one uses a single integer. But surprisingly, from the point of view of an external user (remember: the state is private, so the user can only call `enter` and `maximum`), they are *EQUIVALENT*!

The difference in the amount of information between the two states is impressive, but we can do even better: replace the set of integers with a sequence of integers, so that we will memorize the entire filling history. And again they will be equivalent. In a software context, we will only be interested in a refinement of the left machine by the right one, because we must go from mathematical structures to memory structures. But in a constraint reformulation context, both directions will prove interesting, so here is the refinement of the right one by the left one:

```
REFINEMENT Little_Example_2_r REFINES Little_Example_2

EXTENDS Little_Example_1    /* basically, textual inclusion */
                            /* like a #include "..." in C   */
INVARIANT zz = max(yy\/{0}) /****  the LINKING invariant ****/

END
```

The interesting part is the `INVARIANT` clause, which contains the property $zz = max(yy \cup \{0\})$. This expresses the consistency link between the states of the two machines, hence its name: the *LINKING* invariant (also called the *gluing* invariant).

4 Reformulation via refinement

As already mentioned, we change our point of view on constraint modeling: instead of seeing constraints as a logic formula looking for a logic model (a solution), we see them as the *specification of a dedicated solver* looking for the code of this solver.

Our constraint model will take a very generic form, which begins with a single variable '`vv`' belonging to a certain set '`SET`'. This is by no means restrictive: for example, if `SET` is $\mathbb{N}^2$ (`NATURAL*NATURAL` in B) then `vv` will be a pair of natural numbers, and if `SET` is $\mathbb{N}^{\mathbb{Z}}$ (`INTEGER-->NATURAL` in B), then `vv` will be a function from integers to naturals. Our constraints are represented by a particular subset of `SET`, called '`SOL`' (the SOLutions of the problem). Continuing with our first example, SOL may be the 'Pythagorean doubles' (the pairs (x, y) such that $x^2 + y^2$ is a square), and in the second example, we could look for increasing functions. But we're not going to do that, and let SOL be as generic as possible, to get the most general refinement notion.

Given this context, the most important point of this paper is the simple observation that *the specification of the solver dedicated to SOL is the following substitution:*

```
IF   SOL /= {}                         /* IF SOL is not the empty set */
THEN sat := TRUE    || vv :: SOL       /* THEN a SOLution exists      */
ELSE sat := FALSE   || vv :: SET       /* ELSE no SOLution            */
END
```

The additional variable sat indicates whether vv is just a random element of SET (when SOL is empty) or if vv really is a SOLution. The || symbol means that the left substitution (on sat) and the right one (on vv) are to be performed 'simultaneously' (not relevant in our context).

This substitution must appear in a machine, and there are two possibilities: in the initialisation, or in an operation. In the initialisation, the variables sat and vv must be in the state: we call this the *stateful* style. In an operation, a state is no longer mandatory, because sat and vv can be the output variables: we call this the *stateless* style.

Having or not having a state is a fundamental difference, but there is another important difference: you cannot pass a parameter to an initialisation, while it is a natural thing for an operation. More often that not, constraint models are parameterized (typically the n of the n-queens, the graph in path-finding, ...) so if one day the proposal is implemented, the second style will be used. But we will present both styles, because they give the same formal definition of a reformulation: a sign of the robustness of our proposal.

4.1 A simplified case: problems with solutions

Sometimes we know that there is a solution (e.g. n-queens with $n \geq 4$): formally, SOL /= {}. We will begin with this special case, for which the specification of a solver reduces to vv :: SOL.

Let's begin with the stateful translation. It means that the solution is in the state, and the specification of this solution is in the invariant. The first CSP, CSP1 (for example the raw-based 4-queens), is translated into:

```
MACHINE csp1_st                        /* stateful version of CSP1 */
SETS           SET1
CONSTANTS      SOL1                          /* SOL1 is a non-empty */
PROPERTIES     SOL1 <: SET1 & SOL1 /= {}     /*  subset of SET1     */
VARIABLES      sol1
INVARIANT      sol1 : SOL1
INITIALISATION
    sol1 :: SOL1
OPERATIONS
    sol1_ <-- read1 = BEGIN
        sol1_ := sol1                        /* reading of the state */
    END
END
```

When a user of this machine calls the operation read1, he obtains a solution of the problem (a member of SOL1). The second CSP, CSP2 (for example the square-based 4-queens), is translated into the same machine where all the occurrences of '1' are replaced by '2'.

We now want to formalize the fact that CSP2 is a reformulation of CSP1: we will follow (a part of) the *reducibility* idea of [18] and try to transform solutions of CSP2 into solutions of CSP1. For this, in our framework the most natural thing to do is to build a refinement of csp1_st based on csp2_st. It implies to state a linking invariant between the two states (a *channeling constraint* according to [4]): we name it LINK21, a subset of SET2*SET1 (in our 4-queens example, the set of consistent pairs (square-representation, row-representation)). Let's try this refinement:

```
REFINEMENT csp1_st_r REFINES csp1_st
INCLUDES csp2_st   /* in particular its state sol2,
                      already initialised in SOL2          */
CONSTANTS      LINK21
PROPERTIES     LINK21 : SET2 * SET1  /* the typing is mandatory */
               /**** other properties will have to be added ****/
VARIABLES      sol1
INVARIANT      (sol2,sol1):LINK21
INITIALISATION
    sol1 :: LINK21[{sol2}]  /* a sol1 'compatible' with sol2 */
OPERATIONS
    sol1_ <-- read1 = BEGIN
        sol1_ := sol1
    END
END
```

First, let's explain the semantics. Remember that csp1_st is a black box SOL1 solver. Here, csp1_st_r is a grey box built on two successive black sub-boxes:

1. csp2_st is included, so we have in the state a variable sol2 initialized in SOL2;
2. then we have a complement of initialisation, sol1 :: LINK21[{sol2}], which means: give me one of the sol1's which satisfy (sol2,sol1) : LINK21.

Of course this refinement is not provable, because nothing proves that this sol1 belongs to SOL1: we have not characterized enough LINK21. To do this, we just have to consider the unproved proof obligations (see the annex): they are the weakest properties to add to obtain a correct refinement. The refinement becomes provable when the line /**** other properties will have to be added ****/ is replaced by the two properties:

```
  & SOL2 <: dom(LINK21)
        /* every element of SOL2 is in the domain of LINK21 */
  & LINK21[SOL2] <: SOL1
        /* SOL2 elts are associated only with SOL1 elts */
```

This necessary and sufficient characterization of LINK21 will remain valid in the next section, where more general problems are considered.

The stateless traduction of CSP1 is the following

```
MACHINE    csp1_op
SETS       SET1
CONSTANTS  SOL1
PROPERTIES SOL1 <: SET1 & SOL1 /= {}
OPERATIONS
    sol1 <-- solve1 = BEGIN sol1 :: SOL1 END
END
```

and the (complete) refinement is

```
REFINEMENT csp1_op_r REFINES csp1_op
INCLUDES csp2_op
CONSTANTS LINK21
PROPERTIES LINK21 <: SET2*SET1
         & SOL2 <: dom(LINK21)
         & LINK21[SOL2] <: SOL1
OPERATIONS
    sol1 <-- solve1 = VAR sol2 IN
            sol2 <-- solve2;
            sol1 :: LINK21[{sol2}]
    END
END
```

The sequence "solve SOL2, then translate towards SOL1" is more explicit. The characterization of `LINK21` is performed in the same way (i.e. via the proof obligations) and is the same.

4.2 The general case: problems may be unsolvable

The stateful style needs to express the specification of the problem both in the initialisation and in the invariant (because the initialisation establishes the invariant). Let's recall this substitution:

```
IF   SOL /= {}                          /* IF SOL is not the empty set */
THEN sat := TRUE    || vv :: SOL        /* THEN a SOLution exists      */
ELSE sat := FALSE   || vv :: SET        /* ELSE no SOLution            */
END
```

It can be rewritten as a 'such that' substitution:

```
sat,vv :( ( SOL /= {}  => (sat = TRUE  & vv : SOL) )
        & ( SOL  = {}  => (sat = FALSE & vv : SET) ) )
```

and the predicate inside will be the invariant. The translation of CSP1 is:

```
MACHINE  csp1_st
SETS SET1
CONSTANTS  SOL1
PROPERTIES SOL1 <: SET1
VARIABLES  sat1, sol1
INVARIANT sat1 : BOOL &  sol1: SET1
    & ( SOL1 /= {}  => (sat1 = TRUE  & sol1 : SOL1) )
    & ( SOL1  = {}  => (sat1 = FALSE                ) )
INITIALISATION
    IF   SOL1 /= {}
    THEN sat1 := TRUE    || sol1 :: SOL1
    ELSE sat1 := FALSE   || sol1 :: SET1
    END
OPERATIONS
    sat1_,sol1_ <-- read1 = BEGIN
        sat1_ := sat1  || sol1_ := sol1
    END
END
```

We will omit the refinement `csp1_st_r` and show the refinement only in the stateless case, because the stateless translation is simpler:

```
MACHINE  csp1_op
SETS SET1
CONSTANTS SET1
PROPERTIES SOL1 <: SET1
OPERATIONS
    sat1,sol1 <-- solve1 = BEGIN
        IF   SOL1 /= {}
        THEN sat1 := TRUE  || sol1 :: SOL1
        ELSE sat1 := FALSE || sol1 :: SET1
        END
    END
END
```

The refinement of CSP1 via CSP2 is proved via exactly the same linking relation LINK21, we need only a supplementary condition on SOL1 and SOL2: their *equi-satisfiability*.

```
REFINEMENT csp1_op_r REFINES csp1_op
INCLUDES   csp2_op
CONSTANTS  LINK21
PROPERTIES LINK21 <: SET2 * SET1
         & SOL2 <: dom(LINK21)
         & LINK21[SOL2] <: SOL1
         & ((SOL2 = {}) => (SOL1 = {}))  /* equi-sat 1/2 : needed  */
ASSERTIONS ((SOL1 = {}) => (SOL2 = {}))  /* equi-sat 2/2 : implied */
OPERATIONS
    sat1,sol1 <-- solve1 = VAR sol2 IN
            sat1,sol2 <-- solve2;
            IF sat1 = TRUE
            THEN sol1 :: LINK21[{sol2}]
            ELSE sol1 :: SET1
            END
    END
END
```

Again, the meaning is that to solve CSP1, you first solve CSP2, and then you transform the CSP2 solution into a CSP1 solution with the LINK21 relation (which would be refined into a function if we did the software refinement all the way). For example, if SOL1 is the set of even numbers, and SOL2 is the set of odd numbers, `LINK21(v2,v1)` could be simply: $v2 = v1 + 1$ (ultimately refinable into the substitution $v1 := v2 - 1$)

▶ **Theorem 1.** `csp1_op_r` *(resp.* `csp1_st_r`*) refines* `csp1_op` *(resp.* `csp1_st`*) if and only if the following two conditions are met:*

equi-satisfiability $\mathrm{SOL}_1 = \emptyset \iff \mathrm{SOL}_2 = \emptyset$

transformer *there is a relation* $\mathrm{LINK}_{21} \subseteq \mathrm{SET}_2 * \mathrm{SET}_1$ *satisfying*

$$\mathrm{SOL}_2 \subseteq dom(\mathrm{LINK}_{21}) \quad \wedge \quad \mathrm{LINK}_{21}[\mathrm{SOL}_2] \subseteq \mathrm{SOL}_1$$

Proof. See the appendix for the stateless version. ◀

We propose to consider that *two models are equivalent when they reformulate each other*, which means that we must be able to create two relations `LINK21` and `LINK12` satisfying the constraints above.

4.3 Surprising corollaries, and relevance of the proposal

An easy corollary is that, when SOL1 and SOL2 are empty (i.e. CSP1 and CSP2 are both unsatisfiable), the empty LINK21 satisfies the conditions of the theorem, so *CSP1 and CSP2 reformulate each other*.

A more surprising corollary is that, when SOL1 and SOL2 are not empty (i.e. CSP1 and CSP2 are both satisfiable), again *CSP1 and CSP2 reformulate each other*. Choose any sol1 in SOL1, the following LINK21 satisfies the conditions of the theorem:

$\{ \ (sol2, sol1) \mid sol2 : SOL2 \ \}$

In a way, LINK21 incorporates a CSP1 solver, and this is exactly what [18] have avoided by requiring their transformation to be functional, surjective (onto) and "syntactic". The first two conditions (functional and/or surjective) are easily incorporated in our framework, because they are mathematical. But the third cannot even be stated in the B language. Fortunately, it is naturally taken into account by software engineering coding rules.

From a theoretical point of view, the conditions of theorem 1 have to be completed by surjectivity, because saying that every (satisfiable) problem reformulates every (satisfiable) problem is not very interesting. But from an engineering point of view, surjectivity is too strong a constraint, because sometimes, interesting reformulations loose solutions (in an optimization context, the important point is not to loose good solutions). So in a certification context, we will keep theorem 1 as it is, with a 'coding rules' like supplement.

5 Related works

The practical link between software formal methods and CP is the use of constraint solvers to *animate* abstract specifications (which are usually non-deterministic: it is not possible to simulate them).

The other link is the comparable expressiveness of the languages: the Z-notation and the B-method (the former is the predecessor of the specification part of the latter) are referenced or at least mentioned in [5, 6], and [17] goes further and claims that the Z-notation can be used as a high-level constraint modelling language.

Software refinement is rarely mentioned, a notable exception is [9] who propose to use it to transform specifications into lower-level models suitable for efficient solving.

There is also a notable link between the automatic traduction of high-level models (Essence, Minizinc) towards CP solvers and *automatic software refinement* [2, 11].

6 Conclusion and way forward

The proposed notion of reformulation needs to be reworked in an academic context, but seems adequate in an engineering context, with two weaknesses however.

The first weakness is the non-preservation of the set of solutions. We don't see this as a problem in a constraint satisfaction context, but it is clearly not suited for a constrained optimisation context: how can you ensure that an optimum is preserved if potential solutions are no more considered?

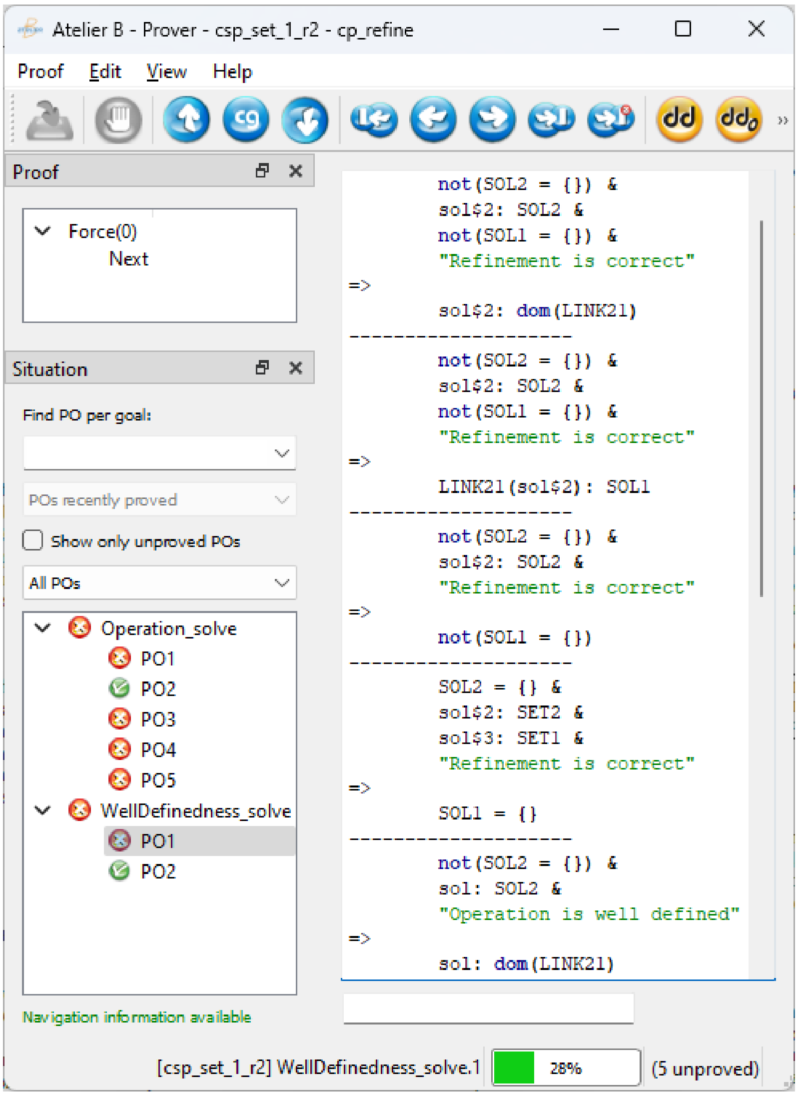
The second weakness is the difficulty of proofs even on simple cases. Here the solution seems closer: the concept of 'representations annotations' [6] [12] may be the key to automatically generate the linking invariant and tactics for guiding the proofs of the proof obligations.

References

1. Jean-Raymond Abrial. *The B-Book - Assigning Programs to Meanings*. Cambridge university press, 1996.
2. Lilian Burdy and Jean-Marc Meynadier. Automatic refinement. *Proceedings of BUGM at FM*, 99, 1999.
3. BMW Cheng, Kenneth M. F. Choi, Jimmy Ho-Man Lee, and JCK Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4:167–192, 1999.
4. BMW Cheng, Jimmy Ho-Man Lee, and JCK Wu. Speeding up constraint propagation by redundant modeling. In *Principles and Practice of Constraint Programming—CP96: Second International Conference, CP96 Cambridge, MA, USA, August 19–22, 1996 Proceedings 2*, pages 91–103. Springer, 1996.
5. Pierre Flener, Justin Pearson, and Magnus Ågren. Introducing esra, a relational language for modelling combinatorial problems. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 214–232. Springer, 2003.
6. Alan M Frisch, Chris Jefferson, Bernadette Martınez-Hernández, and Ian Miguel. The rules of modelling: Automatic generation of constraint programs. In *3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, 2004. `https://conjure.readthedocs.io/en/latest/essence.html`.
7. PA Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proc.: ECAI*, volume 92, pages 31–35, 1992.
8. Google. CP-SAT. `https://developers.google.com/optimization/cp/cp_solver`, 2025.
9. Stefan Hallerstede, Miran Hasanagić, Sebastian Krings, Peter Gorm Larsen, and Michael Leuschel. From software specifications to constraint programming. In *Software Engineering and Formal Methods: 16th International Conference, SEFM 2018, Held as Part of STAF 2018, Toulouse, France, June 27–29, 2018, Proceedings 16*, pages 21–36. Springer, 2018.
10. Yat Chiu Law and Jimmy Ho-Man Lee. Model induction: a new source of csp model redundancy. In *AAAI/IAAI*, pages 54–61, 2002.
11. Thierry Lecomte. Return of experience on automating refinement in b. In *1st International Workshop about Sets and Tools (SETS 2014)*, pages 57–68, 2014.
12. Bernadette Martınez-Hernández and Alan M Frisch. The automatic generation of redundant representations and channelling constraints. In *Proceedings of the 5th International Workshop on Constraint Modelling and Reformulation*, pages 42–56, 2006.
13. Microsoft. Online Z3 Guide. `https://microsoft.github.io/z3guide`, 2025.
14. David Mitchell. On correctness of models and reformulations (preliminary version). In *PTHG-20: The Fourth Workshop on Progress Towards the Holy Grail*, 2020.
15. Bernard A Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert: Intelligent Systems and Their Applications*, 5(3):16–23, 1990.
16. Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007. `https://minizinc.org`, Monash University.
17. Gerrit Renker and Hatem Ahriz. Building models through formal specification. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 395–401. Springer, 2004.
18. Francesca Rossi, Charles J Petrie, and Vasant Dhar. On the equivalence of constraint satisfaction problems. Technical Report ACT-AI-222-89, MCC, Austin, 1989. A shorter version appears in proc. ECAI-90.
19. Wikipedia. B-Method/Atelier B. `https://en.wikipedia.org/wiki/B-Method#Atelier_B`, 2025.
20. Wikipedia. Hoare logic. `https://en.wikipedia.org/wiki/Hoare_logic`, 2025.
21. Wikipedia. Refinement (computing). `https://en.wikipedia.org/wiki/Refinement_(computing)`, 2025.

A Proofs

On the left, a screenshot of Atelier-B with the 5 unproved proof obligations which appear when LINK21 is not characterized and SOL1-SOL2 are not equi-satisfiable. On the right, a translation in standard mathematics.



$$s_2 \in \mathrm{SOL}_2 \Rightarrow s_2 \in dom(\mathrm{LINK}_{21})$$

$$\mathrm{SOL}_2 \neq \emptyset \wedge \mathrm{SOL}_1 \neq \emptyset \Rightarrow \mathrm{LINK}_{21}[\mathrm{SOL}_2] \subseteq \mathrm{SOL}_1$$

$$\mathrm{SOL}_2 \neq \emptyset \Rightarrow \mathrm{SOL}_1 \neq \emptyset$$

$$\mathrm{SOL}_2 = \emptyset \Rightarrow \mathrm{SOL}_1 = \emptyset$$

$$s_2 \in \mathrm{SOL}_2 \Rightarrow s_2 \in dom(\mathrm{LINK}_{21})$$