Solver-Aided Expansion of Loops to Avoid Generate-and-Test

Niklas Dewally
School of Computer Science, University of St Andrews, UK

Özgür Akgün
School of Computer Science, University of St Andrews, UK

Abstract

Constraint modelling languages like MiniZinc and Essence rely on unrolling loops (in the form of quantified expressions and comprehensions) during compilation. Standard approaches generate all combinations of induction variables and use partial evaluation to discard those that simplify to identity elements of associative-commutative operators (e.g. true for conjunction, 0 for summation). This can be inefficient for problems where most combinations are ultimately irrelevant. We present a method that avoids full enumeration by using a solver to compute only the combinations required to generate the final set of constraints. The resulting model is identical to that produced by conventional flattening, but compilation can be significantly faster. This improves the efficiency of translating high-level user models into solver-ready form, particularly when induction variables range over large domains with selective preconditions.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence

Keywords and phrases Constraint modelling, Loop unrolling, Solver-aided reformulation

Supplementary Material Our prototype extension to Conjure is available at `https://github.com/conjure-cp/conjure-oxide`. The models we use in our experiments, scripts we use to run the experiments, raw results and plotting scripts are available at `https://github.com/niklasdewally/2025-comprehension-unrolling-modref`.
Software (Source Code): `https://doi.org/10.5281/zenodo.16723727`
Dataset: `https://doi.org/10.5281/zenodo.16724771`

Acknowledgements We thank the members of the "AI for Decision Making" Vertically Integrated Project at the University of St Andrews for their contributions.

1 Introduction

Constraint modelling languages such as Essence [1], Essence Prime [4], and MiniZinc [3] allow users to specify complex combinatorial problems in a concise and declarative manner. However, the efficiency of compiling these models often hinges on how loops and quantified expressions are expanded, minor syntactic changes can lead to substantial differences in compilation time and memory use. Current tools rely on generate-and-test strategies combined with partial evaluation, making performance fragile and dependent on modeller expertise.

We present a solver-aided method for expanding comprehensions and quantified expressions that is robust to syntactic formulation. Our approach eliminates the need for full enumeration and partial evaluation, ensuring that only relevant combinations of induction variables are generated, regardless of how guards or conditions are expressed. This step moves declarative modelling closer to its ideal: enabling users to express intent without hidden performance traps. The underlying technique is general and could be adapted for any context where side-effect-free loop unrolling is required.

Empirical results on standard benchmarks demonstrate that our solver-aided expansion method robustly scales across different syntactic formulations of constraints. It significantly reduces compilation times compared to traditional generate-and-test approaches, particularly

```
int: n;
array[1..n] of var 1..2: class;
constraint
  forall(a,b,c in 1..n)(
    (a<=b /\ b<=c /\ a^2 + b^2 = c^2) -> (
      class[a] != class[b] \/
      class[b] != class[c] \/
      class[c] != class[a]));
solve satisfy;
```

(a) Naive MiniZinc model. As an indicative figure, this model takes 7.8 seconds to translate, with $n = 200$.

```
int: n;
array[1..n] of var 1..2: class;
constraint
  forall(a,b,c in 1..n)(
    if (a<=b /\ b<=c /\ a^2 + b^2 = c^2)
    then (class[a] != class[b] \/
          class[b] != class[c] \/
          class[c] != class[a])
    else true endif);
solve satisfy;
```

(b) Another MiniZinc model using an if statement. As an indicative figure, this model takes 15.3 seconds to translate, with $n = 200$.

```
int: n;
array[1..n] of var 1..2: class;
constraint
  forall(x in [ class[a] != class[b] \/
                class[b] != class[c] \/
                class[c] != class[a]
              | a,b,c in 1..n
                where (a<=b /\ b<=c /\ a^2 + b^2 = c^2)
              ])(x);
solve satisfy;
```

(c) Faster MiniZinc model using a static comprehension guard. As an indicative figure, this model takes 1.3 seconds to translate, with $n = 200$.

Figure 1 MiniZinc models of the Boolean Pythagorean Triples Problem

when implicit conditions or large domains are involved. This robustness and efficiency makes constraint model compilation more reliable, allowing users to express intent clearly without concern for hidden performance costs.

2 Motivation

Efficient unrolling of loops, quantified expressions, and comprehensions is critical in constraint modelling languages such as MiniZinc and Essence Prime, as it directly affects the scalability and performance of model compilation. Typically, these languages expand loops into explicit constraints through generate-and-test strategies, enumerating all combinations and subsequently filtering irrelevant cases.

```
triples = []
for a in range(1, n+1):
  for b in range(1, n+1):
    for c in range(1, n+1):
      if a <= b <= c and a ** 2 + b ** 2 == c ** 2:
        triples.append([a,b,c])
```

(a) Naive Python implementation. For $n = 1000$, this program takes roughly 3 minutes to run.

```
triples = []
for a in range(1, n+1):
  for b in range(a, n+1):
    c_squared = a**2 + b**2
    c = int(c_squared ** 0.5)
    if c < b or c > n:
      continue
    if c**2 == c_squared:
        triples.append([a, b, c])
```

(b) Optimised Python implementation. For $n = 1000$, this program takes less than a second to run.

Figure 2 Python enumeration of the relevant triples for the Boolean Pythagorean Triples Problem

To illustrate, consider the Boolean Pythagorean Triples Problem [2], a well-known constraint satisfaction problem that partitions integers into two classes such that no Pythagorean triple `(a,b,c)` lies entirely within a single class. A natural MiniZinc formulation of this problem (Figure 1a) explicitly loops through all possible triples `(a,b,c)`, creating constraints for each combination. Although intuitive, this approach scales poorly: MiniZinc takes approximately 1.07 seconds for $n = 100$ and dramatically increases to 7.82 seconds at $n = 200$, due to enumerating and evaluating all n^3 combinations. Using an if statement instead of a logical implication as in Figure 1b roughly doubles the time taken.

In contrast, adding a simple comprehension guard drastically improves MiniZinc's compilation time. Figure 1c, which filters combinations explicitly through a comprehension guard, compiles significantly faster (1.28 seconds for $n = 200$), despite being semantically equivalent to the original formulation. This performance discrepancy highlights how subtle modelling differences critically impact practical usability.

Such issues are not exclusive to constraint modelling languages. A naive implementation of triple generation in Python (Figure 2a) also enumerates all possibilities and performs poorly, taking roughly two minutes for $n = 1000$. While an optimised Python version (Figure 2b) is faster, it requires manual optimisation and increased complexity, contradicting the simplicity and clarity benefits provided by declarative languages.

Essence Prime, the input language of the modelling tool Savile Row, faces similar performance concerns. A Essence Prime formulation using comprehensions without guards (Figure 3c) compiles inefficiently (162.43 seconds for $n = 100$), as all combinations are generated and then filtered using partial evaluation. Conversely, a guarded comprehension (Figure 3b) compiles notably faster (1.95 seconds for $n = 100$), as does a direct formulation using quantifiers (Figure 3a, 5.85 seconds for $n = 100$). These differences are critical, especially as automated model-generation tools like Conjure frequently produce formulations without explicit guards.

Current methods relying on partial evaluation suffer from substantial overhead, particularly in memory consumption and unnecessary computation, severely restricting scalability.

```
given n: int(1..)
find class: matrix indexed by [int(1..n)] of int(1..2)
such that
  forAll a,b,c: int(1..n).
    (a**2 + b**2 = c**2 /\ a<=b /\ b<=c) ->
    or([class[a] != class[b],
        class[b] != class[c],
        class[c] != class[a]])
```

(a) Using a `forall` quantified expression. As an indicative figure, this model takes 9.4 seconds to translate, with $n = 200$.

```
given n: int(1..)
find class: matrix indexed by [int(1..n)] of int(1..2)
such that
  and([or([ class[a] != class[b]
          , class[b] != class[c]
          , class[c] != class[a]])
      | a,b,c: int(1..n), a<=b, b<=c, a**2+b**2=c**2
      ])
```

(b) Using a comprehension with static guards. As an indicative figure, this model takes 11.5 seconds to translate, with $n = 200$.

```
given n: int(1..)
find class: matrix indexed by [int(1..n)] of int(1..2)
such that
  and([(a**2 + b**2 = c**2 /\ a<=b /\ b<=c) ->
       or([class[a] != class[b]
          , class[b] != class[c]
          , class[c] != class[a]])
      | a,b,c: int(1..n)
      ])
```

(c) Using a comprehension without static guards. The translation of this model timed out in an hour, with $n = 200$.

Figure 3 Essence Prime models of the Boolean Pythagorean Triples Problem

Moreover, these methods are highly sensitive to subtle changes in model formulation, placing undue responsibility on modellers to optimise their syntax manually.

Our goal is to develop a robust method that eliminates reliance on generate-and-test and partial evaluation strategies. Instead, we propose using solver-aided expansion to compute only necessary combinations, ensuring consistent, efficient compilation regardless of how loops and comprehensions are expressed.

3 Comprehensions in Essence Prime

Many constraint modelling languages, including MiniZinc and Essence Prime, provide *comprehensions*—constructs that allow the concise specification of collections of expressions. The semantics of comprehensions in these languages are closely related: both allow modellers to describe large numbers of similar constraints in a declarative form, often using guards to restrict the range of assignments to induction variables. For this reason, we describe comprehensions as implemented in Essence Prime; all key points apply to MiniZinc as

well, and our methods can be transferred without loss of generality.

3.1 Syntax and Semantics

In Essence Prime, the general form of a matrix comprehension is:

```
[return_expression | i: domain1, j: domain2, cond1, cond2, ...]
```

Here, *induction variables* (e.g., `i`, `j`) are introduced with their respective domains, and any subsequent expressions separated by commas are *comprehension guards*, Boolean conditions restricting which assignments of the induction variables are included. Only assignments for which all guards hold are considered. The *return expression* specifies the value to be computed for each such assignment.

For example:

```
[ b - a | a: int(1..3), b: int(1..6), b % 3 = 0 ]
```

This comprehension generates a matrix of the values `b - a` for all $a \in \{1, 2, 3\}$, $b \in \{1, \ldots, 6\}$, where b is a multiple of 3. The expansion is limited to those combinations by the guard `b % 3 = 0`.

In both Essence Prime and MiniZinc, comprehensions can be used to succinctly express conjunctions, disjunctions, sums, products, and more, by placing the comprehension within the appropriate aggregate operator (e.g., `and`, `or`, `sum`).

3.2 Static and Dynamic Expressions

A key distinction in the context of comprehension unrolling is between *static* and *dynamic* expressions:

- **Static expressions** reference only induction variables. These can be evaluated entirely at model compilation time.
- **Dynamic expressions** involve non-induction variables (e.g., model variables or decision variables), whose values are only determined during solving.

Comprehension guards must be static, as they determine which assignments are included when unrolling the comprehension at compile time. Conditions that depend on decision variables cannot be used as guards and must be placed within the return expression.

For instance, suppose `m` and `n` are model variables. To specify that `m[i] = n[i]` only if `m[i]` is even, we write:

```
and([ (m[i] % 2 = 0) -> m[i] = n[i] | i: int(1..n) ])
```

Here, `m[i] % 2 = 0` is a dynamic condition and appears inside the return expression rather than as a comprehension guard.

3.3 Quantified Expressions

Essence Prime also includes quantified expressions, such as `forAll` and `exists`, which provide an alternative to comprehensions for expressing universal or existential constraints. The quantifier `forAll a, b: int(1..n). P(a, b)` can be expressed equivalently as:

```
and([ P(a, b) | a: int(1..n), b: int(1..n) ])
```

Similarly, existential quantification corresponds to an `or` of a comprehension. Unlike comprehensions, quantifiers do not support explicit comprehension guards; guards must be incorporated into the quantified condition itself. From a practical standpoint, quantifiers and comprehensions are interchangeable, as quantifiers are internally lowered to comprehensions during model rewriting in many systems, including Conjure.

3.4 Impact of Formulation on Rewriting Performance

Although comprehensions with static guards, comprehensions with guards embedded in the return expression, and quantified expressions are semantically equivalent, only the first can be efficiently unrolled. When guards are omitted or moved inside the return expression, the model compiler is forced to enumerate all possible combinations of induction variables, then use partial evaluation to discard those which turn out to be irrelevant. For large domains, this generates significant memory and computational overhead, and can quickly become infeasible.

For example, consider:

```
and([ (i % 2 = 0) -> m[i] = i | i: int(1..4) ])
```

This comprehension unrolls to four expressions, of which only two matter after partial evaluation. For larger domains, the intermediate blowup can be severe.

In contrast, a comprehension guard such as `i % 2 = 0` restricts the enumeration to only those values where the condition holds, directly controlling the size of the expansion and the resources required.

In summary, both the syntax and placement of conditions in comprehensions have a profound impact on the performance of model compilation. Small syntactic differences can determine whether the compiler must enumerate an infeasible number of combinations, or efficiently target only those that matter.

The next section describes our method for eliminating this fragility, making efficient expansion possible regardless of how conditions are written in the original model.

4 Method: Solver-Aided Expansion of Loops

The scalability of constraint model compilation critically depends on efficient expansion of comprehensions and quantified expressions. In current systems, implicit or poorly-positioned static conditions often result in inefficient generate-and-test enumeration. We present a solver-aided approach that systematically extracts static conditions to ensure only necessary combinations of induction variables are generated, irrespective of the original syntactic form.

4.1 Comprehension Unrolling as a Constraint Problem

We reformulate comprehension unrolling as a constraint satisfaction problem, referred to as a *generator model*. Rather than enumerating all possible assignments to induction variables and filtering them afterwards, we solve a constraints model that finds assignments to the induction variables satisfying the static guards of the comprehension.

Consider the following comprehension:

```
and([ m[i] = i | i: int(1..4), i % 2 = 0 ])
```

The induction variable `i` ranges over $\{1, 2, 3, 4\}$, with the static guard `i % 2 = 0`. The generator model is:

```
find i: int(1..4)
such that i % 2 = 0
```

Solving this gives the assignments `i=2` and `i=4`, producing the constraints `m[2]=2` and `m[4]=4`.

4.2 Lifting Static Guards from Return Expressions

For simpler comprehensions where static guards are explicitly defined as comprehension guards, it is sufficient to use the comprehension guards as the generator model, as described above. However, this approach relies on the user to formulate the model in one of many possible ways, which we want to avoid. For example, consider:

```
and[(i % 2 = 0) -> m[i] = i | i: int(1..4)]
```

A naive approach to finding the static guards would be to look for implications inside `and` quantifiers, add the antecedents, if static, to the generator model, and replace the return expression with the consequent. However, this is not sufficient when static and dynamic guards are combined, such as in:

```
and([!(i % 2 = 0 /\ m[i] % 2 = 0) -> m[i] = i | i: int(1..4)])
```

To solve the problem of identifying static guards in the general case, we convert the entire return expression into a static guard by replacing dynamic sub-expressions with *dummy variables*.

For example, we convert the return type of the above comprehension to a static guard by replacing the dynamic expressions `m[i] % 2 = 0` and `m[i] = i` with boolean variables `Z1` and `Z2`, creating the following generator model:

```
find i: int(1..4)
find Z1, Z2: bool
branching on [i]
such that
  !(i % 2 = 0 /\ Z1) -> Z2 != true
```

As `Z1` and `Z2` are unconstrained, the solver enumerates all possible values of `i` for which the original return expression is false, for some values of `Z1` and `Z2`. As we do not care about the values of the dummy variables, we only branch on the induction values. In Essence Prime, the semantics of the branching on statement means that this generator model returns only unique assignments of the induction variables.

4.3 Aggregates and Identity Values

Aggregate comprehensions (e.g. `and`, `or`, `sum`, `product`) have known identity elements. The identity element can always be removed from an operation's operands without affecting its result; thus, when expanding comprehensions, we exclude expressions that we statically know to be equal to the identity. To do this, in the generator model we ensure that the rewritten return expression is not equal to the identity value of the comprehension. For the example above, the generator model produces the values of `i` where the result of `!(i % 2 = 0 /\ m[i] % 2 = 0)` depends on `m`.

Algorithm 1 Substitute dynamic sub-expressions for dummy variables

Input : The return expression of the comprehension, expr
Input : A list of induction variables
Input : The dummy variable type, dummyVarType
Output : The rewritten return expression
Output : A list of declared dummy variables

```
1  expr ← FirstChild(expr);
2  while expr has a parent do
3      if expr does not reference any non-induction variables then
4          expr ← NextExpression(expr);
5          continue;
6      end
7      hasEligibleChild ← ∃ a descendant d of expr, where TypeOf(d) = dummyVarType,
        and d references non-induction variables;
8      exprIsRightType ← TypeOf(expr) = dummyVarType;
9      if !hasEligibleChild & exprIsRightType then
10         replace expr with a new dummy variable;
11         expr ← NextExpression(expr);
12         continue;
13     end
14     if !hasEligibleChild & !exprIsRightType then
15         walk up the expression tree until expr is the right type;
16         replace expr with a new dummy variable;
17         expr ← NextExpression(expr);
18         continue;
19     end
20     // expr has an eligible child
21     if expr references induction variables then
22         expr ← FirstChild(expr);
23         continue;
24     end
25     else
26         replace expr with a new dummy variable;
27         expr ← NextExpression(expr)
28     end
29     return expr and a list of newly declared dummy variables;
30 end
31 Function NextExpression(expr)
32     return the right sibling expression of expr if one exists, or its parent expression;
33 end
34 Function FirstChild(expr)
35     return the leftmost sub-expression of expr;
36 end
```

4.4 Algorithm

Now we turn our focus to the algorithm used to perform this substitution.

An important consideration is which sub-expressions to replace with a dummy variable. Minimizing the number of dummy variables reduces the number of variables in the generator model, improving performance. For instance, it is better to rewrite `m[i] % 2 = 0 /\ m[i] % 3 = 0` as `Z` than `Z1 /\ Z2`. At the same time, expressions containing induction variables should not be placed inside a dummy variable, so that we do not remove any static guards. Finally, we require the type of dummy variables to be equal to the type of the identity element of the comprehension.

We approach this as a tree traversal over the return expression, the algorithm for which is given in Algorithm 1. For each child expression of the return expression, we check the variables it references. If it references non-induction variables only, and is of the right type, we turn it into a dummy variable. If it also references induction variables, we check to see if there exists a child expression of the right type containing non-induction variables. If so, we move our cursor to the first child of the current expression, and repeat this process.

4.5 Proof of Validity and Correctness

We consider our algorithm to be valid if we can convert all well-typed return expressions into static expressions that do not reference any non-induction variables. Additionally, we consider it correct if the unrolling process does not remove any return expressions whose values depend on assignments of model variables determined at solve-time.

As dummy variables are unconstrained, introducing a dummy variable does not change the possible values the induction variables can take. Furthermore, when a dummy variable removes an induction variable from the return expression it increases the possible combinations of induction variables, delaying the evaluation of statically known guards to post expansion partial evaluation or solve-time. As such, introducing a dummy variable only ever weakens the constraints on the induction variables, and does not affect correctness.

The backtracking case on lines 14 to 19 applies when a sub-expression still contains non-induction variables and no other cases apply. Here, the algorithm traverses up the expression tree until it can find an expression of the right type to convert into a dummy variable. In the worst case, the algorithm will backtrack up to the root of the expression tree, focusing on the entire return expression. As the type of a dummy variable is equal to the expected type of the operands inside the comprehension's operator, and a well-typed return expression must be this type, we can always turn the entire return expression into a dummy variable. Therefore, our algorithm is valid for all well-typed return expressions.

4.6 Sketch of the Unrolling Process

In summary, the solver-aided unrolling process consists of four main steps:

1. The induction variables and explicit guards from the comprehension are added to the generator model.
2. The return expression is rewritten to a static guard by substituting dynamic sub-expressions with dummy variables (Subsection 4.4), and added to the generator model.
3. The generator model is solved to generate valid combinations of induction variables.
4. Each returned combination is substituted into the original return expression to produce the fully expanded set of constraints.

4.7 Unrolling Non-aggregate Comprehensions

So far, we have focused on the efficient unrolling of aggregate comprehensions. However, in Essence Prime, comprehensions can be used inside any operator with a matrix argument, whether aggregate or not: for example, inside of an `allDiff` constraint.

As our return expression rewriting algorithm only works for aggregate comprehensions, we skip this step for non-aggregate comprehensions, instead using a generator model constructed from the induction variables and explicit comprehension guards only.

4.8 Implementation

We implemented this method in a prototype extension to Conjure[1], where generator models are expressed in Essence and solved by Minion. Integration with other tools such as Savile Row and MiniZinc is straightforward. The method introduces negligible overhead when static guards are explicit and provides substantial gains for models where conditions are implicit or not well-placed.

5 Evaluation

To assess the impact and generality of our solver-aided comprehension expansion technique, we compare its performance against existing approaches in both MiniZinc and Essence Prime (Savile Row) across a range of models and problem sizes. Our focus is on the time taken to expand and flatten comprehensions for constraint solving, as this is the main bottleneck addressed by our approach.

We evaluated the following compilation pipelines:

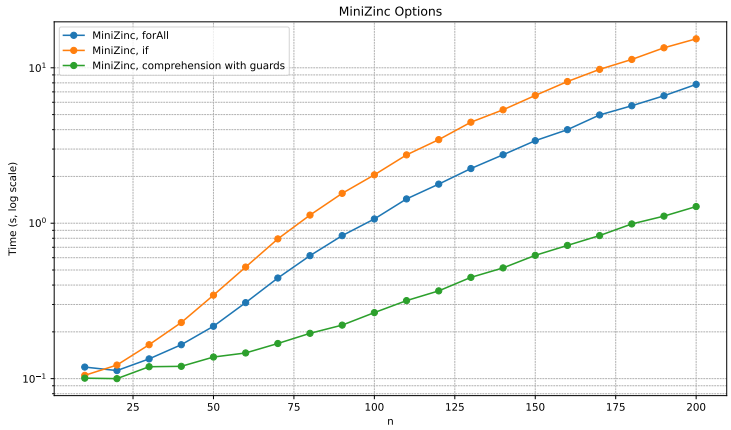
- MiniZinc + Chuffed: Three variants of the Boolean Pythagorean Triples model, differing in how guards are placed (fully explicit comprehension guards, guards embedded in the return expression, and a `forAll` formulation).
- Essence Prime + Savile Row + Minion: The same three variants as above.
- Two variants of our approach: a simple version handling only static guards and a full version that handles both the static guards and introduces dummy variables for dynamic conditions.

For each toolchain, we use three representative models of the Boolean Pythagorean Triples problem. These differ only in the syntactic position of guards; whether conditions on the induction variables appear as comprehension guards, are placed inside return expressions, or are handled by quantifiers. This directly affects the performance of current flattening strategies but should not affect our approach, which is designed to be robust to such variation.

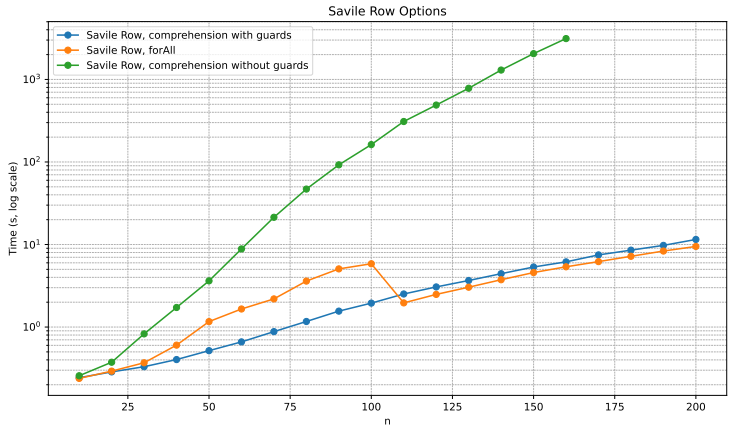
We use the model given in Figure 3c when running our proposed approaches, which places all guards inside the return expression. This model is particularly problematic for Savile Row, which must enumerate all variable combinations before partial evaluation, resulting in the slowest compilation times among the methods tested.

Figure 4c presents the time taken by all eight approaches to expand the models, as n increases. The vertical axis is logarithmic to accommodate the large range of values observed. The figures show that:

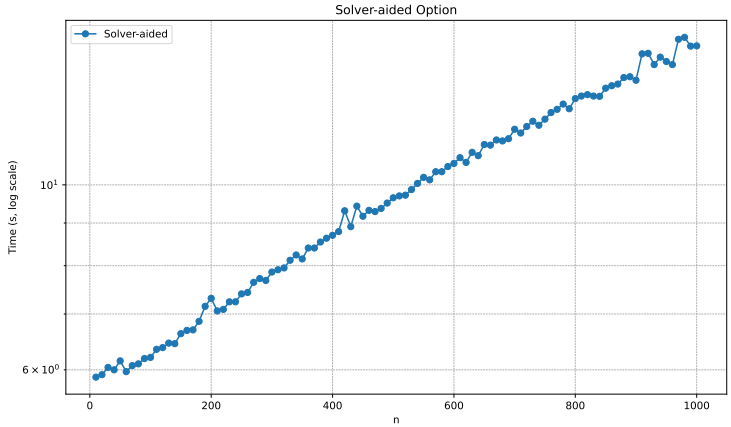
[1] available online at `https://github.com/conjure-cp/conjure-oxide`



(a) The scaling behaviour of the MiniZinc translations, when targeting Chuffed.



(b) The scaling behaviour of the Savile Row translations, when targeting Minion. 1-hour time limit.



(c) The scaling behaviour of the Solver-aided option. Note that this is a prototype implementation that is not feature-complete. We measure time taken to unroll the input model, without converting it to a target solver.

Figure 4 Comparison of model translation or comprehension expansion times for different modelling approaches. Notice that the solver-aided option ranges from $n = 10$ to $n = 1000$ whereas the other options range up to $n = 200$.

- The time required by MiniZinc and Savile Row varies widely depending on how the model is formulated. Poor placement of guards (i.e., inside return expressions rather than as comprehension guards) leads to dramatic increases in time and memory usage.
- Even the most efficient conventional models scale at least linearly with n, but models using generate-and-test can become infeasible at modest problem sizes.

An interesting observation is that the `forAll`-based model for Savile Row exhibits periodic variation in timing (e.g., $n = 120$ is faster than $n = 100$), due to the binary splitting algorithm used to expand quantifiers internally.

The models we use in our experiments, scripts we use to run the experiments, raw results and plotting scripts are available in the accompanying repository on GitHub: `https://github.com/niklasdewally/2025-comprehension-unrolling-modref`.

The results confirm that current tools for constraint model expansion are sensitive to minor syntactic changes in the model. In contrast, our solver-aided method provides consistent, scalable performance, regardless of how guards and conditions are written.

It is important to note that our prototype is not a full reimplementation of the entire compilation toolchain; the times reported reflect only the comprehension expansion phase. In practice, the benefits observed here would be most pronounced in models where comprehensions dominate the rewriting workload, but even in mixed models the elimination of generate-and-test bottlenecks will improve overall performance.

Our method cannot offer a performance benefit in cases where there are no static conditions or guards to exploit (i.e., when all constraints depend on dynamic variables). In our experiments, for these cases the overhead of constructing and solving a trivial generator model is negligible compared to the baseline. Moreover a dedicated native unrolling algorithm can be employed for these trivial cases to avoid the overheads altogether.

6 Conclusion

We introduced a solver-aided approach to comprehension expansion in constraint modelling languages, addressing the inefficiencies associated with conventional generate-and-test methods. By formulating comprehension unrolling as a constraint satisfaction problem, our method systematically extracts and leverages static conditions, thus avoiding the combinatorial explosion typical of current techniques.

Our experimental evaluation demonstrated substantial performance improvements over traditional approaches, especially for models where guards or conditions were implicitly embedded within return expressions. Crucially, the solver-aided approach ensures consistent performance regardless of syntactic variations in the input model, significantly reducing the sensitivity of compilation times to modelling choices.

Future work includes enhancing our prototype to handle a broader set of language constructs, integrating our method fully within popular constraint modelling tools like Savile Row and MiniZinc, and exploring solver-aided approaches to other aspects of constraint compilation. Ultimately, our approach moves declarative modelling closer to its ideal: enabling users to focus on clarity and correctness of their models without hidden performance penalties.

References

1 Özgür Akgün, Alan M Frisch, Ian P Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Conjure: Automatic generation of constraint models from problem specifications. *Artificial Intelligence*, 310:103751, 2022.

2 Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 228–245, Cham, 2016. Springer International Publishing.

3 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.

4 Peter Nightingale, Özgür Akgün, Ian P Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in savile row. *Artificial Intelligence*, 251:35–61, 2017.