Extending Structured Declarative Language with Constraint Programming via MiniZinc

Mario Alviano[1], Carmine Dodaro[1], Thomas Eiter[2], Tobias Geibinger[2] and Ilaria R. Vasile[1,*]

[1]DeMaCS, University of Calabria, 87036 Rende (CS), Italy
[2]Institute of Logic and Computation, TU Wien, 1040 Vienna, Austria

Abstract

Structured Declarative Language (SDL) was introduced as a high-level specification language for combinatorial search and optimization problems, with its semantics originally defined through translation into Answer Set Programming (ASP). In this paper, we present an extension of SDL, enabling dual-target compilation: SDL programs can now be translated to either ASP or MiniZinc, a standard modeling language for Constraint Programming (CP). This extension addresses the semantic gap between the non-monotonic, stable-model semantics of ASP and the monotonic, classical satisfiability interpretation of CP. By mapping SDL constructs to constraint satisfaction formulations, specifically addressing challenges like negation as failure and recursive definitions, we preserve the flexibility and readability of SDL while unlocking the numerical and scheduling efficiency of CP solvers. We demonstrate the viability and coherence of this approach through complex real-world use cases, including the Nurse Scheduling Problem and Operating Room Scheduling.

Keywords

Knowledge Representation, Constraint Programming, MiniZinc, Answer Set Programming, Combinatorial Search and Optimization

1. Introduction

Declarative approaches are widely adopted for modeling complex combinatorial optimization problems due to their high level of abstraction. However, in practice, the comparative evaluation of different solving technologies on the same problem specification remains challenging. The landscape of combinatorial problem solving is largely dominated by two well-established paradigms: Answer Set Programming (ASP) [4, 8] and Constraint Programming (CP) [13]. Despite their conceptual overlap and shared goals, the two paradigms rely on fundamentally different theoretical foundations. ASP is rooted in non-monotonic reasoning under stable model semantics and excels at modeling problems with complex dependencies, recursion, and incomplete information. On the other hand, CP is grounded in classical satisfiability and monotonic interpretation, relying on powerful domain filtering and constraint propagation techniques, and is particularly effective for problems involving large finite domains, arithmetic reasoning, and scheduling constraints.

This divergence creates a practical modeling gap. The choice of solving technology is often dictated by the structural characteristics of the problem instance, yet selecting a solver typically forces the modeler to adopt a specific modeling language. As a result, evaluating the same problem across different paradigms often requires developing and maintaining multiple encodings, increasing complexity and the risk of inconsistencies.

To address the maintainability issues of declarative encodings, recently the Structured Declarative Language (SDL) has been introduced [1]. Designed as a higher-level abstraction, SDL alleviates well-known limitations of ASP-based encodings [5], such as positional notation, rigid arity, and cascading

CILC 2026: The 41st Italian Conference on Computational Logic, June 23–25, 2026, Ferrara, Italy
*Corresponding author.
mario.alviano@unical.it (M. Alviano); carmine.dodaro@unical.it (C. Dodaro); thomas.eiter@tuwien.ac.at (T. Eiter); tobias.geibinger@tuwien.ac.at (T. Geibinger); vasileilaria@gmail.com (I. R. Vasile)
https://alviano.net/ (M. Alviano); https://www.cdodaro.eu (C. Dodaro)
0000-0002-2052-2063 (M. Alviano); 0000-0002-5617-5286 (C. Dodaro)

rule updates. Through typed records, attribute-order independence, dot-notation access, and an SQL-inspired syntax, SDL enables clear, intuitive, and maintainable problem specifications. Originally, SDL was designed as an abstraction over ASP, with its semantics given by a direct translation into ASP rules.

In this paper, we present a compilation of SDL into MiniZinc, providing an alternative compilation target that instantiates SDL's abstract semantics within the monotonic framework of Constraint Programming. We show how SDL programs can be translated from a single specification into either ASP or MiniZinc [11, 12], a standard modeling language for Constraint Programming. This extension allows modelers to retain a uniform, high-level representation while exploiting different solving paradigms depending on the structure of the problem.

Bridging the semantic gap between ASP and CP poses several theoretical and practical challenges. SDL constructs, originally defined in a non-monotonic setting, must be adapted to the monotonic framework of CP. In particular, we address three main challenges: (i) mapping non-deterministic `guess` constructs to finite-domain decision variables; (ii) handling negation as failure through classical reformulations; and (iii) translating recursive `define` instructions into non-recursive constraint encodings using rank-based techniques.

The proposed approach is validated on two representative case studies: the Nurse Scheduling Problem (NSP) [7] and Operating Room Scheduling (ORS) [10], demonstrating that a single SDL specification can be used to derive correct models for both ASP and CP solvers.

2. Background

2.1. Structured Declarative Language: A Brief Recap

To ease the presentation of the language, SDL instructions are grouped into *structure*, *query*, and *model* instructions.

Structure and Query Instructions. Structure instructions define the shape of the data. A *record structure* has the form:

record *RecordName* : *Attributes*;

where *Attributes* is a comma-separated list of pairs *name* : *type*. Each attribute name is unique within the record, and each type is either a primitive type (**int**, **str**) or another record name.

Query instructions specify the output via directives of the form

show *RecordNames*;

where *RecordNames* is a comma-separated list of *RecordName*.

Model Instructions. Model instructions define derived data, search spaces, and constraints. Within model instructions, a record can be referenced via *RecordNameAlias* of the form

RecordName **as** *a*

where the alias provides a local identifier.

A signed *RecordNameAlias* may be preceded by **not** to express negation.

Attributes of a record are accessed via the dot operator (`.`) starting from the record alias; nested records allow chained accesses. A *value* denotes either a record alias, a dot expression, or an immediate constant (integer or string). Values can be combined into expressions using standard arithmetic operators (`+`, `-`, `*`, `/`) and parentheses, yielding integer or string expressions depending on the result type. A *BooleanExpression* is obtained by comparing two value expressions of the same type using standard comparators (i.e., `<`, `<=`, `>`, `>=`, `==`, `!=`).

A *SimpleFromFragment* has the form

from *SignedAliases* **where** *BooleanExpressions*

where *SignedAliases* is a comma-separated list of signed *RecordNameAliases* and *BooleanExpressions* is a conjunction of Boolean expressions. If no condition is specified, the **where** clause can be omitted.

An *AggregateElement* consists of value expressions optionally filtered by a *SimpleFromFragment*. An *aggregate* has the form

$$agg\ \{AggregateElements\}\ \odot\ ValueExpression$$

where $agg \in \{\texttt{sum}, \texttt{count}, \texttt{min}, \texttt{max}\}$ and $\odot$ is a comparison operator.

A *FromFragment* generalizes the simple form by allowing aggregates:

from *SignedAliases* **where** *BooleanExprs* **having** *Aggregates*

A *define* instruction has the form

define *RecordNameAlias FromFragment*;

Multiple definitions for the same record are allowed and are interpreted disjunctively; cyclic dependencies induce a recursive semantics.

A *guess* instruction has the form

guess *FromFragment CardinalityRestriction GuessElements*;

where *CardinalityRestriction* specifies bounds such as **exactly**, **at least**, and **at most**. Each *GuessElement* consists of a *RecordNameAlias* optionally restricted by a *SimpleFromFragment*.

A *deny* instruction has one of the following forms:

deny *FromFragment*; **deny** *FromFragment* **or pay** *Cost* **at** *Level*;

The first form eliminates all solutions satisfying the *FromFragment*, while the second associates a penalty cost at a given priority level.

The semantics of SDL was defined operationally via a mapping Π_S into an ASP program [1]. In the absence of a standalone formal semantics for SDL, we take this ASP translation as our reference semantics. Consequently, our arguments aim to demonstrate that the CP mapping produces results consistent with the stable models generated by the ASP backend.

Illustrative Example. The following SDL specification solves the Hamiltonian Cycle (HC) problem, a classical graph problem that asks whether a cycle exists that visits each node exactly once.

```
record Node: id: int;
record Arc:     first: Node, second: Node;
record InCycle: first: Node, second: Node;
record Reached: node: Node;
record Start:   node: Node;

guess from Node exactly 1
    InCycle from Arc
        where InCycle.first == Arc.first and InCycle.second == Arc.second
              and Node == Arc.first;

deny from Node
    having count { InCycle.first from InCycle
                   where Node == InCycle.second } != 1;

define Reached from Start where Start.node == Reached.node;
define Reached as r1 from Reached as r2, InCycle
    where r2.node == InCycle.first and r1.node == InCycle.second;

deny from Node, not Reached where Node == Reached.node;
show InCycle;
```

The **guess** selects exactly one outgoing arc per node. The first **deny** enforces exactly one incoming arc per node. The two **define** rules derive reachability from a start node, recursively. The last **deny** forces every node to be reached.

SDL Semantics via ASP. The mapping Π_S produces, for the HC example, the following ASP rules:

```
{inCycle(X,Y) : arc(X,Y)} = 1 :- node(X).
:- node(X), #count{Y : inCycle(Y,X)} != 1.
reached(X) :- start(X).
reached(X) :- reached(Y), inCycle(Y,X).
:- node(X), not reached(X).
```

The **guess** construct is mapped to ASP choice rules, representing nondeterministic selections over admissible facts. Each **deny** is translated into an integrity constraint, enforcing hard restrictions on valid solutions. Finally, **define** rules correspond to standard ASP derivation rules, including recursive definitions when applicable.

An *answer set* (stable model) of this program is a minimal set of ground atoms satisfying all rules under the closed-world assumption: an atom is false unless explicitly derived. For the formal definition of ASP we refer the reader to the respective literature [4, 25].

2.2. Constraint Programming and MiniZinc

Constraint Programming (CP) [13] is a declarative paradigm for solving combinatorial search and optimization problems based on finite-domain variables and constraints. Unlike ASP, which relies on non-monotonic reasoning and stable models, CP follows a monotonic, classical satisfiability interpretation. A problem is modeled as a set of decision variables connected by mathematical and logical constraints, and solving combines search (branching) with constraint propagation techniques.

To interface with different CP solvers without committing to a specific implementation, we adopt MiniZinc as target language. MiniZinc is a high-level, solver-independent modeling language that compiles into FlatZinc, a low-level representation supported by most CP solvers.

To formalize the translation from SDL to MiniZinc, we recall core constructs of the latter in Table 1.

Syntax (MiniZinc)	Construct
`int: n;`	Parameter
`var 1..n: x;`	Variable
`array[1..n] of var bool: a;`	Array
`constraint E;`	Assert Boolean expression E
`/\, \/, ->, <->`, not	Logical operators
`forall(i in S where g)(E)`	Universal quantification with optional guard
`exists(i in S)(E)`	Existential quantification
`sum(i in S)(e)`	Summation aggregate
`bool2int(e)`	Coerce Boolean to 0/1 integer
`if c then e1 else e2 endif`	Conditional
`solve satisfy; / minimize x;`	Feasibility / optimization objective
`output ["\(x)"];`	Output

Table 1
Core MiniZinc constructs used in the translation.

Semantics (sketch). A CP model corresponds to a Constraint Satisfaction Problem (CSP) $\mathscr{P} = \langle X, D, \mathscr{C} \rangle$, where X is a set of variables, D assigns finite domains to them, and $\mathscr{C}$ is a set of constraints. A (complete) assignment ν is a solution if it satisfies all constraints under classical logic (including standard negation and universal quantification). Optimization extends CSPs with an objective function minimized over all solutions.

For a formal definition of CP semantics we refer the reader to the standard literature [13, 11].

3. Mapping SDL to MiniZinc

We now present the compilation of SDL specifications into MiniZinc models. Let S be an SDL specification. The mapping produces a MiniZinc model $\mathscr{M}(S)$. We argue, via a correctness sketch detailed in the online appendix (https://github.com/dodaro/SDL/blob/main/Appendix.pdf), that the solutions of $\mathscr{M}(S)$ are consistent with the valid models of S under the reference ASP semantics.

Throughout this section, we extract snippets from two real-world case studies: the Nurse Scheduling Problem (NSP) and Operating Room Scheduling (ORS). In NSP, nurses must be assigned to daily working shifts. In ORS, patients (represented by `Registration` records) are assigned to Medical Surgical Sessions (modeled via the `Mss` record). The discussion and evaluation of these use cases are deferred to Section 4 and 5.

Data Representation. Each record declaration introduces a finite, bounded domain. For each record R with primitive attributes $(a_1, \dots, a_k)$, the mapping defines:

- A finite index set $D_R = \{1, \dots, |R|\}$, where $|R|$ is the number of ground instances.
- A set of constant arrays $A_{R,a_i} : D_R \to \mathbb{Z}$ encoding attribute values.

Dot expressions are interpreted as array lookups:

$$\texttt{for } r \in D_R, \;\; r.a_i \equiv A_{R,a_i}[r].$$

Nested record attributes are *flattened*: if an attribute a_j of R has type R', each primitive attribute b of R' is expanded into a top-level array $A_{R,a_j.b} : D_R \to \mathbb{Z}$, yielding a fully flat representation natively compatible with CP.

Crucially, as CP requires explicitly bounded variables, the translation performs static *domain inference*. If a record depends on another record, its domain is structurally inherited. The minimum and maximum bounds of any numeric attribute are automatically extracted from the provided static facts, ensuring that all generated MiniZinc variables are tightly constrained. To maintain strict homogeneity in the target model, string attributes are systematically mapped to integer indices into a global string table.

Example 1 (NSP Data and Flattening)**.** We declare static shifts with identifiers and durations as follows:

```
record Shift: id: int, hours: int;
define Shift where Shift.id == 1..3 and Shift.hours == 7;
```

The mapping extracts the number of static facts into a length parameter (`shift_len`) and generates parallel arrays for the attributes. Thus, generic accesses like `Record.attribute` (e.g., `Shift.hours`) translate directly to indexed lookups `record_attribute[i]`:

```
int: shift_len = 3;
array[1..shift_len] of int: shift_id = [1, 2, 3];
array[1..shift_len] of int: shift_hours = [7, 7, 7];
```

From-Fragments and Aggregates. A *SimpleFromFragment* `from` $R_1, \dots, R_k$ `where` φ defines the evaluation domain $\Omega = \{\mathbf{x} \in D_{R_1} \times \cdots \times D_{R_k} \mid \varphi(\mathbf{x})\}$, with the Boolean condition φ acting directly as a logical guard expression in the translation. Aggregates over Ω are compiled using an indicator function $\mathbb{I}[\psi(\mathbf{x})] \in \{0, 1\}$:

$$\texttt{count} \mapsto \textstyle\sum_{\mathbf{x}\in\Omega} \mathbb{I}[\psi(\mathbf{x})], \quad \texttt{sum} \mapsto \textstyle\sum_{\mathbf{x}\in\Omega} e(\mathbf{x}) \cdot \mathbb{I}[\psi(\mathbf{x})], \quad \texttt{min/max} \mapsto \min / \max_{\mathbf{x}\in\Omega\wedge\psi(\mathbf{x})} e(\mathbf{x})$$

When $e(\mathbf{x})$ requires accessing an attribute of a dynamically assigned record, the mapping leverages CP *Element constraints* for indirect indexing on the flattened static arrays. Extremum aggregates are implemented via dynamically filtered array comprehensions. The concrete MiniZinc forms are detailed in the paragraph "Aggregate Mapping".

Guess Instructions. SDL `guess` instructions introduce non-determinism and define the search space. The mapping selects among three encoding schemes based on the structure of the guess.

Scheme 1: Functional Encoding. If the guess targets a *single* record T with an `exactly 1` or `at most 1` restriction and no guarded precondition, it maps to an integer decision variable array:

$$\texttt{array}[1..|F_1|, \ldots, 1..|F_k|] \texttt{ of var } \delta..|T|\texttt{: } g$$

where $\delta = 1$ for `exactly 1` and $\delta = 0$ for `at most 1`. The domain bound encodes the cardinality constraint intrinsically, requiring no additional constraint.

Example 2 (Functional encoding – NSP)**.** Each pair (`Day`, `Nurse`) must be assigned exactly one shift:

```
guess from Day, Nurse
   exactly 1
   Assign from Shift
   where Assign.nurse == Nurse and Assign.shift == Shift and Assign.day == Day;
```

Scheme 1 applies, yielding a compact 2D integer array natively bounded by the shift domain:

```
array[1..day_len, 1..nurse_len] of var 1..shift_len: assign;
```

Scheme 2: Relational Fallback Encoding. If the guess involves multiple target records or permits non-functional assignments, the mapping produces a Boolean matrix over all source and target dimensions:

$$\texttt{array}[1..|F_1|, \ldots, 1..|F_k|, 1..|T_1|, \ldots, 1..|T_m|] \texttt{ of var bool: } g$$

Cardinality restrictions become explicit sum constraints:

$$\texttt{constraint forall}(\mathbf{i})\Big(\textstyle\sum_{\mathbf{j}} \texttt{bool2int}(g[\mathbf{i},\mathbf{j}]) \odot c\Big)$$

Boolean conditions in the `where` clause are enforced as logical implications (e.g., $g[\mathbf{i},\mathbf{j}] \rightarrow \varphi(\mathbf{i},\mathbf{j})$).

Example 3 (Relational encoding – ORS)**.** Patients (`Registration`) must be assigned to surgical sessions (`Mss`) of the same medical specialty, but a patient may remain unscheduled:

```
guess from Registration, Mss
   Assign where Assign.registration == Registration and Assign.mss == Mss
     and Registration.specialty == Mss.specialty;
```

Scheme 2 applies. Since the `where` clause restricts valid assignments, the mapping generates a Boolean matrix and translates the compatibility guard into an implication constraint:

```
array[1..registration_len, 1..mss_len] of var bool: assign;
constraint forall(r in 1..registration_len, m in 1..mss_len)(
   assign[r, m] -> (registration_specialty[r] == mss_specialty[m]));
```

Scheme 3: Inclusion Encoding. When the guess has no explicit target domain payload, it acts as a logical selection over the source domain. The mapping produces a Boolean array over the source dimensions only:

$$\texttt{array}[1..|F_1|, \ldots, 1..|F_k|] \texttt{ of var bool: } g$$

Cardinality restrictions are encoded as explicit sum constraints:

$$\texttt{constraint} \sum_{i \in D_{F_1} \times \cdots \times D_{F_k}} \texttt{bool2int}(g[i]) \odot c$$

Guarded preconditions are handled via `if-then-else` blocks inside the sum. The following example illustrates a subset selection with an upper cardinality bound.

Example 4 (Inclusion encoding – Subset selection)**.** Consider a guess selecting a subset of nurses (e.g., to form a special team) with a maximum capacity:

```
guess from Nurse
   at most 3 InTeam
      where Nurse==InTeam.nurse;
```

This is mapped to a 1D Boolean array over the source dimension. The cardinality restriction is enforced via a numerical sum of the coerced Boolean variables:

```
array[1..nurse_len] of var bool: inteam;
constraint sum(i in 1..nurse_len)(bool2int(inteam[i])) <= 3;
```

Define Instructions and Recursive Derivation. A `define` instruction introduces an intensional Boolean relation over existing domains: $R : D_{R_1} \times \cdots \times D_{R_k} \to \{0, 1\}$.

Because the CP paradigm relies strictly on finite, pre-declared variables, the translation materializes R as a multidimensional Boolean array. To determine the exact dimensions of this array statically, the mapping leverages the standard safety requirements of ASP (i.e., every variable must be anchored to a positive domain literal). By computing the Cartesian product of the safely bound attributes' base domains, the translation seamlessly infers the maximum possible bounds for R.

The generated constraints restrict this array, setting to `true` only those cells that are logically deducible. To achieve this, the mapping implements *Clark's Completion* [19] by generating a biconditional constraint across all defining rules B_i:

$$R(\mathbf{x}) \leftrightarrow \bigvee_i B_i(\mathbf{x}).$$

Multiple definitions for the same record are merged disjunctively. For non-recursive instructions this biconditional is sufficient: by Fages' Theorem [18], tight programs yield Clark's Completion models that coincide exactly with the stable models. The full correctness argument is given in the online appendix.

Example 5 (Logical derivation – Hamiltonian Cycle)**.** Referring to the SDL encoding of the Hamiltonian Cycle problem in Example 2.1, a node is reachable if it is the starting node (base case) or if it is connected to a previously reached node (recursive step). The rules are merged into a single biconditional:

```
array[1..node_len] of var bool: reached;
constraint forall(reached_node in 1..node_len)(
 reached[reached_node] <-> ((exists(s in 1..start_len)(start_node_id[s] == reached_node))
  \/ (exists(r2 in 1..node_len, inc in 1..node_len)(
      reached[r2] /\ r2 == arc_first_id[incycle[inc]] /\
      reached_node == arc_second_id[incycle[inc]]))));
```

Positive Cycles and Rank-Based Acyclicity. For *recursive* instructions exhibiting positive loops, Clark's Completion alone admits atoms that evaluate to `true` via mutual support without a grounded base case (unfounded sets).

To mitigate this issue, the mapping introduces a *rank-based acyclicity encoding* for each positively recursive record R, enforcing a dual-replica system:

1. **Logical Equivalence** (`<->`): Clark's Completion establishes whether a fact is logically derivable, decoupled from cyclicity.
2. **Acyclicity Filter** (`->`): A companion integer array $\text{rank}_R : D_R \to \{0, \ldots, |D_R|\}$ imposes a condition designed to prevent unbounded cyclic supports by requiring a lower-ranked premise:

$$R(\mathbf{x}) \rightarrow \bigvee_{\mathbf{y}} \big(B(\mathbf{x}, \mathbf{y}) \wedge \text{rank}_R[\mathbf{x}] > \text{rank}_R[\mathbf{y}]\big).$$

The correctness argument, showing that this encoding precisely excludes unfounded sets via a finite descent argument, is given in the online appendix.

Example 6 (Acyclicity enforcement – Hamiltonian Cycle)**.** Continuing the previous example, the cyclic dependency of `Reached` necessitates the generation of the auxiliary rank array alongside the acyclicity filter:

```
array[1..node_len] of var 0..node_len: rank_reached;
constraint forall(reached_node in 1..node_len)(
  reached[reached_node] -> ((exists(s in 1..start_len)(start_node_id[s] == reached_node))
    \/ (exists(r2 in 1..node_len, inc in 1..node_len)(
        reached[r2] /\ rank_reached[reached_node] > rank_reached[r2] /\
        r2 == arc_first_id[incycle[inc]] /\ reached_node == arc_second_id[incycle[inc]]))));
```

Deny Instructions. A deny instruction of the form `deny from` **R** `where` φ is interpreted as:

$$\forall\, \mathbf{x} \in \Omega \;:\; \neg\, \varphi(\mathbf{x}),$$

mapped to a classical `not` negation inside a `forall` loop. When the condition φ contains a negated alias (`not R`), Negation as Failure is resolved via a negated existential quantifier:

$$\neg\, \exists\, \mathbf{y} \in \Omega_{\text{local}} \;:\; \varphi_{\text{local}}(\mathbf{x}, \mathbf{y}).$$

The soundness of this translation under the Domain Closure Assumption, and why finite domains make classical negation equivalent to NAF, is argued in the online appendix.

Example 7 (NAF-carrying deny – NSP)**.** The SDL constraint requiring that a recovery shift (id 4) can only happen if the nurse worked a night shift (id 3) two days prior:

```
deny from Assign as a1, not Assign as a2
   where a1.nurse.id == a2.nurse.id and a1.shift.id == 4
     and a2.shift.id == 3 and a2.day.id == a1.day.id-2;
```

The NAF atom is replaced by a negated existential quantification:

```
constraint forall(d1 in 1..day_len, n in 1..nurse_len)(
   not (shift_id[assign[d1, n]] == 4 /\
      not exists(d2 in 1..day_len)(day_id[d2] == day_id[d1]-2 /\ shift_id[assign[d2, n]] == 3)));
```

Since the set of days is finite and fully enumerated, the absence of such a d_2 is equivalent to the absence of any `Assign` atom satisfying the condition, thereby realizing Negation as Failure.

Aggregate Mapping. SDL aggregates (`count`, `sum`, `min`, `max`) are mapped to their MiniZinc counterparts, addressing structural challenges tied to the encoding of the search space.

Mapping count: Boolean Coercion. The `count` aggregate evaluates the number of times a logical condition holds over a domain. When the underlying decision variable is non-Boolean (e.g., a functional integer encoding), it cannot be mapped directly to a cardinality constraint. Instead, the mapping projects the condition into a conditional sum via the `bool2int` indicator function. For a domain I and a Boolean condition φ, the general form is:

$$\texttt{sum}(i \in I)(\texttt{bool2int}(\varphi(i)))$$

Numeric Aggregation via Indirect Lookup. When a `sum` aggregate operates on a numeric attribute of a dynamically selected record, the mapping exploits the statically flattened data arrays. The decision variable acts as a dynamic index into the corresponding constant attribute array, yielding an indirect lookup of the form:

$$\texttt{sum}(i \in I)(attr[X[i]])$$

where X is the decision variable array and *attr* is the flattened attribute array of the target record.

Mapping `min`/`max`: Array Comprehensions. Extremum aggregates evaluate the smallest or largest value of a numeric attribute across a filtered domain. These are mapped to the built-in **`min`**`()`/**`max`**`()` functions over a dynamically constructed array comprehension:

$$\texttt{min/max}([\, e(i) \mid i \in I \texttt{ where } \varphi(i) \,])$$

where $e(i)$ is the numeric expression to minimize and $\varphi(i)$ is the optional filtering condition. This structure handles both static domain extrema and dynamically filtered variables uniformly.

To prevent MiniZinc runtime errors on empty arrays, extremum aggregates are safeguarded with an existential guard (`exists(i in I)(...)` $\wedge$ **`min`**`(...)`/**`max`**`(...)`$\odot$ `val`). This ensures empty sets safely evaluate to `false`, preserving ASP semantics.

Aggregates in `define` instructions are supported only for non-recursive definitions. Recursive aggregates would require fixpoint computations over aggregated values, a known open challenge in non-monotonic semantics [22], and are left as future work.

Example 8 (Aggregate mapping – NSP hospital coverage)**.** Consider an SDL `deny` instruction enforcing a minimum number of nurses per shift per day. The `count` aggregate evaluates the number of assignments matching the current day and shift:

```
1 deny from Day, NurseLimits
2   having count { Assign.nurse from Assign
3              where Assign.shift.id == NurseLimits.shift.id
4                and Assign.day.id == Day.id } < NurseLimits.min;
```

Following the classical negation mapping for `deny` (Paragraph "Deny Instructions"), the condition is negated inside a `forall` loop. The aggregate itself is mapped using the Boolean coercion strategy for `count`:

```
1 constraint forall(d_b in 1..day_len, nl_b in 1..nurselimits_len)(
2   not ((sum(d_a in 1..day_len, n_a in 1..nurse_len)(
3     bool2int(shift_id[assign[d_a, n_a]] == nurselimits_shift_id[nl_b] /\ day_id[d_a] ==
          day_id[d_b])))
4     < nurselimits_min[nl_b]));
```

Here, the inner sum acts as a conditional filter via `bool2int`, effectively counting valid assignments against the `nurselimits_min` threshold.

Weak Constraints and Optimization. SDL soft denials (`deny ... or pay` C `at` L) are compiled into a single scalar CP objective via *lexicographic scalarization*. A static multiplier μ_L is assigned to each level L:

$$\mu_1 = 1, \qquad \mu_L = \mu_{L-1} \cdot (MaxCost_{L-1} + 1)$$

where

$$MaxCost_L = \sum_{c \in \mathscr{W}_L} \sum_{\mathbf{x} \in Dom(c)} C_c(\mathbf{x})$$

is the worst-case cumulative cost at level L, with $\mathscr{W}_L$ being the set of soft denials at level L, $Dom(c)$ their evaluation domain, and $C_c(\mathbf{x})$ the penalty cost for a given instance.

The global objective total_penalty $= \sum_L \mu_L \cdot p_L$ is minimized. The proof that this preserves strict lexicographic dominance across all levels is given in the online appendix.

The scalarization requires that total_penalty remains within the solver's integer range. For large instances, the multipliers μ_k may exceed 64-bit bounds, causing overflow; thus, the translation process must statically verify that $\mu_k \leq 2^{63} - 1$. Moreover, only non-negative penalty weights are supported, as negative values would break the ordering preserved by scalarization.

Example 9 (Optimization mapping – ORS priority scheduling)**.** Consider a soft constraint penalizing unassigned patients. The following rule applies to priority-3 registrations:

```
deny from Registration
   where Registration.priority == 3
   having count {Assign.registration
      from Assign where Assign.registration == Registration} == 0
   or pay 1 at 2;
```

A symmetric rule is defined for priority-2 registrations, with the same structure but a higher penalty weight (i.e., cost level 3 instead of 2). The two constraints are mapped as follows:

```
int: max_cost_lvl2 = sum(Registration_base in 1..registration_len where
    registration_priority[Registration_base] == 3)(1 * (1));
int: mult_lvl3 = 1 * (max_cost_lvl2 + 1);
var int: penalty_2 = (sum(r_base in 1..registration_len where registration_priority[r_base] == 3)(
   bool2int((sum(a_reg in 1..registration_len, a_mss in 1..mss_len)(
      bool2int(assign[a_reg, a_mss] /\ registration_id[a_reg] == registration_id[r_base]))) == 0)
         * 1)) * 1;
var int: penalty_1 = (sum(r_base in 1..registration_len where registration_priority[r_base] == 2)(
   bool2int((sum(a_reg in 1..registration_len, a_mss in 1..mss_len)(
      bool2int(assign[a_reg, a_mss] /\ registration_id[a_reg] == registration_id[r_base]))) == 0)
         * 1)) * mult_lvl3;
var int: total_penalty = penalty_2 + penalty_1;
solve minimize total_penalty;
```

The multiplier for level 3 is scaled by the worst-case cost of level 2, ensuring that a single level-3 violation always outweighs any accumulation of level-2 violations.

Output. The show directive restricts the visible output to a specific subset of records. For each target record R, the projected output set O_R is extracted from its corresponding decision variable X according to the encoding scheme:

- **Relational/Inclusion Encodings** (Boolean array): $O_R = \{\, \mathbf{x} \in Dom(X) \mid X[\mathbf{x}] = \texttt{true} \,\}$
- **Functional Encoding** (Integer array): $O_R = \{\, (\mathbf{x}, y) \mid \mathbf{x} \in Dom(X) \wedge y = X[\mathbf{x}] \wedge y > 0 \,\}$

Integer indices are then resolved back to their original lexical strings via the global string table.

4. Use Cases

To demonstrate the practical applicability of SDL and the robustness of the mapping, we present snippets from two real-world combinatorial optimization problems: the Nurse Scheduling Problem (NSP) and the Operating Room Scheduling (ORS) problem. These examples highlight the evaluation of complex domain constraints that were not covered in the fundamental mapping rules.

4.1. Nurse Scheduling Problem (NSP)

The NSP variant considered here (based on [7]) assigns nurses to shifts over a finite planning horizon, subject to workload constraints, shift requirements, and temporal patterns such as rest and recovery periods. The primary guess (one shift per nurse per day) maps directly to Scheme 1 (Example 2).

Example 10 (Sliding Window and NAF – NSP)**.** Consider two constraints involving temporal structure. The first constraint enforces at least two rest shifts (id 5) within any 14-day sliding window; the second forces a recovery shift (id 4) after two consecutive night shifts (id 3):

```
% Sliding window constraint
deny from Nurse, Day where Day.id <= 17
```

```
   having count {
      Assign.day from Assign
      where Assign.nurse.id == Nurse.id and Assign.shift.id == 5
        and Assign.day.id == Day.id .. Day.id+13} < 2;
% Consecutive-night recovery constraint
deny from Assign as a2, Assign as a3, not Assign as a1
   where a1.nurse.id == a2.nurse.id and a1.nurse.id == a3.nurse.id
     and a2.shift.id == 3 and a3.shift.id == 3 and a1.shift.id == 4
     and a3.day.id == a2.day.id + 1
     and a1.day.id == a3.day.id + 1;
```

The rules are mapped as shown below. The NAF over the recovery shift is consistently resolved via negated existential quantification:

```
% Sliding window constraint
constraint forall(n_base in 1..nurse_len, d_base in 1..day_len where day_id[d_base] <= 17)(
   not ((sum(a_day in 1..day_len, a_nurse in 1..nurse_len)(
      bool2int(nurse_id[a_nurse] == nurse_id[n_base] /\
            shift_id[assign[a_day, a_nurse]] == 5 /\
            day_id[a_day] in day_id[d_base]..day_id[d_base]+13)
   )) < 2));
% Consecutive-night recovery constraint
constraint forall(d1 in 1..day_len, n in 1..nurse_len, d2 in 1..day_len where day_id[d2] ==
     day_id[d1]+1)(
   not (shift_id[assign[d1, n]] == 3 /\
       shift_id[assign[d2, n]] == 3 /\
       not exists(d3 in 1..day_len)(
          day_id[d3] == day_id[d2]+1 /\ shift_id[assign[d3, n]] == 4)));
```

4.2. Operating Room Scheduling

The Operating Room Scheduling (ORS) problem [10] assigns patients to surgical sessions, accounting for room availability, time capacities, and bed constraints. The SDL specification models these through records for `Registration`, `Mss` (sessions), `Duration`, `Beds`, and `Day`, with relations `Assign`, `Stay`, and `StayICU` capturing decisions and occupancy.

The decision variable `Assign` uses Scheme 2 (Boolean matrix) as it lacks unit-cardinality restrictions(Example 3). Derived records like `Stay` are defined by non-recursive `define` instructions and encoded as Boolean biconditionals.

Example 11 (Room-capacity and bed-capacity denial – ORS)**.** The SDL constraint bounding total surgery time per room and sequence relies on summing patient durations:

```
deny from Duration
   having sum {Assign.registration.surgery_duration, Assign.registration
      from Assign
      where Assign.mss.room == Duration.room and Assign.mss.sequence ==
          Duration.sequence
   } > Duration.max_minutes;
```

translates to:

```
constraint forall(d_base in 1..duration_len)(
   not ((sum(a_reg in 1..registration_len, a_mss in 1..mss_len)(
      if assign[a_reg, a_mss] /\ mss_room[a_mss] == duration_room[d_base] /\
        mss_sequence[a_mss] == duration_sequence[d_base]
      then registration_surgery_duration[a_reg] else 0 endif))
      > duration_max_minutes[d_base]));
```

The conditional `if-then-else` pattern is seamlessly generated because the `sum` aggregate operating on the Scheme 2 Boolean-matrix encoding must structurally guard on the assignment cell evaluating to `true` before contributing the patient's numerical surgery duration.

Similarly, the bed-capacity constraint shown below natively utilizes the derived `stay` array directly within the condition:

```
constraint forall(b_base in 1..beds_len where beds_specialty[b_base] > 0)(
  not ((sum(s_reg in 1..registration_len, s_day in 1..day_len)(
    bool2int(stay[s_reg, s_day] /\ registration_specialty[s_reg] == beds_specialty[b_base] /\
        day_id[s_day] == beds_day[b_base])))
    > beds_available_beds[b_base]));
```

Hard and soft priority constraints are handled by the combination of a strong denial (priority 1) and weak-constraint penalties (priorities 2 and 3), as described in Example 9.

5. Experimental Evaluation

We present a preliminary evaluation aimed at validating that the generated MiniZinc models preserve the semantics of the original SDL specifications. Experiments were run on a Slurm cluster node equipped with an Intel Xeon Silver 4314 CPU @ 2.40GHz running Ubuntu 22.04.5 LTS, allocating 8 CPU cores and 16 GB of RAM per job. As backends, we employed clingo version 5.8.0 for ASP evaluation, and MiniZinc version 2.9.7 compiled with Chuffed version 0.13.2 [21] and CP-SAT (OR-Tools version 9.15) [20] for the CP models.

The NSP instances share 6 shift types and the same constraint structure, but vary in scale: the small instance involves 5 nurses over a 10-day horizon, while the large instance scales to 10 nurses over a 30-day horizon. The three ORS instances share the same topology (50 patients, 10 operating rooms, 5 specialties, 5-day horizon) but differ in patient composition (priority mix and surgery durations).

		clingo		Chuffed		CP-SAT	
Problem	Inst.	Time	Cost	Time	Cost	Time	Cost
NSP	Small	0.43	–	0.84	–	0.86	–
	Large	0.52	–	17.40	–	2.85	–
ORS	1	0.92	7	>300	27	6.46	7
	2	0.91	7	>300	27	6.34	7
	3	0.83	7	>300	27	6.44	7

Table 2
Solving times (secs) and optimization costs over 10 runs. NSP is a feasibility problem (no cost). ORS cost is the scalarized penalty; optimal is 7. Timeout: 300 s, in line with [7, 10].

Table 2 reports the results. All CP solutions satisfy the full set of hard SDL constraints, confirming the translation is semantically sound with respect to feasibility. For NSP, solutions differ across backends due to non-deterministic search, but all are valid. For ORS, clingo and CP-SAT consistently reach the optimal cost 7, supporting the correctness of the lexicographic scalarization. Chuffed times out on all ORS instances, returning cost 27. While Chuffed also employs clause learning, its lazy clause generation over CP propagators appears less effective on the dense Boolean encodings produced by Scheme 2 compared to the SAT-based approach of CP-SAT.

The performance gap between ASP and CP backends is expected and orthogonal to translation correctness. Future work will consider larger instances and additional backends.

6. Related Work

The question of bridging ASP and CP has received considerable attention. Lierler and Truszczynski [9] provide a unifying framework for modular inference systems capturing connections between ASP and constraint-based solving. This line of work has been further explored in integrated systems such as EZCSP [26] and in studies on the relationship between CASP and SMT [27].

Hybrid systems such as Clingcon [2] extend clingo with CP constraints, combining both paradigms within a single solver. Our work takes a complementary approach: rather than hybridizing solvers, we introduce a single high-level language (SDL) from which either paradigm can be targeted independently. This separation also opens the possibility of targeting hybrid CASP backends (e.g., Clingcon) as a future extension, allowing practitioners to exploit different solving strategies without rewriting the specification.

In previous work by some of the authors, this integration was achieved by compiling Constraint Answer Set Programs directly into FlatZinc [14], enabling CP solvers as backends for CASP. Here, we adopt a more modeling-centric perspective: SDL provides a unified high-level specification that compiles natively into structured MiniZinc, preserving full structural abstraction while supporting both ASP and CP backends.

Both the NSP and the ORS problems have been widely studied. Highly successful ASP encodings were developed in [7, 10], while complementary CP and ILP approaches have also been explored [6]. Our goal is not to introduce new solutions for these domains, but to show that a single SDL specification can abstract away the choice of paradigm and systematically derive correct encodings for both ASP and CP.

Finally, among high-level solver-independent languages, Essence [15] is the closest related work in the CP community. It provides abstract data types such as sets, multisets, and relations, which are automatically refined into constraint models via the Conjure compiler. Unlike SDL, however, Essence remains entirely within the monotonic CP framework and does not support ASP-style constructs such as `guess-and-deny` or default negation. Similarly, Picat [24] integrates logic programming with CP solving, but also operates within a classical monotonic setting. SDL differs from both by originating in the non-monotonic ASP paradigm while enabling dual compilation to ASP and CP, thereby deferring the choice of solving paradigm without modifying the specification.

7. Conclusion

We have presented a dual-target extension of the SDL compiler, enabling the translation of high-level declarative specifications into either ASP or MiniZinc from a single source. This approach effectively bridges the semantic gap between the non-monotonic, stable-model semantics of ASP and the classical satisfiability framework of CP. The key technical contributions are the mapping of non-deterministic `guess` constructs to typed CP decision variables, the rank-based acyclicity encoding for recursive definitions, the classical reformulation of negation as failure under the Domain Closure Assumption, and the lexicographic scalarization of soft constraints. Overall, this work shows that a single high-level specification can be compiled into semantically coherent encodings across fundamentally different solving paradigms, without requiring the modeler to commit in advance to a specific backend.

Future work will pursue several directions. First, we plan to provide a formal, self-contained semantics for SDL, independent of its current definition via translation to other languages. This would clarify its theoretical foundations and enable formal correctness guarantees for all supported constructs. Second, we aim to investigate extensions of SDL with higher-level constructs that can be directly mapped to global constraints in CP, thus improving the efficiency of the generated models while preserving the declarative nature of the language. Finally, we plan to explore alternative backends and hybrid approaches, including the integration of CP-style constraints within ASP solving frameworks, as well as translations toward SMT [3] and MIP [23], further broadening the applicability of SDL across different solver ecosystems.

Declaration on Generative AI

During the preparation of this work, the authors used Gemini 3.1 Pro in order to: Grammar and spelling check. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

Acknowledgments

This work was supported by the Italian Ministry of Health (MSAL) under POS projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by the Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); under PN RIC project ASVIN "Assistente Virtuale Intelligente di Negozio" (CUP B29J24000200005); by Regione Calabria under project NextGenGuides "Innovazione Tecnologica per le Guide Turistiche del Futuro tramite l'ausilio di tecnologie innovative AI e sistemi di geolocalizzazione avanzata" (CUP J39I24002040005); under project MOZART "Modelli e tecniche di mOdernizzazione di applicaZioni e data silos a supporto di clienti esterni, field force e impiegati di backoffice basati su AI GeneRativa e apprendimento auTomatico" (CUP J49I24001740005); by the LAIA lab (part of the SILA labs). Mario Alviano is a member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

References

[1] M. Alviano, C. Dodaro, I. R. Vasile, *Structured Declarative Language*, Proc. CILC 2024, CEUR Workshop Proceedings, Vol. 3733, 2024.

[2] M. Banbara, B. Kaufmann, M. Ostrowski, T. Schaub, *Clingcon: The next generation*, Theory Pract. Log. Program. 17 (2017) 408–461.

[3] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli, *Satisfiability modulo theories*, in: Handbook of Satisfiability, 2nd ed., IOS Press, 2021, pp. 1267–1329.

[4] G. Brewka, T. Eiter, M. Truszczynski, *Answer set programming at a glance*, Commun. ACM 54 (2011) 92–103.

[5] F. Ricca, L. Gallucci, R. Schindlauer, T. Dell'Armi, G. Grasso, N. Leone, *OntoDLV: An ASP-based system for enterprise ontologies*, J. Log. Comput. 19 (2009) 643–670.

[6] P. Cappanera, M. Gavanelli, M. Nonato, M. Roma, *Logic-based Benders decomposition in ASP for chronic outpatients scheduling*, Theory Pract. Log. Program. 23 (2023) 848–864.

[7] C. Dodaro, M. Maratea, *Nurse scheduling via answer set programming*, Proc. LPNMR, LNCS 10377, Springer, 2017, pp. 301–307.

[8] M. Gelfond, V. Lifschitz, *Classical negation in logic programs and disjunctive databases*, New Gener. Comput. 9 (1991) 365–385.

[9] Y. Lierler, M. Truszczynski, *On abstract modular inference systems and solvers*, Artif. Intell. 236 (2016) 65–89.

[10] C. Dodaro, G. Galatà, M. K. Khan, M. Maratea, I. Porro, *Operating room (re)scheduling with bed management via ASP*, Theory Pract. Log. Program. 22 (2022) 229–253.

[11] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, G. Tack, *MiniZinc: Towards a standard CP modelling language*, Proc. CP 2007, LNCS 4741, Springer, 2007, pp. 529–543.

[12] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, J. Fischer, *The MiniZinc challenge 2008–2013*, AI Magazine 35 (2014) 55–60.

[13] F. Rossi, P. van Beek, T. Walsh (Eds.), *Handbook of Constraint Programming*, Elsevier, 2006.

[14] T. Eiter, T. Geibinger, N. Musliu, J. Oetsch, T. Kaminski, *ASP-FZN: A Translation-Based Constraint Answer Set Solver*, Theory Pract. Log. Program. 25 (2025) 649–667.

[15] A. M. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, I. Miguel, *Essence: A constraint language for specifying combinatorial problems*, Constraints 13 (2008) 268–306.

[16] A. Van Gelder, K. A. Ross, J. S. Schlipf, *The well-founded semantics for general logic programs*, J. ACM 38 (1991) 620–650.
[17] K. R. Apt, H. A. Blair, A. Walker, *Towards a theory of declarative knowledge*, in: Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, 1988, pp. 89–148.
[18] F. Fages, *Consistency of Clark's completion and existence of stable models*, J. Methods Log. Comput. Sci. 1 (1994) 51–60.
[19] K. L. Clark, *Negation as failure*, in: Logic and Data Bases, Plenum Press, 1978, pp. 293–322.
[20] L. Perron, V. Furnon, *OR-Tools*, Google, https://developers.google.com/optimization, 2023.
[21] G. Chu, *Improving combinatorial optimization*, Ph.D. thesis, University of Melbourne, 2011.
[22] N. Pelov, M. Denecker, M. Bruynooghe, *Well-founded and stable semantics of logic programs with aggregates*, Theory Pract. Log. Program. 7 (2007) 301–353.
[23] G. L. Nemhauser, L. A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley & Sons, 1988.
[24] N.-F. Zhou, H. Kjellerstrand, J. Fruhman, *Constraint Solving and Planning with Picat*, Springer, 2015.
[25] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, *ASP-Core-2 Input Language Format*, Theory Pract. Log. Program. 20 (2020) 294–309.
[26] M. Balduccini, Y. Lierler, *Constraint answer set solver EZCSP and why integration schemas matter*, Theory Pract. Log. Program. 17 (2017) 462–515.
[27] Y. Lierler, B. Susman, *On relation between constraint answer set programming and satisfiability modulo theories*, Theory Pract. Log. Program. 17 (2017) 559–590.