Towards a Lightweight Standard Search Language

Horst Samulowitz[1], Guido Tack[2], Julien Fischer[1], Mark Wallace[3], and Peter Stuckey[1]

[1] National ICT Australia (NICTA) and University of Melbourne, Victoria, Australia
[2] Katholieke Universiteit Leuven, Belgium
[3] Monash University, Victoria, Australia

Abstract. In this paper we propose a lightweight search language for MiniZinc that allows a user to direct the search process by employing explicitly defined search templates. A challenge in providing a standard search language is to balance expressiveness with ease of implementation. Previous work mainly introduced general frameworks that facilitate user-defined search strategies at virtually arbitrary degrees of freedom (e.g., programmable search). The goal of this paper is to propose a specific set of explicit search templates that is solver-independent, does not impose high requirements on underlying constraint solvers, and enables users to construct search strategies to tackle problems successfully at a sufficiently abstract level by using predefined search templates. We believe a standard for search should start small and expand as the community of solver writers agree to implement it. The ultimate goal of this work is to define a standard set of search templates that form a language that can be employed across various constraint solving systems.

1 Introduction

Various frameworks to define search strategies in the context of Constraint Programming (CP) exist, e.g., [21], [10], [12], [22] [3], [13]. However, most existing work on search languages focuses on *programming* search, either declaratively (e.g., [13]), or in a more procedural fashion (e.g., [21], [10], [12]). While those languages give full control over search to the user, they require both the user to actually design a particular search strategy and a tight integration with the underlying constraint solving system. The design of search can often be a time consuming process and consequently it is most frequently the case that only a very limited range of possible search strategies are explored to tackle a problem at hand. Therefore, the potentially dramatic benefits stemming from tailoring search strategies for solving a class of real-world problems are often not fully exploited.

One of the challenges of comparing constraint programming systems currently is that there is no standards for modelling the problem. MiniZinc [16] to some extent overcomes this, since it is a solver-independent modelling language that is supported by a number of CP systems. Similarly XSCP 2.1 [19] is an instance description language supported by a number of solvers. But comparing solvers on the same model without considering search is a rather opaque comparison, since differences in search strategy will usually overwhelm any other differences in the systems. Hence it is important to be able to compare CP systems using the *same search strategy*. MiniZinc 1.1 [16] includes a rudimentary search language to allow this to happen, but it is far from expressive enough.

In this paper, we propose a language for *specfying* search from a rich set of predefined search templates. We will show how this makes the language much more solver-independent, and thus more suitable for defining search in a standard modelling language like MiniZinc. In order to achieve this goal we deliberately keep the complexity of the proposed search language low, so that constraint solver systems can easily support this language and will eventually result in a standard search language. The major disadvantage of complex search languages that are defined in e.g., [21] is that they impose high requirements (e.g., the support of an entire programming language for search) on the underlying constraint solvers.

As we illustrate in this paper our approach still enables users to construct reasonably complex search strategies at a negligible cost. While our approach obviously restricts the flexibility in designing a search strategy (e.g., in comparison to programmable search), it is exactly this restriction that enables us to provide users with a modular search language that allows the exploration of a vast range of searches without the need of programming every new search explicitly. The ultimate goal of this work is to define a set of search templates that become a standard for designing search strategies. Once such a standard is in place, it will also benefit the development of automated methods to configure search strategies for a given problem (e.g., based on Genetic Programming or Machine Learning).

The language we define here is a simple term language. While we use a MiniZinc [16] realization of this language for this paper, it can be adapted easily to other modelling languages such as Essence [5], Minion [6], and XCSP [19].

2 Background on Search

This section gives a brief overview on search as used in a constraint solver. The presentation mostly follows the terminology of [21]. For a more detailed discussion on search, see [21] [10].

2.1 Exploration Strategy

The exploration strategy defines the order in which to explore the nodes of a search tree. The most commonly used approach in the context of constraint solving is *Depth-First Search*. Other exploration strategies are for instance Breadth-First and Best-First search.

2.2 Branching Strategy

The branching strategy defines the shape of the search tree. At each inner node of the tree, the remaining search space is partitioned, creating branches in the tree that lead to a number of simpler problems to solve.

In constraint solvers, this choice often means selecting a particular variable, and then picking and a particular value in its domain, splitting the domain according to a domain splitting strategy. For instance, the *first fail* variable selection strategy picks the variable with the smallest domain, and a common splitting strategy is to assign the selected variable to its minimum on the left branch, and to exclude the minimum on the right branch.

Different branching strategies can lead to search trees of drastically different shape and size, and obviously also affect where solutions of the problem can be found in the tree. For many hard problems, finding a good selection strategy is of paramount importance for the efficiency of the solving process.

2.3 Limit Strategy

Limit strategies externally control the search process defined by exploration and branching strategy. One of the most frequently employed limit strategies is *restarts*, where search is repeatedly started over according to some dynamically updated measures (e.g., number of encountered failures). More complex limit strategies are for instance approaches like Limited Discrepancy Search [7].

2.4 Composition Strategy

This strategy allows different searches to be composed. For instance, *sequential search* composes multiple searches, where each individual search takes places on a subset of the variables. The searches are performed in a given, sequential order, and all searches have to succeed in order for the sequential search to succeed. Another composition strategy is *parallel search*, which runs multiple searches in parallel, and which succeeds as soon as *one* of them succeeds. Additional composition strategies are for instance approaches based on sampling (e.g., [18]).

3 A Lightweight Search Language

In this section we introduce the basic templates of the search language. These templates can be combined to create more sophisticated searches, as we show in the subsequent section. We start by defining the *Basic Search* template:

Basic Search The basic search implements a branching strategy. It is parameterized by decision variables, variable selection and domain splitting strategies. A basic search template, basic_search, can be one of the following:

```
{bool, int, set}_search(vars, variable_selection, domain_splitting)
```

where vars is an array specifying the variables of the corresponding type to be assigned (ints, bools, or sets respectively), and variable_selection and domain_splitting specify the variable selection and domain splitting strategies (which will be explained in Section 3.2). The search template dealing with float variables takes as additional argument the desired *precision* and terminates as soon as the precision is within eps:

```
    float_search(eps, vars, variable_selection, domain_splitting)
```

Note that all searches presented here are assumed to be *complete* (i.e., exhaustive search) and perform depth-first search. Search applied on the empty set of variables succeeds. Section 3.1 introduces several templates that allow to manipulate the searches introduced here.
We additionally introduce templates without the vars argument:

```
    {bool, int, set}_search_all(variable_selection, domain_splitting)
```

These templates consider all bool/integer/set variables contained in a given problem instance, including auxiliary ones introduced during model transformation. The motivation is that in empirical evaluations, branching on introduced variables (e.g., during model transformations) has often shown to be beneficial.

3.1 Exploration Strategy

The standard exploration strategy for the basic search templates presented in the previous section is complete *Depth-First* search. While other exploration strategies exist (e.g., Breadth-First search, Best-First Search) we do not choose to support other exploration strategies at this point since most constraint solving systems are based on Depth-First search.

3.2 Variable Selection and Domain Splitting Strategy

In this section we introduce a range of variable selection and domain splitting strategies that can be used to control the shape of the search tree.

Variable Selection The template variable_selection specifies how the next variable to be branched on is chosen at each choice point (see Table 1). Note that implicitly tie breaking based on input_order takes place when several variables have exactly the same score according to the used variable selection strategy. It is also possible to use the default template as variable selection strategy which results in employing the default variable selection strategy (which might differ depending on the type of the variable).

Multiple variable selection strategies can be combined using the template seq_vss. It selects a variable according to a sequence of strategies, whose relevance decreases from left to right.

```
    seq_vss([variable_selection, ..., variable_selection])
```

For instance, the following statement uses the variable selection strategy min_lb with tie breaking based on the reverse_input_order of the variables:

```
    seq_vss([min_lb, reverse_input_order])
```

While input_order is implicitly used to perform tie breaking, it is never applied in this example. The following templates on variable ordering strategies can be

Strategy	*Choose variable(s)...*
{reverse_}input_order	in the (reverse) order they appear in vars.
random_order	at random (based on uniform distribution) from vars.
{min,max}_{lb,ub}	with the smallest/largest lower/upper bound.
{min,max}_dom_size	with the smallest/largest domain.
{min,max}_degree	that appears in the smallest/largest number of constraints.
{min,max}_{lb,ub}_regret	with the smallest/largest difference between the two smallest/largest values in its domain.
{min,max}_dom_size_degree	with the smallest/largest ratio of domain size and degree.
{min,max}_dom_size_ weighted_degree	with the smallest/largest ratio of domain size and weighted degree.
{min,max}_impact	that caused the smallest/highest search space reduction in the past.
{min,max}_activity	that has minimal/maximal activity score (e.g., based on VSIDS).

Table 1: Proposed Variable Selection Strategies

used to combine and manipulate variable scores. Note that we implicitly assume that variable strategies are based on numerical scores. The first template can be used to *weigh* the score of the employed variable_selection:

```
weight_score(variable_selection, <Weight>)
```

where <Weight> denotes a float number.
The score of the used variable_selection is multiplied by the given <Weight>. The resulting score is rounded to the closest integer in order to promote further tie breaking. The second template can be used to sum multiple individual scores and the resulting score is then used to rank the variables accordingly:

```
sum_score([variable_selection, ..., variable_selection])
```

In order to use the weighted sum of the individual scores to rank the variables can then be formulated as follows:

```
Weighted Scores
sum_score([
 weight_score(<Variable_selection>, <Weight>), ...,
 weight_score(<Variable_selection>, <Weight>)])
```

Domain Splitting The template domain_splitting specifies how to split the domain of the selected variable. Due to space limitations, the proposed strategies are captured in Table 2 in the Appendix.

Of course, some domain splitting specifications may result in the same selection strategy; e.g., bisect_low and enumerate_lb in bool_search. A domain splitting strategy is applied just once, and the variable may thus not yet be fixed as a result (e.g., with domain bisection). This variable may or may not be selected immediately afterwards, depending on the specified variable selection. It is also possible to use the `default` template as domain splitting strategy, which picks a default strategy that may depend on the variable type and the actual solver that is used. In most cases domain splitting does not assign a variable to a particular value but rather to a range in its domain. Consequently, it is frequently the case that a variable is branched on multiple times (in contrast to branching in e.g., a SAT solver). When employing a non-static variable selection strategy this can result in branching on one variable being interleaved with branching on other variables. In order to completely fix a value to a variable before moving on to the next variable the following template can be used:

```
complete(domain_splitting)
```

It is also possible to provide multiple domain splitting strategies using the `dss_rand` template (which is of type domain_splitting). When specifying this template a domain splitting strategy is picked at *random* per variable from the specified set of strategies[4]. This template is in particular useful when one wants to diversify search (e.g., when sampling the search space):

```
dss_rand([domain_splitting, ...,domain_splitting])
```

3.3 Limit Strategy

Limited Search The template *Limited Search* bounds the search process by measures like time (in seconds), number of failures or number of explored search nodes. Limits may be nested; a search is always constrained by the tightest limit it is scoped by. The *limit* template is also of type basic_search.

```
Limited Search
limit_search(<Measure>, <Limit>, <SearchT>)
```

Measure can take one of the following values:

`fails`	#conflicts	`solutions`	#solutions
`nodes`	#search nodes	`time`	Time in seconds

While limits can be defined in a straightforward way in a non-nested context, the semantics of limits becomes more intriguing when used in more complex search templates like *sequential/parallel search*. We consider the following example to illustrate the semantics of limits in those contexts:

```
Example 1: Nested Limits in Sequential Search
% Overall time limit of 20 seconds
limit_search(time, 20,
 seq_search([
  % Time limit of 5 seconds on the search of variables X
  limit_search(time, 5, search(X, min_degree, bisect_low)),
  % Time limit of 1 second on the search of variables Y
  limit_search(time, 1, search(Y, min_regret, enumerate_lb))]))
```

[4] One could also consider correlating the type of variable (e.g., based on domain size) with particular domain splitting strategies.

The first limit (20 seconds) specifies that the entire duration of the scoped sequential search cannot exceed 20 seconds. The first search on the variables in X within the sequential search template is initialized with a time budget of 5 seconds. If this search fails to find an assignment to all the variables in X within 5 seconds, the sequential search template fails. If the search is able to determine an assignment within 5 seconds (let us say 2 seconds), the sequential search moves on to the search on the variables in Y with a time budget of 1 second. If the search fails to find an assignment within this time budget, the search fails and sequential search backtracks to the search on the variables in X. The remaining time budget for the search on X is now $5 - 2 = 3$ seconds. If the search on X is able to determine an assignment within this time budget, the search on Y is assigned again the full time budget of 1 second (if this does not exceed the outermost limit of 20 seconds).

In general, each search has an allocated time budget. When entering a search it is granted the minimum of the time limits it is scoped by. When backtracking to a search, it is granted the minimal time remaining from the initial budget reduced by the time used already by the search itself since the initial call or the minimal time budget imposed by the limits scoping this search. Exactly the same holds for different measures like node counts or failure counts.

For parallel search, limits are to some extent harder to deal with than with sequential search since one needs to maintain the reduction of the time budget caused by each of the different searches in parallel. However, no interaction between searches within a parallel search takes place with regards to limits.

Restarting Search Any search immediately scoped by a restart template backtracks to the root of the search tree each time the restart condition is hit. As with *Limited Search*, restarts can be defined on several measures (e.g., number of failures). Note that in contrast to limits, restarts can appear nested but are not scoped by other restart templates. The restart template has the format:

```
Restarting Search
restart_<Type>(<Type-Specific-Parameters>, <SearchT>)
```

The following two types of restart strategies are currently supported: *Geometric* and *Luby* based restarts:

```
Geometrically Restarting Search
restart_geometric(<Increment>, <InitLimit>, <Measure>, <SearchT>)
```

```
Luby Restarting Search
restart_luby(<InitValue>, <MaxValue>, <Measure>, <SearchT>)
```

Search Once This template, `once(<SearchT>)`, results in performing a given search exactly once, removing possible choice points.

The following example shows how to define a standard search on the variables X, then how to fix the variables Y (and not to perform backtracking on variables in Y) using *once*, and then to search on variables Z. When search on Z fails, seq_search (see subsequent section) backtracks directly over all variables in Y to the search on X:

Example 2: Search Once
```
seq_search([
    % Search that solves part of the problem involving X
    int_search(X, min_degree,  bisect_low)
    % Assign value (here: lower bound) to remaining variables Y
    once(int_search(Y, min_dom_size, min_lb)),
    % Search that solves part of the problem involving Z
    int_search(Z, min_degree,  bisect_low)]).
```

If one wants to assign all free variables to a value according to an underlying search template without any backtracking, this can be modeled like this:

Assign Once
```
once( limit(failures, 0, int_search(X, min_degree, bisect_low)))
```

In this example, search assigns values to variables according to the given selection strategies. When one assignment fails, the entire search fails.

Limited Discrepancy Search This template results in restricting the scoped search to the specified discrepancy (e.g., only 2 right branches per search) [7].

```
    lds(<Discrepancy>, <SearchT>)
```

The following example shows how to use the `lds` search template. The search scoped by the `lds` template is limited to a discrepancy of at most 1:

Example 3: Limited Discrepancy Search
```
lds(1, int_search(X, min_degree,  bisect_low))
```

3.4 Composition Strategy

Sequential Search The template *Sequential Search* defines multiple searches on variable subsets $X_1, X_2, \ldots, X_n$ of the decision variables X in the given problem instance in a particular order. Typically, $X_1 \cup X_2 ... \cup X_n = X$. The searches are executed in the specified order. If a search fails, sequential search backtracks to the previous search in the sequence. If the first search in the sequence fails, sequential search fails. It succeeds when all searches succeed.

Sequential Search
```
seq_search([<SearchT(X1)>, <SearchT(X2)>, ... <SearchT(Xn)>])
```

As an example consider the following sequential search template:

Example 4: Sequential Search
```
seq_search([
   search(x, min_dom_size_weighted_degree, enumerate_lb),
   search(y, min_dom_size, bisect_low)])
```

First search takes places on the variables in x and once all variables contained in x have been assigned, the search on y is invoked. If the search on y fails, then the sequential search template backtracks to the previous search on x in order to determine a new setting of the variables in x. If there are no more feasible assignments left among the variables in x, sequential search terminates with failure. In order to support a *portfolio* approach to search this template exists:

```
Sequential Or Search
seq_or_search([<SearchT(X)>, <SearchT(Y)>, ... <SearchT(Z)>])
```

In contrast to `seq_search` this template succeeds if *one* of the specified searches succeeds. The searches are executed in the order specified in the sequence. This allows to tackle a problem with several searches in an preferred order.

Parallel Search The searches specified inside the *Parallel Search* template are invoked in parallel, and the search terminates as soon as one of the specified searches is able to determine an answer to the given problem. If all searches fail, the parallel search template fails. Note that all searches inside the *Parallel Search* template are independent of each other and no backtracking between searches as in sequential search takes place. Here, we do not aim at distributed CP solving and for now we simply assume that parallel searches on the same problem benefit from diversification (e.g., different variable selection strategies).

```
Parallel Search
par_search([<SearchT>, <SearchT>, ..., <SearchT>])
```

Sample Search The *Sample Search* template allows information to be passed to variable selection strategies[5] from one search to a subsequent search. The aim of this template is to enable the collection of information relevant to a variable ordering heuristic in order to support the concept of *warm-starts* (see e.g., [11]).

The sample search template takes a variable_selection, and the corresponding information is collected during the first search, which should be some form of limited search in practice. Note that variable selection heuristics that depend on static information only are not a sensible choice (e.g., `input_order`).

```
Sample Search
sample_search(<VarChoiceT>, <SearchT>, <SearchT>)
```

The following is an example for an application of the sample search template:

```
Example 5: Sample Search
sample_search(impact,
 % Sample for 5 sec. to collect information on impact scores
 limit_search(time, 5,
   restart_geometric(1.5, 2000.0, nodes,
     int_search(x, min_lb, dss_rand([bisect_low, bisect_high]))))
 % Initially use pre-initalized ranking of the variables
 int_search(x, seq_vss([impact, min_dom_size]), enumerate_lb))
```

[5] One could also consider sampling for domain splitting strategies or both.

This sample search first performs a search limited by 5 seconds that makes use of restarts and branches on variables in x. During this first search information relevant to the `impact` heuristic is gathered (e.g., search space reduction per variable). The second search accesses this information to perform its initial branching decisions. Note, however that this information is continued to be updated as in the regular `impact` heuristic.

In combination with MiniZinc's array comprehension facilities, fairly sophisticated searches can be programmed. For example, consider the following search for a radiation treatment planning problem:

Example 6: Radiation Search

```
seq_search([
 int_search(N, min_dom_size_weighted_degree, bisect_low),
 seq_search([
  once(int_search([ Q[i,j,b] | j in Columns, b in BTimes],
        max_activity, bisect_activity_min)) | i in Rows])])
```

The search first sets the pattern variables N using a dynamic variable selection strategy. Once they are fixed, each row i in the problem is independent, and therefore failure on one row must be caused by the search on the variables in N, and no backtracking into other rows is necessary. The search then examines each row in turn using a different dynamic search strategy to set the intensity variables $Q[i, j, b]$ for one row at a time.

Backdoor Search Informally, the backdoor search template tries to determine a subset of variables that simplify the problem in an essential fashion once they are fixed to a range in their domain. For a more detailed discussion on backdoors please refer to e.g., [24]. In order to determine such variables, we here perform a limited search to collect information on the variables in the given problem instance. As a result, the variables are divided into *backdoor* variables and *remaining* variables, which are then searched sequentially (see Section 3.4).

In the following specification of backdoor search, X denotes a subset of the problem variables, and `SearchT(X)` denotes a normal search template. After the limited search `SearchT(X)` has finished, the template `backdoor_vars` is replaced with the highest ranked variables in X that fall within the given *ratio* according to the used variable ordering heuristic, while all other variables in X are collected in `remaining_vars`. Then, a sequential search is performed, first on the backdoor variables, then on the remaining variables.

Backdoor Search

```
backdoor_search(<VarChoiceAnn>, <Ratio>,
  limit_search(<Measure>, <Limit>, <SearchT(X)>),
    seq_search([
      <SearchT(backdoor_vars)>, <SearchT(remaining_vars)>]))
```

The following example shows an application of the backdoor template. A first search limited by time is used to gather variable scores according to a variable selection strategy. Then a sequential search is invoked where the top ranked variables according to the scores of the selection strategy are searched on first, and then all remaining variables are searched on in the second search:

Example 7: Backdoor Search
```
% Search that considers 10% of the variables as backdoors
backdoor_search(dom_min_weighted_degree, 0.1,
   % Perform search limited by 10 seconds
   limit_search(time, 10,
       restart_geometric(1.05, 200.0, nodes,
          search(x, min_dom_size_weighted_degree, bisect_low))),
   % backdoor_vars: top 10% ranked dom_min_weighted_degree
   seq_search([
        search(backdoor_vars, min_dom_size, bisect_low)
        search(remaining_vars, input_order, enumerate_lb)]))
```

4 Selecting Search Templates

As mentioned previously we want to propose a specific set of search templates that are instances of more general frameworks for search strategies. Obviously it is important which explicit search strategies to support, since the selected set of templates defines the space of search strategies available to the users. This section tries to motivate why we selected the strategies presented in the previous sections. While we tried to cover the most important search strategies in this first proposal, the selection of search templates is neither complete nor final. As mentioned earlier, in our opinion a standard search language should start small and be extended over time by the community. Having said that, we do not actually expect nor require a constraint solving system to support all features supported in the language (e.g., complex searches like backdoor search).

4.1 Exploration Strategies

Besides Depth-First other search exploration strategies such as Breadth-First and Best-First search are successfully employed in a diverse area of applications (e.g., [4]). We did not consider other exploration strategies here yet since they impose additional requirements on the underlying solver. A number of constraint solvers (e.g., *Eclipse* [22]) are intrinsically based on Depth-First search, and therefore supporting other exploration strategies like Best-First search requires a substantial effort by the solver developers.

4.2 Variable Selection and Domain Splitting Strategies

Most variable selection strategies that we consider here are commonly used in constraint solvers. Lexicographic ordering and selection strategies based on variable domain size and degree are among the most basic variable orderings. The *domain weighted degree* heuristic [23] is a more sophisticated strategy, but still available in many solvers. Impact [17] is a strategy successfully employed by ILOG. Activity-based heuristics are mainly motivated by SAT (e.g., [15]).

Similarly, most considered domain splitting strategies are commonly used in constraint solvers. All strategies that make use of randomization are mainly introduced to enable diversification and are particularly useful for sampling a search space. The weighted combination of variable scores to form a ranking has also shown to be beneficial (e.g., [1]).

4.3 Limit and Composition Strategies

The concept of restarts is successfully employed in various areas (e.g., [24]) and mainly aims at diversifying search as well as redirecting search based on gathered knowledge during previous searches. The limit constructs enable the user to control the search process in terms of various measures (e.g., runtime). Since limits can also be nested they enable a fine grained control on the search process. In addition to the basic composition strategies the *sampling search* construct is mainly motivated by the concept of *warm-starts* (see e.g., [11]). The *backdoor search* construct is motivated by work presented in [24] and the general intuition that a problem can contain a core that when solved simplifies the remaining parts of the problem in an essential way.

5 Requirements on the Underlying Solver

This section briefly outlines some of the requirements that the defined search templates impose on the underlying constraint solver.

5.1 Exploration, Variable and Value Selection Strategies

The only required exploration strategy, Depth-First search, is supported by all constraint solvers. In order to implement the proposed variable selection/domain splitting strategies, the underlying solver needs to support a range of statistics and measures like the following:

1. Variable Domain Size
2. Variable Regret
3. Variable Degree
4. Variable/Domain Value Impact [17]
5. Constraint Failures
6. Variable Activity
7. Current Decision Level
8. Average Search Depth

From those basic measures nearly all other variable and value ordering heuristics can be computed by combining them using basic mathematical expressions (e.g., domain weighted degree [23] is composed of (1), (3), and (5)).

5.2 Limit and Composition Strategies

Since we do allow nested limited searches (e.g., imposing nested time constraints), some bookkeeping facilities have to be supported in order to be able to calculate the appropriate budgets left for a particular search. Search templates such as *sequential search* and *backdoor* search impose additional requirements on the solver. While most requirements can be dealt with by exploiting the accessibility to the measures mentioned in the previous section (e.g., for computing a backdoor score of a variable), constructs like sequential/parallel search also require some additional interfaces to the underlying solver. For instance, the different semantics resulting from introducing sequential search or limited search (e.g., incompleteness) have to be supported.

6 Comparison to Existing Search Languages

The main difference between the work presented here and most existing search languages is the level of abstraction. The focus of most related publications is freely programmable search, while our goal is a library of predefined search templates. Programmable search is, in our opinion, too expressive to become a solver-independent standard. Our approach drastically limits expressivity, it is aimed at being easy to implement in a wide range of constraint solvers.

The work presented here is closely related to the search specifications in OPL, as introduced in [21]. However, OPL provides a general framework to define search strategies that can be used to build arbitrary searches. In other words, the search language presented here is rather a particular instance of the general framework for search available in OPL. The IBM ILOG Script for OPL [9] appears to be more along the lines of the work presented here, but from the documentation provided online, it only seems to support a very limited set of templates that control search. There are variations on the general theme of programmable search as found in OPL. For instance, to some extent ZINC [12] features programmable search similar to how it is done in OPL, and SALSA [10] is an algebraic search language that is mainly concerned with describing search trees. The work presented in [13] introduces a declarative search language. The *Tools* library introduced in [3] supports the design of complex tree search algorithms in the context of hybrid search methods in CP. Localizer [14] is a system for Local Search and also provides users with control over the search process. While basic variable selection and domain splitting strategies are available, more complex search strategies are mainly supported by programmable search. Its successor Comet [8] also supports CP. The platform for constraint programming *Eclipse* [22] provides users with a range of search templates besides the possibility of programmable search. However, Eclipse only seems to cover a subset of the search templates proposed here in a solver dependent fashion.

7 Conclusions and Future Work

We proposed a lightweight search language that defines search templates that can be used to guide and control search. The combination of different search templates enables users to model more complex search strategies. We proposed a set of search templates that are solver independent and that can easily be supported across various different constraint solvers. Clearly, the set of search templates is neither complete nor final, and one aim of this work is to engage in a discussion on a standard for a search language with the community. We have already implemented the majority of the templates proposed here in the context of MiniZinc. In the future we plan to compare the performance achieved by the search specified in our language to the performance of the same search defined in the constraint solving system itself. Furthermore we are also working on an implementation of the search language based on the work presented in [20]. This approach will also provide an abstract implementation of the search language that is likely to aid constraint solver developers that want to support the language in their system.

Although the proposed search language is kept at a simple level, a few issues remain regarding its implementation. For instance, one aim of a standard search language is to achieve settings that allow a fair comparison of different constraint solvers. However, due to slightly different implementations or programming languages standards even rounding/floating point errors can cause search to behave very differently.

Local Search is often a standard component in constraint solving system (e.g., VLNS [2]), but it is not yet considered in the search language proposed here. This is mainly due to the fact that the concepts underlying search here are mainly based on systematic approaches in the sense that search starts at the root node of a search tree and is then expanded in some order towards leaf nodes. Local Search does not follow this pattern and therefore an integration of Local Search requires conceptually a slightly different angle to the control of search. However, it is feasible to design Local Search templates within our language. In addition, developing search templates that also consider hybrid solvers (e.g., based on [3]) and support more complex interactions between for instance parallel searches (e.g., sharing of newly inferred constraints) are desirable as well.

References

1. Tobias Achterberg and Timo Berthold. Hybrid branching. In *CPAIOR09*, pages 309–311, Berlin, Heidelberg, 2009. Springer-Verlag.
2. Ravindra K. Ahuja, Özlem Ergun, James B. Orlin, and Abraham P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Appl. Math.*, 2002.
3. Simon de Givry and Laurent Jeannin. A unified framework for partial and hybrid search methods in constraint programming. *Computers and Operations Research Archive*, 33:2805–2833, 2006.
4. R.E. Fikes and N.J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. Technical Report 43r, SRI International, 1971. SRI Project 8259.
5. Alan M. Frisch, Warwick Harvey, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.
6. Ian P. Gent, Chris Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *In: Proceedings of ECAI 2006*, pages 98–102. IOS Press, 2006.

7. William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *IJCAI'95*, pages 607–613, 1995.
8. Pascal Hentenryck and Laurent Michel. Control abstractions for local search. *Constraints*, 10(2):137–157, 2005.
9. IBM ILOG. IBM ILOG Script for OPL Manual, 2010.
10. François Laburthe and Yves Caseau. Salsa: A language for search algorithms. *Constraints*, 7:255–288, 2002.
11. Jacek Gondzio Marco Colombo and Andreas Grothey. A warm-start approach for large-scale stochastic linear programs. *Mathematical Programming*, 2009.
12. Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the zinc modelling language. *Constraints*, 13:229–267, 2008.
13. Julien Martin, Thierry Martinez, and Francois Fages. On the specification of search tree ordering heuristics by pattern matching in a rule-based modeling language. In *ModRef 08*, 2008.
14. Laurent Michel and Pascal Van Hentenryck. Localizer: a modeling language for local search. *INFORMS J. on Computing*, 11(1):1–14, 1999.
15. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th conference on Design automation*, pages 530–535, 2001.
16. N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 529–543. Springer-Verlag, 2007.
17. Philippe Refalo. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming CP 2004*, 2004.
18. Jean Charles Regin. Solving the maximum clique problem with constraint programming. In *CPAIOR*, 2003.
19. Olivier Roussel and Christophe Lecoutre. XML representation of constraint networks: Format XCSP 2.1. *CoRR*, 2009.
20. Tom Schrijvers, Peter Stuckey, and Philip Wadler. Monadic constraint programming. *J. Funct. Program.*, 2009.
21. Pascal Van Hentenryck, Laurent Perron, and Jean-François Puget. Search and strategies in OPL. *ACM Trans. Comput. Logic*, 1(2):285–320, 2000.
22. M. Wallace, S. Novello, and J. Schimpf. Eclipse - a platform for constraint programming. *ICL Systems Journal*, 12:159–200, 1997.
23. Richard J. Wallace and Diarmuid Grimes. Experimental studies of variable selection strategies based on constraint weights. *Journal of Algorithms*, 63:114–129, 2008.
24. Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In *IJCAI 03*, pages 1173–1178. Morgan Kaufmann, 2003.

`{assign,exclude}_{lb,ub}`	Assign/exclude the smallest/largest value in the variable's domain: $x = \min\{d(x)\} \swarrow\searrow x \neq \min\{d(x)\}$ or $x = \max\{d(x)\} \swarrow\searrow x \neq \max\{d(x)\}$ or $x \neq \min\{d(x)\} \swarrow\searrow x = \min\{d(x)\}$ or $x \neq \max\{d(x)\} \swarrow\searrow x = \max\{d(x)\}$, resp.
`{assign,exclude}_mean`	Assign/exclude the mean value in a variable's domain. $x = m \swarrow\searrow x \neq m$ or $x \neq m \swarrow\searrow x = m$, resp., where $m = \left\lfloor \frac{\min\{d(x)\}+\max\{d(x)\}}{2} \right\rfloor$
`{assign,exclude}_median`	Assign/exclude the median value in a variable's domain. $x = m \swarrow\searrow x \neq m$ or $x \neq m \swarrow\searrow x = m$, resp., where $i = \left\lfloor \frac{\|d(x)\|+1}{2} \right\rfloor$, $d(x) = \{a_1, ..., a_n\}$ with $a_1 < a_2 < \ldots a_n$, and $m = a_i$.
`{assign,exclude}_random`	Assign/exclude a value uniformly[6] sampled from the variable's domain. $x = r \swarrow\searrow x \neq r$ or $x \neq r \swarrow\searrow x = r$, resp., where $r = \mathit{random}\{d(x)\}$
`{assign,exclude}_impact_{min,max}`	Assign/exclude the value in the variable's domain with the minimal/maximal impact score.
`{assign,exclude}_activity_{min,max}`	Assign/exclude the value in the variable's domain with the minimal/maximal activity score (e.g., based on VSIDS).
`{include,exclude}_{min,max}`	Include/exclude the smallest/largest uncertain domain value of a set variable: $x = d_s(x) \cup \{d\} \swarrow\searrow x = d_s(x) - \{d\}$ or $x = d_s(x) - \{d\} \swarrow\searrow x = d_s(x) \cup \{d\}$, resp., where $d = \min\{\max\{d_s(x)\} \setminus \min\{d_s(x)\}\}$ or $d = \max\{\max\{d_s(x)\} \setminus \min\{d_s(x)\}\}$, resp.
`enumerate_{lb,ub}`	Enumerate values from the smallest to the largest/from the largest to the smallest in the variable's domain. $x = a_1 \downarrow x = a_2 \downarrow \ldots \downarrow x = a_n$ or $x = a_n \downarrow x = a_{n-1} \downarrow \ldots \downarrow x = a_1$, resp., where $d(x) = \{a_1, ..., a_n\}$ with $a_1 < \ldots < a_n$
`bisect_{low,high}`	Bisect the variable's domain, excluding the upper/lower half first. $x \leq a \swarrow\searrow x > a$ or $x \geq a \swarrow\searrow x < a$, resp., where $a = \left\lfloor \frac{\min\{d(x)\}+\max\{d(x)\}}{2} \right\rfloor$, resp.
`bisect_median_{low,high}`	Bisect the variable's domain based on the median, excluding the upper/lower half first. $x \leq m \swarrow\searrow x > m$ or $x \geq m \swarrow\searrow x < m$, resp., where $i = \left\lfloor \frac{\|d(x)\|+1}{2} \right\rfloor$, $d(x) = \{a_1, ..., a_n\}$ with $a_1 < \ldots < a_n$ and $m = a_i$.
`bisect_random_{low,high}`	Bisect the variable's domain by splitting on a value selected at random excluding upper/lower half first. $x \leq r \swarrow\searrow x > r$ or $x \geq r \swarrow\searrow x < r$, resp., where $r = \mathit{random}\{d(x)\}$
`bisect_interval_{low,high}`	If the variable's domain consists of several intervals, split the domain into the interval containing the smallest/largest values and the remaining intervals. Otherwise perform `bisect_low`/`bisect_high` on the variable's domain. $x \in [a,b] \swarrow\searrow x \notin [a,b]$ or $x \notin [a,b] \swarrow\searrow x \in [a,b]$, resp., where $[a,b] \subset d(x)$ and $\exists.d \in [a,b]$ s.t. $d = \mathit{min}(d(x))$ or $d = \mathit{max}(d(x))$ resp.
`bisect_impact_{min,max}`	Bisect the variable's domain, excluding the half with minimal/maximal impact score first.
`bisect_activity_{min,max}`	Bisect the variable's domain, excluding the half with minimal/maximal activity score first.

Table 2: Proposed Domain Splitting Strategies