HH-DSL: A Domain Specific Language for Selection Hyper-heuristics

Hilal Kevser Cora
Institute of Science and Technology, Istanbul Technical University, Turkey
subasihi@itu.edu.tr

H. Turgut Uyar
Department of Computer Engineering, Istanbul Technical University, Turkey
uyar@itu.edu.tr

A. Şima Etaner-Uyar
Department of Computer Engineering, Istanbul Technical University, Turkey
etaner@itu.edu.tr

ABSTRACT

A domain specific language (DSL) is a programming language which provides a natural notation and suitable data structures to express solutions to problems of a targeted domain. Although using a general purpose programming language together with a special library for the domain is common practice, it still requires a considerable amount of programming knowledge, making it hard for domain experts who might have limited or no programming skills. In the CHeSC (Cross-domain Heuristic Search Challenge) competition, researchers and practitioners from different research fields use the HyFlex platform to develop hyper-heuristics. The domain specific language proposed in this study aims to help these researchers to focus on hyper-heuristic development rather than the details of Java programming.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search —*Heuristic methods*

General Terms

Languages

Keywords

Hyper-heuristics, Domain Specific Languages

1. INTRODUCTION

A domain specific language (DSL) is a programming language which aims to make it easier to develop programs in a certain domain [17]. A well designed DSL provides a more natural notation and more suitable data structures to express solutions to problems of the targeted domain as opposed to a general purpose language (GPL). Although using a GPL in combination with a library or package for the domain is a very common approach, it still assumes a considerable amount of programming knowledge, therefore making it harder to be productive for domain experts who might have limited or no programming skills. A study for comparing DSLs and GPLs is given in [12] where conducted experiments indicate that learning, understanding and developing programs in a domain using a DSL designed for that domain is easier than doing the same using a GPL.

The development of a DSL is a difficult task because it requires expertise both in the targeted domain and in programming language design and implementation. The phases of development are outlined as decision, analysis, design and implementation [13]. First, it has to be decided if a new DSL is really needed instead of using existing languages and tools. The advantages of domain related notation and data structures have to be evaluated. In the analysis phase, domain knowledge is gathered from documents, domain experts and existing program codes developed for the domain. Next, the DSL is designed in light of the design patterns which emerged in the analysis phase. And finally, the DSL is implemented by developing a language processor such as an interpreter or a compiler or by embedding it in a GPL.

There are many studies about DSL design and implementation. In [11], a DSL for developing data acquisition systems is proposed. Using this DSL, domain experts can design their own data acquisition systems without any knowledge of programming. As another example, the EasyTime DSL is used to measure time in sports competitions [8]. It offers a flexible timing system which can be adapted to different types of competitions with small changes. Experiments showed that using this DSL eliminated the need for hiring specialized companies to measure time.

Since developing a new DSL is a complex process, it is recommended to use language development tools or frameworks [13]. Therefore, beside the research on developing DSLs, there are also many studies on developing frameworks to generate and implement DSLs. For example, the Delite compiler framework and runtime aims to simplify the development of parallel, embedded DSLs [15]. In this study, the Spoofax language development framework is used to design and implement the proposed DSL.

Hyper-heuristics are high-level methodologies which were proposed as an alternative to heuristics tailored to a specific problem domain [3]. There are two types of hyper-heuristics in literature: heuristics to choose heuristics (selection hyper-heuristics) and heuristics to generate heuristics [4]. In this study, we work with only the first type of hyper-heuristics.

GECCO'13 Companion, July 6-10, 2013, Amsterdam, The Netherlands.

Therefore, in this paper, the term hyper-heuristics refers to selection hyper-heuristics.

A hyper-heuristic selects and applies a low-level heuristic without using any problem specific information [16]. Low-level heuristics are heuristics tailored to a specific problem domain. Therefore, low-level heuristics operate on the problem search space, whereas hyper-heuristics operate on the low-level heuristics search space (Fig. 1).

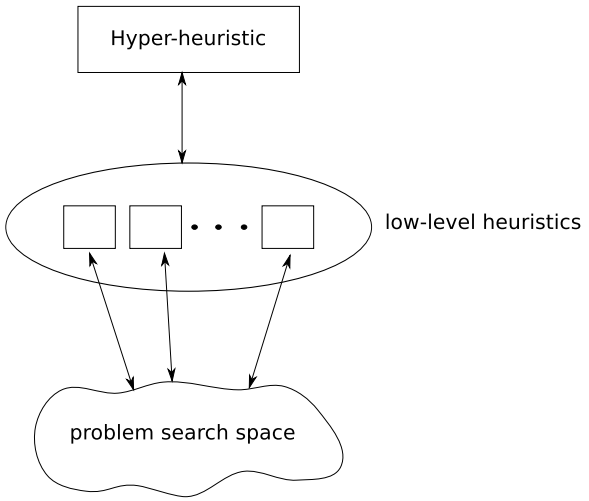

Figure 1: A hyper-heuristic communicates with the low-level heuristics, while they in turn communicate with the underlying problem implementation.

Over the years, research on hyper-heuristics has shown an increase [3, 4, 5]. Recently, HyFlex (Hyper-heuristics Flexible framework) [14], a hyper-heuristics framework which provides easy implementation of hyper-heuristics and allows comparisons between them [2] has been implemented. HyFlex can be used with different hyper-heuristics and different problem domains. In the CHeSC[1] (Cross-domain Heuristic Search Challenge) competitions, the HyFlex framework is used as the underlying system for hyper-heuristic development.

CHeSC competitions aim to bring together researchers and practitioners from operational research and computer science fields to develop more general approaches to problem solving across many domains. Since researchers and practitioners come from different fields, programming skills may be a drawback for some. To write hyper-heuristic code working on HyFlex, one needs to have good Java programming skills. To alleviate this requirement and allow the researchers to focus on hyper-heuristic development rather than dealing with the details of Java programming, we decided to develop a DSL for hyper-heuristic research.

The proposed DSL is called HH-DSL and in particular, it aims to generate code which can be executed on HyFlex. An example HyFlex Java code and its corresponding HH-DSL code are shown in Figure 2 and Figure 3.

2. HYPER-HEURISTICS AND HYFLEX

Hyper-heuristics are composed of two components: heuristic selection schemes and acceptance criteria. Heuristic selection schemes deal with deciding which low-level heuristic to choose and apply to the current candidate solution. Acceptance criteria are used to determine whether to accept or reject the new candidate solution created as a result of applying the selected low-level heuristic. A general outline of a hyper-heuristic program is given in Figure 4.

The initial candidate solution s is usually generated randomly. Through a selection scheme, one of the low-level heuristics h is chosen. Then, the chosen low-level heuristic h is applied to the current candidate solution s, generating a new candidate solution s'. Based on the acceptance mechanism, the new candidate solution s' may be accepted, replacing the current candidate solution s. This process con-

[1]http://www.asap.cs.nott.ac.uk/external/chesc2011

```
import AbstractClasses.HyperHeuristic;
import AbstractClasses.ProblemDomain;
public class ExampleHyperHeuristic1
    extends HyperHeuristic {
    public ExampleHyperHeuristic1(long seed){
        super(seed);
    }
    public void solve(ProblemDomain problem) {
        int numOfHeuristics =
            problem.getNumberOfHeuristics();
        double currObjVal =
            Double.POSITIVE_INFINITY;
        problem.initialiseSolution(0);
        while (!hasTimeExpired()) {
            int h = rng.nextInt(numOfHeuristics);
            double newObjVal =
                problem.applyHeuristic(h, 0, 1);
            double delta = currObjVal - newObjVal;
            if (delta > 0) {
                problem.copySolution(1, 0);
                currObjVal = newObjVal;
            } else {
              if (rng.nextBoolean()) {
                  problem.copySolution(1, 0);
                  currObjVal = newObjVal;
                }
            }
        }
    }
}
```

Figure 2: Example hyper-heuristic code in Java.

```
h1 = selection simple_random
acc = acceptance nonimproving_with_probability 0.5
initialize solution

loop 100%{
    h = next h1
    s = first solution
    n = apply h1 s
    check acc n replace s
}
```

Figure 3: Example hyper-heuristic code in HH-DSL.

tinues until some stopping criteria are satisfied. The final candidate s is returned as the solution found by the hyper-heuristic.

```
generate initial candidate solution s
while stopping criteria not satisfied do
{
    select low-level heuristic h
    apply h to s generating s'
    check whether s' will be accepted
    if accepted
        s = s'
}
return s
```

Figure 4: Pseudo-code for a general hyper-heuristic framework.

In [6], several heuristic selection schemes are presented.

- Greedy (GR) applies all low-level heuristics separately to the candidate solution and selects the one which provides the largest improvement.
- Simple Random (SR) chooses a low-level heuristic randomly and applies it.
- Random Descent (RD) is the same as SR, but the selected heuristic is applied repeatedly until there is no improvement.
- Random Permutation (RP) creates a random permutation of all low-level heuristics and applies them in the order given by the permutation. Each low-level heuristic is applied only once.
- Random Permutation Descent (RPD) is the same as RP, but each heuristic is applied repeatedly until there is no improvement.
- The choice function heuristic selection (CF) and the reinforcement learning heuristic selection (RL) both use a scoring approach based on the historical performance of the low-level heuristics and select the one with the highest score at each iteration.

Move acceptance strategies can be deterministic or non-deterministic [3].

- All Moves (AM) directly accepts the new solution candidate.
- Only Improving (OI) accepts only those solution candidates that provide an improvement.
- Improving and Equal (IE) accepts the solution candidates which are as good as or better than the current solution candidate.

There are also more complex acceptance schemes in literature, such as Monte Carlo, Simulated Annealing and Great Deluge acceptance methods [5].

HyFlex, the hyper-heuristics framework, provides easy implementation of hyper-heuristics and allows comparisons between them [2] on various problem domains. In this study, the proposed HH-DSL is designed to work with HyFlex version 2012. Any problem domain can be implemented in HyFlex. This implies that the low-level heuristics for that domain are also included along with the problem definition. Hyper-heuristics to work with these implemented problem domains can be developed on HyFlex. In HyFlex version 2012, six test problem domains are included as minimization problems. These are One Dimensional Bin Packing, Permutation Flow Shop, Personnel Scheduling, Boolean Satisfiability, Traveling Salesman and Vehicle Routing problems. Each problem domain contains a set of low-level heuristics which can be grouped under four categories as mutational heuristics, ruin-recreate heuristics, local search heuristics and crossover heuristics. Not all problem domains have all four types of low-level heuristics. These low-level heuristics work on the problem search space while the implemented hyper-heuristic controls these low-level heuristics and conducts a search on the heuristics search space through HyFlex.

3. THE LANGUAGE DEVELOPMENT FRAMEWORK

Domain specific languages can be designed as internal or external languages. Internal DSLs are integrated into an existing general purpose language and use the features of the original language [18]. However, internal DSLs might be complex for non-programmers because they still require the knowledge of host language syntax and semantics. Non-programmers might feel more confident when using external DSLs where the DSL is an independent language which has its own constructs. An advantage of internal DSLs is that they can use the tools -such as editors and debuggers- for the host language, whereas for external DSLs these tools need to be implemented at an extra cost. In recent years, language development frameworks have emerged that can, in addition to language implementation, also provide basic tool support for the generated DSL [18].

In this study, the Spoofax language development framework[2] is used [10]. When using this framework, two aspects of the DSL have to be expressed: the grammar and the code generation rules. From these specifications, Spoofax generates the necessary components like the parser and the code generator. It also prepares an environment which supports assistive features like syntax highlighting for the target DSL.

The grammar of the DSL is defined using the Syntax Definition Formalism (SDF) [7]. Below are examples of two grammar production rules in SDF:

```
ID "=" Expression
        -> Stm        {cons("Assignment")}

HeuristicType? "selection"
    HeuristicSelectionType Params*
        -> Expression {cons("SelectionType")}
```

The "->" symbol means that the right hand side of the arrow can be replaced with the left hand side.

Production rules are labeled with the `cons` attribute. The example given above shows two production rules labeled "Assignment" and "SelectionType".

In the Assignment rule, the `Stm` non-terminal is expanded to the `ID` non-terminal, followed by an equals sign and the

[2]http://strategoxt.org/Spoofax

`Expression` non-terminal. There can also be other production rules for the `Stm` non-terminal.

In the SelectionType rule, the `Expression` non-terminal is expanded to an optional `HeuristicType` non-terminal, followed by the `selection` keyword, the `HeuristicSelectionType` non-terminal and any number of `Params` non-terminals.

Some syntax definition patterns in SDF are listed in Table 1 [9].

The code generation rules are written using the Stratego program transformation language [1] which is integrated into Spoofax. The code generation process produces Java code by applying rewrite rules to terms. Code generation rules are matched with the labels of the production rules.

For example, the code generation rule for the assignment statement for which the syntax was given above, is given in Figure 5. In this example, for the `Stm` non-terminal labeled as "Assignment", the code generation takes two parameters where the second parameter is a SelectionType `Expression` with its own three parameters. The template for the code to be generated is listed between square brackets and it can refer to the parameters. The `where` part is used to constrain the application of the rule. In this example, the rule is applied only when the `selectionMethod` parameter has the value `simple_random`.

```
to-java:
Assignment(VarName, SelectionType(None(),
    selectionMethod, params*)) ->
$[
SimpleRandomSelection [VarName] =
    new SimpleRandomSelection(hList, rng);
]
where
<eq>(selectionMethod, "simple_random");
<set-global> (VarName, "simple_random")
```

Figure 5: The code generation rule for heuristic selection.

4. PROPOSED HH-DSL

4.1 The Syntax

The syntax of the language is specified using the SDF notation as required by the Spoofax platform. An outline of the language grammar is given in Figure 6. Some of the reserved keywords of HH-DSL are `best`, `first`, `random`, `select`, `solution`, `heuristic`, `apply` and `replace`, which map to concepts in the hyper-heuristics domain. Other than these, some standard programming language constructs like `if-else` and `loop` are also supported. The `loop` iteration construct can take two forms. Beside the usual counter controlled repetition which allows repeating for a certain number of times, loops can also be limited by the time they are allowed to spend; so the total time available to the program can be divided between loops. This feature is implemented as a convenience for programs developed for the CHeSC competition.

While some of these types of general purpose constructs are necessary when expressing hyper-heuristic algorithms, adding too many such constructs might damage the aim of being a DSL and the language might start resembling a GPL. Some higher level features of HH-DSL are primarily meant for defining new hyper-heuristic operators instead of using standard operators which are already included in the platform.

As required by the domain, one of the basic operations in HH-DSL is selecting an element randomly from a collection. The `random` function takes a collection and the number of elements to select as parameters. If the number of elements is not specified, only one element will be selected. If the number is greater than 1 and an exclamation mark is appended to the number, it will select that many distinct elements.

Typically, a program in HH-DSL starts by setting the environment for later stages, such as configuring a selection heuristic method and an acceptance method. Then a set of initial solutions will be generated and the algorithm will proceed to a loop where the chosen methods will be applied to candidate solutions.

Heuristic Selection.

Hyper-heuristic algorithms need different mechanisms for selecting heuristics. Some mechanisms make their selection regardless of any history (such as simple random selection) whereas other mechanisms need to keep track of what has been selected so far (such as permutation based methods). Moreover, it might be desirable to select a certain type of heuristic, for example, a mutation heuristic. Therefore, in order to support these different cases, first a selector has to be created and configured, using an expression that consists of the heuristic type, followed by the `selection` keyword and the selection method. The valid heuristic types are: mutation, crossover, local search, ruin recreate, or any combination of these. Valid methods for selecting a heuristic of the given type are: simple random, random permutation, reinforcement learning, greedy selection. Specific heuristic selection methodologies can also be defined and used in HH-DSL. Once created, the `next` operation can be used to request the next heuristic from the selector. For example, the following code shows how to create a random mutation selector (assigned to the variable `hs`) and how to request a heuristic from it (assigned to the variable `h`).

```
hs = mutation selection simple_random
h = next hs
```

To choose from multiple types of heuristics, a combination of the types can be specified. For example, to choose a random mutation or local search heuristic:

```
hs = mutation+local_search selection simple_random
```

Acceptance Method Selection.

Another setting that needs to be determined concerns the acceptance of new solutions. There are four predefined acceptance methods: accept all, only improving, improving or equal, non-improving with probability. An accepter is created by giving a method name after the `acceptance` keyword.

```
ac = acceptance accept_all
```

Table 1: Some syntax definition patterns in SDF.

Pattern	Explanation	Examples
A*	Zero or more symbols A.	Stm*, [a-zA-Z]*
A+	One or more symbols A.	TypeDec+, [a-zA-Z]+
A?	Optional symbol A.	Expr?, [fFdD]?
{A0 A1}*	Zero or more symbols A0 separated by A1.	{Exp ","}*, {FormalParam ","}*
{A0 A1}+	One or more symbols A0 separated by A1.	{Id "."}+, {InterfaceType ","}+
A0 | A1	Alternative of symbol A0 or A1.	{Expr ","}* | LocalVarDec

Initialization.

The syntax for initializing solutions consists of the keywords `initialize solution` and optionally followed by the number of solutions to initialize. If no number is given, only one solution will be initialized. For example, the following statement will create and randomly initialize 10 solutions:

```
initialize solution 10
```

Applying Heuristics.

A heuristic can be applied to a solution using the `apply` statement. For example, the following code applies a heuristic (`h`) to a solution (`s`) and assigns the generated solution to a new variable (`n`):

```
n = apply h s
```

Accepting Solutions.

A generated solution can be accepted or rejected by requesting a check from an accepter. This operation takes the old and new solutions as parameters and decides whether to accept the new solution or not. The action to take if the new solution is accepted (such as replacing the old solution with the new one) can also be specified. In the example given below, the accepter (`ac`) will replace the old solution (`s`) with the new solution (`n`), provided the acceptance conditions have been met.

```
check ac n replace s
```

Defining Structures.

Users of HH-DSL may want to define their own methodologies. Structures can be defined in this case. A structure consists of statements and methods. In the example given below a structure (`myHeuristic`) for heuristic selection is defined. A permutation of heuristics is generated using `permute` statement and is assigned to an initial array. The method `next` is used to select and return the next heuristic from the permutation array.

```
define myHeuristic
{
    h[] = permute heuristic
    current = -1

    next() : Heuristic
    {
        current = current + 1
        return h[current]
    }
}
```

4.2 Code Generation

In the code generation phase, our system generates Java code which can be directly executed on the HyFlex platform. The Spoofax framework uses string interpolation for code generation. In this approach, the code to generate is expressed as a template containing references to variables and these variables will be replaced by their values.

As an example, the code generation rule for the solution initialization construct is given in Figure 7. The template for the code to be generated is written between "[" and "]" and it starts on line 3 in the listing. This template takes 1 parameter, called `MemorySize`, which gets referenced in the first line of the template. This parameter will be replaced with a value during actual code generation. For example, if this rule is invoked for the statement `initialize solution 10`, the `MemorySize` parameter takes the value 10 and the code given in Figure 8 gets generated.

As another example, consider the code generation rule for heuristic selection given in Figure 5. The `where` part restricts the application of this rule to only the case where the `selectionMethod` parameter has the value `simple_random`, therefore `hs = selection simple_random` statement generates the code given in Figure 9.

4.3 Working Environment

HH-DSL is written using the Spoofax language platform which is integrated into Eclipse, one of the leading modern development environments. Therefore, the working environment for HH-DSL can take advantage of the features provided by Eclipse and Spoofax, such as syntax highlighting and code folding.

After editing the HH-DSL code in the editor component of the working environment, code generation is invoked from the menu. An Eclipse Java project is generated automatically when code generation is invoked.

5. EXAMPLES AND DISCUSSION

The example codes given here are based on the examples provided in the HyFlex platform. These examples were tested on the system developed in this study to make sure that the generated code was equivalent to the code given in the HyFlex documentation. The DSL design and implementation were also tested with code not covered in the HyFlex documentation.

Figure 10 shows a simple hyper-heuristic example which works with a single candidate solution at a given time. Methods for heuristic selection will be chosen at random from all supported types of heuristics. A generated solution will definitely be accepted if it improves on the original solution, but only with probability 0.5 if it does not improve. At every iteration, a random heuristic will be applied to the solution

```
Stm* StructDcl*
  -> Start        {cons("TypesAndStatements")}
ID "=" Expression
  -> Stm          {cons("Assignment")}
Array "=" ArrayExpr
  -> Stm          {cons("ArrayAssignment")}
"check" ID ID "replace" ID ("[" INT "]")?
  -> Stm          {cons("Acceptance")}
"initialize" "solution" INT?
  -> Stm          {cons("Initialization")}
"define" ID "{" Stm* Methods* "}"
  -> StructDcl    {cons("StructDeclaration")}
"define" ID "(" StructParams ")"
    "{" Stm* Methods* "}"
  -> StructDcl    {cons("StructDclWithParams")}
ID"() : " VariableType? "{" Stm* "}"
  -> Methods      {cons("Methods")}
HeuristicType? "selection" HeuristicSelectionType
    Params*
  -> Expression   {cons("SelectionType")}
"acceptance" AcceptanceSelectionType
    AcceptanceParams*
  -> Expression   {cons("AcceptanceType")}
"next" ID
  -> Expression   {cons("HeuristicSelection")}
"apply" Arguments
  -> Expression   {cons("ApplyHeuristic")}
AdjExpr HSType
  -> Expression   {cons("HSSelection")}
"permute" "heuristic"
  -> ArrayExpr    {cons("PermuteHeuristic")}
AdjExpr "solution" INT "!"
  -> ArrayExpr    {cons("ArraySelection")}
{Heuristic "+"}+
  -> HeuristicType{cons("MultipleHeuristics")}
ID "in" ID
  -> Expr         {cons("CheckType")}
ID "[]"
  -> Array        {cons("Array")}
"best"
  -> AdjExpr      {cons("Best")}
"first"
  -> AdjExpr      {cons("First")}
"random"
  -> AdjExpr      {cons("Random")}
"heuristic"
  -> HSType       {cons("HeuristicType")}
"solution"
  -> HSType       {cons("SolutionType")}
"#" HSType
  -> Size         {cons("Size")}
Size
  -> Expression   {cons("SizeExpression")}
"if" Expr "{" Stm* "}" ("else" "{" Stm* "}")?
  -> Stm          {cons("IfElse")}
"loop" LoopCondition "{" Stm* "}"
  -> Stm          {cons("Loop")}
```

Figure 6: Some grammar rules of the proposed language.

```
to-java:
Initialization(Some(MemorySize)) -> $[
int solutionmemorysize = [MemorySize];
problem.setMemorySize(solutionmemorysize);
double["[]"] current_obj_function_values =
    new double["["]solutionmemorysize["]"];
for (int x = 0; x < solutionmemorysize; x++) {
    problem.initialiseSolution(x);
    current_obj_function_values["[x]"] =
        problem.getFunctionValue(x);
}
]
```

Figure 7: Code generation rule for solution initialization.

```
int solutionmemorysize = 10;
problem.setMemorySize(solutionmemorysize);
double[] current_obj_function_values =
    new double[10];
for (int x = 0; x < solutionmemorysize; x++) {
    problem.initialiseSolution(x);
    current_obj_function_values[x] =
        problem.getFunctionValue(x);
}
```

Figure 8: The generated Java code for solution initialization.

```
SimpleRandomSelection hs =
    new SimpleRandomSelection(hList, rng);
```

Figure 9: The generated Java code for simple random heuristic selection.

to obtain a new solution, which will then be checked for acceptance and if it is accepted, it will replace the original solution. This loop will use up all the time slot available to the program.

```
hs = selection simple_random
ac = acceptance nonimproving_with_probability 0.5
initialize solution
loop 100% {
    h1 = next hs
    s = first solution
    n1 = apply h1 s
    h2 = next hs
    n2 = apply h2 n1
    check ac n2 replace s
}
```

Figure 10: Example program with 1 candidate solution, simple random heuristic selection and probabilistic acceptance of non-improving solutions.

The second example program shown in Figure 11 works with 10 candidate solutions. Three heuristic selection methods are used for different stages of the algorithm. All generated candidate solutions will be accepted. At every iteration, a random mutational heuristic is applied to the best candidate solution in the current solution set. If the new candidate solution is accepted, it takes the place of the original best candidate solution. Next, a random local search heuristic is selected and applied to the best candidate solution. Again, if the new candidate solution is accepted, it takes the place of the original one. Then, two distinct candidate solutions are selected randomly and a random crossover heuristic is applied to these. If the obtained solution is accepted, it replaces the first one of the candidate solutions in the crossover operation. At the end of every iteration, the new best candidate solution is determined.

```
hs1 = mutation selection simple_random
hs2 = local_search selection simple_random
hs3 = crossover_heuristic selection simple_random
ac = acceptance accept_all
initialize solution 10
loop 100% {
    h1 = next hs1
    s = best solution
    n1 = apply h1 s
    check ac n1 replace s
    h2 = next hs2
    n2 = apply h2 s
    check ac n2 replace s
    s3[] = random solution 2!
    h3 = next hs3
    n3 = apply h3 s3[]
    check ac n3 replace s3[0]
    currentBest = best solution
}
```

Figure 11: Example program with 10 solutions, three heuristic selection methods and acceptance of all generated solutions.

Figure 12 shows an example of how to use conditional branches based on the types of selected heuristics. At every iteration, a mutation or local search heuristic is applied to a solution. Then, if that first heuristic is a mutation heuristic, a local search heuristic will be used which does not change between iterations.

Figure 13 shows an example of how to define and use a structure for heuristic selection. A custom structure named `myHeuristic` is defined. In this method, first, a permutation of all supported types of heuristics is generated. The `next` method is defined to select the next heuristic from the permutation array. This structure is used as the heuristic selection strategy in the program. All generated solutions are accepted in this example.

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed HH-DSL, a domain specific language for developing hyper-heuristic programs, which can be executed on HyFlex. Through examples, we illustrated how hyper-heuristic programs can be developed using HH-

```
hs1= mutation+local_search selection simple_random
hs2= local_search selection simple_random
ac = acceptance accept_all
h2 = next hs2
initialize solution
loop 100% {
    h1 = next hs1
    s = first solution
    n = apply h1 s
    if h1 in mutation {
        n = apply h2 n
    }
    check ac n replace s
}
```

Figure 12: Example program with check for heuristic type.

```
hs1 = selection myHeuristic
ac = acceptance accept_all
initialize solution 10
loop i in [1:20]{
    h1  = next hs1
    s = first solution
    n = apply h1 s
    check ac n replace s
}

define myHeuristic
{
    h[] = permute heuristic
    current = -1

    next() : Heuristic
    {
        current = current + 1
        return h[current]
    }
}
```

Figure 13: Example program with defining heuristic selection mechanism.

DSL. The examples show that writing a hyper-heuristic program in HH-DSL is easier than writing the same in Java. Currently, the features provided in HH-DSL are designed to work with hyper-heuristic development on HyFlex, however, features can be added to make it a more general tool for hyper-heuristic research. The advantage of working with HyFlex is that HyFlex provides the problem domain implementations. Therefore, researchers can focus on hyper-heuristic development rather than implementing the problems to solve. It also allows researchers to benchmark their approaches on standard implementations of these problem domains.

This paper presents a minimal, functional implementation of HH-DSL. It is a work in progress and the language is still being improved. The source code for HH-DSL is available at `https://bitbucket.org/hcora/hh-dsl`.

7. REFERENCES

[1] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[2] E. K. Burke, T. Curtois, M. Hyde, G. Kendall, G. Ochoa, S. Petrovic, and J. A. Vazquez-Rodriguez. HyFlex: A flexible framework for the design and analysis of hyper-heuristics. In *MISTA*, 2009.

[3] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and Q. Rong. A survey of hyper-heuristics. Technical report, 2009.

[4] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward. A classification of hyper-heuristic approaches. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research and Management Science*, pages 449–468. Springer, 2010.

[5] K. Chakhlevitch and P. Cowling. Hyperheuristics: Recent developments. volume 136 of *Studies in Computational Intelligence*, pages 3–29. Springer, 2008.

[6] P. Cowling, G. Kendall, and E. Soubeiga. A hyper-heuristic approach to scheduling a sales summit. In *Practice and Theory of Automated Timetabling III : Third International Conference, PATAT 2000*, volume LNCS 2079, August 2000.

[7] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual, 1992.

[8] I. F. Jr., I. Fister, M. Mernik, and J. Brest. Design and implementation of domain-specific language EasyTime. *Comput. Lang. Syst. Struct.*, 37(4):151–167, Oct. 2011.

[9] K. T. Kalleberg and E. Visser. Syntax definition in SDF, Jan. 2013.

[10] L. C. L. Kats and E. Visser. The Spoofax language workbench. In *Companion to the Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH 2010)*. ACM, 2010.

[11] T. Kos, T. Kosar, and M. Mernik. Development of data acquisition systems by using a domain-specific modeling language. *Comput. Ind.*, 63(3):181–192, Apr. 2012.

[12] T. Kosar, M. Mernik, and J. C. Carver. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Softw. Engg.*, 17(3):276–304, June 2012.

[13] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.

[14] G. Ochoa, M. Hyde, T. Curtois, J. A. Vazquez-Rodriguez, J. Walker, M. Gendreau, G. Kendall, B. McCollum, A. J. Parkes, S. Petrovic, and E. K. Burke. *HyFlex: A Benchmark Framework for Cross-domain Heuristic Search*, volume 7245 of *LNCS*. Springer, 2012.

[15] K. Olukotun. High performance embedded domain specific languages. *SIGPLAN Not.*, 47(9):139–140, Sept. 2012.

[16] E. Özcan, B. Bilgin, and E. E. Korkmaz. A comprehensive analysis of hyper-heuristics. *Intelligent Data Analysis*, 12:3–23, January 2008.

[17] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, Jan. 2001.

[18] M. Voelter. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, 2013.