



Addressing Problem Drift in UNHCR Fund Allocation

Sameela Suharshani Wijesundara ✉ 

Department of Data Science and AI, Faculty of IT, Monash University, Clayton, Australia
ARC Industrial Training and Transformation Centre OPTIMA, Clayton, Australia

Maria Garcia de la Banda ✉ 

Department of Data Science and AI, Faculty of IT, Monash University, Clayton, Australia
ARC Industrial Training and Transformation Centre OPTIMA, Clayton, Australia

Guido Tack ✉ 

Department of Data Science and AI, Faculty of IT, Monash University, Clayton, Australia
ARC Industrial Training and Transformation Centre OPTIMA, Clayton, Australia

Abstract

Optimisation models are concise mathematical representations of real-world problems, usually developed by modelling experts in consultation with domain experts. Typically, domain experts are only indirectly involved in the problem modelling process, providing information and feedback, and thus perceive the deployed model as a black box. Unfortunately, real-world problems “drift” over time, where changes in the input data parameters and/or requirements cause the developed model to fail. This requires modelling experts to revisit and update deployed models. This paper identifies the issue of problem drift in optimisation problems using as case study a model we developed for the United Nations High Commissioner for Refugees (UNHCR) to help them allocate funds to different crises. We describe the initial model and the challenges due to problem drift that occurred over the following years. We then use this case study to explore techniques for mitigating problem drift by including domain experts in the modelling process via techniques such as domain specific languages.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Software and its engineering → Constraint and logic languages

Keywords and phrases Fund Allocation, Problem Drift, Domain Specific Languages, MiniZinc

Digital Object Identifier 10.4230/LIPIcs.CP.2023.37

1 Introduction

Most software engineering projects suffer from one or several forms of *drift*, where a change in requirements during the lifetime of the project causes an existing, deployed software to degrade, to fail, or in the worst case, to produce incorrect solutions. Software projects that include optimisation components are no exception. In this paper, we present a real-world problem as a case study in *problem drift*, and discuss how we can mitigate its effects by enabling the problem owners to become part of the modelling and maintenance workflow.

Our case study is the United Nations High Commissioner for Refugees (UNHCR) fund allocation. UNHCR works in 135 countries and territories to provide crucial assistance to the millions of people forced to flee as a result of conflicts, crises, persecution or natural disasters; recently estimated as 112.6 million [19]. To achieve this, every year UNHCR proposes an annual *budget* for each of the many *projects* it wants to tackle and releases a Global Appeal requesting donations to cover it. For 2023 alone the Global Appeal was more than US\$10 billion [18]. The large amount of donations, or *funding*, received as a result need to be distributed among the budgeted projects according to different priorities, while ensuring the allocation respects any constraints set by the donors. For example, it is common for funding to be “earmarked” for a specific crisis, or a geographical area such as a region or a country. These allocation constraints often make it difficult for the funds to be optimally allocated.



© Sameela Suharshani Wijesundara, Maria Garcia de la Banda, and Guido Tack;
licensed under Creative Commons License CC-BY 4.0

29th International Conference on Principles and Practice of Constraint Programming (CP 2023).

Editor: Roland H. C. Yap; Article No. 37; pp. 37:1–37:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

As a result, excess funds would be carried forward from one year to the next rather than being directed to UNHCR’s activities such as life-saving assistance. The aim of UNHCR is to maximise the funds available for activities.

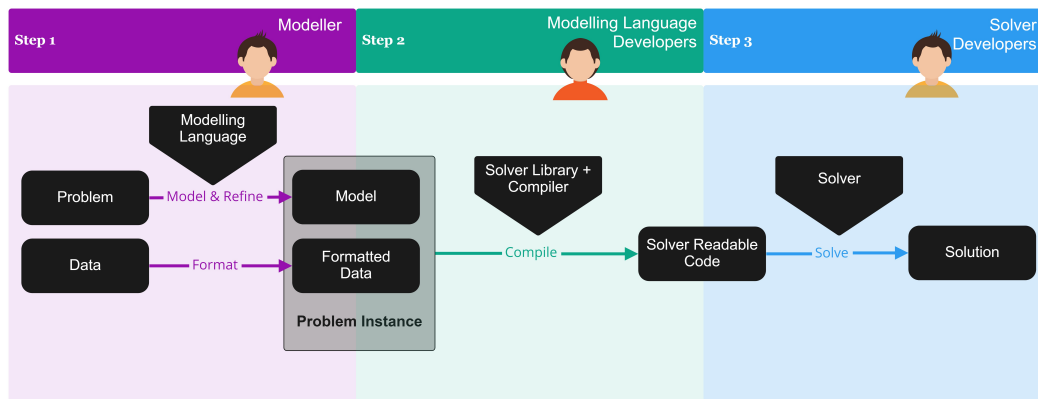
This *fund allocation problem* can be represented as a combinatorial optimisation problem. As such, it can be modelled in a high-level constraint modelling language (e.g., [5, 7, 14, 22]) and solved with state-of-the-art optimisation solving technology (e.g., [1, 17, 23]). In 2016, we developed a model for this problem that would have been able to provide US\$400 million more in allocations than their previous method, leaving only US\$100 million in unallocated funds. As it is common when developing such models, UNHCR experts were only indirectly involved in the modelling process: they provided information and feedback but did not fully understand the implemented model as they lacked the required expertise. This is unfortunate since, as it often happens with real-world problems, the problem requirements *drifted* over time, experiencing changes in the format and kind of the input data, as well as in the rules for allocating funds. After a while, the drift grew to the point where the developed model became unusable. In addition, personnel changes led to a loss of organisational memory, severing the contact between the organisation and us, the developers. As a result, UNHCR never implemented a full system using the model and continued to use their original greedy allocation algorithm, which they were able to maintain and update using in-house expertise.

This paper makes **three contributions**. The first is identifying the issue of problem drift in optimisation models and highlighting its importance to the community as an area worth investigating. The second contribution is our modelling of the UNHCR fund allocation problem: an initial model developed in the first phase of our work in 2016, and a new model developed in a second phase to incorporate the changes that caused the problem to drift. Our third and final contribution results from an analysis of this problem drift, and a subsequent exploration of techniques to mitigate this issue. We present an approach to include domain experts in the modelling process by adapting known software engineering techniques such as domain specific languages. This phase of our work is, so far, exploratory. However, we believe our approach can increase the domain experts’ trust in the developed solution, by making them an integral part of the development process, and can also enable the domain experts to adapt the model over time to keep up with problem drift.

2 Background

Combinatorial optimisation problems require finding a combination of choices (i.e., a **solution**) that satisfies a set of requirements and optimises the quality of the solutions. Modern approaches for solving these problems first specify a *model* of the problem that formally describes its choices, in terms of **variables** representing the decisions and their **domains** representing the choices available; its requirements, in terms of **constraints** defined over the variables; and the quality of the solutions in terms of an **objective function** also defined over the variables. Models are often specified using **parameters** so that input data can be provided independently of the model. This allows models to be used to solve any **instance** of the problem by instantiating the model parameters with the concrete input data and translating these instances into input for a **solver**, which then produces solutions to the problem. At a high level, this modelling and solving approach can be depicted as the step-wise process shown in Figure 1.

- *Step 1* formalises the real world problem via a model. This is currently a manual process where modelling experts communicate with domain experts to capture all relevant requirements, often requiring several iterations to progressively improve the formalisation (**refinement process**). The result is a model of the problem requirements written in a high-level modelling language such as MiniZinc [14], Essence [7], OPL [22] or AMPL [5].



■ **Figure 1** Step-wise illustration of the combinatorial problem modelling and solving process.

- *Step 2* takes the model along with concrete input data formatted according to the model's parameters and representing an instance of the problem, and compiles this instance into a format suitable for the solver selected by the modeller. Each of the modelling languages mentioned in *Step 1* has its own compilation methods and lower level languages such as FlatZinc for MiniZinc [14] and Essence' for Essence [7].
- *Step 3* solves the compiled instance using the selected solver. Solvers use state-of-the-art algorithms and heuristics that are efficient in tackling the unique challenges inherent in the kind of constraints they support. There are many different solving technology paradigms including MIP [23] (for Mixed Integer Programming), CP [17] (for Constraint Programming), and SAT [1] (for Boolean Satisfiability Problems).

Steps 2 and *3* are extensively automated thanks to many years of research on modelling language and solver design. However, *Step 1* is a manual process and, in practice, considerably more complex than depicted in Figure 1, involving multiple refinement and feedback loops.

3 Phase 1: The 2016 UNHCR Fund Allocation Problem and Model

3.1 Problem Description

In 2016 we worked with UNHCR on an optimisation model for their fund allocation. According to the rules at the time, part of or all of the donations pledged by an organisation (a fund) could be either *earmarked* or not, indicating whether the donor put some restrictions on the kind of projects the fund could cover or not, respectively. The kind of restrictions supported by the algorithm used at the time by UNHCR were based on the following fund and project attributes:

- **Level:** can take one of three values: **Region**, **Sub-region**, or **Country**, indicating the geographic scale of a project, e.g., a continent such as Europe, a part of a given continent such as Eastern Europe, or a country such as Ukraine. For a fund, the level is used to restrict its allocation, and can have an additional value **Global** to indicate that the fund is not restricted to a geographical area.
- **Region:** the region of the world where the project is located or to which the fund is allocated. Only relevant when the level is **Region**; disregarded otherwise.
- **Sub-region:** the sub-region of the world where the project is located or to which the fund is allocated. Only relevant when the level is **Sub-region**; disregarded otherwise.

37:4 Addressing Problem Drift in UNHCR Fund Allocation

- Area of Budgetary Control (ABC): the name of the UNHCR office which could be responsible either for a single country or a collection of countries each with fewer activities.
- Pillar: the broad target area of support. Pillar 1 stands for refugees, Pillar 2 for stateless people, Pillar 3 for reintegration, and Pillar 4 for internally displaced people. Pillars can be combined, e.g., Pillars 1-2. A fund that can be spent in any pillar is given the value `All Pillars`.
- Situation: the particular issue a project is tackling or a fund can be spent on, e.g., `South Sudan Situation`. A fund that can be spent on any situation is given the value `Country/Regular Program`.

The input data is given by an Excel sheet where each row is referred to as an `item`. The first six columns correspond to an item's pillar, situation, level, region, sub-region, and ABC. The next two columns contain the *budget* and *income*. An item with an empty budget column is a fund, and with a non-empty budget column is a project. Note that the budget of some projects can already be partially covered by money from other sources (e.g., bank interest). In such cases both the budget and income columns will have non-zero amounts.

The objective is to move money from fund items to project items to minimise the total amount of money left unspent in the funds, while ensuring the allocation constraints are satisfied. This means, ensuring money does **not** move from item A to item B if they have:

1. Different pillars, unless the pillar of item A is the generic `All Pillars`.
2. Different regions if the level of item A is `Region`.
3. Different sub-regions if the level of item A is `Sub-region`.
4. Different ABCs if the level of item A is `Country`.
5. Different situations, unless the situation of item A is `Country/Regular Programme`, its level is `Country`, its pillar is `All Pillars` and item A is in surplus for its country.

The first four preclude a fund earmarked for a non-generic pillar, region, sub-region, or country to be spent on a different one. The last one ensures that funds for generic country situations are spent first on their country. There are also common-sense rules ensuring, for example, that funds are not overspent and projects do not get more income than budgeted.

3.2 Model

Modelling this problem in MiniZinc was easy. We will not reproduce the full model here, but we will give enough details for later discussions.

Parameters. The Excel input was fed into 9 MiniZinc parameters: integer `n` representing the number of rows and used to build set `ITEM` of `1..n`, and eight one-dimensional arrays indexed by `ITEM` representing the values of the eight columns for each `ITEM`, i.e., its pillar, situation, level, region, sub-region, ABC, budget and income. The following MiniZinc code shows the definition of some of these parameters together with two enumerated data types (`enums`) whose definitions (strings from the input data) are the values for `REGION` and `SUBREGION`, an auxiliary parameter `maxinc` set to the maximum income, and an array `excess`, where `excess[i]` is the initial amount of money either available in fund `i`, or needed by project `i`:

```
set of int: ITEM = 1..n;
enum REGION;
enum SUBREGION;
array[ITEM] of REGION: region;
array[ITEM] of SUBREGION: subregion;
array[ITEM] of int: income;
array[ITEM] of int: budget;
int: maxinc = max(income);
array[ITEM] of int: excess = [ income[i] - budget[i] | i in ITEM ];
```

While enums were not used in the model, they are more intuitive and used here for brevity.

Variables. The decision variables are stored in a two-dimensional array `movement`, where `movement[i,j]` represents the amount of money moved from item `i` to item `j`, and is constrained to be between 0 and `maxinc`. Note that if `i` is a project, the value of `movement[i,j]` will be later set to 0. This array is defined as follows:

```
array[ITEM,ITEM] of var 0..maxinc: movement;
```

In addition, several auxiliary variables are defined based on `movement`. For example, `out[i]` and `inn[i]` represent, respectively, the amount of money moved out of item `i` or into it. They are defined as follows:

```
array[ITEM] of var 0..maxinc: out =
  [ sum(j in ITEM)(movement[i,j]) | i in ITEM ];
array[ITEM] of var 0..maxinc: inn =
  [ sum(j in ITEM)(movement[j,i]) | i in ITEM ];
```

Constraints. Are quite simple and modelled using the above parameters and variables. For example, the common-sense constraint ensuring funds are not overspent is modelled as:

```
constraint forall(i in ITEM where excess[i] > 0)(out[i] <= excess[i]);
constraint forall(i in ITEM where excess[i] <= 0)
  (forall(j in ITEM)(movement[i,j] = 0);
```

ensuring that if item `i` is a fund (`excess[i]>0`), the amount of money that moves out of `i` is not higher than its excess, and otherwise no money moves out of `i`. Similarly, constraint 2 which correctly earmarks funds based on region is modelled as follows:

```
constraint forall(i,j in ITEM
  where region[i] != region[j] /\ level[i] = Region)
  (movement[i,j] = 0);
```

ensuring no money moves from `i` to `j` if level of `i` is `Region` and their regions do not match. Note that constraints 3 and 4 follow exactly the same structure.

Our last example is constraint 5; the most complex. First we define auxiliary parameters `surplus_to_country_regular_situation[i]` as either a surplus if `i` is a generic country fund (i.e., one whose pillar is `all_pillars`, level `Country` and situation `country_regular_program`) or 0, otherwise. The surplus is defined as the fund's income minus the money needed by that country. We also define Boolean parameter `has_surplus_to_country_regular_situation[i]` to true iff the associated surplus parameter is positive.

```
array[ITEM] of -maxinc..maxinc: surplus_to_country_regular_situation =
  [ if situation[i] != country_regular_programme
    \ / level[i] != Country
    \ / pillar[i] != all_pillars
  then 0
  else income[i] -
    sum(j in ITEM where abc[i] = abc[j] /\
      situation[j] = country_regular_programme)
    (budget[j] - income[j])
  endif
  | i in ITEM ];
array[ITEM] of bool: has_surplus_to_country_regular_situation =
  [ surplus_to_country_regular_situation[i] > 0 | i in ITEM ];
```

These auxiliary parameters are then used to model part of constraint 5 as follows:

37:6 Addressing Problem Drift in UNHCR Fund Allocation

```
constraint forall(i in ITEM
    where situation[i] = country_regular_programme
    /\ level[i] = Country
    /\ pillar[i] = all_pillars
    /\ has_surplus_to_country_regular_situation[i])
    (sum(j in ITEM where situation[i] != situation[j]) (movement[i,j])
    <= surplus_to_country_regular_situation[i]);
```

If there is no surplus, the other part (not shown) sets `movement[i,j]=0` for every `j` in the same country but different situation.

Objective. Minimises the excess of funds and deficit of projects after allocation by means of the intermediate variable `difference`, which captures the excess for fund items (positive number) and the deficit for projects (negative number). The sum `obj` of the absolute value `absdiff` of these values is then computed and minimised as follows.

```
array[ITEM] of var -maxinc .. maxinc: difference =
    [ excess[i] - out[i] + inn[i] | i in ITEM ];
array[ITEM] of var 0..maxinc: absdiff;

constraint forall(i in ITEM)(absdiff[i] >= difference[i] /\
    absdiff[i] >= -difference[i]);

var int: obj = sum(i in ITEM)(absdiff[i]);
solve minimize obj;
```

This model solves the above fund allocation problem in less than 10s for the sample data we were given. With the 2016 sample data, the model was able to allocate US\$400 million more than the UNHCR greedy algorithm used at the time. UNHCR domain experts saw great value in the results produced by the optimisation model, and used it occasionally to provide supplementary input into their decision making process.

4 Phase 2: The 2022 UNHCR Fund Allocation Problem and Model

4.1 Problem Changes

When we contacted UNHCR again several years later, we learned that problem drift had made our model obsolete. After a number of discussions that spanned several months, we identified the following main changes to the problem.

Data format. Instead of the data being provided as a table where each row was an item representing both funds and projects, it is now provided as two separate tables (that we will refer to as *entities*), one for funds and one for projects.

Attributes. Some attributes changed names (e.g., fund income was now called *Balance*), others disappeared (e.g., funds no longer had budget as attribute, since they were always empty), and new ones appeared. The complete list of attributes for funds is: *Unique ID, Level, Domain, Region, Sub-Region, Country, Cost Center, Pillar, Situation, Account Status, Account Type, Balance*. Projects have the same attributes except that instead of *Balance*, they have a *Requirement* attribute that is equivalent to previously subtracting income from budget. Also, while previously the smallest geographical operational area provided in the sample data was the value of attribute *ABC*, countries are now further divided into smaller geographical areas defined by the attribute *Cost Centres*. More importantly, funds and

projects are now given priority levels via two different attributes. One is *Account Type*, which can take values OL (*operational level*) and AOL (*above operational level*), the latter having lesser priority [20]. The other is *Domain* which can take several values such as, in decreasing priority order, Headquarters (for leadership, management, policy guidance, etc), Global (for global programmes undertaken at the headquarters but of direct benefit to field operations), and Field (for field operations, which still takes 88% of the funding according to the 2022 Global Report [19]).

Constraints. Are now defined in terms of these new attributes. In particular, the constraints for Cost Centre and Domain are similar to those for other attributes.

Objective. While the main objective remained unchanged, it now also prioritised the allocation of funds based on their Account Type and Domain, as well as funds carried over from previous years and those with the highest number of constraints.

4.2 Model Changes

Parameters. Changes to the parameters of the model were mainly caused by UNHCR's decision to split ITEM into two entities, FUNDS and PROJECTS, and change their associated attributes. We modelled this similarly to before but using a different set of indexed arrays per entity, as follows:

```
set of int: PROJECTS = 1..numProjects;
set of int: FUNDS = 1..numFunds;

array[PROJECTS] of LEVEL: p_level;
array[PROJECTS] of DOMAIN: p_domain;
array[PROJECTS] of REGION: p_region;
array[PROJECTS] of ACCOUNT_TYPE: p_account_type;
array[PROJECTS] of int: p_requirement ;

array[FUNDS] of LEVEL: f_level;
array[FUNDS] of DOMAIN: f_domain;
array[FUNDS] of REGION: f_region;
array[FUNDS] of ACCOUNT_TYPE: f_account_type;
array[FUNDS] of int: f_balance;
```

Variables. The main decision variables are almost identical to those in the first model, except for the index sets now being known to be funds for the first dimension and projects for the second, and the maximum value being represented by fund parameter `f_balance`.

```
array[FUNDS,PROJECTS] of var 0..max(f_balance) : movement;
```

Most auxiliary variables also undergo very superficial changes, such as `out[i]` and `iin[i]`:

```
array[FUNDS] of var int : out =
  [ sum(p in PROJECTS)(movement[f,p]) | f in FUNDS ];
array[PROJECTS] of int : iin =
  [ sum(f in FUNDS)(movement[f,p]) | p in PROJECTS ];
```

Constraints. Some constraints encoding the same rules only require relatively superficial changes. For example, the constraint that earmarks funds based on region looks very similar:

37:8 Addressing Problem Drift in UNHCR Fund Allocation

```
constraint forall(f in FUNDS, p in PROJECTS
  where f_region[f] != p_region[p] /\ f_level[f] == L3)
  (movement[f,p] = 0);
```

if one knows L3 is equivalent to the old `Region`. Others change more, e.g., ensuring funds are not overspent is much simpler now that funds and projects are clearly separated:

```
constraint forall(f in FUNDS)(out[f] <= f_balance[f]);
```

In addition, we need new constraints to correctly earmark funds based on new attributes, such as cost centre, which look very similar to those implemented for Phase 1 attributes:

```
constraint forall(f in FUNDS, p in PROJECTS
  where f_cost_centre[f] != p_cost_centre[p] /\ f_level[f] == L6)
  (movement[f,p] = 0);
```

Objective. As described above, the old objective maximising the overall fund allocation was now extended to prioritise carry-over funds from the previous round, tightly earmarked funds (funds with the tightest restrictions), and funds allocated according to their Domain and Account Type. The new objective function was implemented as a weighted sum of these different aspects in order to achieve the required prioritisation.

4.3 Implementation and Adoption

We implemented the changes to the model as described above over the course of several months, in consultation with the UNHCR domain expert. Compared to the in-house algorithm, the new model is faster (it solves the sample data in under 3 minutes) and more effective in terms of total allocation and priority order. We were also able to experiment with extensions of the model, such as achieving fairness and balance in the allocation, which is easier to do using optimisation technology compared to greedy algorithms.

Unfortunately, the new model was not adopted by UNHCR. However, as a result of our work with them, they proposed the use of optimisation techniques with the new re-implementation of their enterprise resource planning (ERP) system. The viability of this is currently under investigation.

5 Phase 3: Addressing Problem Drift via Interactive Modelling DSLs

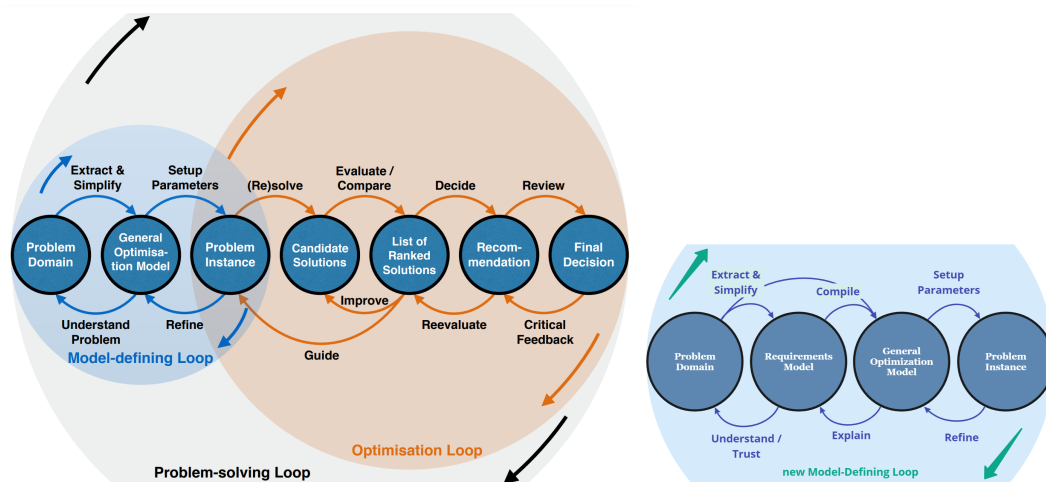
The transition from Phase 1 to Phase 2 of the UNHCR project encountered the usual challenges that exist for optimisation models even after they are successfully integrated into real-world workflows, which is an already challenging task. First, the problem had *drifted* between the two phases and the end users, i.e., the domain experts, did not have sufficient in-house expertise to adjust the implementation to tackle the drift. As a result, they continued to use their original method which was less effective but much more familiar. And second, due to subsequent personnel changes in UNHCR, Phase 2 required not just updating the implemented model, but on-boarding a new domain expert who had no reason to be invested in a new method other than our promises of a better future. While we were lucky enough for our new domain expert to trust us and become invested in developing a better approach, it is easier for organisations to simply continue using their old systems and move on.

In this section, we will present our *initial exploration* of how domain experts could be integrated better into the modelling process. Our proposal is based on our experience with Phases 1 and 2 of the UNHCR problem. Our goal is to develop a new *framework* for integrating domain experts into the modelling process in such a way they can (a) understand and verify how their user requirements have been mapped to the model, thus increasing the model's *transparency*; and (b) modify the model to cope with some level of problem drift, thus increasing the model's *flexibility*. Additionally, such an approach may help organisations obtain a better optimisation model, achieve higher levels of trust in the model, and develop more in-house expertise to manage the life cycle of the model. The rest of this section discusses this framework.

5.1 Related Work

The field of *Interactive Optimisation* has a long tradition of introducing domain experts into the development process early on. Meignan et al. [12] present a comprehensive overview of interactive optimisation approaches from operations research studies where domain experts are involved via a User Interface (UI) to guide the *solving* process. It also presents the high level components of an interactive optimisation system that includes a *preference model*, that is, an intermediate model between the UI and the optimisation model that captures user preferences when guiding the solver towards the preferred solution. This approach can be useful for users to find their preferred solutions, particularly for multi-objective optimisation problems with a large set of solutions. However, it can be achieved without domain experts understanding much about the model or being able to modify anything except its search and/or objective function.

Liu et al. [10] introduce a theoretical framework for interactive optimisation problem solving, called the *Problem Solving Loop* (left of Figure 2) and based on the *Sense Making Loop* [16] from the field of Visual Analytics [9]. The Problem Solving Loop aims at directly engaging the domain experts in the optimisation process. In doing this, it captures the high-level user goals and tasks in two major sub-loops: the *model-defining loop* and the *optimisation loop*, which correspond to the modelling design phase and the decision making phase, respectively.



■ **Figure 2** Problem solving loop presented by Liu et al. [10] and updated Model Defining Loop.

However, Liu et al. focus mostly on the latter, which covers the utilisation of the model by the domain experts from the moment they execute a *Problem Instance* of the *General Optimisation Model*, to the moment they make a final decision based on a *List of Ranked Solutions* (see left of Figure 2). Their model-defining loop does capture the fact that, in practice, problems are not converted to models in a singular step; rather, this involves multiple iterations of discussion among different stakeholders. However, they mostly ignore this sub-loop because they argue that its tasks, which lead to the creation of the *General Optimisation Model* and its *Problem Instance*, are usually done by modelling experts. While this is certainly the case currently, it does not mean it is appropriate, as we discuss below.

5.2 An Updated Model-Defining Loop

Our experience in several real-world projects indicates that to increase the model’s transparency and flexibility, we must deeply involve domain experts in the model-defining loop. Further, we need to do so by giving them a formal role and a formal language with which to communicate not only with the modelling experts but with the model itself. This will enable domain experts to verify the modelling experts’ perception of the problem, as well as to directly refine and modify the model, both as part of its initial definition and its ongoing maintenance. Thus, it should help address problem drift. To this end, we focus on extending this loop with the addition of a *Requirements Model* as depicted on the right of Figure 2. The Requirements Model captures user requirements in a *high-level, expressive intermediate formal language* at the early stages of the project. This language will have to be different for each application area, in order to allow both the modelling experts and the domain experts to express the parts of the model they are responsible for. Thus, we propose for the Requirements Model to be expressed in a *Domain Specific Language* (DSL) [13, 21, 8]. DSLs offer powerful expressivity while being tailored to a specific application domain, and have proven successful in many different application areas [6].

We call the DSLs that are used for the Requirements Model *Modelling DSLs, or MDSLs*. Our proposal is for the Requirements Model, expressed in an application-specific MDSL, to be compiled into a General Optimisation Model, which can then be instantiated and solved in the usual way. It is the responsibility of the MDSL designers to develop a compiler (often referred to as a generator in the DSL literature [15]) that generates efficient and correct code for the General Optimisation Model.

An important design decision is the level of abstraction and complexity required of an MDSL. On the one hand, we could make it as expressive as a General Purpose Modelling Language (GPML) such as MiniZinc, but this would defeat the purpose of simplicity and easy communication with the domain experts. On the other hand, significantly restricting its expressivity could severely limit its usefulness. For this reason, we propose to split the General Optimisation Model into two parts. A *fixed part*, which is implemented by the modelling expert; and a *flexible part*, which is generated (compiled) from an MDSL specification.

In the following, we will discuss which parts of an optimisation model such an MDSL should be able to express, and at what level of abstraction the MDSL should be designed. We will then see a concrete example of an MDSL for the UNHCR fund allocation problem.

5.3 MDSL expressivity

When designing an MDSL, it is important to select an appropriate level of expressivity, striking a balance between ease of use (especially for non-experts) and usefulness. While there are guidelines for general DSL design [8, 21, 13], we will focus here on the specific aspects related to Modelling DSLs. Let us look at the basic parts of an optimisation model and discuss how those parts are amenable to being expressed within MDSLs.

5.3.1 Variables

The choice of variables for an optimisation problem is one of the fundamental modelling decisions. It has a profound impact on the way the constraints are modelled and on the performance of solving algorithms – a fact that domain experts would usually not be aware of. Therefore, while it is important to define variables in a way that is easy to understand by the domain experts, we argue that the choice itself should not be left to domain experts. The modelling experts should define the variables, which are then made available to domain experts to be used (but not modified) via the MDSL.

This results in an architecture where the General Optimisation Model in our updated model-defining loop is split into two parts: a *fixed* part and a *flexible* one. The modelling experts design the former, which includes the choice of decision variables and is expressed in a General Purpose Modelling Language. This fixed part of the model cannot be modified through the MDSL, and any updates require input from a modelling expert. In addition, the modelling and domain experts collaborate on the *flexible model part*, which captures the Requirements Model and is defined using an MDSL. The two parts together are compiled into the General Optimisation Model, which can then be instantiated with data and solved.

Figure 3 shows a more detailed view of the new model-defining loop that includes this split into a fixed core and a flexible MDSL model.

5.3.2 Parameters

Parameters capture the core objects and data that the model is concerned with. We can split parameters into two kinds: *entities* and *attributes*. *Entities* define the main objects in a problem. For example, in Phase 1 of the UNHCR model, the main entity was an `ITEM`, while Phase 2 had two entities, `PROJECT` and `FUND`. *Attributes* define the *properties* of the entities. For example, in our Phase 1 model the `ITEM` entity had attributes such as `income` and `budget`, while in Phase 2 we saw `requirement` for `PROJECT` and `balance` for `FUND`.

Entities and their attributes commonly appear in the definitions of the constraints and/or the objective function in the model. For example, the balance of a fund will appear in the rules that govern how that fund can be used. The transition from Phase 1 to Phase 2 in the UNHCR problem saw the addition of new attributes (such as cost centre) and the removal of others (such as budget), which required changes to the corresponding funding rules. An MDSL can accommodate attribute changes by adding support for defining attributes and then using them consistently in the constraints and objective function. To do this we can build on the numerous DSLs commonly used in practice for this purpose, such as UML diagramming languages [2], class diagrams used in the object oriented paradigm and entity-relationship diagramming languages [3] used in database design. Note that this part of the MDSL is not application specific and can be used across many domains.

Accommodating changes in entities is more complex, as user decisions often relate to entities. For example, in the UNHCR problem we had to decide how much funding to move from each *fund* to each *project*. Thus, entity changes often require changes in the variables.

37:12 Addressing Problem Drift in UNHCR Fund Allocation

Based on our rationale above for leaving the choice of variables to the modelling experts, we do not propose yet to add support into MDSLs for defining new entities. However, a more abstract version of entity might be needed to avoid simple changes in input format.

5.3.3 Constraints

Supporting a close collaboration between modelling and domain experts during the definition and refinement of constraints is key for ensuring transparency. Supporting domain experts in modifying them is key for flexibility. As a result, any MDSL will need to be designed such that constraints can be specified and modified in a way that is natural for the domain experts. The Requirements Model, expressed in the MDSL, will serve as a means of communication and documentation between the modelling and domain experts. Therefore, the main challenge when applying our approach is designing an application-specific MDSL with sufficient expressivity to capture current and likely future constraints, yet sufficient simplicity to be understandable by domain experts. A prototype MDSL for the UNHCR problem is presented in Section 5.4.

5.3.4 Objective function

Changes to the objective function are also common, both during the modelling process and during the life cycle of a model [11]. To support this, MDSLs can provide basic arithmetic expressions involving all model variables and parameters that can then be used to define and/or modify the objective. This however is not enough for defining extra variables that may be needed to modify the objective. Since we do not want domain experts to introduce new variables using the MDSL, modelling experts will have to foresee and pre-define any auxiliary variables that may be exposed via the MDSL. For example, if the auxiliary variables for `total_allocation` and `minimum_allocation` are pre-modelled, domain experts will subsequently be able to formulate new objective functions using them.

The combination of basic arithmetic expressions and pre-defined variables also allows MDSLs to support domain experts in experimenting with multi-objective solving by combining different objectives in a weighted sum. These weights can then be changed by domain experts (including setting them to 0 to remove certain terms from the objective). Domain experts can even add new terms into the objective if required.

5.3.5 Proposed framework

Given the above discussion, we designed our framework to expand the functionality of the updated model-defining loop suggested in Section 5.2 with a new workflow depicted in Figure 3. As discussed earlier, the Requirements Model is used to *extract and simplify* the change requests and to capture them using an MDSL. Domain experts use it to *understand* how constraints are mapped to the model and to build *trust* in the model. The Requirements Model is then *compiled* to generate GPML code to be added to the ***General Optimisation Model***. An interesting extension would be to support bidirectional compilation, where GPML code can be reflected back into the Requirements Model. This would in turn allow the model to be *explained* in terms of the MDSL in the Requirements Model.

5.3.6 Natural vs Formal Languages for MDSLs

The main function of the MDSL is to serve as the interface between the domain expert and the optimisation model. Therefore, it has to strike a balance between being easy to understand by the domain expert, easy to implement, and easy to compile into a general

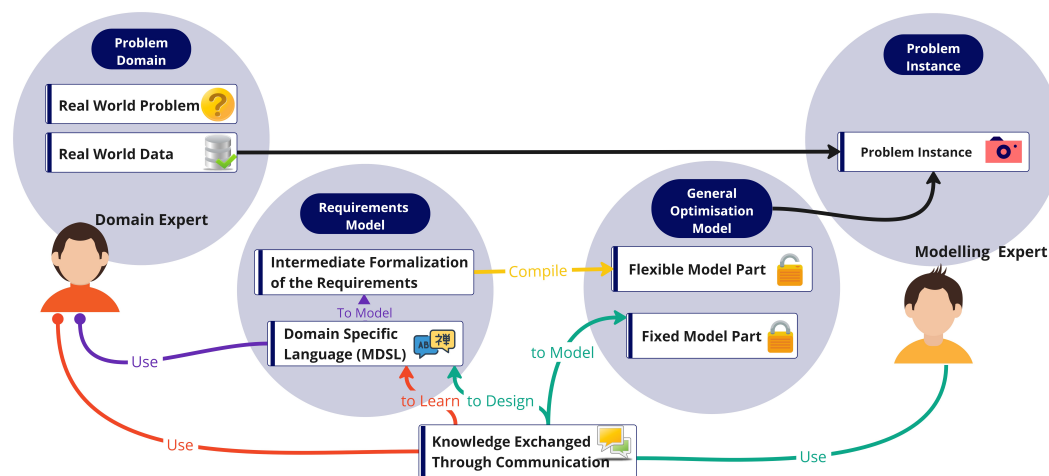
optimisation model. Being easy to understand suggests designing a language closer to the natural language a domain expert would use to describe the problem in a conversation; being easy to implement and to compile suggests one closer to the mathematical model that is going to be the result of the compilation process. Unfortunately, the vague and ambiguous nature of natural language can be problematic for an optimisation model, as its aim is to capture a problem specification precisely and unambiguously.

Let us illustrate this with an example from our UNHCR model. The following are statements from our domain experts, communicating to us their rules for transferring money:

1. Can transfer money from a fund to a project if the pillar of the fund is “all pillars“ or the pillars of the fund and the projects match
2. Can transfer money from a fund to a project if the situation of the fund is “regular situation“ or the situation of the fund and the projects match
3. Can transfer money from a fund to a project if the level of fund is equal to 2 and the domains of the fund and project match
4. Can transfer money from a fund to a project if the level of the fund is equal to 3 and the regions of the fund and project match

These statements have several issues. First, while they are expressed as *Can transfer*, expressing “possibility“ in an optimisation model is not easy. For example, our model’s `movement[i,j]` variable represents whether there is a transfer (`movement[i,j] > 0`) or not (`movement[i,j]=0`). We can represent this rule as *If movement[i,j] > 0, then the pillar of the fund must be “all pillars“, or the pillars of the fund and the projects must match*. While this captures the correct meaning, it may not be immediately intuitive for the domain expert (or anyone else). An alternative would be to state the contrapositive: *If the pillar of the fund is not “all pillars“ and the pillars of the fund and the projects do not match, then we must not transfer funds, so movement[i,j]=0*. This may be more intuitive, but it requires negating all conditions in the rules, which might be difficult to implement.

The second issue is the ambiguity regarding whether the statements are logically connected by “and“ or by “or“, or if they are supposed to be mutually exclusive. This is where the statements in contra-positive form have an advantage, as each of them stands on its own and it is clearer they should be connected by an “and“. However, even then we still have a third source of ambiguity, as we do not know whether the statements are exhaustive or not. In



■ **Figure 3** Further detailed model-defining loop incorporating MDSLs.

other words: if none of the rules holds, should we set `movement[i,j] = 0` or not? While it turns out the answer is “no“, the opposite could have been true, which is more difficult to achieve in an optimisation model.

A final point to note is that rules 3 and 4, when interpreted as mentioned above, in fact contradict each other: Rule 3 would mean that if the level is not 2, or the domains do not match, no movement is possible, while rule 4 states the same for level 3 and the regions. Clearly, the intended interpretation here is that if `movement[i,j] > 0` *and* the level is 2, then the domains must match.

These examples make it clear that, while a language closer to natural language seems convenient for domain experts, it is not precise enough to capture a unique logical meaning. As capturing the constraints in a mathematically precise format is crucial for optimisation modelling, care must be taken for the MDSLs to strike the right balance between being easy to understand *and* mathematically precise. Given the narrow scope of the language, domain experts will require some training to use an MDSL. However, we anticipate this to be only a minor hurdle when compared to learning a complete modelling language such as MiniZinc.

With the current fast advance of AI-based large-language-model discourse systems, it would be interesting to explore how this technology could be used to bridge the gap between precise mathematical MDSL and natural language. However, it is unclear how the inherent ambiguity in natural language formulations could be avoided.

5.4 An MDSL for UNHCR fund allocation problem

Our design goal for the MDSL grammar of the new UNHCR problem was to support domain experts in formalising any new required changes to the model in terms of the given set of parameters and variables. Compared to the full MiniZinc language, the MDSL grammar should be significantly simpler, covering only a small sub-set of MiniZinc’s capabilities specific to this problem domain. Most of the constraints in the UNHCR problem are expressed in the form of ad-hoc *rules* that consist of *conditions* and *consequences* in the form of *if-then-else* statements. This level of modelling is close to the actual constraints, while still being relatively easy to understand even for someone without any programming experience.

We first define the *abstract* syntax for the MDSL in the form of the following grammar:

```

<rule>      ::= <logical-expr> | <iteration> | <if-then-else>
<logical-expr> ::= <expr> <compare-op> <expr>
                | "not" <logical-expr>
                | <logical-expr> "and" <logical-expr>
                | <logical-expr> "or" <logical-expr>
<iteration>  ::= "forall" (<identifier>, ...) <logical-expr>
<if-then-else> ::= "if" <logical-expr> "then" <logical-expr> ("else" <logical-expr>)
<compare-op> ::= "=" | "!=" | "<" | "<=" | ">" | ">="
<expr>      ::= <named-expr> | <number> | <expr> <arithmetic-op> <expr>
<named-expr> ::= <identifier> | <identifier> "of" (<identifier>, ...)
                | <identifier> "from" <identifier> "to" <identifier>
<arithmetic-op> ::= "+" | "-" | "*" | "/"

```

The resulting abstract grammar is much simpler than that of MiniZinc. In particular, if-then-else expressions can only have Boolean type and cannot be nested; the grammar only supports one kind of iteration (`forall`); and it restricts the way variables and parameters are addressed. These restrictions were identified as sufficient for UNHCR problem.

While the grammar defines the *kind* of constraints that can be expressed in the MDSL, it does not define the concrete representation of the constraints the domain experts would use. Thus, the grammar can be implemented in several different ways. We now show a concrete text-based language as well as a simple graphical interface below to illustrate the possibilities. However, we do not claim that those are the most intuitive or best suited for this purpose.

A concrete text-based representation of the above abstract MDSL grammar could use syntax that is close to a more natural way of stating constraints, while still being concise and unambiguous. The following example shows a rule as defined in MiniZinc and in a concrete version of the abstract MDSL grammar:

MiniZinc constraint

```
constraint forall(f in FUNDS, p in PROJECTS where
  f_level[f] == L4 /\ f_subregion[f] != p_subregion[p]
  (movement[f,p] = 0);
```

Constraint represented using a concrete version of the abstract MDSL grammar

```
For Every FUND, PROJECT :
  IF [((level OF FUND )== (L4)) and
      ((subregion OF FUND )!= (subregion OF PROJECT ))]
  THEN
    [(movement FROM FUND TO PROJECT )== (0)]
```

Note how the concrete MDSL grammar simplifies the access to entity attributes, by using notation such as `level OF FUND` instead of having to introduce a new identifier `f` and corresponding expressions `f_level[f]`, as it is done in the MiniZinc model. For two-dimensional arrays like `movement`, it defines an explicit syntax `movement FROM FUND TO PROJECT` that avoids the ambiguity of the direction of movement present in the MiniZinc model. Many other concrete versions of the abstract MDSL grammar are also possible. For example, it would be easy to make the concrete grammar more verbose by writing the iteration as `For Every FUND and PROJECT`. Importantly, the MDSL grammar does not have hard-wired

Rule Base

- For Every FUND, PAIR OF PROJECT : IF [((pillar OF FUND)!= (pillar OF PROJECT1))and ((pillar OF FUND)!= (all_pillar))] THEN [(movement FROM FUND TO PROJECT1)= (0)] ELSE [[edit](#)]
- For Every FUND, PROJECT : IF [((level OF FUND)== (4))and ((subregion OF FUND)!= (subregion OF PROJECT))] THEN [(movement FROM FUND TO PROJECT)= (0)] ELSE [[edit](#)]

Rule Constructor

```
For Every FUND, PROJECT : IF [((level OF FUND )== (4))and ((subregion OF FUND )!= (subregion OF PROJECT ))] THEN [( movement FROM FUND TO PROJECT )= ( 0 )] ELSE []
```

For Every :

IF

and

==

level

!=

subregion subregion

THEN

=

movement

0

ELSE

Figure 4 User Interface for forming intermediate formalisation.

attribute names. If the set of attributes of an entity is later extended, the MDSL can easily be extended accordingly. Another important choice is that of operator precedences. We opted for full explicit bracketing, in order to avoid ambiguity. However, there are many other approaches, such as indentation-based grouping of expressions that belong together. It is beyond the scope of this paper to explore the advantages and disadvantages of different choices in concrete syntax.

In addition to the text-based representation, a graphical representation of the MDSL specifications could be used. Figure 4 shows a prototype browser-based interface that is still quite close to the text version. It has two main sections: The *Rule Base* section displays the already defined rules, and allows users to turn them on and off as needed. The *Rule Constructor* section allows users to define new rules using drop down list boxes containing various components of the language grammar, such as parameters (entity attributes) and variables. A graphical representation like this (or an integration of the text-based language into a graphical development environment) may have several advantages over pure text-based languages. For instance, precedence between different expressions can be indicated graphically, which may help avoid mistakes in the specification. Furthermore, the graphical interface can restrict the choices to those that are valid in a particular situation.

6 Conclusion and Future Work

Many organisations need to solve optimisation problems as part of their core operation, but they still perceive the introduction of technologies like model-based optimisation in their decision support systems as a significant risk. They may appreciate that this technology can solve difficult problems, that it can produce better outcomes than simple algorithms or manual solutions, and that it can incorporate additional factors such as uncertainty, fairness or diversity of solutions. However, the lack of in-house expertise for implementation and maintenance and the perceived black-box nature of the technology are often hurdles that are difficult to overcome. The change of input data formats or requirements after an optimisation solution has been deployed, which we refer to as *problem drift*, adds further risks to this process.

This paper presents the UNHCR fund allocation problem, a combinatorial optimisation problem that deals with the distribution of donated funds to humanitarian projects according to ad-hoc rules set by the donors and operational requirements of UNHCR. While the initial model we implemented in 2016 worked well for the sample data, the problem drift that occurred over the next few years prevented the organisation from fully adopting it, instead continuing to use their in-house algorithm. Phase 2 of our project focused on adapting the initial optimisation model to the changed requirements.

Based on this experience, this paper identifies problem drift in optimisation models and highlights its importance to the community as an area worth investigating. It then proposes to tackle it via an extended framework for model development that incorporates a *Requirements Model* into the Model Defining Loop defined by Liu et al. [10]. The requirements model is based on an application specific Modelling Domain Specific Language (MDSL). We have explored guidelines for the trade-off between general expressivity and ease of use of MDSLs. Our recommendations include that certain parts of a model such as the choice of decision variables need to be left to the optimisation expert, while other parts, such as entity attributes, constraints and objective function, can be exposed in the MDSL. We believe that the introduction of a Requirements Model and MDSL into the modelling process can facilitate collaboration between modelling and domain experts, achieve higher levels of trust with the end users of optimisation technology, as well as equip them with the skills and tools to address certain forms of problem drift in-house. sp An important area for future

work is to evaluate the feasibility, suitability and effectiveness of our proposed framework for addressing problem drift in real-world applications. We acknowledge that the development of an application-specific MDSL together with a model may seem prohibitively expensive. Future work will therefore include exploring the use of template MDSLs for larger problem classes to enable code reuse across applications. Furthermore, we will investigate how to speed up and streamline the MDSL development process using tools such as Language Workbenches [6, 4]. We will also study graphical and hybrid text/graphical MDSLs, which can draw on techniques from research areas such as interactive optimisation and visual analytics. Finally, it would be interesting to explore the use of AI-based dialogue systems such as large language models as a user-facing representation of an MDSL, where the potential ambiguity of natural language becomes a challenging research topic.

References

- 1 A. Biere, M. Heule, and H. van Maaren. *Handbook of Satisfiability*. IOS Press, January 2009.
- 2 Grady Booch, James E. Rumbaugh, and Ivar Jacobson. *The unified modeling language user guide - covers UML 2.0, Second Edition*. Addison Wesley object technology series. Addison-Wesley, 2005.
- 3 Peter P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976. doi:10.1145/320434.320440.
- 4 Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, volume 8225, pages 197–217. Springer International Publishing, Cham, 2013. Series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-319-02654-1_11.
- 5 Robert Fourer, David M Gay, and Brian W Kernighan. A modeling language for mathematical programming. *Management Science*, 36(5):519–554, 1990.
- 6 Martin Fowler. *Domain-Specific Languages*. Pearson Education, September 2010.
- 7 Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, September 2008. doi:10.1007/s10601-008-9047-y.
- 8 Paul Hudak. Domain Specific Languages. *Handbook of programming languages*, 3(39-60):23, 1997.
- 9 Youn-ah Kang and John Stasko. Examining the Use of a Visual Analytics System for Sensemaking Tasks: Case Studies with Domain Experts. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2869–2878, December 2012. doi:10.1109/TVCG.2012.224.
- 10 Jie Liu, Tim Dwyer, Kim Marriott, Jeremy Millar, and Annette Haworth. Understanding the Relationship Between Interactive Optimisation and Visual Analytics in the Context of Prostate Brachytherapy. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):319–329, January 2018. doi:10.1109/TVCG.2017.2744418.
- 11 Jie Liu, Tim Dwyer, Guido Tack, Samuel Gratzl, and Kim Marriott. Supporting the Problem-Solving Loop: Designing Highly Interactive Optimisation Systems. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1764–1774, February 2021. doi:10.1109/TVCG.2020.3030364.

- 12 David Meignan, Sigrid Knust, Jean-Marc Frayret, Gilles Pesant, and Nicolas Gaud. A Review and Taxonomy of Interactive Optimization Methods in Operations Research. *ACM Transactions on Interactive Intelligent Systems*, 5(3):1–43, October 2015. doi:10.1145/2808234.
- 13 Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.
- 14 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, Lecture Notes in Computer Science, pages 529–543, Berlin, Heidelberg, 2007. Springer. doi:10.1007/978-3-540-74970-7_38.
- 15 Václav Pech. JetBrains MPS: Why Modern Language Workbenches Matter. In Antonio Bucchiarone, Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio, editors, *Domain-Specific Languages in Practice: with JetBrains MPS*, pages 1–22. Springer International Publishing, Cham, 2021. doi:10.1007/978-3-030-73758-0_1.
- 16 P. Pirolli and S. Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of international conference on intelligence analysis*, volume 5. McLean, VA, USA, 2005.
- 17 Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, August 2006.
- 18 UNHCR Global Appeal 2023. Accessed on 5 May, 2023. URL: <https://reporting.unhcr.org/globalappeal2023>.
- 19 UNHCR Global Report 2022. Accessed on 5 May, 2023. URL: <https://reporting.unhcr.org/global-report-2022>.
- 20 UNHCR operations plan in emergencies. Accessed on 5 May, 2023. URL: <https://emergency.unhcr.org/support-response/planning-and-programming/unhcr-operations-plan-emergencies>.
- 21 Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000. doi:10.1145/352029.352035.
- 22 Pascal Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999.
- 23 Laurence A. Wolsey. *Integer Programming*. John Wiley & Sons, September 2020.