Computational Monotone Co-Design

Compiled on 2025-04-18.

Contents

Invitation to computational co-design | Part A.

The *Alphorn* is a "labrophone", consisting of a straight multi-meter-long wooden natural horn and a wooden cup-shaped mouthpiece. It is used by mountain dwellers in Swiss alps as musical instruments and communication tools.

1. A tour of MCDPL

This chapter describes the basic concepts of the MCDPL language.

Switzerland is famous for high quality chocolate. In particular, Switzerland is renowned for its milk chocolate, invented in Vevey. The per capita yearly chocolate consumption in Switzerland is about 10 kg, and the industry counts over 4,400 employees.

This chapter provides a tutorial on the language MCDPL and related tools.

1.1. What MCDPL is

MCDPL is a *modeling language* that can be used to formally describe *monotone* co-design problems. This means that MCDPL allows describing variables and systems of constraints between variables. MCDPL is not a generic *programming* language. It is not possible to write loops or conditional statements.

MCDPL is designed to represent all and only MCDPs (Monotone Co-Design Problems). This means that variables belong to partially ordered sets, and all relations are monotone. For example, multiplying by a negative number is a syntax error.

MCDPs can be interconnected and composed hierarchically. Several features help organize the models in reusable "libraries".

Once an MCDP model is described, then we can *query* it in various ways. Another way to see this is that an MCDP model is not a single optimization problem, but rather a collection of optimization problems.

This chapter describes the MCDPL modeling language by way of a tutorial. A more formal description is given in Part B.

1.2. Graphical representations of design problems

Monotone design problems are graphically represented as in Fig. 1.

A design problem is represented by a box with curved corners. On the left, solid green arrows represent functionalities; on the right, dashed red arrows represent requirements.

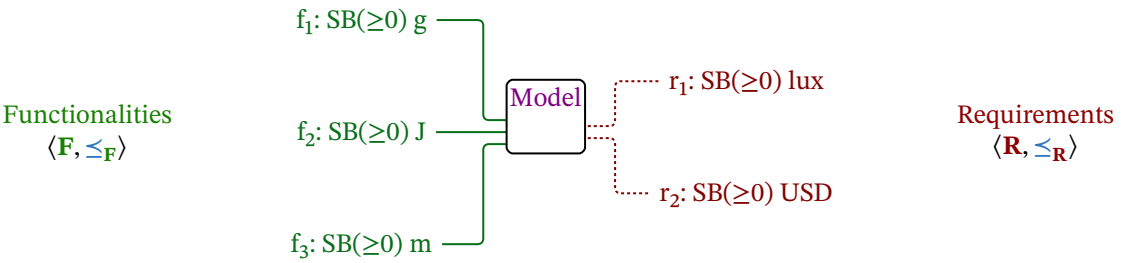


Figure 1.: Representation of a design problem with three functionalities (`f_1`, `f_2`, `f_3`) and two requirements (`r_1`, `r_2`). In this case, the functionality space **F** is the product of three copies of $\overline{\mathbb{R}}_{\geq 0}$: $\mathbf{F} = \overline{\mathbb{R}}_{\geq 0}[\mathrm{g}] \times \overline{\mathbb{R}}_{\geq 0}[\mathrm{J}] \times \overline{\mathbb{R}}_{\geq 0}[\mathrm{m}]$ and $\mathbf{R} = \overline{\mathbb{R}}_{\geq 0}[\mathrm{lux}] \times \overline{\mathbb{R}}_{\geq 0}[\mathrm{USD}]$.

The graphical representation of a co-design problem is as a set of design problems that are interconnected (Fig. 2). A functionality and a requirement edge can be joined using a $\preceq$ sign. This is called a "co-design constraint".

In the figure below, there are two subproblems called a and b, and the co-design constraints are $r_1 \preceq f_3$ and $r_3 \preceq f_2$.

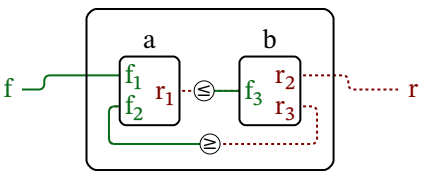


Figure 2.: Example of a *co-design diagram* with two design problems, a and b. The *co-design constraints* are $r_1 \preceq f_3$ and $r_3 \preceq f_2$.

1.3. Your first model

An MCDP is described using the keyword `mcdp`. The simplest MCDP is shown in Listing 1. In this case, the body is empty, and that means that there are no functionality and no requirements.

The representation of this MCDP is as in Fig. 3: a simple box with no signals in or out.

Listing 1: `empty.mcdp`

```
mcdp {

}
```

Figure 3.

Adding functionality and requirements

The functionality and requirements of an MCDP are defined using the keywords `provides` and `requires`.

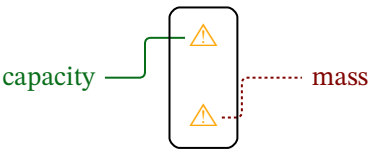
The code in Listing 2 defines an MCDP with one functionality, `capacity`, measured in joules, and one requirement, `mass`, measured in grams.

The graphical representation is in Fig. 4.

Listing 2: `model1.mcdp`

```
mcdp {
    provides capacity [J]
    requires mass [g]
}
```

Figure 4.



That is, the functionality space is $\mathbf{F} = \overline{\mathbb{R}}_{\geq 0}[\mathrm{J}]$ and the resource space is $\mathbf{R} = \overline{\mathbb{R}}_{\geq 0}[\mathrm{g}]$. Here, let $\overline{\mathbb{R}}_{\geq 0}[\mathrm{g}]$ refer to the nonnegative real numbers with units of grams.

The MCDP defined above is, however, incomplete, because we have not specified how `capacity` and `mass` relate to one another. In the graphical notation, the co-design diagram has unconnected arrows (Fig. 4).

Constraining functionality and requirements

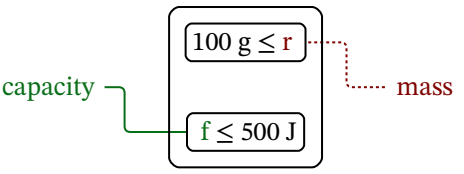
In the body of the `mcdp` declaration one can refer to the values of the functionality and requirements using the expressions `provided` *functionality name* and `required` *requirement name*. For example, Listing 3 shows the completion of the previous MCDP, with hard bounds given to both `capacity` and `mass`.

The visualization of these constraints is as in Fig. 5.

Listing 3

```
mcdp {
    provides capacity [J]
    requires mass [g]
    provided capacity ≤ 500 J
    required mass ≥ 100 g
}
```

Figure 5.



It is possible to query this minimal example. For example, we can ask through the command-line interface to solve for the minimal requirements given the functionality constraint of 400 J, using the command:

```
mcdp-solve minimal "400 J"
```

The answer is:

```
Minimal resources needed: mass = ↑ {100 g}
```

If we ask for more than the MCDP can provide:

```
mcdp-solve minimal "600 J"
```

we obtain no solutions: this problem is unfeasible.

Use of Unicode glyphs in the language

To describe the inequality constraints, MCDPL allows to use "<=", ">=", as well as the Unicode versions "≤", "≥", "≤", "≥".

These two snippets are equivalent:

using ASCII characters

```
provided capacity <= 500 J
required mass >= 100g
```

using Unicode characters

```
provided capacity ≤ 500 J
required mass ≥ 100g
```

MCDPL also allows to use some special letters in identifiers, such as Greek letters and subscripts.

These two snippets are equivalent:

using ASCII characters

```
alpha_1 = beta^3 + 9.81 m/s^2
```

using Unicode characters

```
α₁ = β³ + 9.81 m/s²
```

Aside: a helpful interpreter

The MCDPL interpreter tries to be very helpful by being liberal in what it accepts and suggests fixes to common mistakes.

For example, if one forgets the keyword **provided**, the interpreter will give the following warning:

```
Please use "provided capacity" rather than just "capacity".

line 2 |    provides capacity [J]
line 3 |    requires mass [g]
line 4 |
line 5 |    capacity <= 500 J
...... |    ^^^^^^^^
```

The web IDE will automatically insert the keyword using the "beautify" function.

All inequalities are of the kind:

$$\text{resources} \preceq \text{functionality}. \tag{1}$$

If you mistakenly put functionality and requirements on the wrong side of the inequality, as in:

```
provided capacity ≥ 500 J # incorrect expression
```

then the interpreter will display an error like:

```
DPSemanticError: This constraint is invalid.
Both sides are requirements.

line 5 |    provided capacity <= 500 J
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~↑
```

1.4. Describing relations between functionality and resources

In MCDPs, functionality and requirements can depend on each other using any monotone relations.

MCDPL implements some primitive relations, such as addition, multiplication, and exponentiation by a constant.

For example, we can describe a linear relation between mass and capacity, given by the specific energy ρ:

$$\text{capacity} = \rho \times \text{mass}. \tag{2}$$

This relation can be described in MCDPL as either of the following:

```
provided capacity ≤ ρ • required mass
```

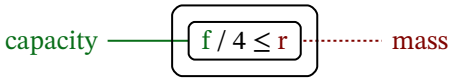
```
provided capacity / ρ ≤ required mass
```

In the graphical representation (Fig. 6), there is now a connection between `capacity` and `mass`, with a DP that multiplies by the inverse of the specific energy.

Listing 4: `model4.mcdp`

```
mcdp {
    provides capacity [J]
    requires mass [g]
    ρ = 4 J / g
    required mass ≥ provided capacity / ρ
}
```

Figure 6.



For example, if we ask for 600 J:

```
mcdp-solve linear "600 J"
```

we obtain this answer:

```
Minimal resources needed: mass = ↑{150 g}
```

Units

Units are taken seriously. The interpreter will complain if there is any dimensionality inconsistency in the expressions. However, as long as the dimensionality is correct, it will automatically convert to and from equivalent units.

For example, in Fig. 7 the specific energy is given in `kWh/kg`. PyMCDP will take care of the needed conversions, and will introduce a conversion from `J*kg/kWh` to `g` (Fig. 8).

```
mcdp {
        '''
        Simple model of a battery as a linear relation
        between capacity and mass.
        '''
    provides capacity [J] 'Capacity provided by the battery'
    requires mass [g] 'Battery mass'
    ρ = 200 kWh / kg 'Specific energy'
    required mass ≥ provided capacity / ρ
}
```

Figure 7.: Automatic conversion among `g`, `kg`, `J`, `kWh`.

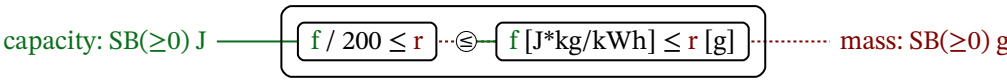


Figure 8.

Fig. 7 also shows the syntax for comments. MCDPL allows adding a Python-style documentation string at the beginning of the model, delimited by three quotes. It also allows giving a short description for each functionality, requirement, or constant declaration, by writing a Python-style string at the end of the line.

1.5. Catalogs

The previous example used a linear relation between functionality and requirement. However, in general, MCDPs do not make any assumption about continuity and differentiability of the functionality-requirement relation. The MCDPL language has a construct called `catalog` that allows defining an arbitrary discrete relation.

Recall from the theory that a design problem is defined from a triplet of functionality space, implementation space, and requirement space.

According to the diagram in Fig. 9, one should define the two maps `req` and `prov`, which map an implementation to the functionality it provides and the requirements it requires.

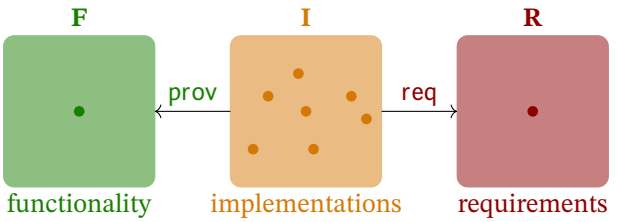


Figure 9.: A design problem is defined from an implementation space **I**, a functionality space **F**, a requirement space **R**, and the maps `req` and `prov` that relate the three spaces.

MCDPL allows defining arbitrary maps `req` and `prov`, and therefore arbitrary relations from functionality to requirements, using the `catalog` construction.

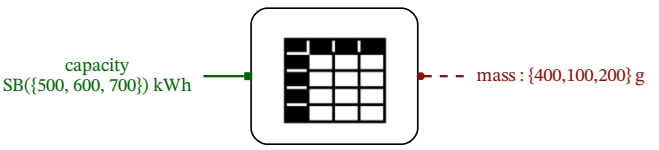
An example is shown in Listing 5. In this case, the implementation space contains the three elements `model1`,`model2`, `model3`. Each model is explicitly associated with a value in the functionality and the requirements space.

The icon for this construction is meant to remind us of a spreadsheet (Fig. 10).

Listing 5: `catalog1.mcdp`

```
catalog {
    provides capacity [kWh]
    requires mass [g]
    500 kWh ↤ model1 ↦ 100 g
    600 kWh ↤ model2 ↦ 200 g
    700 kWh ↤ model3 ↦ 400 g
}
```

Figure 10.



In Fig. 10 we can see how the type of the mass is not just grams, but rather given as `{100, 200, 400 g}`. The solver propagates the bounds information about functionalities and requirements and uses this information to prune the search space during querying.

Multiple minimal solutions

The `catalog` construct is the first construct we encountered that allows defining MCDPs that have *multiple minimal solutions*. To see this, let's expand the model in Listing 5 to include a few more models and one more requirement, `cost`.

Listing 6: `catalog2.mcdp`

```
catalog {
    provides capacity [kWh]
    requires mass [g]
    requires cost [USD]

    500 kWh ↤ model1 ↦ 100 g, 10 USD
    600 kWh ↤ model2 ↦ 200 g, 200 USD
    600 kWh ↤ model3 ↦ 250 g, 150 USD
    700 kWh ↤ model4 ↦ 400 g, 400 USD
}
```

The numbers (not realistic) were chosen so that `model2` and `model3` do not dominate each other: they provide the same functionality (`600 kWh`) but one is cheaper but heavier, and the other is more expensive but lighter. This means that for the functionality value of `600 kWh` there are two minimal solutions: either `⟨200 g, 200 USD⟩` or `⟨250 g, 150 USD⟩`.

The number of minimal solutions is not constant: for this example, we have four cases as a function of f (Table 1.1). As f increases, there are 1, 2, 1, and 0 minimal solutions.

Table 1.1.: Solution cases for the model in Listing 6.

Functionality required	Optimal implementation(s)	Minimal resources needed
`0 kWh` $\leq f \leq$ `500 kWh`	`model1`	`⟨100 g, 10 USD⟩`
`500 kWh` $< f \leq$ `600 kWH`	`model2` or `model3`	`⟨200 g, 200 USD⟩` or `⟨250 g, 150 USD⟩`
`600 kWh` $< f \leq$ `700 kWH`	`model4`	`⟨400 g, 400 USD⟩`
`700 kWh` $< f \leq$ `Top kWh`	(unfeasible)	$\emptyset$

We can verify these with `mcdp-solve`. We also use the switch `--imp` to ask the program to give the name of the implementations; without the switch, it only prints the value of the minimal resources.

For example, for $f =$ `50 kWH`:

```
mcdp-solve --imp catalog2 "50 kWh"
```

we obtain one solution:

```
Minimal resources needed: mass, cost = ↑{⟨mass:100 g, cost:10 USD⟩}
r = ⟨mass:100 g, cost:10 USD⟩
implementation 1 of 1: m = 'model1'
```

For $f =$ `550 kWH`:

```
mcdp-solve --imp catalog2 "550 kWh"
```

we obtain two solutions:

```
Minimal resources needed:
mass, cost = ↑{⟨mass:200 g, cost:200 USD⟩, ⟨mass:250 g, cost:150 USD⟩}

r = ⟨mass:250 g, cost:150 USD⟩
implementation 1 of 1: m = 'model3'

r = ⟨mass:200 g, cost:200 USD⟩
implementation 1 of 1: m = 'model2'
```

`mcdp-solve` displays first the set of minimal resources required; then, for each value of the resource, it displays the name of the implementations; in general, there could be multiple implementations that have the same resource consumption.

1.6. Union/choice of design problems

The “choice” construct allows describing the idea of "alternatives".

As an example, let us consider how to model the choice between different battery technologies.

Consider the model of a battery, in which we take the functionality to be the `capacity` and the requirements to be the `mass [g]` and the `cost [USD]`.

Consider two different battery technologies, characterized by their specific energy (`Wh/kg`) and specific cost (`Wh/USD`).

Specifically, consider Nickel-Hidrogen batteries (NiH2) and Lithium-Polymer (LiPo) batteries. One technology is cheaper but leads to heavier batteries and vice versa. Because of this fact, there might be designs in which we prefer either.

First, we model the two battery technologies separately as two MCDPs that have the same interface (same requirements and same functionalities).

Listing 7: `Battery1_LiPo.mcdp`

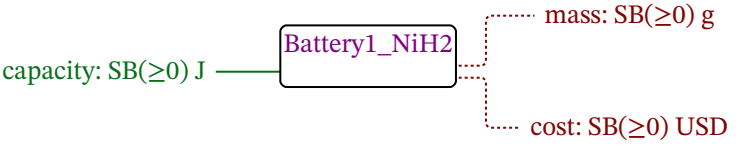
```
mcdp {
    provides capacity [J]
    requires mass [g]
    requires cost [USD]
    ρ = 150 Wh/kg
    α = 2.50 Wh/USD
    required mass ≥ provided capacity / ρ
    required cost ≥ provided capacity / α
}
```

Figure 11.

Listing 8: `Battery1_NiH2.mcdp`

```
mcdp {
    provides capacity [J]
    requires mass [g]
    requires cost [USD]
    ρ = 45 Wh/kg
    α = 10.50 Wh/USD
    required mass ≥ provided capacity / ρ
    required cost ≥ provided capacity / α
}
```

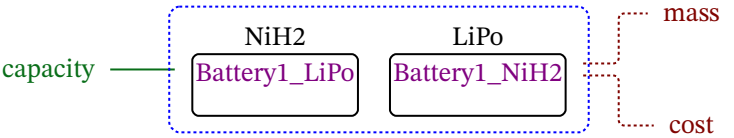
Figure 12.



Listing 9: `Batteries.mcdp`

```
choose(
NiH2: `Battery1_LiPo,
LiPo: `Battery1_NiH2
)
```

Figure 13.



1.7. Composing design problems

The MCDPL language encourages composition and code reuse through composition of design problems.

Implicit composition of design problems from formulas

All the mathematical operations (addition, multiplication, power, etc.) are each represented by their own design problem, even though they do not have explicit names.

For example, let's define the MCDP `Actuation1` to represent the actuation in a drone. We model it as a quadratic relation from `lift` to `power`, as in Listing 10.

Listing 10: `Actuation1.mcdp`

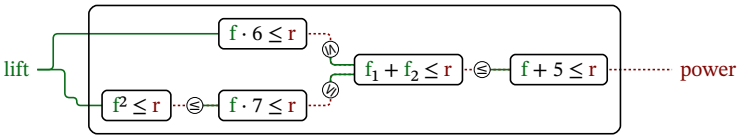
```
mcdp {
    provides lift [N]
    requires power [W]

    l = provided lift
    p₀ = 5 W
    p₁ = 6 W/N
    p₂ = 7 W/N²
    required power ≥ p₀ + p₁ • l + p₂ • l²
}
```

Figure 14.

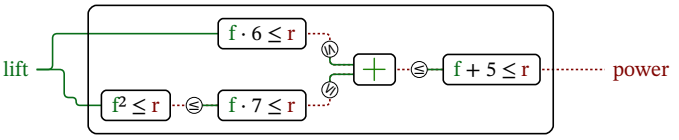
Note that the relation between `lift` and `power` is described by the polynomial relation

```
required power ≥ p₀ + p₁ • l + p₂ • l²
```

This relation can be written as the composition of several DPs, corresponding to sum, multiplication, and exponentiation (Fig. 14).



In the following, we will use icons for the operations:



Explicit composition of design problems

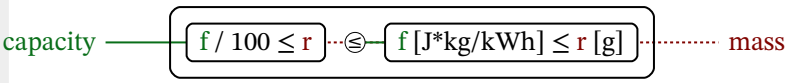
The other way to compose design problems is for the user to define them separately and then connect them explicitly.

Suppose we define a model called `Battery` as in Fig. 15.

Listing 11: `Battery1.mcdp`

```
mcdp {
    provides capacity [J]
    requires mass [g]
    ρ = 100 kWh / kg # specific_energy
    required mass ≥ provided capacity / ρ
}
```

Figure 15.

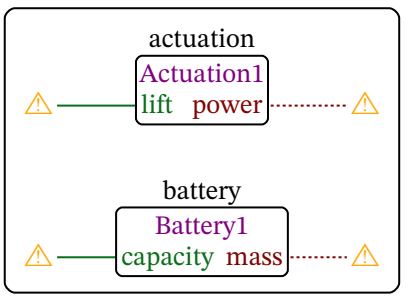


We can now put the battery model and the actuation model using the following syntax.

Listing 12: `combined1.mcdp`

```
mcdp {
    sub actuation = instance `Actuation1
    sub battery = instance `Battery1
}
```

Figure 16.



The syntax to re-use previously defined MCDPs is `instance` `` `Name ``. The backtick means "load the MCDP from the library, from the file called `Name.mcdp`".

Simply instancing the models puts them side-to-side in the same box; we need to connect them explicitly. The model in Listing 12 is not usable yet because some of the edges are unconnected Fig. 16. We can create a complete model by adding co-design constraints.

Adding co-design constraints

For example, suppose that we know the desired `endurance` for the design. Then we know that the `capacity provided by the battery` must exceed the `energy` required by actuation, which is the product of power and endurance. All of this can be expressed directly in MCDPL using the syntax:

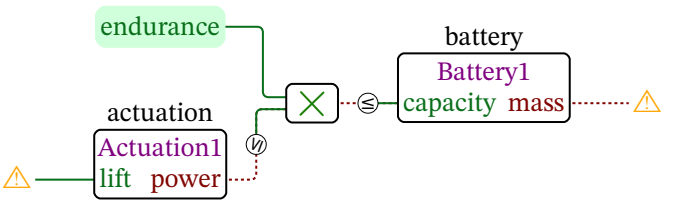
```
energy = provided endurance · (power required by actuation)
capacity provided by battery ≥ energy
```

The visualization of the resulting model has a connection between the two design problems representing the co-design constraint (Listing 13).

Listing 13: `combined2.mcdp`

```
mcdp {
    provides endurance [s]

    sub actuation = instance `Actuation1
    sub battery = instance `Battery1

        # battery must provide power for actuation
    energy = provided endurance · power required by actuation
    capacity provided by battery ≥ energy
        # still incomplete...
}
```

Figure 17.



We can create a model with a loop by introducing another constraint.

Take `payload` to represent the user payload that we must carry.

Then the lift provided by the actuator must be at least the mass of the battery plus the mass of the payload times gravity:

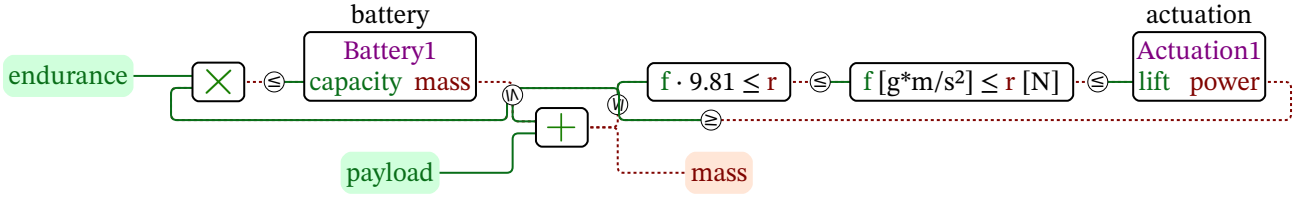
```
constant gravity = 9.81 m/s²
total_mass = (mass required by battery + provided payload)
weight = total_mass · gravity
lift provided by actuation ≥ weight
```

Now there is a loop in the co-design diagram (Fig. 18).

Listing 14: `Composition.mcdp`

```
from library . import model Battery1, Actuation1
mcdp {
    provides endurance [s]
    provides payload [g]

    sub actuation = instance Actuation1
    sub battery = instance Battery1

        # battery must provide power for actuation
    energy = provided endurance * (
        power required by actuation)

    capacity provided by battery ≥ energy

        # actuation must carry payload + battery
    constant gravity = 9.81 m/s²
    total_mass = (mass required by battery +
        provided payload)
    weight = total_mass * gravity
    lift provided by actuation ≥ weight

        # minimize total mass
    requires mass [g]
    required mass ≥ total_mass
}
```

Figure 18.



1.8. Re-usable design patterns using templates

"Templates" are a way to describe reusable design patterns.

For example, the code in Listing 14 composes a particular battery model, called `Battery1`, and a particular actuator model, called `Actuation1`.

However, it is clear that the pattern *interconnect battery and actuators* is independent of the particular battery and actuator selected. MCDPL allows describing this situation by using "templates".

Templates are contained in files with extension `.mcdp_template`. The syntax is:

```
template [name1: interface1, name2: interface2]
mcdp {
        # usual definition here
}
```

In the brackets put pairs of names and NDPs that will be used to specify the interface. Interfaces are contained in files with extension `.mcdp_interface`. For example, Fig. 19 defines an interface with a functionality and a requirement.

ExampleInterface.mcdp_interface

```
interface {
    provides f [ℕ]
    requires r [ℕ]
}
```

Figure 19.

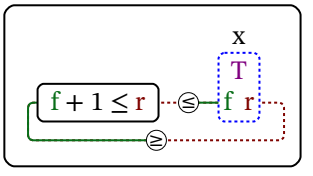
Then we can declare a template as in Listing 15. The template is visualized as a diagram with a hole (Fig. 20).

Listing 15

ExampleTemplate.mcdp_template

```
template [T: `ExampleInterface]
mcdp {
    sub x = instance T
    f provided by x ≥ r required by x + 1
}
```

Figure 20.



Example

Here is the application to the previous example of battery and actuation. Suppose that we define their interfaces as in Listing 16 and Listing 17.

Listing 16: `BatteryInterface.mcdp_interface`

```
interface  {
    provides capacity [J]
    requires mass [g]
}
```

Listing 17: `ActuationInterface.mcdp_interface`

```
interface {
    provides lift [N]
    requires power [W]
}
```

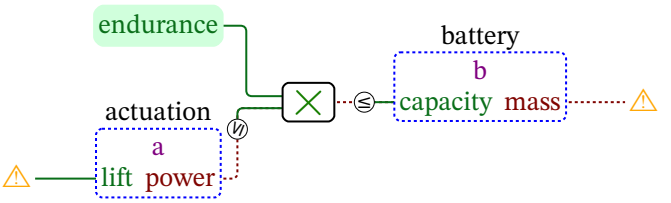
Then we can define a template that uses them. For example the code in Listing 18 specifies that the templates requires two parameters, called `generic_actuation` and

`generic_battery`, and they must have the interfaces defined by `` `ActuationInterface `` and `` `BatteryInterface ``.

Listing 18: `CompositionTemplate.mcdp_template`

```
template [
        a: `ActuationInterface,
        b: `BatteryInterface
]
mcdp {
    sub actuation = instance a
    sub battery = instance b
        # battery must provide power for actuation
    provides endurance [s]
    energy = provided endurance • (power required by actuation)
    capacity provided by battery ≥ energy
        # only partial code
        # ...
}
```

Figure 21.

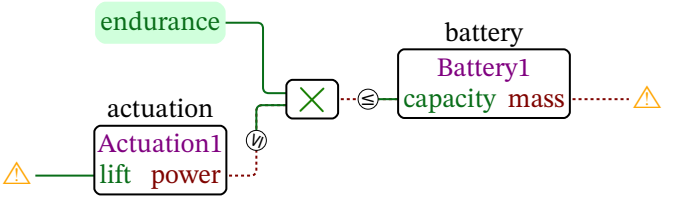


The diagram in Fig. 21 has two "holes" in which we can plug any compatible design problem.

To fill the holes with the models previously defined, we can use the keyword "**`specialize`**" as in Listing 19.

Listing 19: `TemplateUse.mcdp`

```
specialize [
        a: `Actuation1,
        b: `Battery1
] `CompositionTemplate
```



Part B.MCDPL reference

The *Alphorn* is a "labrophone", consisting of a straight multi-meter-long wooden natural horn and a wooden cup-shaped mouthpiece. It is used by mountain dwellers in Swiss alps as musical instruments and communication tools.

2. Introduction

This chapter describes the basic concepts of the MCDPL language.

Fondue is a Swiss dish consisting of melted cheese, served in a *caquelon* (communal pot) over a *réchaud* (portable stove). It is best enjoyed with boiled potatoes, bread, and pickled vegetables.

2.1. Introduction

Kinds

MCDPL has 7 *kinds* of data:

1. *Posets*;
2. *Values* that belong to a poset;
3. *Intervals* of values, also called *uncertain values*;
4. *Primitive DPs*: corresponding to the morphisms in the category **DP**. These are the basic building blocks of the language but they are not used directly by the user.
5. *Named DPs (NDPs)*: these are DPs with functionality and requirements begin tuples of posets with names associated to them. These can be:
   - ▹ *atomic*, often a simple wrapping of a primitive DP.
   - ▹ *composite*, which are defined as the interconnection of other NDPs. These are also called *Monotone Co-Design Problems (MCDPs)*.
6. *Interfaces*: these are the types of the ports of an NDP.
7. *Templates*: These are *diagrams with holes* that can be *specialized* to create NDPs.

Table 2.1 shows the various kinds and examples of expressions belonging to them. The *extension* column shows the file extension used for files containing expressions of that kind.

Table 2.1.: *Kinds* in MCDPL

kind	mathematical concept	extension	code snippet example
Posets	(sub)posets	`.mcdp_poset`	`product(mass: g, volume: m³)`
Values	elements of posets	`.mcdp_value`	`(10 g, 20 l)`
Uncertain values	intervals of values	n/a	`between 10 g and 12 g`
Primitive DPs	morphisms of **DP**	`.mcdp_primitivedp`	`yaml resource('dpc1.dpc.yaml')`

MCDPs	diagrams of **DP** with signal names	`.mcdp`	`mcdp {`
			`  requires r = 10 g`
			`}`

Interfaces	hom-sets of **DP** with signal names	`.mcdp_interfaces`	`interface {`
			`  requires r [g]`
			`  provides f [W]`
			`}`

Templates	operad of **DP** with signal names	`.mcdp_template`	`template [] mcdp {` `}`

3. Posets and values

The “Contra dam” is commonly known as the “Verzasca dam” and is an arch dam on the Verzasca river, in Ticino, Switzerland. The dam, which supports a large hydroelectric power station, is 220 m tall and 380 m long. Fun fact: the dam appeared in James Bond’s 1995 GoldenEye movie, in which a stuntman performed bungee jumping in the opening scene.

poset

3.1. Defining finite posets

It is possible to define custom finite posets in files `*.mcdp_poset` using the keyword **poset**.

For example, to define the poset $\mathbf{Q} = \langle \mathbf{Q}, \preceq_{\mathbf{Q}} \rangle$ with elements

$$\mathbf{Q} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}\} \tag{1}$$

and the order relations

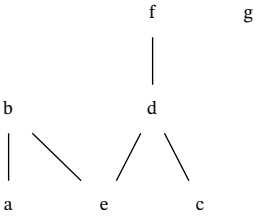
$$\mathtt{a} \preceq_{\mathbf{Q}} \mathtt{b}, \qquad \mathtt{c} \preceq_{\mathbf{Q}} \mathtt{d}, \qquad \mathtt{e} \preceq_{\mathbf{Q}} \mathtt{d} \preceq_{\mathbf{Q}} \mathtt{f}, \qquad \mathtt{e} \preceq_{\mathbf{Q}} \mathtt{b}, \tag{2}$$

we create a file named `Q.mcdp_poset` with code as in Listing 20.

Listing 20: `Q.mcdp_poset`

```
poset {
    a ≤ b
    c ≤ d
    e ≤ d ≤ f
    e ≤ b
    g
}
```

Figure 1.: Hasse diagram of the poset defined in Listing 20



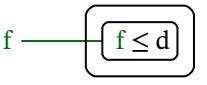
After the poset has been defined, it can be used in the definition of an MCDP, by referring to it by name using the backtick notation, as in "`` `Q ``".

To refer to its elements, use the notation `` `Q: element `` (Listing 21).

Listing 21: `onep.mcdp_poset`

```
mcdp {
    provides f [`Q]
    provided f ≤ `Q: d
}
```

Figure 2.



3.2. Extrema of posets

Top and bottom

`Top` `Bottom`

To indicate top and bottom of a poset, use this syntax:

`Top` *poset*	`⊤` *poset*
`Bottom` *poset*	`⊥` *poset*

For example, `Top ∨` indicates the top of the poset `∨`.

Minimals and maximals

`Minimals` `Maximals`

The expressions `Minimals` *poset* and `Maximals` *poset* denote the set of minimal and maximal elements of a poset.

For example, assume that the poset `MyPoset` is defined as in Fig. 1. Then the expression `Maximals` `` `MyPoset `` is equivalent to the set with two elements `b` and `d`, and the expression `Minimals` `` `MyPoset `` is equivalent to a set with the elements `a`, `e`, `c`.

These assertions can be checked with the following code:

```
assert_equal(Maximals `MyPoset, {`MyPoset: b, `MyPoset: d})
```

```
assert_equal(Minimals `MyPoset, {`MyPoset: a, `MyPoset: e, `MyPoset: c})
```

3.3. Numerical posets

MCDPL works with numerical posets that are subposets of the completion of $\mathbb{R}$, with $-\infty$ and $+\infty$ as bottom and top.

These subposets are parametrized by:

1. A lower bound;
2. An upper bound;
3. A discretization step (possibly 0, to mean no discretization).

You can refer directly to the following subposets:

poset	lower bound	upper bound	discretization
`Nat`	0	$+\infty$	1
`Int`	$-\infty$	$+\infty$	1
`Rcomp`	0	$+\infty$	0

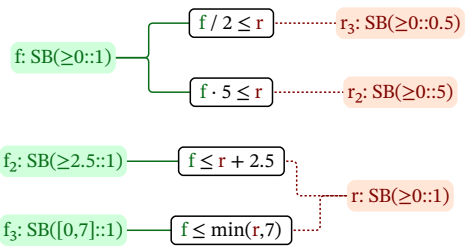
The other variations cannot be accessed directly, but they are derived implicitly by being careful about how the (monotone) operations can transform the first subsets.

Listing 22 shows an example in which some functionality/resource is defined as a natural number, and other derived quantities are inferred to be in subposets.

Listing 22: `simplenumerics.mcdp`

```
mcdp {
    requires r [ℕ]
    provides f [ℕ]

    provides f₂ = required r + 2.5
    provides f₃ = min(required r, 7)
    requires r₂ = provided f * 5
    requires r₃ = provided f / 2
}
```

Figure 3.



Numerical precision

The details of how these posets are represented internally for computation is considered an *implementation detail* and several variations are implemented, including:

- **Default**: Numbers are represented as *decimal values* with 9 digits of precision.
- **Floats**: Numbers are represented as *double precision floating point numbers* (IEEE 754).
- **Intervals**: Numbers are represented as *intervals* of floats or decimals.

Non-default implementations cannot be accessed by the user at this time.

Guarantees of operations

Note that monotone co-design theory does not require that the operations are exact; in particular, it does not require that the operations are associative or commutative. Therefore, some of the usual concerns of numerical analysis do not apply.

It is not a problem if $a + b \neq b + a$. What matters is that the operations are

correct in the sense that they are *conservative*. For example, the operation $a + b$ operating with finite precision needs to be rounded *up* if we are computing a feasible *upper set*, and rounded down if we are computing a feasible *lower set*, as those are *conservative* choices.

3.4. Numbers with units

Numerical values can also have *units* associated to them. So we can distinguish $\overline{\mathbb{R}}_{\geq 0}$[m] from $\overline{\mathbb{R}}_{\geq 0}$[s] and even $\overline{\mathbb{R}}_{\geq 0}$[m] from $\overline{\mathbb{R}}_{\geq 0}$[km].

These posets and their values are indicated using the syntax in Table 3.1.

Table 3.1.: Numbers with units

syntax for posets	`dimensionless`	`g`	`J`	`m/s`
syntax for values	`23`	`1.2 g`	`20 J`	`23 m/s`

In general, units behave as one might expect. Units are implemented using the library Pint; please see its documentation for more information.

The following is the formal definition of the operations involving units.

Units form a group with an equivalence relation.

Call this group of units U and its elements $u, v, \cdots \in U$. By $\mathbb{F}[u]$, we mean a field $\mathbb{F}$ enriched with an annotation of units $u \in U$.

Multiplication is defined for all pairs of units. Let $|x|$ denote the absolute numerical value of x (stripping the unit away). Then, if $x \in \mathbb{F}[u]$ and $y \in \mathbb{F}[v]$, their product is $x \cdot y \in \mathbb{F}[uv]$ and $|x \cdot y| = |x| \cdot |y|$.

Addition is defined only for compatible pairs of units (e.g., `m` and `km`), but it is not possible to sum, say, `m` and `s`. If $x \in \mathbb{F}[u]$ and $y \in \mathbb{F}[v]$, then $x + y \in \mathbb{F}[u]$, and $x + y = |x| + \alpha_v^u |y|$, where α_v^u is a table of conversion factors, and $|x|$ is the absolute numerical value of x.

In practice, this means that MCDPL thinks that `1 kg + 1 g` is equal to `1.001 kg`. Addition is not strictly commutative, because `1 g + 1 kg` is equal to `1001 g`, which is equivalent, but not equal, to `1.001 kg`.

http://pint.readthedocs.org/

3.5. Poset products

MCDPL allows the definition of finite Cartesian products, which can be anonymous or named.

Anonymous Poset Products

Use the Unicode symbol "×" or the simple letter "x" to create a poset product, using the syntax:

```
P1 x P2 x ... x Pn
```

For example, the expression `J × A` represents a product of Joules and Amperes.

The elements of a poset product are tuples. To define a tuple, use angular brackets "<" and ">".

For example, the expression "`<2 J, 1 A>`" denotes a tuple with two elements, equal to `2 J` and `2 A`. An alternative syntax uses the fancy Unicode brackets "⟨" and "⟩", as in "`⟨Ø J, 1 A⟩`".

Tuples can be nested. For example, you can describe a tuple like

```
⟨⟨Ø J, 1 A⟩, ⟨1 m, 1 s, 42⟩⟩,
```

and its poset is denoted as

```
(J × A) × (m × s × ℕ).
```

Named Poset Products

`product`

MCDPL also supports "named products". These are semantically equivalent to products, however, there is also a name associated to each entry. This allows to easily refer to the elements. For example, the following declares a product of the two spaces `J` and `A` with the two entries named "energy" and "current".

```
product(energy: J, current: A)
```

The names for the fields must be valid identifiers (starts with a letter, contains letters, underscore, and numbers).

The attribute can be referenced using the following syntax:

```
(requirement).field
```

For example:

```
mcdp {
    provides out [product(energy: J, current: A)]

    (provided out).energy ≤ 10 J
    (provided out).current ≤ 2 A
}
```

3.6. Posets of subsets

powerset

Power sets

MCDPL allows to describe the set of subsets of a poset, i.e. its *power set*.

The syntax is either of these:

`powerset(p)` `℘(P)`

To describe the values in a powerset (subsets), use this set-building notation:

`{value, value, ..., value}`

For example, the value `{1, 2, 3}` is an element of the poset `powerset(Nat)`.

EmptySet

The empty set

To denote the empty set, use the keyword **`EmptySet`** or the equivalent unicode:

`EmptySet P` `∅ P`

Note that empty sets are typed. This is different from set theory: here, a set of apples without apples and a set of oranges without oranges are two different sets, while in traditional set theory they are the same set.

`∅ J` is an empty set of energies, and `∅ V` is an empty set of voltages, and the two are not equivalent.

3.7. Upper and lower sets

UpperSets

The poset of upper sets of **P** can be described by the syntax

```
UpperSets(P)
```

For example, `UpperSets(ℕ)` is the poset of upper sets for the natural numbers.

upperclosure

To describe an upper set (i.e. an element of the space of upper sets), use the keyword **upperclosure** or its abbreviation ↑. The syntax is:

```
upperclosure set   ↑ set
```

For example: `↑{(2 g, 1 m)}` denotes the principal upper set of `(2 g, 1 m)` in the poset `g x m`.

LowerSets

Similarly you can define the poset of lowersets of **P** as:

```
LowerSets(P)
```

lowerclosure

and you can obtain elements of this poset using the keyword **lowerclosure** or its abbreviation ↓.

```
lowerclosure set   ↓ set
```

between

3.8. Defining uncertain constants

MCDPL allows describing interval uncertainty for variables and expressions.

There are three syntaxes accepted:

1. The first is using explicit bounds:

   ```
   x = between lower bound and upper bound
   ```

2. Median plus or minus absolute tolerance:

   ```
   x = median +- tolerance
   ```
   ```
   x = median ± tolerance
   ```

3. Median plus or minus percent:

   ```
   x = median +- percent tolerance %
   ```
   ```
   x = median ± percent tolerance %
   ```

For example, Table 3.2 shows the different ways in which a constant can be declared to be between `9.95 kg` and `10.05 kg`:

Table 3.2.: Equivalent ways to declare an uncertain constant

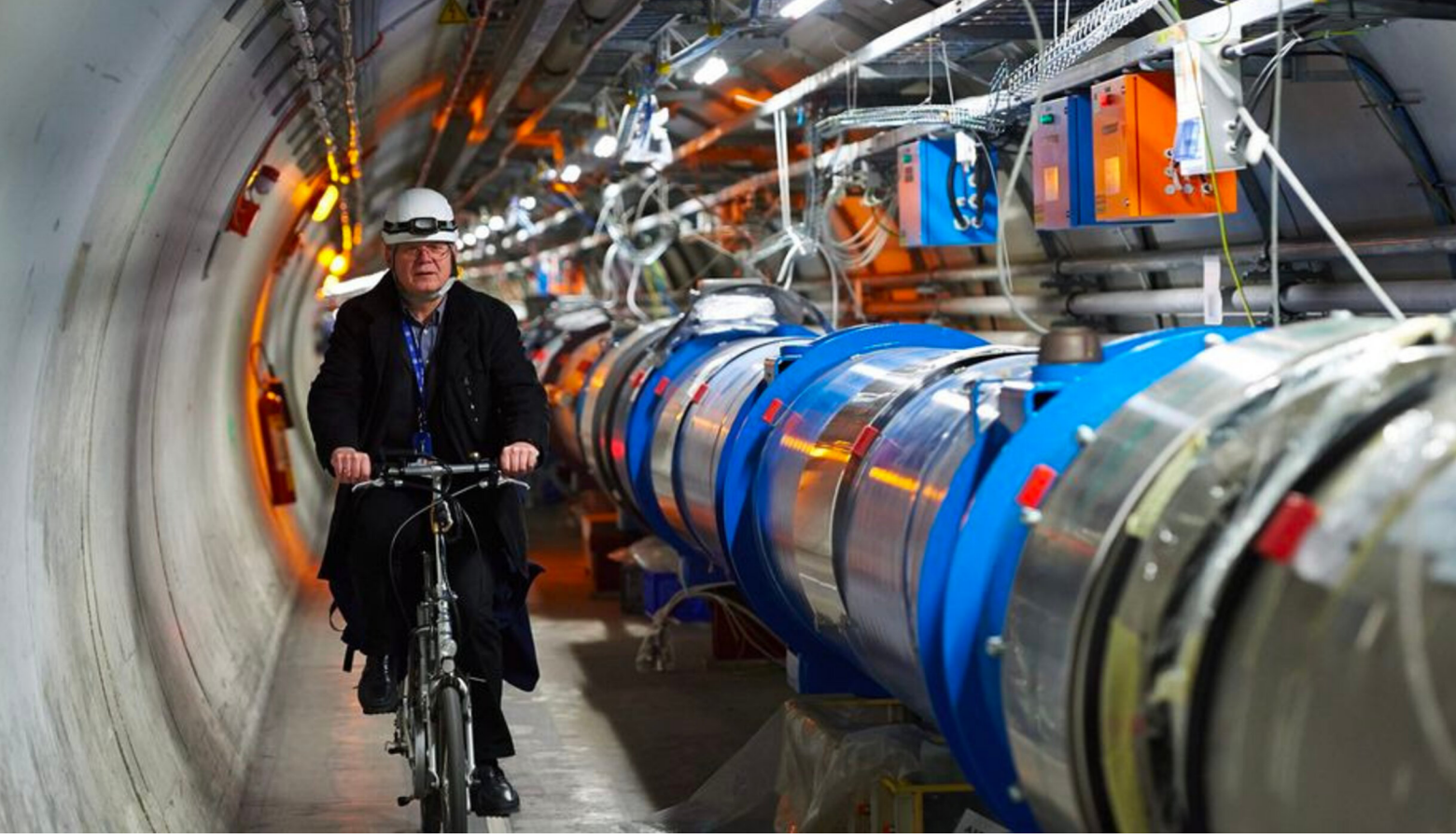
```
x = between 9.95 kg and 10.05 kg
```

```
x = 10 kg ± 50 g
```

```
constant c = 10 kg
δ = 50 g
x = between c - δ and c + δ
```

It is also possible to describe parametric uncertain relations by simply using uncertain constants in place of regular constants:

```
mcdp {
    provides energy [J]
    requires mass [kg]

    specific_energy = between 100 kWh/kg and 120 kWh/kg
    required mass · specific_energy ≥ provided energy
}
```



4. Named DPs

The *Large Hadron Collider* (*LHC*) is the world's largest and highest-energy particle collider. It is located near Geneva and it was built by the European Organization for Nuclear Research (CERN) between 1998 and 2008.

4.1. Defining NDPs

An NDP (*named* design problem) is a design problem that has associated names for its functionality and requirements.

There are several ways of defining an NDP:

1. by constructing one using the `catalog` syntax;
2. by constructing one using the `mcdp` syntax;
3. by constructing one using the `dp` syntax;
4. by specializing a template, using the `specialize` keyword.
5. by constructing one using operations such as `compact`, `abstract`, etc.

4.2. Constructing NDPs as catalogs

An NDP can be created using a "catalog" declaration, which enumerates the possible ways to provide functionalities and requirements explcitly.

A catalog declaration is comprised of zero or more functionality/requirement declarations and zero or more *catalog records*:

```
catalog {
    provides f1 [W]
    requires r1 [s]

        # records go here
}
```

There are two types of records that can be used (but cannot be mixed):

1. Records that specify the implementation names explicitly;
2. Records that do not specify the implementation names implicitly.

```
catalog {
    provides f1 [W]
    requires r1 [s]

    10 W ↤ imp1 ↦ 10 s
    20 W ↤ imp2 ↦ 20 s
}
```

```
catalog {
    provides f1 [W]
    requires r1 [s]

    10 W ↔ 10 s
    20 W ↔ 20 s
}
```

For multiple functionalities and resources, partition the elements using commas:

```
catalog {
    provides f1 [W]
    provides f2 [m]
    requires r1 [s]
    requires r2 [s]

    5 W, 5 m ↤ imp1 ↦ 10 s, 10 s
}
```

Note that in the catalog rows, you must use units, which might be different from the units used in the declaration of the functionalities and requirements. For example, in the following we use standard units (meters and seconds) for the catalog rows, but we use different units in the declaration of the functionalities and requirements (miles and hours):

```
catalog {
    provides distance [m]
    requires duration [s]
    5 miles ↔ 10 hours
}
```

In case there are no functionalities, use an empty tuple on the left; and if there are no requirements, use an empty tuple on the right.

```
catalog {
    requires r1 [s]
    () ↤ imp1 ↦ 10 s
}
```

```
catalog {
    provides f1 [s]
    5s ↤ imp1 ↦ ()
}
```

True and false

The empty catalog is valid: it is the NDP with no functionality and no requirements and no implementations. That is:

```
catalog {
}
```

This is “false”: even though nothing is asked, there is no way to provide it.

Dually, the following catalogs definitions are equivalent to “true”. One has an anonymous implementation, the other has an explicit implementation.

```
catalog {
    () ↤ imp1 ↦ ()
}
```

```
catalog {
    () ↔ ()
}
```

Of course, we can create a catalog that is *even more true*, by adding more implementations:

```
catalog {
    () ↤ even    ↦ ()
    () ↤ more    ↦ ()
    () ↤ ways    ↦ ()
    () ↤ to      ↦ ()
    () ↤ provide ↦ ()
    () ↤ nothing ↦ ()
}
```

4.3. Constructing NDPs from YAML files

dp

An NDP can be created using a “dp” declaration, which is used to import from external YAML data.

The syntax is as follows:

```
dp {
        # usual provides/requires declaration

        # only one line of this form
    implemented-by yaml resource(”data.yaml”)
}
```

For example:

```
dp {
    provides task [`task]
    requires sensor_requirements [`sensor_reqs]
    requires vehicle_properties [`vehicle_properties]
    requires computation [ops]
    requires discomfort [dimensionless]

    implemented-by yaml resource(”planner_catalog-Mar-12-2024-08-24.dpc.yaml”)
}
```

An example YAML file is as follows:

```
F:
- "`task"
R:
- "`sensor_reqs"
- "`vehicle_properties"
- "ops"
- "dimensionless"
implementations:
  planning_4125:
    f_max:
    - "`task: task_urban_car_cr_AA_50_3"
    r_min:
    - "`sensor_reqs: sensor_reqs_4125"
    - "`vehicle_properties: car_van_chrycler_pacifica"
    - "6.00 ops"
    - "13 dimensionless"
  planning_4102:
    f_max:
    - "`task: task_urban_car_cr_AA_30_3"
    r_min:
    - "`sensor_reqs: sensor_reqs_4102"
    - "`vehicle_properties: car_van_chrycler_pacifica"
    - "6.00 ops"
    - "11 dimensionless"
```

The files should contain three fields: `F`, `R`, and `implementations`.

The first two fields are lists of strings. These represent, in MCDPL, the names of the interfaces that the NDP requires and provides.

The the third field is a dictionary, where the keys are the names of the implementations, and the values are dictionaries with two keys: `f_max` and `r_min`. These are lists of strings, representing the requirements and capabilities of the implementation in MCDPL.

mcdp

4.4. Describing Monotone Co-Design Problems

An NDP can be created using a declaration enclosed in the `mcdp` construct. The declaration must be comprised of an optional comment and zero or more statements; a statement can be either a declaration or a constraint, and these can be mixed.

provides, requires

Declaring functionality and requirements explicitly

A functionality declaration is of this form:

```
provides f [poset]
```

Symmetrically, a requirement declaration is of this form:

```
requires r [poset]
```

Declaring functionality and requirements implicitly using expressions

There are shortcuts one can use to declare functionalities and requirements given a value:

```
provides f ≤ expression
```

This is equivalent to:

```
provides f [poset]
provided f ≤ expression
```

Symmetrically, there are the same constructs for defining requirements:

```
requires r ≥ expression
```

This is equivalent to:

```
requires r [poset]
required r ≥ expression
```

using, for

Forwarding functionalities and requirements from other subproblems

The second shortcut is used to declare one or more functionalities and specify which DP is responsible for providing them:

```
provides f1, f2, ... using dp
```

This is equivalent to

```
provides f1 ≤ f1 provided by dp
provides f2 ≤ f2 provided by dp
```

The symmetric construct is used to declare one or more requirements and specify which DP is responsible for providing them:

```
requires r1, r2, ... for dp
```

which is equivalent to:

```
requires r1 ≥ r1 required by dp
requires r2 ≥ r2 required by dp
```

Note that the syntax is "requires … for …" rather than "provides … using …".

Declaring functionality and requirements using interfaces

Another way to declare functionalities and requirements is to declare that one is implementing a certain interface.

For example, given the following interface defition:

```
interface  {
    provides f [W]
    requires r [g]
}
```

We can use the keyword `implements` to declare that the current MCDP is implementing the interface:

```
mcdp {
    implements `MyInterface

    provided f ≤ 10 W
    required r ≥ 1 kg
}
```

Ignoring functionality/requirements of subproblems

`ignore`

The `ignore` command allows to "ignore" some of the functionality or requirements of an NDP. We "ignore" them by connecting them to DPs that will make the constraints always true.

Suppose the functionality `f` has type F. The syntax

```
ignore f provided by dp
```

is equivalent to

```
f provided by dp ≥ any-of(Minimals F)
```

Equivalently, suppose the requirement `r` has type R. The syntax

```
ignore r required by dp
```

is equivalent to

```
r required by dp ≤ any-of(Maximals R)
```

ignore, propagate unconnected

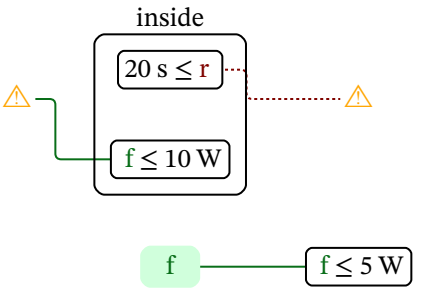
Ignore/propagate unconnected ports

There are two special declarations that can be used to ignore or propagate unconnected ports.

`ignore unconnected` `propagate unconnected`

Consider the following example, in which there is a subproblem called `inside` that provides and requires some ports, but these ports are not connected to anything:

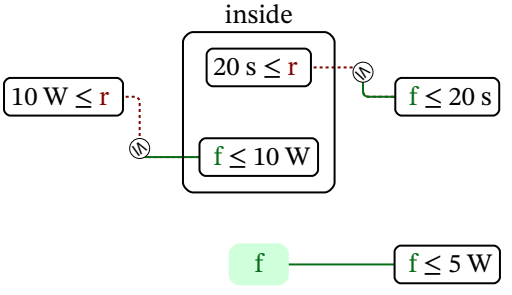
```
mcdp {
    provides f ≤ 5 W

    sub inside = instance mcdp {
            provides f_sub ≤ 10 W
            requires r_sub ≥ 20 s
        }

        # f_sub, r_sub are unconnected
}
```



In the editor there will be warnings about unconnected ports.

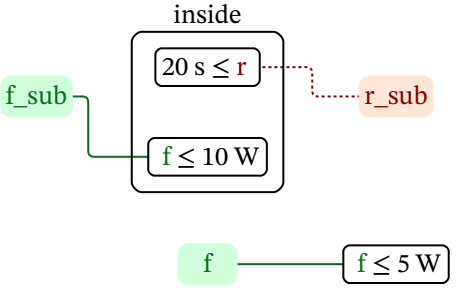
If we add the declaration `ignore unconnected`, then the unconnected ports will be ignored; they will be connected to a special DP that is always true.

```
mcdp {
    provides f ≤ 5 W

    sub inside = instance mcdp {
            provides f_sub ≤ 10 W
            requires r_sub ≥ 20 s
        }

    ignore unconnected
}
```



If we add the declaration `propagate unconnected`, then the unconnected ports will be propagated to the parent DP.

```
mcdp {
    provides f ≤ 5 W

    sub inside = instance mcdp {
            provides f_sub ≤ 10 W
            requires r_sub ≥ 20 s
        }

    propagate unconnected
}
```



Shortcuts for summing over functionalities and requirements

One common pattern is to sum over many requirements of the same name. For example, a design might consist of several components, and the budgets must be summed together (Listing 23). In this case, it is possible to use the shortcut `sum` (or its equivalent symbol $\sum$) that allows to sum over requirements required with the same name (Listing 24) with the syntax

```
sum budget required by *
```

An error will be generated if there are no subproblems with a requirement of the given name.

The two MCDP in Listing 23 and Listing 24 are equivalent.

Listing 23: `sumc-example.mcdp`

```
mcdp {
    requires total_budget [USD]

    sub c1 = instance `Component1
    sub c2 = instance `Component2
    sub c3 = instance `Component3

    required total_budget ≥ (
budget required by c1 +
budget required by c2 +
budget required by c3
)
}
```

Listing 24: `sumc2.mcdp`

```
mcdp {
requires total_budget [USD]

c1 = instance `Component1
c2 = instance `Component2
c3 = instance `Component3

# this sums over all components
required total_budget >= sum budget required by *

}
```

The dual syntax for functionality is also available (Listings 25 and 26).

Listing 25: `sumc3.mcdp`

```
mcdp {
    provides total_power [W]

    sub g1 = instance `Generator1
    sub g2 = instance `Generator2
    sub g3 = instance `Generator3

    provided total_power ≤ (
power provided by g1 +
power provided by g2 +
power provided by g3)
}
```

Listing 26: `sumf3.mcdp`

```
mcdp {
provides total_power [W]

g1 = instance `Generator1
g2 = instance `Generator2
g3 = instance `Generator3

provided total_power <= sum power provided by *

}
```

4.5. Mathematical relations between functionalities and requirements

Once functionalities and resources are defined, it is possible to define relations between them by using various mathematical operations.

The available math relations are shown in Table 4.1.

Table 4.1.: Math relations available

algebra	addition
	multiplication
	division (by a constant)
	substraction (of a constant)
ceil/floor	`ceil`
	`ceil0`
	`floor`
	`floor0`
exponentiation	`a^2`
	`sqrt`
	`pow`
min/max	`max`
	`min`
approximation	`approx`

One thing to keep in mind is that these are not “functions”; rather, they are *relations*.

Abstract interpretation

The following examples (Listings 27 and 28) show how the interpreter is able to use *abstract interpretation* to infer the type of the result of the operations.

In Listing 27 we start with one requirement defined to be a a natural number, and then we define several functionalities that are the result of applying different operations to that requirement. The interepreter is able to to infer the subposet of numbers that are possible feasible values for each relation.

For example, for the line

```
provides f4 = r + 2.5
```

the interpreter is able to infer that the smallest set that contains the optimal feasible functionalities for the relation is the set of numbers of the form $r + 2.5$ where r is a natural number. This is written compactly in the figure as `>=2.5::1`.

Likewise, for the line

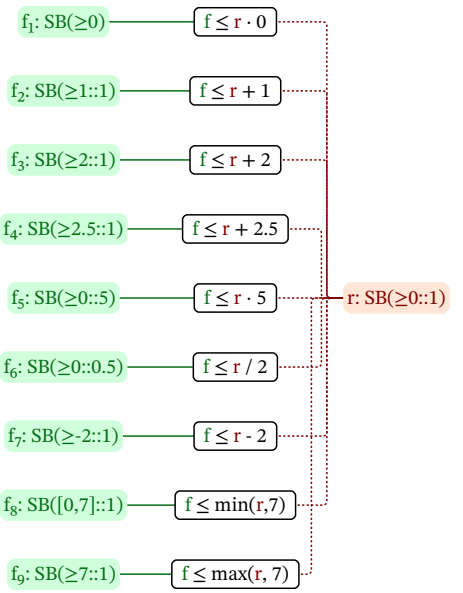
```
provides f8 = min(r, 7)
```

the interpreter is able to infer that the smallest set that contains the optimal feasible functionalities for the relation is the set of integers from 0 to 7, written compactly in the figure as `[0,7]::1`.

Listing 27: `numerics2.mcdp`

```
mcdp {
    requires r [ℕ]
    provides f1 = required r * 0
    provides f2 = required r + 1
    provides f3 = required r + 2.0
    provides f4 = required r + 2.5
    provides f5 = required r * 5
    provides f6 = required r / 2
    provides f7 = required r - 2
    provides f8 = min(required r, 7)
    provides f9 = max(required r, 7)
}
```

Figure 1.

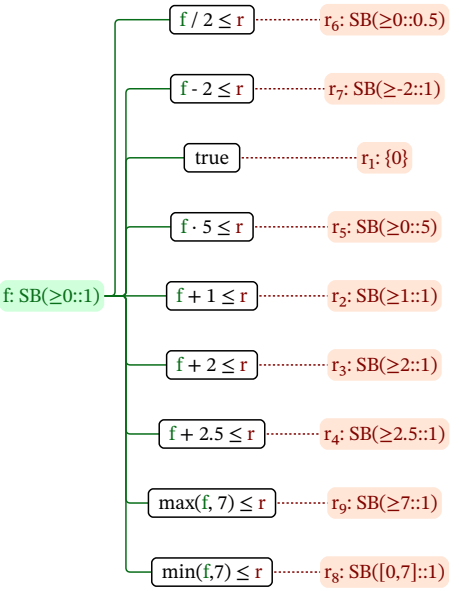


The next example is symmetric, with the functionality being a natural number and the requirements being the result of applying different operations to that functionality.

Listing 28: `numerics.mcdp`

```
mcdp {
    provides f [ℕ]
    requires r₁ = provided f • 0
    requires r₂ = provided f + 1
    requires r₃ = provided f + 2.0
    requires r₄ = provided f + 2.5
    requires r₅ = provided f • 5
    requires r₆ = provided f / 2
    requires r₇ = provided f - 2
    requires r₈ = min(provided f, 7)
    requires r₉ = max(provided f, 7)
}
```

Figure 2.



Min and max

min max

In most cases, we can think of the min and max symbols as the usual mathematical functions. (Corresponding to the "meet" and "join" operations in lattice theory.)

However, in the context of MCDPs, the semantics is more subtle, and it holds even when the poset in question is not a lattice, so that the meet and join are not necessarily defined.

The following table shows the semantics of the min and max operations in MCDPs.

MCDPL expression	Equivalent semantics
`r ≥ min(a, b)`	$(r \geq a) \vee (r \geq b)$
`r ≥ max(a, b)`	$(r \geq a) \wedge (r \geq b)$
`f ≤ min(c, d)`	$(f \leq c) \wedge (f \leq d)$
`f ≤ max(c, d)`	$(f \leq c) \vee (f \leq d)$

If the poset is a lattice, then the semantics of the min and max operations are the same as the meet and join operations.

Consider the following poset with two elements:

Listing 29: `discrete.mcdp_poset`

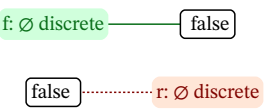
```
poset {
    a b
}
```

Because it is a discrete poset, the meet and join of the two elements do not exist. So, $\min(a, b)$ and $\max(a, b)$ are not defined if we consider them as mathematical functions. However, using the relation semantics we can write the following MCDP. We can see in the figure that the relation is reduced to the false relation.

Listing 30: `exampleminmax1.mcdp`

```
mcdp {
    provides f [`discrete]
    requires r [`discrete]
    provided f ≤ min(`discrete: a, `discrete: b)
    required r ≥ max(`discrete: a, `discrete: b)
}
```

Figure 3.



floor floor0 ceil ceil0

Floor and ceil relations

The `floor` and `ceil` operations are used to round down and up, respectively.

The `floor0` and `ceil0` are variants, defined as follows:

$$\text{floor}_0(x) = \begin{cases} 0 & \text{for } x = 0 \\ \text{ceil}(x-1) & \text{for } x > 0 \end{cases} \tag{1}$$

$$\text{ceil}_0(x) = \begin{cases} 0 & \text{for } x = 0 \\ \text{floor}(x+1) & \text{for } x > 0 \end{cases} \tag{2}$$

The functions floor and floor_0 agree everywhere except at nonzero integers. For example, $\text{floor}(2) = 2$ and $\text{floor}_0(2) = 1$. Likewise, ceil and ceil_0 agree everywhere, except at nonzero integers: $\text{ceil}(2) = 2$ and $\text{ceil}_0(2) = 3$.

The reason we need these extra operations is that relations need to be upper-semicontinuous if acting on requirements, and lower-semicontinuous if acting on functionalities.

In fact, we know that floor is upper semi-continuous:

$$\lim_{x \to 3^+} \text{floor}(x) = 3 \tag{3}$$

but that it is not lower semi-continuous:

$$\lim_{x \to 3^-} \text{floor}(x) = \text{does not exist} \tag{4}$$

The variant floor_0 is lower semi-continuous:

$$\lim_{x \to 3^-} \text{floor}_0(x) = 3 \tag{5}$$

but it is not upper semi-continuous:

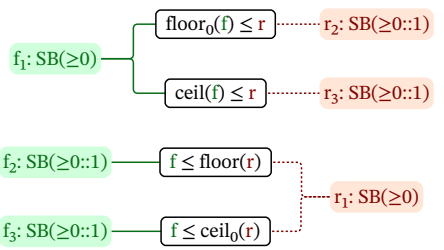
$$\lim_{x \to 3^+} \text{floor}_0(x) = \text{does not exist} \tag{6}$$

The following example shows the use of these operations.

Listing 31: `numerics3.mcdp`

```
mcdp {
    requires r1 [dimensionless]
    provides f1 [dimensionless]
    provides f2 = floor(required r1)
    provides f3 = ceil0(required r1)
    requires r2 = floor0(provided f1)
    requires r3 = ceil(provided f1)
}
```

Figure 4.



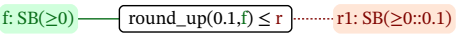
Built-in approximations

The `approx` keyword is used to discretize a signal.

Listing 32: `approx1.mcdp`

```
mcdp {
    provides f [dimensionless]

    requires r1 = approx(provided f, 0.1)
}
```

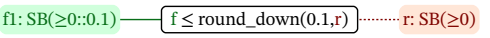
Figure 5.



Listing 33: `approx2.mcdp`

```
mcdp {
    requires r [dimensionless]

    provides f1 = approx(required r, 0.1)
}
```

Figure 6.



take

4.6. Accessing elements of a product

We have seen in Section 3.5 how to define anononymous and named products.

Accessing elements of an anonymous product

The "take" operation allows us to access the elements of a product. The syntax is:

```
take(signal, index)
```

For example:

```
mcdp {
    provides out [J x A]

    take(provided out, 0) ≤ 10 J
    take(provided out, 1) ≤ 2 A
}
```

This is equivalent to

```
mcdp {
    provides out [J x A]
    provided out ≤ (10 J, 2 A)
}
```

Accessing elements of a named product by name

If the product is a named product, it is possible to index those entries using one of these two syntaxes:

```
take(requirement, label)
```

```
take(functionality, label)
```

There is also a syntax with the dot, reminiscent of the syntax used in object-oriented languages:

```
(requirement).label
```

```
(functionality).label
```

For example:

```
mcdp {
    provides out [product(energy: J, current: A)]

    (provided out).energy ≤ 10 J
    (provided out).current ≤ 2 A
}
```

4.7. Operations on NDPS

The software allows the manipulation of NDPs using a set of operations.

These are the operations possible on NDPs:

Abstraction	`abstract` *NDP*
Compactification	`compact` *NDP*
Flattening	`flatten` *NDP*
Canonical form	`canonical` *NDP*

compact

Compactification

The construct `compact` takes an NDP and produces another in which parallel edges are compacted into one edge. This is one of the steps required for the solution of the MCDP.

The syntax is:

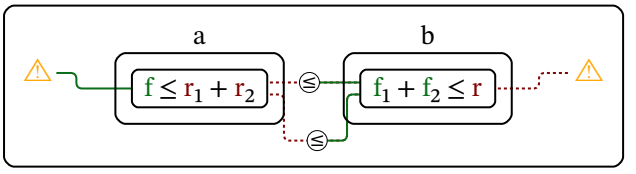
```
compact NDP
```

For every pair of NDPS that have more than one edge between them, those edges are being replaced.

For example, consider this MCDP:

Listing 34: compact_example_1.mcdp

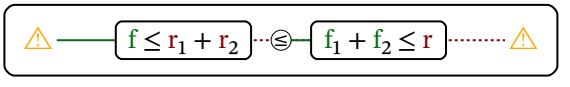
```
mcdp {
    sub a = instance mcdp {
            provides f [ℕ]
            requires r1 [ℕ]
            requires r2 [ℕ]

            provided f ≤ (required r1
 + required r2)
        }
    sub b = instance mcdp {
            provides f1 [ℕ]
            provides f2 [ℕ]
            requires r [ℕ]

            required r ≥ (provided f1
 + provided f2)
        }
    r1 required by a ≤ f1 provided by b
    r2 required by a ≤ f2 provided by b
}
```



The compacted version has only one edge between the two NDPs, with the corresponding poset being the product of the two posets:

Listing 35: compact_example_2.mcdp

```
compact `compact_example1
```



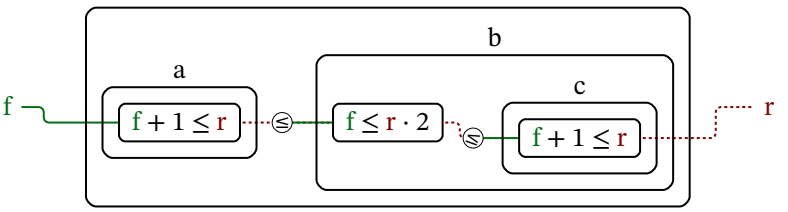
Flattening

flatten

It is easy to create recursive composition in MCDPL, in which we have NDPs that contain other NDPs.

Listing 36: `Composition1.mcdp`

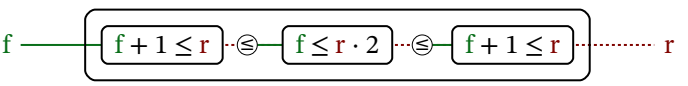
```
mcdp {
    T = mcdp {
            provides f [ℕ]
            requires r [ℕ]
            provided f + 1 ≤ required r
        }
    U = mcdp {
            provides f [ℕ]
            requires r [ℕ]
            sub c = instance T
            provided f ≤ 2 • f provided by c
            required r ≥ r required by c
        }
    sub a = instance T
    sub b = instance U
    r required by a ≤ f provided by b
    requires r for b
    provides f using a
}
```



The "flattening" operation erases the borders between subproblems.

Listing 37: `Composition1_flattened.mcdp`

```
flatten `Composition1
```



abstract

Abstraction

The command **abstract** takes an NDP and creates another NDP that forgets the internal structure.

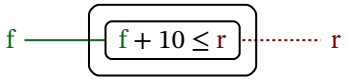
The syntax is:

```
abstract NDP
```

The resulting NDP is guaranteed to be equivalent to the initial one, but the internal structure is forgotten.

Listing 38: m0.mcdp

```
mcdp {
    provides f [m]
    requires r [m]
    required r ≥ provided f + 10 m
}
```



Listing 39: m0_abstracted.mcdp

```
abstract `m0
```

canonical

Canonical form

This puts the MCDP in a canonical form:

```
canonical NDP
```

The canonical form is obtained by compacting all the loops into one single loop.

4.8. Other ways to compose NDPs

choose

Union/choice of design problems

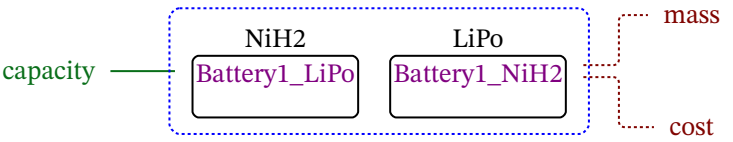
The “choice” construct allows describing the idea of "alternatives".

The syntax is as follows:

Listing 40: `Batteries2.mcdp`

```
choose(
NiH2: `Battery1_LiPo,
LiPo: `Battery1_NiH2
)
```

Figure 7.



For a full example, see Section 1.6.

5. Second-order modeling

"Krampus" is a horned figure, which, in Alpine folklore, during the Christmas season, scares children who have misbehaved, helping Saint Nicholas. More recently, the character has been imported in Hollywood horror films, and has become part of the American popular culture.

5.1. Interfaces

A template has parameters that are constrained to have a certain *interface*.

In MCDPL, interfaces are first-class citizens. They reside in `.mcdp_interface` files, and can be imported and used in other libraries just like the other top-level entities.

and are defined using the `interface` keyword.

`interface`

Declaring interfaces

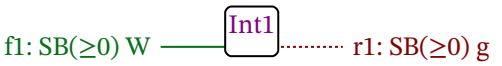
The syntax for declaring an interface is a subset of the syntax for declaring a MCDP. Only the definitions of functionality and resources are allowed.

For example, this interface declares one functionality and one resource:

Listing 41: `Int1.mcdp_interface`

```
interface {
    provides f1 [W]
    requires r1 [g]
}
```

Figure 1.



`extends`

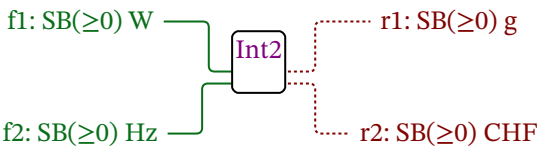
Extending interfaces

It is possible to *extend* an interface by adding more functionality or resources. This is done using the `extends` keyword.

Listing 42: `Int2.mcdp_interface`

```
interface {
    extends `Int1
    provides f2 [Hz]
    requires r2 [CHF]
}
```

Figure 2.



`implements`

Using interfaces when defining models

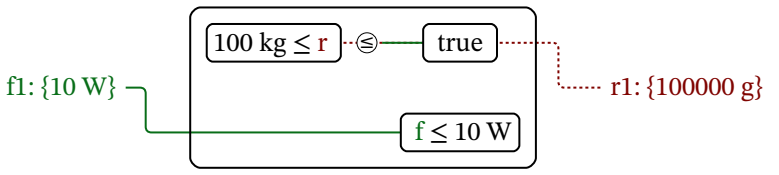
When defining a model, you can specify that it *implements* an interface. This means that the model must provide the functionality and resources required by the interface.

This is done using the `implements` keyword:

Listing 43: `Impl1.mcdp`

```
mcdp {
    implements `Int1
    provided f1 ≤ 10 W
    required r1 ≥ 100 kg
}
```

Figure 3.



5.2. Templates

Templates are contained in files with extension `.mcdp_template`.

Declaring templates

`template`

The syntax uses the keyword "`template`". It is followed by square brackets, which specify the names of the interfaces for the template holes.

```
template [name1: interface1, name2: interface2]
mcdp {
        # usual definition here
}
```

For example, we can define the following simple interface:

Listing 44: Scalar.mcdp_interface

```
interface {
    provides f [dimensionless]
    requires r [dimensionless]
}
```

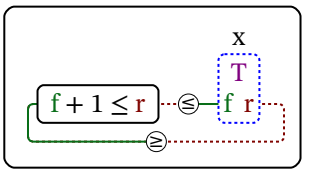
Figure 4.

And then we can define a template that uses this interface:

Listing 45: Loop1.mcdp_template

```
template [T: `Scalar]
mcdp {
    sub x = instance T
    f provided by x ≥ r required by x + 1
}
```

Figure 5.



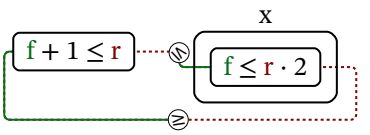
Instantiating templates

`specialize`

To instantiate a template, we need to specify the models for its holes. We use the keyword "`specialize`" as follows:

Listing 46: Loop1Instance.mcdp

```
specialize [
       T: mcdp {
           implements `Scalar
           provided f ≤ 2 • required r
       }
] `Loop1
```



6. Queries

Switzerland produces timepieces since the 14th century. Indeed, a “Swiss made” watch is very unique: according to official rules, the mechanics, casing, and inspection of a watch must be carried out in Switzerland to earn the hallmark. Today, Switzerland is the world’s largest watch exporter, counting over 60,000 employees and exporting over $ 13.7 billion worth products.

6.1. Defining queries

Queries are specified in files with extension `.mcdp_query`. This file contains a YAML-document that specifies:

- the model that the query is about;
- the type of query;
- the parameters of the query;

There are two types of queries: `FixFunMinRes` and `FixResMaxFun`.

FixFunMinRes

This query type is used to find the minimum amount of requirements required to achieve a certain functionality.

Consider a model `query1_model` that has 2 functionalities and 2 requirements:

```
mcdp {
    provides f1 [m]
    provides f2 [m]

    requires r1 [m]
    requires r2 [m]

    provided f1 ≤ floor(r1)
    r1 ≥ 1 m
    r2 ≥ f2 + 1 m
}
```

A query that uses this model could look like this:

```
title: Query 1
description: ''
model: "`query1_model"
query:
    query_type: FixFunMinRes
    min_f:
        f_1: '1 m'
        f_2: '2 m'
    max_r:
        r_1: '3 m'
        r_2: '4 m'
    optimize_for: [r_1]
```

The fields `title` and `description` are just used for metadata.

The `model` field should be a string that evaluates to a model.

The `query` field specifies the details of the query:

- The field `type` should be set to `FixFunMinRes`.
- The field `min_f` is a dictionary that specifies the minimum values for the functionalities.
- The field `max_r` is a dictionary that specifies the upper bound for the requirements.
- The field `optimize_for` is a list that specifies the resources to optimize for.

In this case, the query is equivalent to this optimization problem:

$$\begin{aligned} \min \quad & r_1 && (1)\\ \text{such that} \quad & \text{query1\_model}(\langle 1\text{m}, 2\text{m}\rangle, \langle r_1, r_2\rangle) \text{ is feasible} && (2)\\ & r_1 \leq 3\text{m} && (3)\\ & r_2 \leq 4\text{m} && (4) \end{aligned}$$

We are minimizing the value of `r_1`, which is the only element in the `optimize_for` list. The value of `r_2` is not relevant, although it is still bounded.

FixResMaxFun

This query type is used to find the maximum amount of functionalities that can be achieved with a certain amount of resources.

The syntax is exactly similar as the previous type of query:

```
title: Query 2
description: ''
model: "`query1_model"
query:
    query_type: FixResMaxFun
    min_f:
        f_1: '1 m'
        f_2: '2 m'
    max_r:
        r_1: '3 m'
        r_2: '4 m'
    optimize_for: [f_1]
```

The query is equivalent to this optimization problem:

$$\begin{aligned} \max \quad & f_1 && (5)\\ \text{such that} \quad & \text{query1\_model}(\langle f_1, f_2\rangle, \langle 3\text{m}, 4\text{m}\rangle) \text{ is feasible} && (6)\\ & f_1 \geq 1\text{m} && (7)\\ & f_2 \geq 2\text{m} && (8) \end{aligned}$$

7. Syntax

7.1. MCDPL syntax

Characters

A MCDP file is a sequence of Unicode code-points that belong to one of the classes described in Table 7.1. All files are assumed to be encoded in UTF-8.

Table 7.1.: Character classes

class	characters
Latin letters	`abcdefghijklmnopqrstuvxwyz`
	`ABCDEFGHIJKLMNOPQRSTUVXWYZ`
Underscore	`_`
Greek letters	$\alpha\beta\gamma\delta\epsilon\zeta\eta\theta\iota\kappa\lambda\mu\nu\xi\pi\rho\sigma\tau\upsilon\phi\chi\psi\omega$
	$\Gamma\Delta\Theta\Lambda\Xi\Pi\Sigma\Upsilon\Phi\Psi\Omega$
Digits	`0123456789`
Superscripts	x^1 x^2 x^3 x^4 x^5 x^6 x^7 x^8 x^9
Subscripts	x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9
Comment delimiter	`#`
String delimiters	`'"`
Backtick	`` ` ``
Parentheses	`[](){}`
Operators	`<= =>` $\preceq\succeq$, $\preceq\succeq$
Tuple-making	`< >` $\langle\rangle$
Arrows glyphs	`<--| <--> |-->` $\mapsto \leftrightarrow \mapsto$
Math	`= . * - + ^`
Other glyphs	$\times \top \bot \mathcal{P} \mathbb{N} \mathbb{R} \mathbb{Z} \sum \pm \uparrow \downarrow \infty \emptyset \$$

Comments

Comments work as in Python. Anything between the symbol `#` and a newline is ignored. Comments can include any Unicode character.

Reserved keywords

The reserved keywords are shown in Table 7.2.

Syntactic equivalence

MCDPL allows a number of Unicode glyphs as abbreviations of a few operators.

For example, every occurrence of a superscript of the digit d is interpreted as a power `^d`. It is syntactically equivalent to write `x^2` or x^2.

Other syntactic equivalences are shown in Table 7.3.

Identifiers

An *identifier* is a string that is not a reserved keyword and follows these rules:

1. It starts with a Latin or Greek letter (except underscore).
2. It contains Latin letters, Greek letters, underscore, digit,
3. It ends with Latin letters, Greek letters, underscore, digit, or a subscript.

Table 7.2.: Reserved keywords

abstract	by	implements	required
add_bottom	canonical	instance	requires
and	catalog	Int	Reals
approx_lower	choose	interface	solve_f
approx_upper	code	lowerclosure	solve_r
approx	compact	lowersets	solve
approxu	constant	maximals	specialize
assert_empty	coproduct	mcdp	sum
assert_equal	emptyset	minimals	take
assert_geq	eversion	namedproduct	template
assert_gt	extends	Nat	top
assert_leq	flatten	poset	uncertain
assert_lt	for	powerset	upperclosure
assert_nonempty	ignore_resources	product	uppersets
between	ignore	provided	using
bottom	implemented-by	provides	variable

Unicode	ASCII
$\leq$ or $\preceq$	`<=`
$\geq$ or $\succeq$	`>=`
$\cdot$	`*`
$\langle\rangle$	`< >`
$\top$	`Top`
$\bot$	`Bottom`
$\mathcal{P}$	`powerset`
$\pm$	`+-`
$\sum$	`sum`
$\mapsto$	`|-->`
$\mapsfrom$	`<--|`
$\leftrightarrow$	`<-->`
$\emptyset$	`Emptyset`
$\mathbb{N}$	`Nat`
$\mathbb{R}$	`Rcomp`
$\mathbb{Z}$	`Int`
$\uparrow$	`upperclosure`
$\downarrow$	`lowerclosure`
$\times$	`x`

Table 7.3.: Unicode glyphs and syntactically equivalent ASCII

A regular expression that captures these rules is:

```
identifier = [latin|greek][latin|greek|_|digit]*[latin|greek|_|digit|subscript]?
```

Here are some examples of good identifiers: `a`, `a_4`, a_4, `alpha`, α.

Use of Greek letters as part of identifiers

MCDPL allows to use some Unicode characters, Greek letters and subscripts, also in identifiers and expressions. For example, it is equivalent to write `alpha_1` and `α₁`.

For subscripts, every occurrence of a subscript of the digit d is converted to the fragment `_d`.

Subscripts can only occur at the end of an identifier: a_1 is valid, while $a_1 b$ is not

a valid identifier.

Every Greek letter is converted to its name. It is syntactically equivalent to write `alpha_material` or `α_material`.

Greek letter names are only considered at the beginning of the identifier and only if they are followed by a non-word character. For example, the identifer `alphabet` is not converted to αbet.

Part C. Software manual

Hiking and climbing are favourite sport activites in Switzerland. On average, each Swiss spends 60 hours a year hiking on trails. Notably, hikes can last multiple days: often, hikers stop for the night in one of the 152 Alps huts, offering over 9,200 bed spaces for guests.

8. Command-line usage

This chapter describes how to use the command-line tools using Docker.

When one thinks about Switzerland, one of the symbols that comes to mind is usually cows. Recent statistics show that Switzerland has around 1.6 million cows (roughly one cow per five residents). The canton of Bern leads the rankings, being the one owning the most cows. Interestingly, in the canton of Appenzell Innerrhoden, the ratio of cows and humans is close to 1:1.

8.1. Using MCDP with Docker

Install Docker

First, you must be familiar with Docker and have it installed. Follow the instructions at `https://docs.docker.com/get-started/`.

Pull the image (often)

The docker image `zupermind/mcdp:2024` contains the MCDP tools.

You can pull the image with:

```
docker pull zupermind/mcdp:2024
```

Additional tags

Whenever we release a new version of MCDP, we will tag 3 images:

- `zupermind/mcdp:YYYY`
- `zupermind/mcdp:YYYYMM`
- `zupermind/mcdp:YYYYMMDD`

So, `zupermind/mcdp:2024` is always the latest version of MCDP released in 2024.

For reproducibility purposes, you should use a specific tag.

The list of released tags is available at `https://hub.docker.com/r/zupermind/mcdp/tags`.

Running the command-line tools (Linux/OS X)

The magic command line to use is something like:

```
docker run -it --rm -v $PWD:$PWD -w $PWD zupermind/mcdp:2024 bash
```

This command line drops you into a shell inside the container, with the current directory mounted.

Explanation of the flags:

- `docker run`: starts a container
- `-it`: makes the container interactive
- `-rm`: removes the container when it exits
- `-v $PWD:$PWD`: mounts the current directory inside the container
- `-w $PWD`: sets the working directory to the current directory
- `zupermind/mcdp:2024`: the image to use
- `bash`: the command to run inside the container

It is useful to create an alias for this command, for example:

```
alias mcdp-solve="docker run -it --rm \
    -v $PWD:$PWD -w $PWD \
    zupermind/mcdp:2024 bash"
```

8.2. mcdp-plot

`mcdp-plot` is a command-line tool that can be used to draw MCDP problems in a variety of formats. It can be used to generate the various visualizations used in this book and more.

The calling syntax is:

```
mcdp-plot --plots PLOT_NAMES THING_NAME
```

where the plot names are separated by commas. To see the complete set of plot names, look at the output of

```
mcdp-plot --help
```

and the thing name is the name of the MCDP problem to plot.

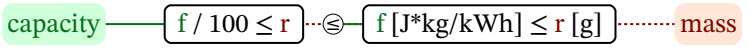
For example, consider the following model:

Listing 47: `ExampleModel.mcdp`

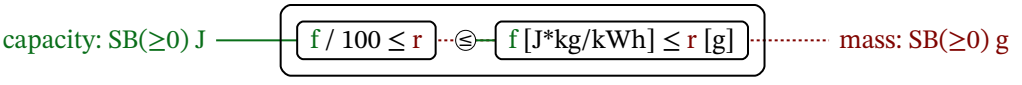
```
mcdp {
    provides capacity [J]
    requires mass [g]
    ρ = 100 kWh / kg # specific_energy
    required mass ≥ provided capacity / ρ
}
```

Here are some examples of plots that can be generated.

```
mcdp-plot --plots ndp_gojs_nowrap_noexpand ExampleModel
```




```
mcdp-plot --plots ndp_gojs_wrap_noexpand_units ExampleModel
```




```
mcdp-plot --plots ndp_gojs_interface ExampleModel
```

8.3. mcdp-solve-query

`mcdp-solve-query` is a command-line tool that can be used to solve queries on MCDP problems.

The calling syntax is:

```
mcdp-solve-query QUERY_NAME
```

The options `--optimistic` and `--pessimistic` can be used to specify the resolution for the solution.

```
mcdp-solve-query --optimistic 10 --pessimistic 10 myquery
```