Language and Tools for Monotone Co-Design

Compiled on 2023-10-05.

Contents

The MCDPL language | 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.

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

1.1. What MCDPL is

MCDPL is a *modeling language* that can be used to formally describe co-design problems. This means that MCDPL allows to describe 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.

MCDPL is extensible. There is a core of posets/relations built into the language, and there are extension mechanisms to implement additional posets and relations.

MCDPs can be interconnected and composed hierarchically. There are several features that 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 Chapter 2.

1.2. Graphical representations of design problems

MCDPL allows to define design problems, which are represented as in Fig. 1: a box with dashed green arrows for functionalities and solid red arrows for requirements.

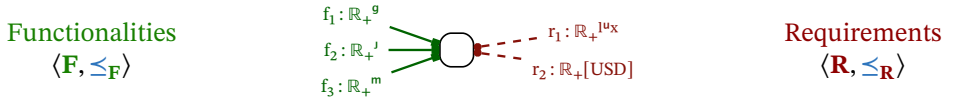


Figure 1.: Representation of a design problem with three functionalities (`f_1`, `f_2`, `f_3`) and two requirements (`r_1`, `r_2`). In this case, the functionality space **F** is the product of three copies of $\overline{\mathbb{R}}_{\geq 0}$: $\mathbf{F} = \overline{\mathbb{R}}_{\geq 0}[\mathrm{g}]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

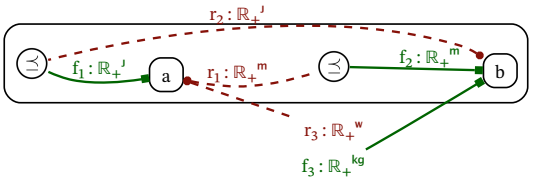


Figure 2.: Example of a *co-design diagram* with two design design problems, a and b. The *co-design constraints* are $f_1 \succeq r_2$ and $f_2 \succeq r_1$.

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`.

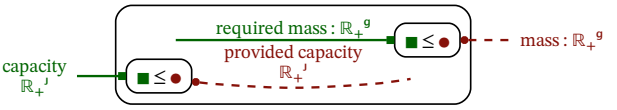
The code in Listing 2 defines an MCDP with one functionality, `capacity`, measured in joules, and one requirement, `mass`, measured in grams.

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. (Of course, internally this is represented using decimal numbers of fixed precision.

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

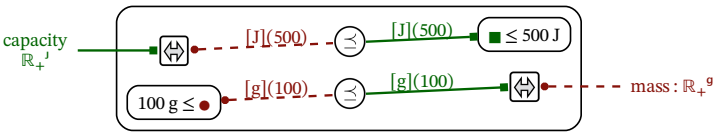
In the body of the `mcdp` declaration one can refer to the values of the functionality and requirements using the expressions `provided` *functionality name* and `required` *requirement name*. For example, Listing 3 shows the completion of the previous MCDP, with hard bounds given to both `capacity`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

The visualization of these constraints is as in Fig. 5. Note that there is always a $\preceq$ node between a dashed green and a solid red edge.

Listing 3

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

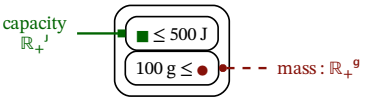
Figure 5.



The visualization above is quite verbose. It shows one node for each functionality and requirement; here, a node can be thought of a variable on which we are optimizing. This is the view shown in the editor; it is helpful because it shows that while `capacity` is a functionality from outside the MCDP, from inside `provided capacity` is a requirement.

The less verbose visualization, as in Fig. 6, skips the visualization of the extra nodes.

Figure 6.: Visualization of Listing 3 without the conversion nodes.



It is possible to query this minimal example. For example, we can ask through the commmand line interface to solve for the minimal requirements given the fucntionality 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.

1.4. Beyond ASCII - 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
```

1.5. Aside: an helpful interpreter

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

Fixing the omission of `provided` and `required`

If it is possible to disambiguate from the context, the MCDPL interpreter also allows to drop the keywords **`provided`** and **`required`**, although it will give a warning.

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.

Catching other problems

All inequalities will always be of the kind:

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

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
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~↑
```

1.6. Describing relations between functionality and resources

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

MCDPL contains as primitive operations addition, multiplication, and exponenentiation 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

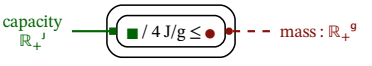
```
ρ = 4 J / g
required mass ≥ provided capacity / ρ
```

In the graphical representation (Fig. 7), 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 7.



For example, if we ask for 600 J:

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

we obtain this answer:

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

1.7. 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. 8 the specific energy is given in `kWh/kg`. PyMCDP will take care of the conversions that are needed, and will introduce a conversion from `J*kg/kWh` to `g` (Fig. 9).

```
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 8.: Automatic conversion among `g`, `kg`, `J`, `kWh`.

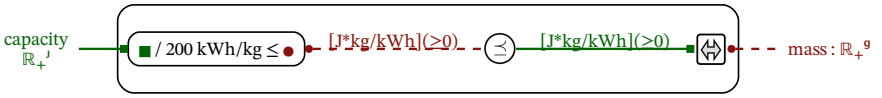


Figure 9.: A conversion from `J*kg/kWh` to `g` is automatically introduced.

Fig. 8 also shows the syntax for comments. MCDPL allows to add a Python-style documentation string at the beginning of the model, delimited by three quotes. It also allows to give a short description for each functionality, requirement, or constant declaration, by writing a Python-style string at the end of the line.

1.8. Catalogues

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 `catalogue` 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. 10, one should define the two maps `req` and `prov`, which map an implementation to the functionality it provides and the requirements it requires.

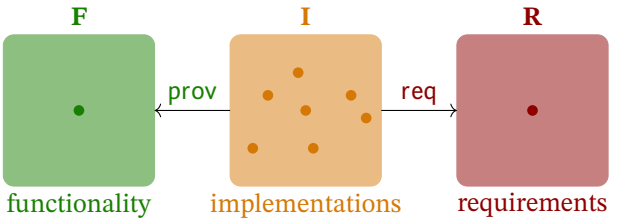


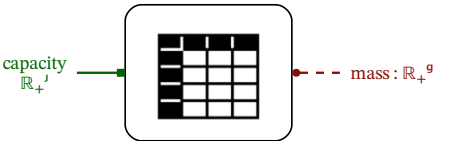
Figure 10.: A design problem is defined from an implementation space **I**, a functionality space **F**, a requirement space **R**, and the maps `req` and `prov` that relate the three spaces.

MCDPL allows to define arbitrary maps `req` and `prov`, and therefore arbitrary relations from functionality to requirements, using the `catalogue` 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 to a value in the functionality and in the requirement space. The icon for this construction is meant to remind of a spreadsheet (Fig. 11).

Listing 5: `model3.mcdp`

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

Figure 11.



Multiple minimal solutions

The `catalogue` construct is the first construct we encountered that allows to define 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: `catalogue2.mcdp`

```
catalogue {
    provides capacity [J]
    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.: Cases for 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 also 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 catalogue2_try "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 catalogue2_try "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 exactly the same resource consumption.

1.9. Union/choice of design problems

The “union” 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`).

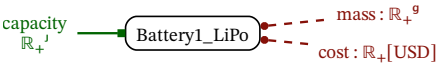
Specifically, consider Nickel-Hidrogen batteries (NiH2) and Lithium-Polymer (LiPo) batteries. One technology is cheaper but leads to heavier batteries and viceversa. Because of this fact, there might be designs in which we prefer either.

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
}
```

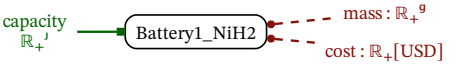
Figure 12.



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
}
```

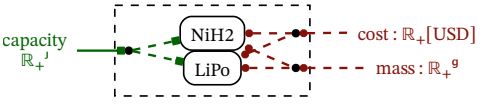
Figure 13.



Listing 9: `Batteries.mcdp`

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

Figure 14.



1.10. Composing design problems

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

Implicit composition of design problems from formulas

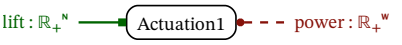
All the mathematical operations (addition, multiplication, power, etc.) are each represented by their own design problem, even though they do not have explicit names.

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
}
```

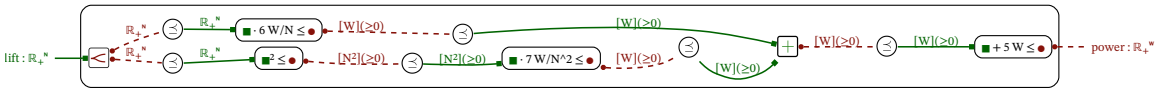
Figure 15.



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

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

This is really the composition of several DPs, correponding to sum, multipication, and exponentiation (Fig. 15).



Explicit composition of design problems

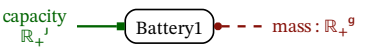
The other way to compose design problems is for the user to define them separately and then connect them explicitly.

Suppose we define a model called `Battery` as in Fig. 16.

Listing 11: `Battery1.mcdp`

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

Figure 16.

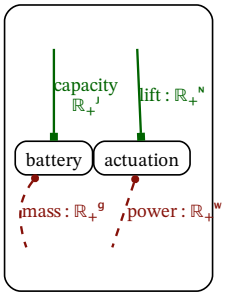


We can now put the battery model and the actuation model using the following syntax.

Listing 12: `combined1.mcdp`

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

Figure 17.



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. 17. We can create a complete model by 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:

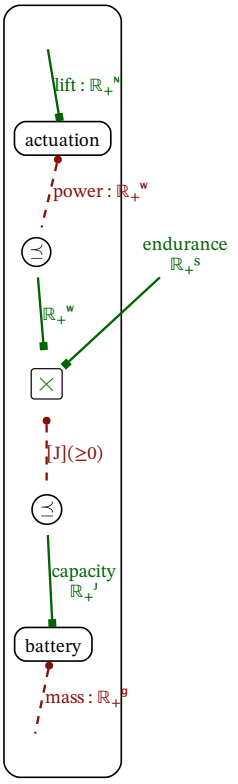
```
energy = provided endurance · (power required by actuation)
capacity provided by battery ≥ energy
```

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]

    actuation = instance `Actuation1
    battery = instance `Battery1

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

Figure 18.



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:

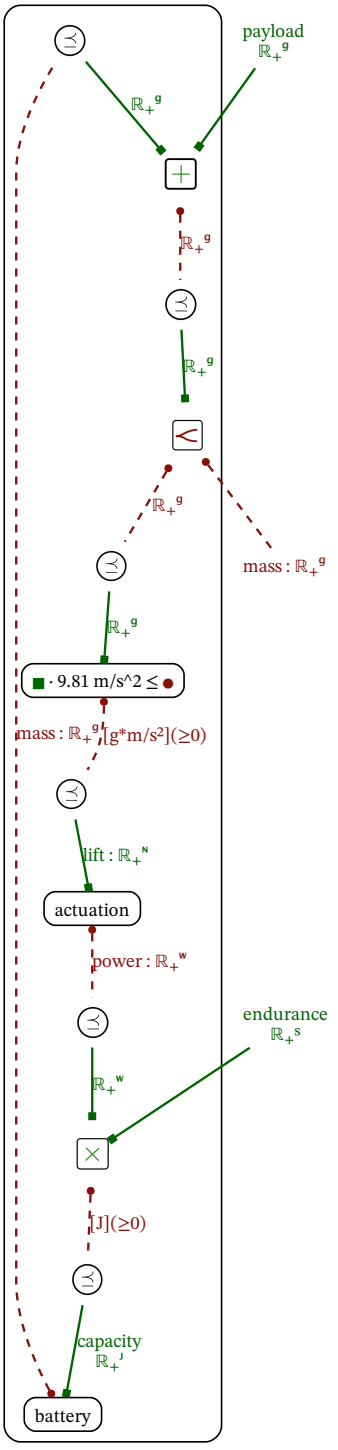
```
gravity = 9.81 m/s²
total_mass = (mass required by battery + provided payload)
weight = total_mass * gravity
lift provided by actuation ≥ weight
```

Now there is a loop in the co-design diagram (Fig. 19).

Listing 14: `Composition.mcdp`

```
mcdp {
    provides endurance [s]
    provides payload [g]

    actuation = instance `Actuation1
    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
    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 19.



1.11. Describing templates (reusable design patterns)

"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 to describe this situation by using the idea of "template".

Templates are described using the keyword **`template`**.

The syntax is:

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

In the brackets put pairs of name and NDPs that will be used to specify the interface.

For example, suppose that there is an interface defined in `Interface.mcdp` as in Listing 1.1.

Listing 1.1: `ExampleInterface.mcdp`

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

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

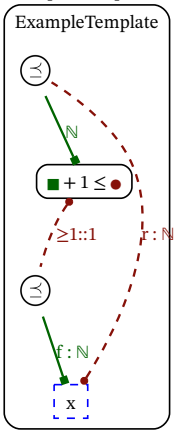
Listing 15

`ExampleTemplate.mcdp_template`

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

Figure 20.

`ExampleTemplate.mcdp_template`



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

Listing 17: `ActuationInterface.mcdp`

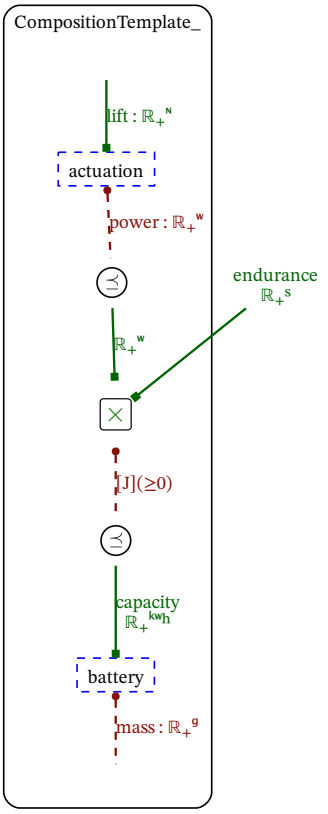
```
interface mcdp {
    provides lift [N]
    requires power [W]
}
```

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 [
    generic_actuation: `ActuationInterface,
    generic_battery: `BatteryInterface
]
mcdp {
    actuation = instance generic_actuation
    battery = instance generic_battery
    # 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 [
        generic_battery: `Battery1,
        generic_actuation: `Actuation1
] `CompositionTemplate
```

1.12. 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 20). 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 21) 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 20 and Listing 21 are equivalent.

Listing 20: `sumc-example.mcdp`

```
mcdp {
    requires total_budget [USD]

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

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

Listing 21: `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 22 and 23).

Listing 22: `sumc3.mcdp`

```
mcdp {
    provides total_power [W]

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

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

Listing 23: `sumf3.mcdp`

```
mcdp {
    provides total_power [W]

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

    provided total_power <= \
    sum power provided by *

}
```

2. Reference

This chapter is a detailed reference for the MCDP language.

2.1. Kinds

MCDPL has 6 *kinds* of data:

1. *Posets* (or, more technically "subposets").
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**.
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. *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	morphisms of ⟨**DP**⟩ with signal names	`.mcdp`	`mcdp {`
			`  requires r = 10 g`
			`}`

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

poset

2.2. Defining finite posets

It is possible to define custom finite posets using the keyword `poset`.

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

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

and the order relations

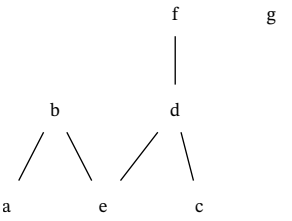
$$a \preceq_Q b, \qquad c \preceq_Q d, \qquad e \preceq_Q d \preceq_Q f, \qquad e \preceq_Q b, \tag{2}$$

we create a file named `Q.mcdp_poset` with code as in Listing 24.

Listing 24: `Q.mcdp_poset`

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

Figure 1.: Hasse diagram of the poset defined in Listing 24



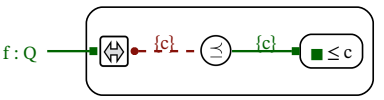
After the poset has been defined, it can be used in the definition of an MCDP, by referring to it by name using the backtick notation, as in "`` `Q ``".

To refer to its elements, use the notation `` `Q: element `` (Listing 25).

Listing 25: `onep.mcdp_poset`

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

Figure 2.

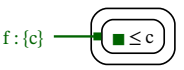


Note that technically there is a conversion in the MCDP in Fig. 2 because the constant DP belongs to the subposet $\mathbf{Q}_c$ of $\mathbf{Q}$ that contains only the element c.

If we modify slightly the MCDP as in Listing 26 we obtain that the functionality is the subsposet $\mathbf{Q}_c$ and there is no need for a conversion node.

Listing 26: `onep2.mcdp_poset`

```
mcdp {
    provides f = `Q: c
}
```



2.3. 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

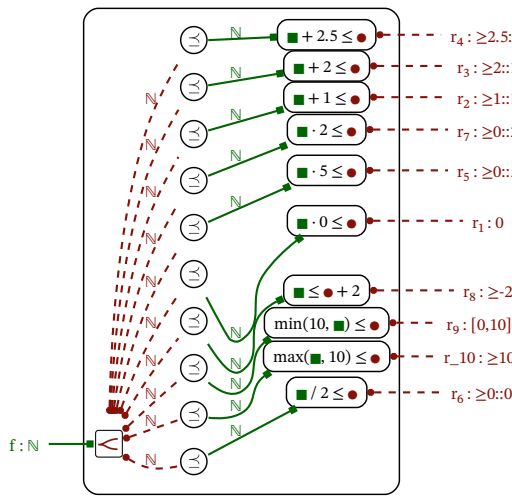
The other variations cannot be accessed directly, but they are derived implicitly by by being careful about how the (monotone) operations can transform the first subsets.

Listing 27 shows an example in which the functionality is defined as a natural number, but the various requirements have a different poset type that depends on the operation.

Listing 27: `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 = provided f - 2
    requires r₉ = min(provided f, 10)
    requires r_10 = max(provided f, 10)
}
```

Figure 3.

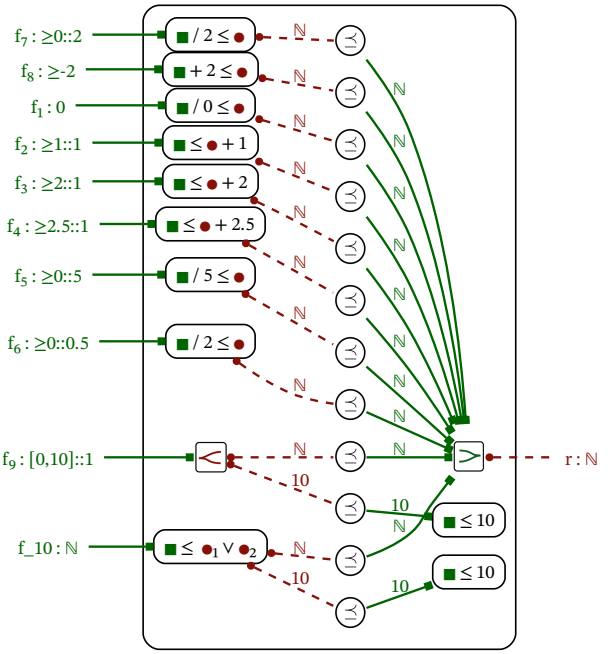


Listing 28 shows the equivalent demo for the requirements side. Note that the for f_{10} the interpreted DPs are slightly different as current one optimization is not implemented.

Listing 28: `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 = required r − 2
    provides f₉ = min(required r, 10)
    provides f_10 = max(required r, 10)
}
```

Figure 4.



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.

2.4. Numbers with units

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

These posets and their values are indicated using the syntax in Table 2.2.

Table 2.2.: Numbers with units

ideal poset	$\langle \overline{\mathbb{R}}_{\geq 0}[\mathrm{g}], \leq \rangle$	$\langle \overline{\mathbb{R}}_{\geq 0}[\mathrm{J}], \leq \rangle$	$\langle \overline{\mathbb{R}}_{\geq 0}[\mathrm{m/s}], \leq \rangle$
syntax for poset	`g`	`J`	`m/s`
syntax for values	`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/

2.5. Poset products

MCDPL allows the definition of finite Cartesian 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 "`⟨∅ J, 1 A⟩`".

Tuples can be nested. For example, you can describe a tuple like `⟨⟨∅ 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
}
```

Accessing elements of a product

`take`

To access the elements of a tuple, use the syntax

```
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:

```
(requirement).label
```

```
(functionality).label
```

For example:

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

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

2.6. Posets of subsets

Power sets

powerset

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)`.

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 the element `⟨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*

The empty set

EmptySet

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.

between

2.7. 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 2.3 shows the different ways in which a constant can be declared to be between `9.95 kg` and `10.05 kg`:

Table 2.3.: Equivalent ways to declare an uncertain constant

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

```
x = 10 kg ± 50 g
```

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

These expressions can be used also in tables, as follows.

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

    100 kWh ± 5 kWh ⟷ 1.2 kg ± 100 g
}
```

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
}
```

2.8. Other constant values

Top Bottom

Top and 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 Maximals

Minimals and 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 `Maximals` `` `MyPoset `` is equivalent to `b` and `d`, and `Minimals` `` `MyPoset `` is equivalent to `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})
```

2.9. 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 loading one from the library, using the syntax `` `name `` or `` `library.name ``.
2. by constructing one using the `mcdp` syntax;
3. by constructing one using the `catalogue` syntax;
4. by specializing a template, using the `specialize` keyword.
5. by constructing one using operations such as `compact` and `abstract`.

2.10. Loading an external NDP

Loading an external NDP from the same library

It's possible to refer to an NDP defined in another file in the same library using the backtick syntax. In the next examples, we expect that there is a file called `other_model.mcdp` in the same directory as the current file.

```
mcdp {
    T = `other_model
    a = instance T
}
```

```
mcdp {
    T = load other_model
    a = instance T
}
```

Loading an external NDP from a different library in the same repository

We can refer to a file in a different library in the same repository using the backtick syntax and specifying the library name

```
mcdp {
    T = `other_library.other_model
    a = instance T
}
```

Loading an external NDP from a different repository

We can also refer to a model in a different repository using the import syntax. We can import the library by name:

```
from shelf "github.com/org/repo@branch" import library other_library
mcdp {
    T = `other_library.other_model
    a = instance T
}
```

Or we can import the specificy model in the library:

```
from shelf "https://github.com/org/repo@branch" import model other_library.other_model
mcdp {
    a = instance other_model
}
```

2.11. Describing Monotone Co-Design Problems (MCDPs)

`mcdp`

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.

A declaration is either a functionality declaration or a requirement declaration.

A functionality declaration is of this form:

```
provides fn [poset]
```

There are also two shortcuts.

The first is used to declare a functionality and provide an expression for it:

```
provides fn = expression
```

This is equivalent to:

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

with the poset being inferred from the expression.

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 [poset of f1]
provides f2 [poset of f2]
provided f1 ≤ f1 provided by dp
provided f2 ≤ f2 provided by dp
```

The posets are inferred from the definition of the DP.

Symmetrically, there are the same constructs for defining requirements:

```
requires fn = expression
```

```
requires r1, r2, r3 for dp
```

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

2.12. Defining NDPs as catalogues

catalogue

An NDP can be created using a "catalogue" declaration, of which there are two types:

A catalogue declaration is comprised of zero or more functionality/requirement declarations and zero or more *catalogue records*.

```
catalogue {
    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.

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

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

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

    10 W ⟷ 10 s
    20 W ⟷ 20 s
}
```

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

```
catalogue {
    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 catalogue 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 catalogue rows, but we use different units in the declaration of the functionalities and requirements (miles and hours):

```
catalogue {
    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.

```
catalogue {
    requires r1 [s]
    ⟨⟩ ↤ imp1 ⟼ 10 s
}
```

```
catalogue {
    provides f1 [s]
    5s ↤ imp1 ⟼ ⟨⟩
}
```

True and false

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

```
catalogue {}
```

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

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

```
catalogue {
    ⟨⟩ ↤ imp1 ⟼ ⟨⟩
}
```

```
catalogue {
    ⟨⟩ ↔ ⟨⟩
}
```

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

```
catalogue {
    ⟨⟩ ↤ even    ⟼ ⟨⟩
    ⟨⟩ ↤ more    ⟼ ⟨⟩
    ⟨⟩ ↤ ways    ⟼ ⟨⟩
    ⟨⟩ ↤ to      ⟼ ⟨⟩
    ⟨⟩ ↤ provide ⟼ ⟨⟩
    ⟨⟩ ↤ nothing ⟼ ⟨⟩
}
```

2.13. 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*

Compactification

`compact`

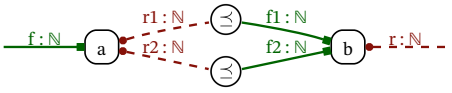
The construct `compact` takes an NDP and produces another in which parallel edges are compacted into one edge. This is one of the steps required towards the solution of the MCDP.

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, the NDP in Listing 2.1 can be visualized as follows:



Instead, the compacted version has only one edge between the two NDPs:


```
mcdp {
    a = instance template mcdp {
        provides f [ℕ]
        requires r1 [ℕ]
        requires r2 [ℕ]
    }
    b = instance template mcdp {
        provides f1 [ℕ]
        provides f2 [ℕ]
        requires r [ℕ]
    }
    r1 required by a ≤ f1 provided by b
    r2 required by a ≤ f2 provided by b
}
```

Ignoring functionality/requirements

`ignore`

The `ignore` command allows to ignore some of the functionality or requirements of an NDP.

Suppose the functionality `f` has type `F`. Then:

```
ignore functionality provided by dp
```

is equivalent to

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

Equivalently,

```
ignore requirement required by dp
```

is equivalent to

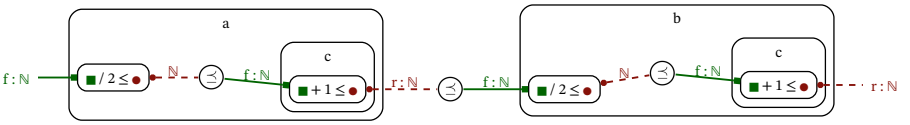
```
requirement required by dp ≤ any-of(Maximals space)
```

flatten

Flattening

It is easy to create recursive composition in MCDPL.

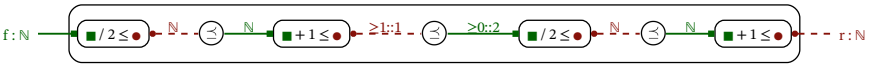
For example, the code on the side describes an NDP composed of nested NDPs.



The "flattening" operation erases the borders between subproblems. The syntax is:

```
flatten `Composition1
```

The result is as follows:




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

abstract

Abstraction

The command `abstract` takes an NDP and created another NDP that forgets the internal structure.

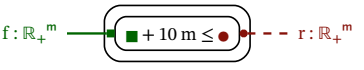
The syntax is:

```
abstract NDP
```

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

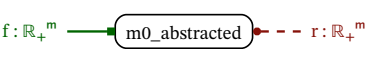
Listing 30: `m0.mcdp`

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



Listing 31: `m0_abstracted.mcdp`

```
abstract `m0
```



2.14. MCDPL syntax

Characters

A MCDP file is a sequence of Unicode code-points that belong to one of the classes described in Table 2.4. All files are assumed to be encoded in UTF-8.

Table 2.4.: 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 2.5.

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 2.6.

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 2.5.: Reserved keywords

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

Table 2.6.: Unicode glyphs and syntactically equivalent ASCII

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`

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 α_1.

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.