CHAPTER 6
Language and tools tutorial

MCDPL is a modeling language that can be used to formally describe co-design problems. MCDPL was designed to represent all and only MCDPs (Monotone Co-Design Problems). For example, multiplying by a negative number is a syntax error. There is a core of features implemented (posets, primitive relations, etc.) that are built into the language, and there are extension mechanisms.

MCDPL is a *modeling* language, not a *programming* language. This means that MCDPL allows to describe variables and systems of relations between variables. Once the model is described, then it can be *queried*; and the "interpreter" `mcdp-solve` runs the computation necessary to obtain the answers.

This chapter describes the MCDPL modeling language, by way of a tutorial. A more formal description is given in Chapter 12.

6.1. Graphical representations of design problems

MCDPL allows to define design problems, which are represented as in Fig. 32: a box with green arrows for functionalities and red arrows for resources.

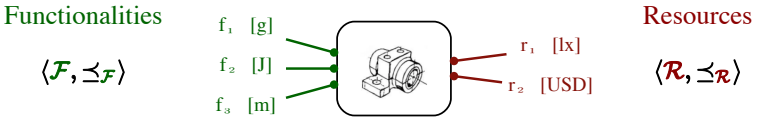


Figure 32. Representation of a design problem with three functionalities (`f_1`, `f_2`, `f_3`) and two resources (`r_1`, `r_2`). In this case, the functionality space $\mathcal{F}$ is the product of three copies of $\overline{\mathbb{R}}_+$: $\mathcal{F} = \overline{\mathbb{R}}_+^{[g]} \times \overline{\mathbb{R}}_+^{[J]} \times \overline{\mathbb{R}}_+^{[m]}$ and $\mathcal{R} = \overline{\mathbb{R}}_+^{[lux]} \times \overline{\mathbb{R}}_+^{[USD]}$.

The graphical representation of a co-design problem is as a set of design problems that are interconnected (Fig. 33). A functionality and a resource edge can be joined using a $\preceq$ sign. This is called a "co-design constraint".

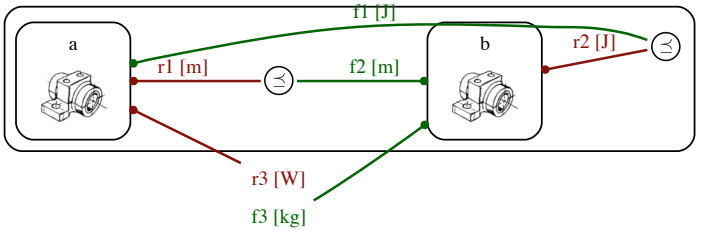


Figure 33. Example of a co-design diagram with two design design problems, a and b.

6.2. Hello, world!

The "hello world" example of an MCDP is an MCDP that has zero functionalities and zero resources. This MCDP will be able to tell us that to do nothing, nothing is needed. While you might have an intuitive understanding of this fact, you might appreciate having a formal proof.

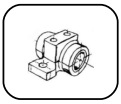
An MCDP is described in MCDPL using the construct `mcdp {…}`, as in Listing 1. The body is empty, and that means that there are no functionality and resources. Comments in MCDPL work like in Python: everything after "#" is ignored by the interpreter.

Listing 1

empty.mcdp

```
mcdp {
    # an empty MCDP
    # (comments are ignored)
}
```

Figure 34



Formally, the functionality and resources spaces are $\mathcal{F} = \mathbf{1}$, $\mathcal{R} = \mathbf{1}$. The space $\mathbf{1} = \{\langle\rangle\}$ is the empty product (Def. 46). $\mathbf{1}$ contains only one element, the empty tuple $\langle\rangle$.

The MCDP above can be queried using the program `mcdp-solve`:

```
$ mcdp-solve empty "<>"
```

The command above means "for the MCDP `empty`, find the minimal resources needed to perform the functionality $f = \langle\rangle$". The command produces the output:

```
Minimal resources needed = ↑{⟨⟩}
```

The output means "it is possible to perform the functionality specified, and the minimal resources needed are $r^\star = \langle\rangle$".

We have proved that:

1. it is possible to do nothing; and
2. we need nothing to do nothing.

6.3. Defining functionality and resources

The functionality and resources 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 resource, `mass`, measured in grams.

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

Listing 2

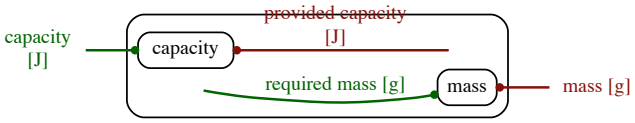


Figure 35

That is, the functionality space is $\mathcal{F} = \overline{\mathbb{R}}_+^{[J]}$ and the resource space is $\mathcal{R} = \overline{\mathbb{R}}_+^{[g]}$. Here, let $\overline{\mathbb{R}}_+^{[g]}$ refer to the nonnegative real numbers with units of grams. (Of course, internally this is represented using floating point numbers. See Section 12.3-B for more details.)

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. 35).

6.4. Constraining functionality and resources

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

Listing 3

```
mcdp {
    provides capacity [J]
    requires mass [g]
    provided capacity <= 500 J
    required mass >= 100 g
}
```

The visualization of these constraints is as in Fig. 36. Note that there is always a "$\preceq$" node between a green and a red edge.

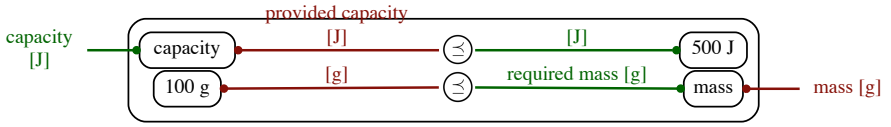
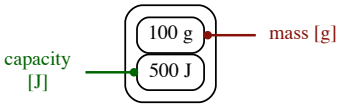


Figure 36

The visualization above is quite verbose. It shows one node for each functionality and resources; 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 resource.

The less verbose visualization, as in Fig. 36, skips the visualization of the extra nodes.

Figure 37



It is possible to query this minimal example. For example:

```
$ 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 (the empty set):

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

The notation "$\uparrow\{\,\}$" means "the upper closure of the empty set $\emptyset$" (Def. 44), which is equal to $\emptyset$.

6.5. Beyond ASCII - Use of Unicode glyphs in the language

To describe the inequality constraints, MCDPL allows to use "<=", ">=", as well as their fancy Unicode version "≤", "≥". These two expressions are equivalent:

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

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

MCDPL also allows to use some special letters in identifiers. These two are equivalent:

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

```
α₁ = β³ + 9.81 m/s²
```

6.6. Aside: an helpful interpreter

The MCDPL interpreter tries to be very helpful.

a) Fixing omissions

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 IDE will actually automatically insert the keyword.

b) Catching other problems

All inequalities will always be of the kind:

$$\textbf{functionality} \succeq \textbf{resources}.$$

If you mistakenly put functionality and resources 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 resources.

line  5 |    provided capacity >= 500 J
                               ↑
```

6.7. Describing relations between functionality and resources

In MCDPs, functionality and resources can depend on each other using any monotone relations (Def. 43). MCDPL contains as primitives addition, multiplication, and division.

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

$$\textbf{capacity} = \rho \times \textbf{mass}.$$

This relation can be described in MCDPL as

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

In the graphical representation (Fig. 38), there is now a connection between capacity and mass, with a DP that multiplies by the inverse of the specific energy.

linear.mcdp

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

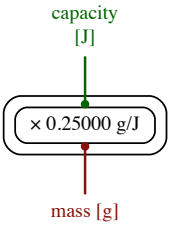


Figure 38

For example, if we ask for `600 J`:

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

we obtain this answer:

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

6.8. Units

PyMCDP is picky about units. It 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 Listing 4 the specific energy given in `kWh/kg`. The two MCDPs are equivalent. PyMCDP will take care of the conversions that are needed, and will introduce a conversion from `J*kg/kWh` to `g` (Fig. 39).

For example, Listing 4 is the same example with the specific energy given in `kWh/kg`. The output of the two models are completely equivalent (up to numerical errors).

Listing 4 also shows the syntax for comments. MCDPL allows to add a Python-style documentation strings at the beginning of the model, delimited by three quotes. It also allows to give a short description for each functionality, resource, or constant declaration, by writing a string at the end of the line.

```
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
}
```

Listing 4. Automatic conversion among `g`, `kg`, `J`, `kWh`.

Figure 39. A conversion from `J*kg/kWh` to `g` is automatically introduced.

6.9. Catalogues (enumeration)

The previous example used a linear relation between functionality and resource. However, in general, MCDPs do not make any assumption about continuity and differentiability of the functionality-resource 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 generally defined from a triplet of functionality space, implementation space, and resource space. According to the diagram in Fig. 40, one should define the two maps **eval** and **exec**, which map an implementation to the functionality it provides and the resources it requires.

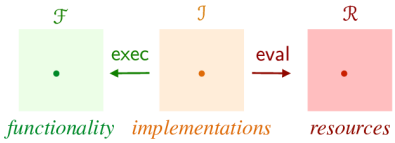


Figure 40. A design problem is defined from an implementation space $\mathcal{I}$, functionality space $\mathcal{F}$, resource space $\mathcal{R}$, and the maps **eval** and **exec** that relate the three spaces.

MCDPL allows to define arbitrary maps **eval** and **exec**, and therefore arbitrary relations from functionality to resources, 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 resource space.

```
catalogue {
    provides capacity [J]
    requires mass [g]

    500 kWh ⟷ model1 ⟼ 100 g
    600 kWh ⟷ model2 ⟼ 200 g
    700 kWh ⟷ model3 ⟼ 400 g
}
```

Listing 5. The `catalogue` construct allows to define an arbitrary relation between functionality, resources, and implementation.

The icon for this construction is meant to remind of a spreadsheet (Fig. 41).

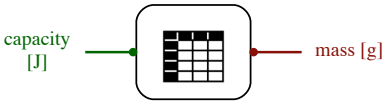


Figure 41

6.10. 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 resource, `cost`.

```
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
}
```

Listing 6

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). As **f** increases, there are 1, 2, 1, and 0 minimal solutions.

TABLE 1.

CASES FOR MODEL IN LISTING 6

Functionality requested	Optimal implementation(s)	Minimal resources needed
0 kWh ≤ **f** ≤ 500 kWh	model1	⟨100 g, 10 USD⟩
500 kWh < **f** ≤ 600 kWH	model2	⟨200 g, 200 USD⟩
	or	or
	model3	⟨250 g, 150 USD⟩
600 kWh < **f** ≤ 700 kWH	model4	⟨400 g, 400 USD⟩
700 kWh < **f** ≤ T kWh	∅	∅

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.

6.11. Coproducts (alternatives)

The *coproduct* construct allows to describe 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 resources to be the mass [g] and the cost [$].

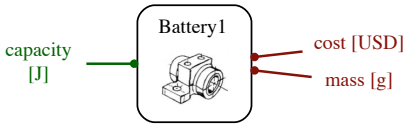


Figure 42

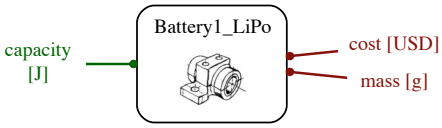
Consider two different battery technologies, characterized by their specific energy [Wh/kg] and specific cost [Wh/$].

Specifically, consider Nickel-Hidrogen batteries and Lithium-Polymer 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 MCDP using the same interface (same resources and same functionality).

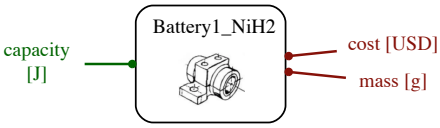
Battery_LiPo.mcdp

```
mcdp {
    provides capacity [J]
    requires mass [g]
    requires cost [$]

    ρ = 150 Wh/kg
    α = 2.50 Wh/$

    required mass ≥ provided capacity / ρ
    required cost ≥ provided capacity / α
}
```



Battery1_NiH2.mcdp

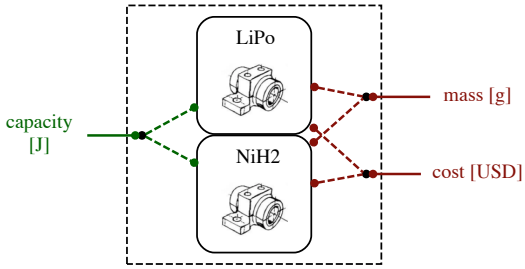
```
mcdp {
    provides capacity [J]
    requires mass [g]
    requires cost [$]

    ρ = 45 Wh/kg
    α = 10.50 Wh/$

    required mass ≥ provided capacity / ρ
    required cost ≥ provided capacity / α
}
```



Then we can define the **coproduct** of the two using the keyword `choose`. Graphically, the choice is indicated through dashed lines.

Batteries.mcdp

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



TODO: Add numerical example

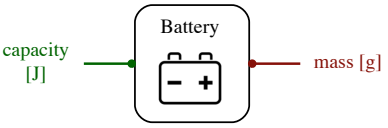
6.12. Composing MCDPs

MCDPL encourages composition and code reuse.

Suppose we define a simple model called `Battery` as follows:

`Battery.mcdp`

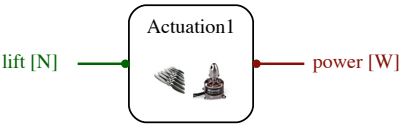
```
mcdp {
    provides capacity [J]
    requires mass [g]
    ρ = 100 kWh / kg # specific_energy
    ρ₁ = 2
    required mass ≥ provided capacity / ρ
}
```



Let's also define the MCDP `Actuation1`, as a relation from lift to power, as in Listing 7.

`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
}
```



Listing 7

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 five DPs, correponding to sum, multiplaction, and exponentiation (Fig. 43).

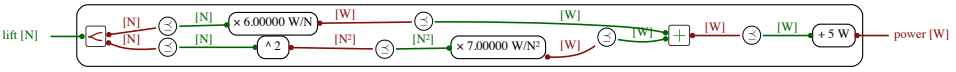


Figure 43

Let us combine these two together.

The syntax to re-use previously defined MCDPs is:

```
instance `Name
```

The backtick means "load the symbols from the library, from the file called `Name.mcdp`".

The following creates two sub-design problems, for now unconnected.

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

Listing 8

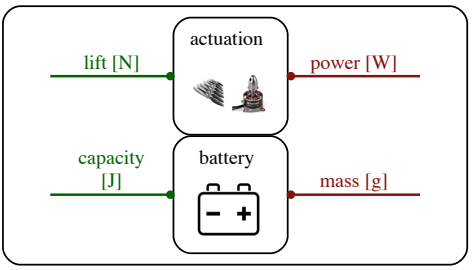


Figure 44

The model in Listing 8 is not usable yet because some of the edges are unconnected. We can create a complete model by adding a co-design constraint.

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 (Fig. 45).

Listing 9

```
mcdp {
    provides endurance [s]

    actuation = instance `Actuation1
    battery = instance `Battery

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

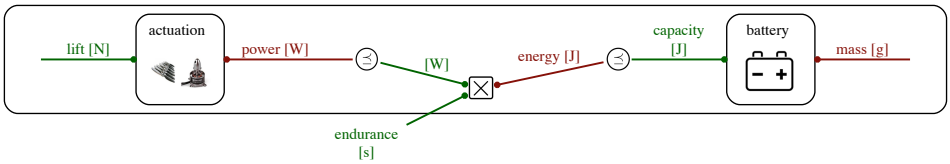


Figure 45

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

Take extra_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. 46).

TODO: Add example output

6.13. Describing design patterns

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

For example, the code in Listing 10 composes a particular battery model, called `Battery, and a particular actuator model, called `Actuation1. However, it is clear that the pattern of "interconnect battery and actuators" is independent of the particular battery and actuator. 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 {
    …
}
```

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 11.

`Interface.mcdp`

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

Listing 11

Then we can declare a template as in Listing 12. The template is visualized as a diagram with a hole (Fig. 47).

`Composition.mcdp`

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

    actuation = instance `Actuation1
    battery = instance `Battery

    # 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
}
```

Listing 10

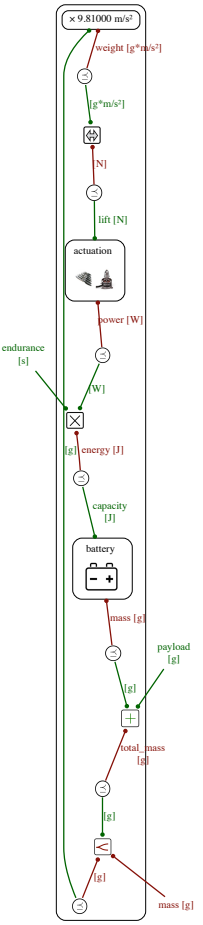


Figure 46

ExampleTemplate.mcdp_template

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

Listing 12

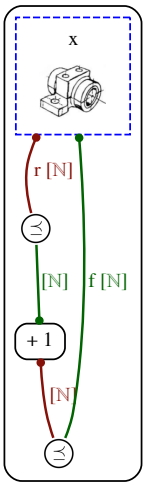


Figure 47

Here is the application to the previous example of battery and actuation. Suppose that we define their "interfaces" as in Listing 13 and Listing 14.

BatteryInterface.mcdp

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

Listing 13

ActuationInterface.mcdp

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

Listing 14

Then we can define a template that uses them. For example the code in Listing 15 specifies that the templates requires two parameters, called `generic_actuation` and `generic_battery`, and they must have the interfaces defined by `ActuationInterface and `BatteryInterface.

The diagram in Fig. 48 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 16.

```
specialize [
    generic_battery: `Battery,
    generic_actuation: `Actuation1
] `CompositionTemplate
```

Listing 16

TODO: Add example of battery composition

CompositionTemplate.mcdp

```
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
}
```

Listing 15

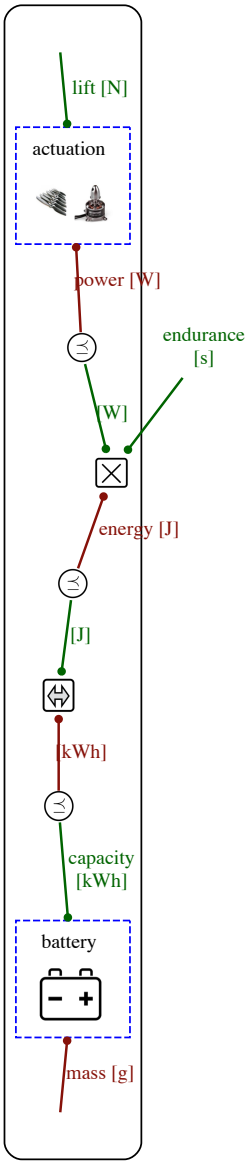


Figure 48

CHAPTER 7
Installation and quickstart

7.1. Supported platforms

The code has been tested on the following platforms:

- Ubuntu 14.04 and 16.04, using the system Python.
- OS X using the Enthought Python distribution.

Windows support:

In principle, there is nothing that prevents the software to run on Windows, however, there are probably lots of subtle adjustments to be done. Porting contributions are welcome.

Python 3 support:

PyMCDP works on Python 2.7, not Python 3. Porting is not difficult, but it is not done yet.

7.2. Installing dependencies

a) Installing dependencies on Ubuntu 16.x

Install git:

```
$ sudo apt-get install git
```

Install Python and libraries:

```
$ sudo apt-get install python-numpy python-matplotlib python-yaml python-pip python-dev
python-setproctitle python-psutil python-lxml
```

Install other external programs used for creating the graphics:

```
$ sudo apt-get install graphviz pdftk imagemagick
```

For the manual, need to install LyX ([instructions(https://launchpad.net/ lyx-devel/+archive/ubuntu/release)]):

```
$ sudo add-apt-repository ppa:lyx-devel/release
$ sudo apt-get install lyx
```

Fonts:

```
https://fontlibrary.org/en/font/anka-coder-narrow
DOLLAR sudo mv *.ttf  /.fonts/
```

and run ***sudocd/usr/share/fonts*** sudo fc-cache -fv

Optional extra dependencies for compiling the manual:

Compiling the manual takes a bit more effort.

For math support:

```
$ sudo apt-get install nodejs npm
$ sudo npm install -g MathJax-node jsdom less stylelint
```

If the last step fails, try the following first:

```
$ sudo ln -s /usr/bin/nodejs /usr/local/bin/node
```

For printing to PDF, install Prince from `https://www.princexml.com>`. Download the .deb for Ubuntu and use `dpkg -i xxx.deb`.

b) Installing dependencies on OS X

The Python distribution used for developing is Enthought Canopy, but any other distribution should do.

7.3. Installing PyMCDP

a) Option 1: Install using (`pip`)

This is marginally easier than option 2.

Run this command:

```
$ sudo pip install -U PyMCDP conftools quickapp decentlogs systemcmd
```

Note that if you omit the `sudo`, modern Ubuntu 16 will install (correctly) in the directory `/.local/`. In this case, make sure you have `/.local/bin/` in your `PATH`.

b) Option 2: Installation from source (preferred)

Clone the repo using:

```
$ git clone https://github.com/AndreaCensi/mcdp.git
```

Jump into the directory and install the main module:

```
$ cd mcdp
$ sudo python setup.py develop
```

Omit the command `sudo` if you have already set up a virtual environment.

7.4. Verifying that the installation worked

a) Running the web interface

Run the command:

```
$ mcdp-web
```

Then point your browser to the address `http://127.0.0.1:8080/`.

b) Running command-line programs

The program `mcdp-solve` is a solver.

```
$ mcdp-solve -d <library> <model_name> <functionality>
```

For example, to solve the MCDP specified in the file `battery.mcdp` in the library `src/mcdp_data/libraries/examples/example-battery.mcdplib`, use:

```
$ mcdp-solve -d src/mcdp_data/libraries/examples/example-battery.mcdplib battery "<1 hour, 0.1 kg, 1 W>"
```

The expected output is:

```
...
Minimal resources needed: mass = ↑{0.039404 kg}
```

This is the case of unreasonable demands (1 kg of extra payload):

```
$ mcdp-solve -d src/mcdp_data/libraries/examples/example-battery.mcdplib battery "<1 hour, 1.0 kg, 1 W>"
```

This is the expected output:

```
Minimal resources needed: mass = ↑{+∞ kg}
```

CHAPTER 12
MCDPL Language reference

This chapter gives a formal description of the MCDPL language.

12.1. MCDPL grammar

a) Characters

A MCDP file is a sequence of Unicode code-points that belong to one of the classes described in Table 4.

In the current implementation, all files on disk are assumed to be encoded in UTF-8.

TABLE 4. CHARACTER CLASSES

Class	Characters	Class	Characters
Latin letters	`abcdefghijklm nopqrstuvxwyz ABCDEFGHIJKLM NOPQRSTUVXWYZ`	Subscripts	`0123456789`
Underscore	`_`	Comment delimiter	`#`
Greek letters	`αβγδεζηθικλμν ξοπρστυφχψω ΓΔΕΖΗΘΙΚΛΜΝΞ ΟΠΡΣΤΥΦΧΨΩ`	String delimiters	`'"`
		Backtick	`` ` ``
		Parentheses	`[]()`
		Operators	`<= > ≤≥ ≮≯ =`
		Tuple-making	`<>??`
		Arrows glyphs	`↔? ?↤?↦`
		Math	`·*−+^`
Digits	`0123456789`	Other glyphs	`×⊤⊥∅ℕℝ /±↑↓%$u∅`
Superscripts	`123456789`		

b) Comments

Comments work as in Python. Anything between the symbol `#` and a newline is ignored.

Comments can include any Unicode character.

c) Identifiers and reserved keywords

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

1. It starts with a Latin or Greek letter (not underscore).
2. It contains Latin letters, Greek letters, underscore, digit,
3. It ends with Latin letters, Greek letters, underscore, digit, or a subscript.

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

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

The reserved keywords are shown in Table 5.

TABLE 5. RESERVED KEYWORDS

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

TABLE 6. DEPRECATED KEYWORDS

	s	interval	
deprecated	finite_poset		experimental
ceilsqrt	dp	no need to be keyword	addmake
set-of	from		mcdp-type
any-of	sub	pow	new
load	dptype	dimensionless	

12.2. Syntactic equivalents

MCDPL allows a number of Unicode glyphs as an abbreviations of a few operators (Table 7).

a) Superscripts

Every occurrence of a superscript of the digit *d* is interpreted as a power "`^d`". It is syntactically equivalent to write "`x^2`" or "`x²`".

TABLE 7. UNICODE GLYPHS AND SYNTACTICALLY EQUIVALENT ASCII

Unicode	ASCII	Unicode	ASCII	Unicode	ASCII
⩾ or ≥	>=	ℝ	Rcomp	a_0	a_0
⩽ or ≤	<=	ℤ	Int	a_1	a_1
·	*	↑	upperclosure	a_2	a_2
⟨···⟩	<···>	↓	lowerclosure	a_3	a_3
⊤	Top	a^1	a^1	a_4	a_4
⊥	Bottom	a^2	a^2	a_5	a_5
℘	powerset	a^3	a^3	a_6	a_6
±	+-	a^4	a^4	a_7	a_7
⊖ or ⟷	<-->	a^5	a^5	a_8	a_8
↤ or ⟻	<--|	a^6	a^6	a_9	a_9
↦ or ⟼	|-->	a^7	a^7	×	x
∅	Emptyset	a^8	a^8		
ℕ	Nat	a^9	a^9		

b) Subscripts

For subscripts, every occurrence of a subscript of the digit *d* is converted to the fragment "_d". It is syntactically equivalent to write "_1" or "$_1$".

Subscripts can only occur at the end of an identifier: a_1 is valid, while "a_1b" is not a valid identifier.

c) 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".

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".

Table 8 shows the Greek letters supported and their translitteration. Note that there is a difference between lower case and upper case.

12.3. Posets and their values

All values of functionality and resources belong to posets. PyMCDP knows a few built-in posets, and gives you the possibility of creating your own. Table 9 shows the basic posets

TABLE 8. GREEK LETTERS SUPPORTED BY MCDPL

α	alpha	Ι	Iota	Ο	Omicron	Σ	Sigma
β	beta	ι	iota	ο	omicron	σ	sigma
Χ	Gamma	Κ	Kappa	Φ	Phi	Τ	Tau
χ	gamma	κ	kappa	φ	phi	τ	tau
Δ	Delta	Λ	Lambda	Π	Pi	Θ	Theta
δ	delta	λ	lambda	π	pi	ϑ	theta
Ε	Epsilon	Μ	Mu	Ψ	Psi	Υ	Upsilon
ε	epsilon	μ	mu	ψ	psi	υ	upsilon
Η	Eta	Ν	Nu	Χ	Chi	Ξ	Xi
η	eta	ν	nu	χ	chi	ξ	xi
Γ	Gamma	Ω	Omega	Ρ	Rho	Ζ	Zeta
γ	gamma	ω	omega	ρ	rho	ζ	zeta

that are built in: natural numbers, nonnegative real numbers, and nonnegative real numbers that have units associated to them.

TABLE 9. BUILT-IN POSETS

MCDPL poset	ideal poset	Python representation
`Nat`	$\langle \mathbb{N} \cup \top, \leq \rangle$	`int` plus a special `Top` object
`Rcomp`	$\langle \mathbb{R}_+ \cup \top, \leq \rangle$	`float` plus a special `Top` object
`g`	$\langle \mathbb{R}_+^{[g]} \cup \top, \leq \rangle$	`float` plus a special `Top` object

a) Natural numbers Nat Nat

By $\overline{\mathbb{N}}$ we mean the completion of $\mathbb{N}$ to include a top element $\top$. This makes the poset a complete partial order (Def. 10).

The natural numbers with completion are expressed as "Nat" or with the Unicode letter "Nat". Their values are denoted using the syntax "Nat:42" or simply "42".

Internally, $\mathbb{N}$ is represented by the Python type `int`, which is equivalent to the 32 bits `signed long` type in C. So, it is really a chain of $\mathbf{2^{31} + 1}$ elements.

b) Nonnegative floating point numbers Rcomp Rcomp

Let $\mathbb{R}_+ = \{x \mid x \geq 0\}$ be the nonnegative real numbers and let $\overline{\mathbb{R}}_+ = \mathbb{R}_+ \cup \top$ be its completion. The $+$ and $\times$ operations are extended from $\mathbb{R}$ to $\overline{\mathbb{R}}_+$ by defining the following:

$$\begin{aligned} \forall a: \quad & a + \top = \top \\ \forall a: \quad & a \times \top = \top \end{aligned}$$

This poset is indicated in MCDPL by Rcomp or Rcomp. Values belonging to this poset are indicates with the syntax their values as Rcomp:42.0, Rcomp:42.0, or simply 42.0.

Internally, $\overline{\mathbb{R}}_+$ is approximated using double precision point numbers (IEEE 754), corresponding to the `float` type used by Python and the `double` type in C (in most implementations of C). Of course, the floating point implementations of $+$ and $\times$ are also not associative or commutative due to rounding errors. PyMCDP does not assume commutativity or associativity; the assumption is just that they are monotone (Def. 42) in each argument (which they are).

c) Nonnegative floating point numbers with units

Floating point numbers can also have **units** associated to them. So we can distinguish $\mathbb{R}^{[m]}$ from $\mathbb{R}^{[s]}$ and even $\mathbb{R}^{[m]}$ from $\mathbb{R}^{[km]}$. These posets and their values are indicated using the syntax in Table 10.

TABLE 10. NUMBERS WITH UNITS

ideal poset	$\langle \overline{\mathbb{R}}_+^{[g]}, \leq \rangle$	$\langle \overline{\mathbb{R}}_+^{[J]}, \leq \rangle$	$\langle \overline{\mathbb{R}}_+^{[m/s]}, \leq \rangle$
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 in Pint form a group. 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 $u \in U$.

with an equivalence relation.

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.

d) Defining custom posets (`poset`)

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

For example, suppose that we create a file named "`MyPoset.mcdp_poset`" containing the definition in Listing 23. This declaration defines a poset with 5 elements `a`, `b`, `c`, `d`, `e` and with the given order relations, as displayed in Fig. 52.

`MyPoset.mcdp_poset`

```
poset {
    a b c d e

    a ≤ b
    c ≤ d
    e ≤ d
    e ≤ b
}
```

Listing 23. Definition of a custom poset

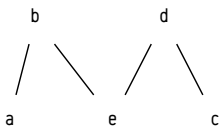


Figure 52

The name of the poset, `MyPoset`, comes from the filename `MyPoset.mcdp_poset`. 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 "`MyPoset".

To refer to its elements, use the notation `MyPoset: element (Listing 24).

```
mcdp {
    provides f [`MyPoset]

    provided f ≤ `MyPoset : c
}
```

Listing 24. Referring to an element of a custom poset

e) Poset products (`×`)

MCDPL allows the definition of finite Cartesian products (Def. 0).

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

⟨poset⟩ × ⟨poset⟩ × … × ⟨poset⟩ ⟨poset⟩ × ⟨poset⟩ × … × ⟨poset⟩

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

The elements of a poset product are called "tuples". These correspond exactly to Python's tuples. To define a tuple, use angular brackets "<" and ">". The syntax is:

<⟨value⟩, ⟨value⟩, …, ⟨value⟩>

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 "?0 J, 1 A?".

Tuples can be nested. For example, you can describe a tuple like ? ?0 J, 1 A?, ?1 m, 1 s, 42? ?, and its poset is denoted as ((J × A) × (m × s × Nat)).

f) 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).

g) Power sets (`set-of`, ℘)

MCDPL allows to describe the set of subsets of a poset, i.e. its power set (Def. 39).

The syntax is either of these:

```
℘(⟨poset⟩)      set-of(⟨poset⟩)
```

To describe values in a powerset, i.e. subsets, use this set-building notation:

```
{ ⟨value⟩, ⟨value⟩, …, ⟨value⟩}
```

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

h) Upper and lower sets and closures (`UpperSets`, `LowerSets`, `upperclosure`, ↑, `lowerclosure`, ↓)

Upper sets (Def. 40) and lower sets (Def. 41) can be described by the syntax

```
UpperSets(⟨poset⟩)      LowerSets(⟨poset⟩)
```

For example, UpperSets(Nat) represents the space of upper sets for the natural numbers.

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.

12.4. Types universe

MCDPL has several "types universes", which we are going to call "Posets", "Values", "Named DPs (NDPs)", "Primitive DPs (PDPs)" and "Templates". Every expression in the language belongs to one of these universes (Table 11).

The syntax is designed in such a way that the type of each expression can be inferred directly from the abstract syntax tree (AST).

TABLE 11. TYPES UNIVERSE

Type universe	example	Semantics
Posets	`ℕ, m/s²`	A "poset" describes a set of objects and an order relation.
Values	`42, 9.81 m/s²`	Values are elements of a posets.
Upper and Lower Sets		These are represented as antichains
Uncertain values (intervals)	`between 42 and 43`	
Interfaces		
Primitive DPs	`XXX`	These correspond to the idea of the DP category. They have a functionality space and a resource space.
Named DPs	`mcdp{...}`	These are Primitive DPs + information about the names of ports.
Composite Named DPs	`mcdp{...}`	These are Named DPs that are described as the composition of other DPs.
templates	`template[...]{...}`	These are templates for Composite Named DPs (they could be considered morphisms in an operad).
R-quantity	`provided a + 1`	An expression that refers to a resource
F-quantity	`required a + 1`	An expression that refers to a functionality

There are numerous relations among these universes (Fig. 0).

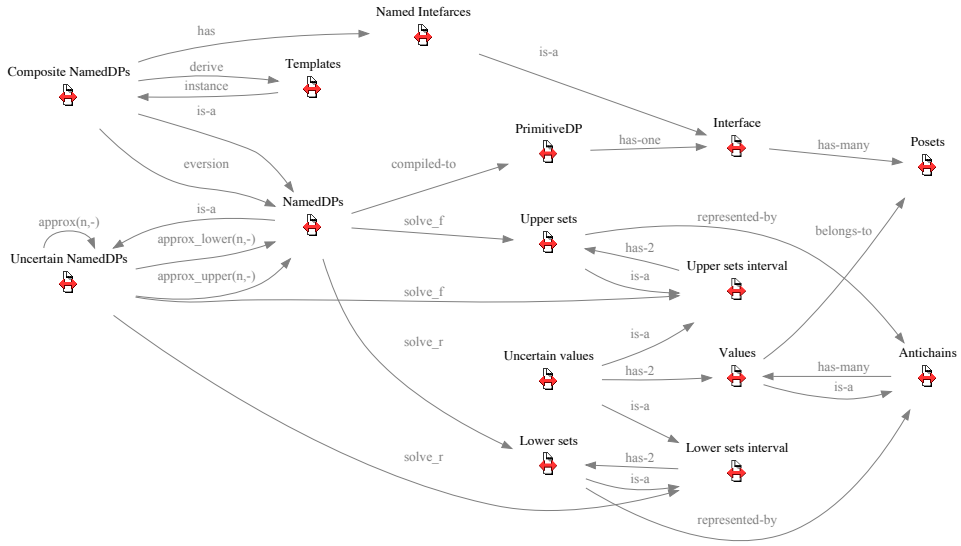


Figure 53. Relationships among the types universes

12.5. Other ways to specify values

a) Top and bottom (`Top`, `Bottom`)

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

`Top` ⟨poset⟩ `T` ⟨poset⟩
`Bottom` ⟨poset⟩ `⊥` ⟨poset⟩

For example, "Top V" indicates the top of the V.

b) 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. 54. Then Maximals `MyPoset is equivalent to `b` and `d`, and Minimals `MyPoset is equivalent to `a`, `e`, `c`.

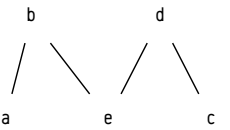


Figure 54

c) The empty set (`EmptySet`)

To denote the empty set, use the keyword `EmptySet`:

`EmptySet` ⟨poset⟩

Note that empty sets are typed—this is different from set theory. EmptySet J is an empty set of energies, and EmptySet V is an empty set of voltages, and the two are not equivalent.

12.6. Defining NDPs

a) (`mcdp{...}`) syntax

`provides` ⟨functionality⟩ `using` ⟨dp⟩

`requires` ⟨resource⟩ `for` ⟨dp⟩

b) (`catalogue{...}`) syntax

12.7. Primitive DPs (`implemented-by`)

…

12.8. Operations on NDPs

a) (`abstract`)

`abstract` ⟨mcdp⟩

b) (`compact`)

The command `compact` takes an MCDP and produces another with "compacted" edges:

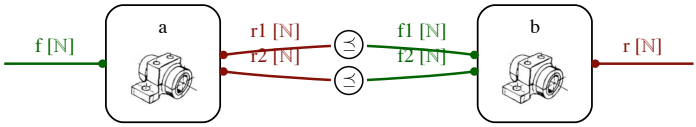
`compact` ⟨mcdp⟩

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

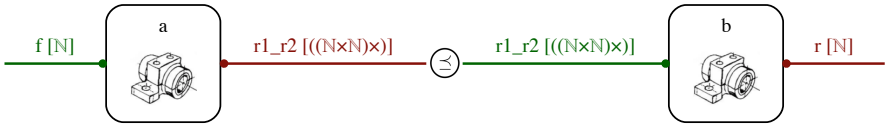
replaced.

```
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
}
```

Original:



Compacted:



c) (template)

```
template ⟨mcdp⟩
```

d) (flatten)

```
flatten ⟨mcdp⟩
```

e) (canonical)

This puts the MCDP in a canonical form:

```
canonical ⟨mcdp⟩
```

f) (approx_lower, approx_upper)

This creates a lower and upper bound for the MCDP:

```
approx_lower(⟨n⟩, ⟨mcdp⟩)
```

```
approx_upper(⟨n⟩, ⟨mcdp⟩)
```

g) (ignore)

Suppose f has type F. Then:

```
ignore ⟨functionality⟩ provided by ⟨dp⟩
```

is equivalent to

```
⟨functionality⟩ provided by ⟨dp⟩ ≥ any-of(Minimals ⟨space⟩)
```

Equivalently,

```
ignore ⟨resource⟩ required by ⟨dp⟩
```

is equivalent to

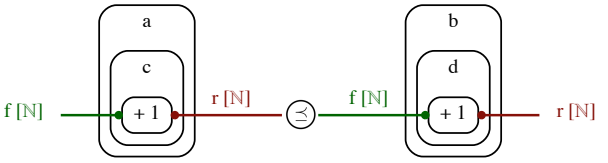
```
⟨resource⟩ required by ⟨dp⟩ ≤ any-of(Maximals ⟨space⟩)
```

h) Abstraction and flattening (`abstract`, `flatten`)

It is easy to create recursive composition in MCDP.

Composition1.mcdp

```
mcdp {
    a = instance mcdp {
      c = instance mcdp {
        provides f [N]
        requires r [N]
        provided f + N:1 ≤ required r
    }

    provides f using c
    requires r for   c
  }

  b = instance mcdp {
      d = instance mcdp {
        provides f [N]
        requires r [N]
        provided f + N:1 ≤ required r
    }

    provides f using d
    requires r for   d
  }

  r required by a ≤ f provided by b
  requires r for b
  provides f using a
}
```

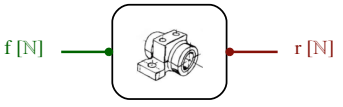


We can completely abstract an MCDP, using the `abstract` keyword.

```
abstract ⟨mcdp⟩
```

For example:

```
abstract `Composition1
```



And we can also completely **flatten** it, by erasing the border between subproblems:

```
flatten `Composition1
```