Formal tools for co-design

Contents

Part A - Tutorial and quickstart .................................................................2

Part B - MCDPL Language reference .....................................................20
PART A
Tutorial and quickstart

This part contains an installation guide as well as a short tutorial to give you the feeling of how this new approach to co-design works.

Contents

UNIT A-1 - Installation and quickstart ................................................................. 3
UNIT A-2 - Language and tools tutorial ............................................................... 4
You can use Docker to quickly start the PyMCDP server and utilities.

1.1. **Running the web interface on your computer in read only mode**

PyMCDP comes with an experimental web interface. You can quickly run the web interface using:

```
$ docker run -it -p 8080:8080 andreacensi/mcdp_server:master
```

**Note:** This only affords *read only* access. See

1.2. **Running the command-line examples mentioned in the manual**

Use this command to drop into a shell where the programs mentioned in the manual (such as `mcdp-solve`) are available:

```
$ docker run -it -v $PWD:$PWD --rm andreacensi/mcdp:master
```

Alternatively, you can also introduce an alias like this:

```
$ alias mcdp-solve='docker run -it -v $PWD:$PWD --rm andreacensi/mcdp:master mcdp-solve'
```

and then just use `mcdp-solve` at your shell.

1.3. **Running the webserver with write access to the models**

To run the command line examples, you need to clone the repository:

`https://github.com/andreacensi/mcdp`

and then run

```
$ docker-compose up --build
```
UNIT A-2
Language and tools tutorial

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

Contents

Section 2.1 - What MCDPL is .............................................................................................................. 4
Section 2.2 - Graphical representations of design problems......................................................... 5
Section 2.3 - Defining functionality and resources .......................................................................... 5
Section 2.4 - Constraining functionality and resources .................................................................. 6
Section 2.5 - Beyond ASCII - Use of Unicode glyphs in the language ........................................ 7
Section 2.6 - Aside: an helpful interpreter ....................................................................................... 7
Section 2.7 - Describing relations between functionality and resources ...................................... 8
Section 2.8 - Units .................................................................................................................................. 9
Section 2.9 - Catalogues (enumeration) .......................................................................................... 10
Section 2.10 - Multiple minimal solutions ........................................................................................ 11
Section 2.11 - Coproducts (alternatives) .......................................................................................... 12
Section 2.12 - Composing design problems ...................................................................................... 13
Section 2.13 - Describing design patterns ......................................................................................... 16
Section 2.14 - Shortcuts for sums sum ............................................................................................ 19

2.1. What MCDPL is

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

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

This chapter describes the MCDPL modeling language by way of a tutorial. A more formal description is given in Part B - MCDPL Language reference.
2.2. Graphical representations of design problems

MCDPL allows to define design problems, which are represented as in Figure 2.1: a box with dashed green arrows for functionalities and solid red arrows for resources.

![Graphical representation of a design problem](image)

**Figure 2.1.** 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 $F$ is the product of three copies of $\mathbb{R}_+$: $F = \mathbb{R}_+^*[g] \times \mathbb{R}_+^*[J] \times \mathbb{R}_+^*[m]$ and $R = \mathbb{R}_+^*[lux] \times \mathbb{R}_+^*[USD]$.

The graphical representation of a co-design problem is as a set of design problems that are interconnected (Figure 2.2). A functionality and a resource edge can be joined using a $\preceq$ sign. This is called a “co-design constraint”. In the figure, the co-design constraints are $f_1 \preceq r_2$ and $f_2 \preceq r_1$.

![Co-design diagram](image)

**Figure 2.2.** Example of a co-design diagram with two design problems, a and b. The co-design constraints are $f_1 \preceq r_2$ and $f_2 \preceq r_1$.

2.3. Defining functionality and resources

An MCDP is described in MCDPL using the construct `mcdp {...}`, as in Listing 2.1. In this case, the body is empty, and that means that there are no functionality and no resources.

**Listing 2.1**

```mcdp
mcdp {
    # an empty MCDP
    # (everything after # is ignored)
}
```

Figures and code are shown to illustrate the concepts. The functionality and resources of an MCDP are defined using the keywords `provides` and `requires`. The code in Figure 2.5 defines an MCDP with one functionality, `capacity`, measured in joules, and one resource, `mass`, measured in grams. The graphical representation is in Figure 2.5.
That is, the functionality space is \( \mathcal{F} = \mathbb{R}^+_* [J] \) and the resource space is \( \mathcal{R} = \mathbb{R}^+_* [g] \). Here, let \( \mathbb{R}^+_* [g] \) refer to the nonnegative real numbers with units of grams. (Of course, internally this is represented using floating point numbers. See Section 2.2 - Nonnegative floating point numbers \( \mathbb{R}_{\text{comp}} \) for more details.)

The MCDP defined above is, however, incomplete, because we have not specified how \textit{capacity} and \textit{mass} relate to one another. In the graphical notation, the co-design diagram has unconnected arrows (Figure 2.5).

### 2.4. Constraining functionality and resources

In the body of the \texttt{mcdp(...) \ldots} declaration one can refer to the values of the functionality and resources using the expressions \texttt{provided (functionality name)} and \texttt{required (resource name)}. For example, Listing 2.2 shows the completion of the previous MCDP, with hard bounds given to both \textit{capacity} and \textit{mass}.

```
Listing 2.2
mcdp {
    provides capacity [J]
    requires mass [g]
    provided capacity <= 500 J
    required mass >= 100 g
}
```

The visualization of these constraints is as in Figure 2.6. Note that there is always a “\(\leq\)” node between a dashed green and a solid red edge.

```
Figure 2.6
```

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 \textit{capacity} is a functionality from outside the MCDP, from inside \texttt{provided capacity} is a resource.
The less verbose visualization, as in Figure 2.6, skips the visualization of the extra nodes.

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

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

The answer is:

```
Minimal resources needed: mass = + (100 g)
```

If we ask for more than the MCDP can provide:

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

we obtain no solutions:

```
This problem is unfeasible.
```

### 2.5. 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 snippets are equivalent:

<table>
<thead>
<tr>
<th>Using ASCII characters</th>
<th>Using Unicode characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>provided capacity &lt;= 500 J</td>
<td>provided capacity ≤ 500 J</td>
</tr>
<tr>
<td>required mass &gt;= 100g</td>
<td>required mass ≥ 100g</td>
</tr>
</tbody>
</table>

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

These two snippets are equivalent:

<table>
<thead>
<tr>
<th>Using ASCII characters</th>
<th>Using Unicode characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{alpha}_1 = \beta^3 + 9.81 \text{ m/s}^2 )</td>
<td>( \alpha_1 = \beta^3 + 9.81 \text{ m/s}^2 )</td>
</tr>
</tbody>
</table>

### 2.6. 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.
1) 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".
```

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

2) Catching other problems

All inequalities will always be of the kind:

```
functionality ≥ 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.
```

2.7. Describing relations between functionality and resources

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

MCDPL contains as primitive operations addition and multiplication.

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

```
capacity = \rho \times mass.
```

This relation can be described in MCDPL as

```
\rho = 4 J / g
required mass ≥ provided capacity / \rho
```

In the graphical representation (Figure 2.8), there is now a connection between capacity and mass, with a DP that multiplies by the inverse of the specific energy.
For example, if we ask for $600 \text{ J}$:

```
$ mcdp\text{-solve} \ linear \ "600 \text{ J}"$
```

we obtain this answer:

```
Minimal resources needed: mass = \{150 \text{ g}\}
```

### 2.8. 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 2.3 the specific energy is given in $\text{kWh/kg}$. PyMCDP will take care of the conversions that are needed, and will introduce a conversion from $\text{J*kg/kWh}$ to $\text{g}$ (Figure 2.9).

For example, Listing 2.3 is the same example with the specific energy given in $\text{kWh/kg}$. The output of the two models are completely equivalent.

Listing 2.3 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, resource, or constant declaration, by writing a string at the end of the line.
2.9. 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 defined from a triplet of functionality space, implementation space, and resource space. According to the diagram in Figure 2.10, one should define the two maps eval and exec, which map an implementation to the functionality it provides and the resources it requires.

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 2.4. 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.

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

Listing 2.4. 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 (Figure 2.11).
2.10. Multiple minimal solutions

The catalogue construct is the first construct we encountered that allows to define MCDPs that have \textit{multiple minimal solutions}. To see this, let’s expand the model in Listing 2.4 to include a few more models and one more resource, \texttt{cost}.

\begin{verbatim}
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
}
\end{verbatim}

Listing 2.5

The numbers (not realistic) were chosen so that \texttt{model2} and \texttt{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 \langle 200 g, 200 USD \rangle or \langle 250 g, 150 USD \rangle.

The number of minimal solutions is not constant: for this example, we have four cases as a function of \texttt{f} (Table 2.1). As \texttt{f} increases, there are 1, 2, 1, and 0 minimal solutions.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Functionality requested & Optimal implementation(s) & Minimal resources needed \\
\hline
0 kWh ≤ \texttt{f} ≤ 500 kWh & model1 & \langle 100 g, 10 USD \rangle \\
500 kWh < \texttt{f} ≤ 600 kWh & model2 & \langle 200 g, 200 USD \rangle \\
& or & \langle 250 g, 150 USD \rangle \\
600 kWh < \texttt{f} ≤ 700 kWh & model4 & \langle 400 g, 400 USD \rangle \\
700 kWh < \texttt{f} ≤ T kWh & (unfeasible) & \emptyset \\
\hline
\end{tabular}
\caption{Cases for model in Listing 2.5}
\end{table}

We can verify these with \texttt{mcdp-solve}. We also use the switch \texttt{"--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 \texttt{f = 50 kWh}:

\begin{verbatim}
$ mcdp-solve --imp catalogue2.try "50 kWh"
\end{verbatim}

we obtain one solution:

\begin{verbatim}
Minimal resources needed: mass, cost = \star\{mass:100 g, cost:10 USD\}
r = \{mass:100 g, cost:10 USD\}
implementation 1 of 1: m = 'model1'
\end{verbatim}
For \( f = 550 \text{kWh} \):

```
$ mcdp-solve --imp catalogue2_try "550 kWh"
```

we obtain two solutions:

Minimal resources needed: mass, cost = \{\langle \text{mass:}\ 200 \text{ g, cost:}\ 200 \text{ USD}\rangle, \langle \text{mass:}\ 250 \text{ g, cost:}\ 150 \text{ USD}\rangle\}

\[ r = \langle \text{mass:}\ 250 \text{ g, cost:}\ 150 \text{ USD}\rangle \]

implementation 1 of 1: \( m = \text{model3} \)

\[ r = \langle \text{mass:}\ 200 \text{ g, cost:}\ 200 \text{ USD}\rangle \]

implementation 1 of 1: \( m = \text{model2} \)

\text{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.

2.11. Coproducts (alternatives)

The \textit{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 \textit{capacity} and the resources to be the \textit{mass \text{[g]}} and the \textit{cost \text{[USD]}} (Figure 2.12).

![Battery1](image)

Figure 2.12

Consider two different battery technologies, characterized by their specific energy \( \text{Wh/kg} \) and specific cost \( \text{Wh/\$} \).

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

First we model the two battery technologies separately as two MCDPs that have the same interface (same resources and same functionality).
Then we can define the **coprod**uct of the two using the keyword `choose` (Listing 2.6). Graphically, the choice is indicated through dashed lines (Figure 2.13).

### 2.12. Composing design problems

The MCDPL language encourages composition and code reuse.

Suppose we define a simple model called `Battery` as in Figure 2.14.
Let us also define the MCDP `Actuation1`, as a relation from `lift` to `power`, as in Listing 2.7.

Suppose the relation between `lift` and `power` is described by the polynomial relation

\[
\text{required power} \geq p_0 + p_1 \cdot l + p_2 \cdot l^2
\]

This is really the composition of five DPs, corresponding to sum, multiplication, and exponentiation (Figure 2.15).

Let us combine these two together.

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`”.

The code in Listing 2.8 creates two sub-design problems, for now unconnected.
The model in Listing 2.8 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:

\[
\text{energy} = \text{provided endurance} \cdot (\text{power required by actuation})
\]

\[
\text{capacity provided by battery} \geq \text{energy}
\]

The visualization of the resulting model has a connection between the two design problems representing the co-design constraint (Figure 2.17).

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:

\[
\text{gravity} = 9.81 \text{ m/s}^2 \\
\text{total mass} = (\text{mass required by battery} + \text{provided payload}) \\
\text{weight} = \text{total mass} \cdot \text{gravity} \\
\text{lift provided by actuation} \geq \text{weight}
\]

Now there is a loop in the co-design diagram (Figure 2.18).

```
Composition.mcdp

mcdp ( 
    provides endurance [s] 
    provides payload [g] 

    actuation = instance `Actuation1 
    battery = instance `Battery 

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

    capacity provided by battery \geq energy 

    # actuation must carry payload + battery 
    gravity = 9.81 \text{ m/s}^2 
    total mass = (\text{mass required by battery} + \text{provided payload}) 
    weight = \text{total mass} \cdot \text{gravity} 
    lift provided by actuation \geq \text{weight} 

    # minimize total mass 
    requires mass [g] 
    required mass \geq \text{total mass}
)
```

Listing 2.10

Figure 2.18

2.13. Describing design patterns

“Templates” are a way to describe reusable design patterns.

For example, the code in Listing 2.10 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:
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 2.11.

```
Interface.mcdp

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

Listing 2.11

Then we can declare a template as in Listing 2.12. The template is visualized as a diagram with a hole (Figure 2.19).

```
ExampleTemplate.mcdp_template

template [
  # T is a generic type that implements the given interface
  T: `Interface
]

mcdp {
  x = instance T
  f provided by x ≥ r required by x + 1
}
```

Listing 2.12

Here is the application to the previous example of battery and actuation. Suppose that we define their “interfaces” as in Listing 2.13 and Listing 2.14.
Then we can define a template that uses them. For example the code in Listing 2.15 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 Figure 2.20 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 2.16.

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

```
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 2.20
2.14. Shortcuts for sums \texttt{sum}

One common pattern is to sum over many resources of the same name. For example, a design might consist of several components, and the budgets must be summed together (Listing 2.17). In this case, it is possible to use the shortcut \( \sum \) (or \texttt{sum}) that allows to sum over resources required with the same name (Listing 2.18).

![Listing 2.17](image1)

![Listing 2.18](image2)

Figure 2.21. Use of the sum shortcut

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

The dual syntax for functionality is also available (Listing 2.19).

![Listing 2.19](image3)

![Listing 2.20](image4)

Listing 2.20. Example use of the \( \sum \) syntax for functionalities.
PART B
MCDPL Language reference

This part gives a formal description of the MCDPL language.

Contents

Unit B-1 - MCDPL grammar .................................................................21
Unit B-2 - Posets and their values .......................................................25
Unit B-3 - Defining NDPS .................................................................31
Unit B-4 - Operations on NDPS ..........................................................35
1.1. Characters
A MCDP file is a sequence of Unicode code-points that belong to one of the classes described in Table 1.1.

All files on disk are assumed to be encoded in UTF-8.

<table>
<thead>
<tr>
<th>Class</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latin letters</td>
<td>abcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td></td>
<td>ABCDEFGHJKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>Underscore</td>
<td>_</td>
</tr>
<tr>
<td>Greek letters</td>
<td>αβγδεζηθικλμνξοπρστυφχψω</td>
</tr>
<tr>
<td></td>
<td>ΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩ</td>
</tr>
<tr>
<td>Digits</td>
<td>0123456789</td>
</tr>
<tr>
<td>Subscripts</td>
<td>0123456789</td>
</tr>
<tr>
<td>Comment delimiter</td>
<td>#</td>
</tr>
</tbody>
</table>

**Table 1.1. Character classes**

<table>
<thead>
<tr>
<th>Class</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>String delimiters</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>Backtick</td>
<td>\</td>
</tr>
<tr>
<td>Parentheses</td>
<td><a href=""> </a></td>
</tr>
<tr>
<td>Operators</td>
<td>&lt;= &gt;= ≤ ≥ ≼ ≽ ≈ =</td>
</tr>
<tr>
<td>Tuple-making</td>
<td>&lt; &gt; ( )</td>
</tr>
<tr>
<td>Arrows glyphs</td>
<td>↔ ⟷ ⟹ ⟟ ⟸ ⟹ ⟹ ⟻ ⟺ ⟼ ⟼</td>
</tr>
<tr>
<td>Math</td>
<td>* - + ^</td>
</tr>
<tr>
<td>Other glyphs</td>
<td>× T ⊥ ∅ N</td>
</tr>
<tr>
<td></td>
<td>⊤ ⊥ ⊥ ⊥ ⊥ \ / ±</td>
</tr>
<tr>
<td></td>
<td>↑ ↓ % $ \ ⊥ \</td>
</tr>
</tbody>
</table>

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

1.3. 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|subscript]?
```

Here are some examples of good identifiers: a, a₄, α, alpha, α.

The reserved keywords are shown in Table 1.2.
Table 1.2. Reserved keywords

<table>
<thead>
<tr>
<th>Reserved keywords</th>
<th>approx_lower</th>
<th>constant</th>
<th>product</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>approx_upper</td>
<td>coproduct</td>
<td>provided</td>
</tr>
<tr>
<td>between</td>
<td>approxu</td>
<td>eversion</td>
<td>provides</td>
</tr>
<tr>
<td>bottom</td>
<td>assert_empty</td>
<td>flatten</td>
<td>required</td>
</tr>
<tr>
<td>emptyset</td>
<td>assert_equal</td>
<td>for</td>
<td>requires</td>
</tr>
<tr>
<td>Int</td>
<td>assert_geq</td>
<td>ignore</td>
<td>solve</td>
</tr>
<tr>
<td>lowersets</td>
<td>assert_gt</td>
<td>ignore_resources</td>
<td>solve_f</td>
</tr>
<tr>
<td>maximals</td>
<td>assert_leq</td>
<td>implemented-by</td>
<td>solve_r</td>
</tr>
<tr>
<td>imals</td>
<td>assert lt</td>
<td>implements</td>
<td>sum</td>
</tr>
<tr>
<td>Nat</td>
<td>assert_nonempty</td>
<td>instance</td>
<td>specialize</td>
</tr>
<tr>
<td>Rcomp</td>
<td>by</td>
<td>interface</td>
<td>take</td>
</tr>
<tr>
<td>top</td>
<td>canonical</td>
<td>lowerclosure</td>
<td>template</td>
</tr>
<tr>
<td>uncertain</td>
<td>catalogue</td>
<td>mcdp</td>
<td>upperclosure</td>
</tr>
<tr>
<td>uppersets</td>
<td>choose</td>
<td>namedproduct</td>
<td>using</td>
</tr>
<tr>
<td>abstract</td>
<td>code</td>
<td>poset</td>
<td>variable</td>
</tr>
<tr>
<td>add_bottom</td>
<td>compact</td>
<td>powerset</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.3. Deprecated keywords

<table>
<thead>
<tr>
<th>Deprecated keywords</th>
<th>interval</th>
<th>experimental</th>
</tr>
</thead>
<tbody>
<tr>
<td>deprecated</td>
<td>s</td>
<td>interval</td>
</tr>
<tr>
<td>ceilsqrt</td>
<td>finite_poset</td>
<td>no need to be keyword</td>
</tr>
<tr>
<td>set-of</td>
<td>dp</td>
<td>pow</td>
</tr>
<tr>
<td>any-of</td>
<td>from</td>
<td>dimensionless</td>
</tr>
<tr>
<td>load</td>
<td>sub</td>
<td>new</td>
</tr>
<tr>
<td></td>
<td>dptype</td>
<td></td>
</tr>
</tbody>
</table>

1.4. Syntactic equivalents

MCDPL allows a number of Unicode glyphs as an abbreviations of a few operators (Table 1.4).

1) 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 “$x^t$”.
### Table 1.4. Unicode glyphs and syntactically equivalent ASCII

<table>
<thead>
<tr>
<th>Unicode</th>
<th>ASCII</th>
<th>Unicode</th>
<th>ASCII</th>
<th>Unicode</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>≧ or ≫</td>
<td>&gt;=</td>
<td>≼ or ≪</td>
<td>&lt;=</td>
<td>≪ or ≻</td>
<td>&gt;=</td>
</tr>
<tr>
<td>⋮</td>
<td>⟨⋯⟩</td>
<td>‡</td>
<td>⊤</td>
<td>⊤</td>
<td>Top</td>
</tr>
<tr>
<td>⟨⋯⟩</td>
<td>⟨⋯⟩</td>
<td>≺</td>
<td>⊥</td>
<td>⊥</td>
<td>Bottom</td>
</tr>
<tr>
<td>⊤</td>
<td>Top</td>
<td>⊤</td>
<td>⊤</td>
<td>⊤</td>
<td>Top</td>
</tr>
<tr>
<td>⊥</td>
<td>Bottom</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>Bottom</td>
</tr>
<tr>
<td>⊤</td>
<td>Top</td>
<td>⊤</td>
<td>⊤</td>
<td>⊤</td>
<td>Top</td>
</tr>
<tr>
<td>⊥</td>
<td>Bottom</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>Bottom</td>
</tr>
<tr>
<td>a¹</td>
<td>a^1</td>
<td>a²</td>
<td>a^2</td>
<td>a³</td>
<td>a^3</td>
</tr>
<tr>
<td>a⁴</td>
<td>a^4</td>
<td>a⁵</td>
<td>a^5</td>
<td>a⁶</td>
<td>a^6</td>
</tr>
<tr>
<td>a⁷</td>
<td>a^7</td>
<td>a⁸</td>
<td>a^8</td>
<td>a⁹</td>
<td>a^9</td>
</tr>
<tr>
<td>×</td>
<td>x</td>
<td>a₀</td>
<td>a_0</td>
<td>a₁</td>
<td>a_1</td>
</tr>
<tr>
<td>a₂</td>
<td>a_2</td>
<td>a₃</td>
<td>a_3</td>
<td>a₄</td>
<td>a_4</td>
</tr>
<tr>
<td>a⁵</td>
<td>a_5</td>
<td>a₆</td>
<td>a_6</td>
<td>a₇</td>
<td>a_7</td>
</tr>
<tr>
<td>a₈</td>
<td>a_8</td>
<td>a₉</td>
<td>a_9</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

#### 2) 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: $a_i$ is valid, while “$a_i b$” is not a valid identifier.

### 1.5. 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 “$\text{alpha}_1$” and “$\text{α}_1$”.

Every Greek letter is converted to its name. It is syntactically equivalent to write “$\text{alpha}_\text{material}$” or “$\text{α}_\text{material}$”.

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 identifier “$\text{alphabet}$” is not converted to “$\text{αbet}$”.

Table 1.5 shows the Greek letters supported and their translitteration. Note that there is a difference between lower case and upper case.
<table>
<thead>
<tr>
<th>Greek Letter</th>
<th>Lowercase</th>
<th>Uppercase</th>
<th>Lowercase</th>
<th>Uppercase</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>alpha</td>
<td>I</td>
<td>Iota</td>
<td>O</td>
</tr>
<tr>
<td>β</td>
<td>beta</td>
<td>ι</td>
<td>iota</td>
<td>o</td>
</tr>
<tr>
<td>Γ</td>
<td>Gamma</td>
<td>K</td>
<td>Kappa</td>
<td>Φ</td>
</tr>
<tr>
<td>χ</td>
<td>gamma</td>
<td>κ</td>
<td>kappa</td>
<td>ψ</td>
</tr>
<tr>
<td>Δ</td>
<td>Delta</td>
<td>Λ</td>
<td>Lambda</td>
<td>Π</td>
</tr>
<tr>
<td>δ</td>
<td>delta</td>
<td>λ</td>
<td>lambda</td>
<td>π</td>
</tr>
<tr>
<td>Ε</td>
<td>Epsilon</td>
<td>Μ</td>
<td>Mu</td>
<td>Ψ</td>
</tr>
<tr>
<td>ε</td>
<td>epsilon</td>
<td>μ</td>
<td>mu</td>
<td>ψ</td>
</tr>
<tr>
<td>Η</td>
<td>Eta</td>
<td>Ν</td>
<td>Nu</td>
<td>Χ</td>
</tr>
<tr>
<td>η</td>
<td>eta</td>
<td>ν</td>
<td>nu</td>
<td>χ</td>
</tr>
<tr>
<td>Γ</td>
<td>Gamma</td>
<td>Ω</td>
<td>Omega</td>
<td>Ρ</td>
</tr>
<tr>
<td>γ</td>
<td>gamma</td>
<td>ω</td>
<td>omega</td>
<td>ρ</td>
</tr>
</tbody>
</table>
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 2.1 shows the basic posets that are built in: natural numbers, nonnegative real numbers, and nonnegative real numbers that have units associated to them.

### Table 2.1. Built-in posets

<table>
<thead>
<tr>
<th>MCDPL poset</th>
<th>ideal poset</th>
<th>Python representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nat</td>
<td>( (\mathbb{N} \cup T, \leq) )</td>
<td>int plus a special Top object</td>
</tr>
<tr>
<td>Rcomp</td>
<td>( (\mathbb{R}^+ \cup T, \leq) )</td>
<td>float plus a special Top object</td>
</tr>
<tr>
<td>g</td>
<td>( (\mathbb{R}^*_+ [g] \cup T, \leq) )</td>
<td>float plus a special Top object</td>
</tr>
</tbody>
</table>

#### 2.1. Natural numbers \( \mathbb{N} \)

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

The natural numbers with completion are expressed as “Nat” or with the Unicode letter “\( \mathbb{N} \)”. Their values are denoted using the syntax “Nat:42” or simply “42”.

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

#### 2.2. Nonnegative floating point numbers \( \mathbb{R}^+ \)

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

\[
\forall a: \quad a + T = T \\
\forall a: \quad a \times T = T
\]

This poset is indicated in MCDPL by \( \text{Rcomp} \) or \( \mathbb{R} \). Values belonging to this poset are indicated with the syntax their values as \( \text{Rcomp:}42.0 \), \( \mathbb{R}:42.0 \), or simply \( 42.0 \).

Internally, \( \mathbb{R}^*_+ \) 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 in each argument (which they are).

#### 2.3. 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 indi-
cated using the syntax in Table 2.2.

<table>
<thead>
<tr>
<th>ideal poset syntax for poset syntax for values</th>
<th>( \mathbb{R}_+^* [g], \leq )</th>
<th>( \mathbb{R}_+^* [J], \leq )</th>
<th>( \mathbb{R}_+^* [m/s], \leq )</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>J</td>
<td>m/s</td>
<td></td>
</tr>
<tr>
<td>1.2 g</td>
<td>20 J</td>
<td>23 m/s</td>
<td></td>
</tr>
</tbody>
</table>

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 with an equivalence relation. Call this group of units \( U \) and its elements \( u, v, \ldots \in U \). 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 \( x + y = |x| + \alpha_u^v |y| \), where \( \alpha_u^v \) is a table of conversion factors, and \( |x| \) is the absolute numerical value of \( x \).

In practice, this means that MCDPL thinks that \( 1 \, \text{kg} + 1 \, \text{g} \) is equal to \( 1.001 \, \text{kg} \). Addition is not strictly commutative, because \( 1 \, \text{g} + 1 \, \text{kg} \) is equal to \( 1001 \, \text{g} \), which is equivalent, but not equal, to \( 1.001 \, \text{kg} \).

### 2.4. 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 2.21. This declaration defines a poset with 5 elements \( a, b, c, d, e \) and with the given order relations, as displayed in Figure 2.1.

```plaintext
MyPoset.mcdp_poset

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

Figure 2.1

Listing 2.21. Definition of a custom poset

The name of the poset, \texttt{MyPoset}, comes from the filename \texttt{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 “\texttt{`MyPoset’}”.

To refer to its elements, use the notation \texttt{`MyPoset: element’} (Listing 2.22).
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:

\[
\langle\text{poset}\rangle \times \langle\text{poset}\rangle \times \cdots \times \langle\text{poset}\rangle
\]

For example, the expression \( J \times A \) 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:

\[
<\text{value}, \text{value}, \ldots, \text{value}>
\]

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 \( (J \times A) \times (m \times s \times \text{Nat}) \).

2.6. **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.

\[
\text{product(energy:J, current:A)}
\]

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:

\[
\langle\text{resource}\rangle.\langle\text{label}\rangle
\]

For example:
2.7. Power sets set-of $\mathcal{P}$

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

The syntax is either of these:

$$\mathcal{P}((\text{poset}))$$

$$\text{set-of}((\text{poset}))$$

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

$$\{ \text{(value)}, \text{(value)}, \ldots, \text{(value)} \}$$

For example, the value $\{1,2,3\}$ is an element of the poset $\mathcal{P}(\mathbb{N})$.

1) Upper and lower sets and closures $\text{UpperSets LowerSets upperclosure + lowerclosure}$

Upper sets and lower sets can be described by the syntax

$$\text{UpperSets}(\text{poset})$$

$$\text{LowerSets}(\text{poset})$$

For example, $\text{UpperSets}(\mathbb{N})$ 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 $\text{upperclosure}$ or its abbreviation $\uparrow$. The syntax is:

$$\text{upperclosure} \ (\text{set})$$

$$\uparrow \ (\text{set})$$

For example: $\uparrow \{(2 \ g, \ 1 \ m)\}$ denotes the principal upper set of the element $\{2 \ g, \ 1 \ m\}$ in the poset $\mathbb{g} \times \mathbb{m}$.

2.8. Defining uncertain constants between ±

MCDPL allows to describe interval uncertainty for variables and expressions.

There are three syntaxes accepted (Table 2.3).

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit bounds</td>
<td>$x = \text{between} \ (\text{lower bound}) \ \text{and} \ (\text{upper bound})$</td>
</tr>
<tr>
<td>Median plus or minus tolerance</td>
<td>$x = (\text{median}) \ \pm \ (\text{absolute tolerance})$</td>
</tr>
<tr>
<td>Median plus or minus percent</td>
<td>$x = (\text{median}) \ \pm \ (\text{percent tolerance}) \ %$</td>
</tr>
</tbody>
</table>

The character “±” can be written as “$\pm$”.

For example, Table 2.4 shows the different ways in which a constant can be declared to be between 9.95 kg and 10.05 kg.
### Table 2.4

**Explicit bounds**

\[ c = \text{between } 9.95 \, \text{kg and } 10.05 \, \text{kg} \]

\[ c = 10 \, \text{kg} \]

\[ \delta = 50 \, \text{g} \]

\[ x = \text{between } c-\delta \text{ and } c+\delta \]

**Median plus or minus tolerance**

\[ c = 10 \, \text{kg} \pm 50 \, \text{g} \]

**Median plus or minus percent**

\[ c = 10 \, \text{kg} \pm 0.5\% \]

These expressions can be used also in tables (Listing 2.23).

```plaintext
catalogue {
    provides energy [J]
    requires mass [kg]
    100 kWh ± 5% ↔ 1.2 kg ± 100 g
}
```

Listing 2.23

It is also possible to describe parametric uncertain relations (Listing 2.24).

```plaintext
mcdp {
    provides energy [J]
    requires mass [kg]
    specific_energy = \text{between } 100 \, \text{kWh/kg and } 120 \, \text{kWh/kg}
    required mass \cdot \text{specific_energy} \geq \text{provided energy}
}
```

Listing 2.24

### 2.9. Other ways to specify values

1) **Top and bottom**

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

   \[
   \text{Top (poset)} \quad \text{T (poset)} \\
   \text{Bottom (poset)} \quad \text{⊥ (poset)}
   \]

   For example, \( \text{T V} \) indicates the top of the \( V \).

2) **Minimals and maximals**

   The expressions \text{Minimals (poset)} \text{ and Maximals (poset)} denote the set of minimal and maximal elements of a poset.
For example, assume that the poset $\text{MyPoset}$ is defined as in Figure 2.2. Then $\text{Maximals } \text{MyPoset}$ is equivalent to $b$ and $d$, and $\text{Minimals } \text{MyPoset}$ is equivalent to $a$, $e$, $c$.

![Figure 2.2](image)

3) The empty set $\text{EmptySet}$

To denote the empty set, use the keyword $\text{EmptySet}$:

\[
\text{EmptySet} \langle \text{poset} \rangle
\]

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.) $\text{EmptySet} J$ is an empty set of energies, and $\text{EmptySet} V$ is an empty set of voltages, and the two are not equivalent.
An NDP (named design problem) is a design problem that has associated names for its functionality and resources.

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`.

3.1. Loading an NDP from library

Just like any other universe, it is possible to refer to an NDP defined in another file or another library by using the backtick syntax.

If this syntax is used:

```
NDP |= `""" name
```

then the interpreter will look for a file called `name.mcdp` and continue loading it.

Alternatively, the syntax can be

```
NDP |= `""" library "."" name
```

The interpreter will then look for `name.mcdp` in the library `library`.

3.2. The `mcdp{ }` syntax

An NDP can be created using a declaration enclosed in `mcdp { ... }`/ The declaration must be comprised of an optional comment and zero or more line statements:

```
NDP |= "mcdp" "(" [comment] line-statement* ")"
```

A line statement can be either a declaration or a constraint:

```
line-statement = declaration | constraint
```

A declaration is either a functionality declaration or a resource declaration:
A functionality declaration can take one of the four forms in Table 3.1.

**Table 3.1. Syntax for functionality declaration**

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Equivalent constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>provides (fn) [ (poset) ]</td>
<td>provides (fn) [ (poset) ] provided (fn) ≤ (fn) provided by (dp)</td>
</tr>
<tr>
<td>provides (fn) = (expression)</td>
<td>provides (fn) [ (poset) ] provided (fn) ≤ (expression)</td>
</tr>
<tr>
<td>provides (fn) using (dp)</td>
<td></td>
</tr>
<tr>
<td>provides (fn1), (fname), … using (dp)</td>
<td></td>
</tr>
</tbody>
</table>

The forms 2,3,4 are shortcuts. Their equivalent constructs are shown in Table 3.2.

**Table 3.2. Equivalence of constructs for functionality declaration**

A resource declaration can take one of the forms in Table 3.3.

**Table 3.3. Syntax for resource declaration**

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Equivalent constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>requires (rn) [ (poset) ]</td>
<td></td>
</tr>
<tr>
<td>requires (rn) = (expression)</td>
<td></td>
</tr>
<tr>
<td>requires (rn) for (dp)</td>
<td></td>
</tr>
<tr>
<td>requires (rn1), (rn2), … for (dp)</td>
<td></td>
</tr>
</tbody>
</table>

The forms 2,3,4 are shortcuts. Their equivalent constructs are shown in Table 3.4.
3.3. The catalogue syntax

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

\[
\text{NDP} \models \text{catalogue1 | catalogue2}
\]

A catalogue declaration is comprised of zero or more functionality/resource declarations and zero or more records. There are two types of records to be used:

\[
\text{catalogue1} = \text{"catalogue" \{ \"declarations\* record1\* \" \"} \\
\text{catalogue2} = \text{"catalogue" \{ \"declarations\* record2\* \" \"}
\]

In the first syntax, the user lists the elements of the implementation space as strings, and then assigns to each implementation a certain value of the functionality and resources:

\[
\text{record1} = f_1, f_2, \ldots \ "\leftrightarrow\" \text{imname} \ "\leftrightarrow\" r_1, r_2, \ldots
\]

In the second syntax, the user specifies a relation between functionality and resources, and the implementation names are implicit:

\[
\text{record2} = f_1, f_2, \ldots \ "\leftrightarrow\" r_1, r_2, \ldots
\]

Figure 3.1 shows examples of the two syntaxes.
Figure 3.1. Two ways to describe a catalogue.
UNIT B-4
Operations on NDPs

The software allows the manipulation of NDPs using a set of operations. These are the operations possible on NDPs:

- `abstract ⟨NDP⟩` Abstraction
- `compact ⟨NDP⟩` Compactification
- `flatten ⟨NDP⟩` Flattening
- `canonical ⟨NDP⟩` Canonical form
- `approx_lower ⟨n⟩, ⟨NDP⟩` Approximation (lower bound)
- `approx_upper ⟨n⟩, ⟨NDP⟩` Approximation (upper bound)

4.1. Compactification (compact)
The command `compact ⟨NDP⟩` 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, if this is the original:

```plaintext
cmdp {
  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
}
```
The compacted version has only one edge between the two NDPs:

4.2. **Ignoring resources/functionality (ignore)**

The `ignore` command allows to ignore some of the functionality or resources of an NDP. Suppose the functionality \( f \) has type \( F \). Then:

\[
\text{ignore } \langle f \rangle \text{ provided by } \langle dp \rangle
\]

is equivalent to

\[
\langle f \rangle \text{ provided by } \langle dp \rangle \geq \text{any-of}(\text{Minimals } (F))
\]

Equivalently,

\[
\text{ignore } \langle \text{resource} \rangle \text{ required by } \langle dp \rangle
\]

is equivalent to

\[
\langle \text{resource} \rangle \text{ required by } \langle dp \rangle \leq \text{any-of}(\text{Maximals } (\text{space}))
\]

4.3. **Flattening (flatten)**

It is easy to create recursive composition in MCDP.

For example, the code in Listing 4.25 describes an MCDP composed of nested MCDPs.
The "flattening" operation erases the borders between subproblems. The syntax is:

```
flatten Composition1
```

The result for the example is shown in Figure 4.1.

**4.4. Abstraction (abstract)**

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.

```
abstract1.mcdp

cdp {
  provides f [m]
  requires r [m]

  required r ≽ provided f + 10 m
}
```

```
abstract `abstract1
```

Figure 4.2. Example use of the `abstract` keyword.
Bibliography

