Reasoning about Study Regulations in Answer Set Programming (Preliminary Report)

Susana Hahn[1,2], Cedric Martens[1], Amadé Nemes[1], Henry Otunuya[1,*], Javier Romero[1], Torsten Schaub[1,2] and Sebastian Schellhorn[1]

[1]*University of Potsdam, Germany*
[2]*Potassco Solutions, Germany*

Abstract

We are interested in automatizing reasoning with and about study regulations, catering to various stakeholders, ranging from administrators, over faculty, to students at different stages. Our work builds on an extensive analysis of various study programs at the University of Potsdam. The conceptualization of the underlying principles provides us with a formal account of study regulations. In particular, the formalization reveals the properties of admissible study plans. With these at end, we propose an encoding of study regulations in Answer Set Programming that produces corresponding study plans. Finally, we show how this approach can be extended to a generic user interface for exploring study plans.

Keywords

Answer Set Programming, Study regulations and plans

1. Introduction

Study regulations govern our teaching at universities. by specifying requirements to be met by students to earn a degree. This involves different stakeholders: faculty members designing study programs, administrative and legal staff warranting criteria, like studyability, faculty members teaching the corresponding programs as well as supervising their execution on examination boards, study advisors consulting students, and of course, students studying accordingly.

Given this impressive spectrum of use-cases, it is quite remarkable that study regulations are usually rather sparse and leave many aspects to the commonsense of the respective users. This is needed to cope with their inherent incomplete, inconsistent, and evolving nature. For instance, often study regulations leave open minor dependencies among modules. Sometimes associated courses overlap and certain modules cannot be taken in the same semester. And finally, studying happens over time, students' perspectives may change and faculty may rotate. Often these phenomena are compensated by changes, preferences, recommendations, defaults, etc. In fact, this richness in issues and notions from Knowledge Representation and Reasoning

ASPOCP'23: 16th Workshop on Answer Set Programming and Other Computing Paradigms, London, UK, July, 2023
*Corresponding author.
otunuya@uni-potsdam.de (H. Otunuya)
0000-0002-1012-7887 (H. Otunuya); 0000-0002-7456-041X (T. Schaub)

(KRR) makes study regulations a prime candidate for a comprehensive benchmark for KRR formalisms.

This work is part of a project conducted at the University of Potsdam to assist different users by automatizing study regulations. These users range from study administrators, over faculty in different functions, to prospective and advanced students. We started by analyzing more than a dozen different study regulations in order to identify their underlying principles. The conceptualization of the basic principles led us to a formal account of basic study regulations, presented in Section 2. For illustration, we provide the formalization of the master program *Cognitive Systems*. Further, more specialized concepts in other study regulations are presented in Section 3. The formalization of study regulations reveals the properties of admissible study plans. To automatize reasoning about study regulations and their study plans, we capture their properties in Answer Set Programming (ASP; [1]), a declarative problem solving paradigm, tailored for knowledge representation and reasoning. The ASP-based encoding of basic study regulations is discussed in Section 4. Moreover, we show in Section 5 how this encoding can be used together with an ASP-driven user interface to browse through study plans of given study regulations. We conclude in Section 7.

2. Conceptualizing study regulations

The basic concept of our study regulations are modules. Accordingly, a semester is composed of a set of modules and a study plan is a finite sequence of semesters. More formally, given a set M of modules, a *study plan* of n semesters is a sequence $(S_i)_{i=1}^n$ where $S_i \subseteq M$ for $1 \leq i \leq n$. Study regulations specify legal study plans. To capture this, we propose an abstract characterization of study regulations and show how they induce legal study plans.

A basic *study regulation* is a tuple $(M, G, c, s, l, u, R_g, R_t)$, where

1. M is a set of modules,
2. $G \subseteq 2^M$ distinguishes certain groups of modules,
3. $c : M \to \mathbb{N}$ gives the credits of each module,
4. $s : M \to \{w, s, e\}$ assigns a regular semester to a module,
5. $l : G \to \mathbb{N}$ returns the lower-bound of the credits of a module group,
6. $u : G \to \mathbb{N}$ returns the upper-bound of the credits of a module group,
7. $R_g \subseteq 2^{(2^M)^n}$ is a set of global constraints expressing study regulations, and
8. $R_t \subseteq 2^{(2^M)^n}$ is a set of temporal constraints expressing study regulations.

The module groups in G allow us to structure the modules and to express group-wise regulations. Functions c and s give the credit points of a module and its turnus,[1] viz. in winter, summer, or each semester (indicated by *w*, *s*, *e*), respectively. The elements l and u are partial functions delineating the number of credits obtained per module group; a specific number of credits is captured by an equal lower and upper bound. The regulation constraints in R_g and R_t are represented extensionally: each constraint r in $R_g \cup R_t$ is represented by the sequences of sets of modules $r \subseteq (2^M)^n$ satisfying it. While the sets of constraints R_g and R_t share the same

[1]Winter and summer semesters are associated with odd and even positions in a sequence, respectively (see below).

mathematical structure, they differ in purpose and are therefore separated for clarity's sake. R_t expresses temporal constraints over the sequence of semesters, while R_g does not make use of this sequential structure, and rather expresses global constraints over the entire set of modules.

A study plan for a basic study regulation is a finite sequence of sets of modules satisfying all regulation constraints. More precisely, a sequence $(S_i)_{i=1}^n$ of modules of length n such that $S_i \subseteq M$ for $1 \leq i \leq n$ is a *study plan* for $(M, G, c, s, l, u, R_g, R_t)$ if $(S_i)_{i=1}^n \in \bigcap_{r \in R_g \cup R_t} r$.

Finally, we call modules exogenous if they are imposed by external means, eg. by an examination board. The specific choice of these modules is determined case by case.

Example 1 (Cognitive systems). *As an example, consider the study regulations of the master program* Cognitive Systems *offered at the University of Potsdam.*[2] *This program offers a combination of modules in Natural language processing, Machine learning, and Knowledge representation and reasoning.*

These subjects are reflected by the three mandatory base modules, joined in B in (3)*. Since each module yields 9 credits, their obligation is achieved by requiring that the modules stemming from B must account for 27 credits.*[3] *While the amount is specified in* (12) *and* (13)*, the actual constraint is imposed in* (15)*. The same constraint is used for choosing two among three possible project modules from P. The optional modules in group O are handled similarly, just that only 24 credits from 36 possible credits are admissible. That is, four out of nine modules must be taken. The freedom of which four the student may choose is restricted by the examination board by imposing the study of up to two foundational modules $E \subseteq F$, which must be the only modules taken from module group F, as formalized in constraint* (14)*. The total number of credits over all modules must equal 120.*

Finally, an internship, im, and a thesis, msc, are imposed in (16)*. This brings us to the temporal regulation in* (20) *requiring that at least 90 credits are accumulated before conducting a thesis. The temporal regulations in* (18) *and* (19) *ensure that modules are taken in the right season. And finally* (17) *makes sure that modules are chosen at most once.*

Let $E \subseteq F$ be some exogenous set of modules in the following example; and let $\overline{S}$ stand for $\bigcup_{i=1}^n S_i$, and $M_w = \{m \in M \mid s(m) = w\}$ and $M_s = \{m \in M \mid s(m) = s\}$. Then, the study regulations of the master program Cognitive Systems *with respect to E can be formalized as follows.*

$$M = B \cup F \cup A \cup P \cup \{im, msc\} \tag{1}$$
$$G = \{B, F, A, O, P, M\} \tag{2}$$
$$B = \{bm_i \mid i = 1..3\} \tag{3}$$
$$F = \{fm_i \mid i = 1..3\} \tag{4}$$
$$A = \{am_{i,j} \mid i = 1..3, j = 1, 2\} \tag{5}$$
$$O = F \cup A \tag{6}$$
$$P = \{pm_i \mid i = 1..3\} \tag{7}$$
$$c = \{bm_i \mapsto 9 \mid i = 1..3\} \cup \{am_{i,j} \mapsto 6 \mid i = 1..3, j = 1, 2\} \cup \tag{8}$$

[2] Available at https://www.uni-potsdam.de/fileadmin/projects/studium/docs/03_studium_konkret/07_rechtsgrundlagen/studienordnungen/StO_CogSys_EN.pdf

[3] This is not our way of modeling mandatory courses but rather reflects the actual regulation.

$$\{fm_i \mapsto 6 \mid i = 1..3\} \cup \{pm_i \mapsto 12 \mid i = 1..3\} \cup \{im \mapsto 15, msc \mapsto 30\} \tag{9}$$

$$s = \{bm_1 \mapsto w, bm_2 \mapsto s, bm_3 \mapsto w\} \cup \{am_{i,j} \mapsto e \mid i = 1..3, j = 1, 2\} \cup \tag{10}$$

$$\{fm_i \mapsto w \mid i = 1..3\} \cup \{pm_i \mapsto e \mid i = 1..3\} \cup \{im \mapsto e, msc \mapsto e\} \tag{11}$$

$$l = \{B \mapsto 27, O \mapsto 24, P \mapsto 24, M \mapsto 120\} \tag{12}$$

$$u = \{B \mapsto 27, O \mapsto 24, P \mapsto 24, M \mapsto 120\} \tag{13}$$

$$R_g = \{\, \{(S_i)_{i=1}^n \subseteq M^n \mid |E| \leq 2, \overline{S} \cap F = E\}, \tag{14}$$

$$\{(S_i)_{i=1}^n \subseteq M^n \mid l(H) \leq \textstyle\sum_{m \in H \cap \overline{S}} c(m) \leq u(H)\} \text{ for all } H \in \{B, O, P, M\}, \tag{15}$$

$$\{(S_i)_{i=1}^n \subseteq M^n \mid \{im, msc\} \subseteq \overline{S}\}\, \} \tag{16}$$

$$R_t = \{\, \{(S_i)_{i=1}^n \subseteq M^n \mid S_i \cap S_j = \emptyset, 1 \leq i < j \leq n\} \tag{17}$$

$$\{(S_i)_{i=1}^n \subseteq M^n \mid M_w \cap S_{2k} = \emptyset, 1 \leq 2k \leq n, k \in \mathbb{N}\} \tag{18}$$

$$\{(S_i)_{i=1}^n \subseteq M^n \mid M_s \cap S_{2k-1} = \emptyset, 1 \leq 2k-1 \leq n, k \in \mathbb{N}\} \tag{19}$$

$$\{(S_i)_{i=1}^n \subseteq M^n \mid msc \in S_k \text{ implies } \textstyle\sum_{1 \leq i < k} \sum_{m \in S_i} c(m) \geq 90, k \in \mathbb{N}\}\, \} \tag{20}$$

If the set of exogenous modules given by the examination board is, for example, $E = \{fm_1\}$, one admissible study plan spanning four semesters is $S = (S_i)_{i=1}^4$, where

$$S_1 = \{bm_1, bm_3, fm_1, am_{1,2}\} \tag{21}$$

$$S_2 = \{bm_2, am_{2,1}, pm_1\} \tag{22}$$

$$S_3 = \{im, pm_3, am_{3,1}\} \tag{23}$$

$$S_4 = \{msc\} \tag{24}$$

This plan comprises 120 credits, although the load per semester varies.

For illustration, let us verify that our study plan belongs to the ones in (14) *and* (15) *for $H = O$. Indeed constraint* (14) *is satisfied as we have $F \cap \overline{S} = \{fm_1\} = E$, and thus S is an element of constraint* (14)*. With regards to* (15)*, we have*

$$O \cap \overline{S} = \{fm_1, am_{1,2}, am_{2,1}, am_{3,1}\} \tag{25}$$

which makes us check whether our study plan satisfies

$$l(O) = 24 \leq \textstyle\sum_{m \in \{fm_1, am_{1,2}, am_{2,1}, am_{3,1}\}} c(m) \leq 24 = u(O) \tag{26}$$

This is indeed the case since $c(fm_1) + c(am_{1,2}) + c(am_{2,1}) + c(am_{3,1}) = 24$. Hence, our study plan is an element of constraint (15)*.*

Although the above specification reflects the legal study regulation, it leaves lots of ambiguities behind. For instance, the number of credits per semester is left open, as is the order of the modules. The guideline is usually to take around 30 credits per semesters but this is not enforced. Similarly, basic modules in B should be taken before advances ones in A, again this is neither enforced nor always possible. Since these constraints are usually soft, they are left to the students and/or their study advisors.

3. Refinements

This section presents some concepts of interest that go beyond basic study regulations.

3.1. Specialization

Some study regulations allow for specializations. This provides students the opportunity to specialize in some area within their field of study. For example, in a biology program, one may specialize in ecology, genetics, or zoology. Each focus may contain different groups of modules.

To govern such specializations, which we refer to as focuses, we augment our formalization of study regulations by the following concepts:

- $F \subseteq 2^{2^M}$ is a set of focuses, and
- $R_g(H)$ and $R_t(H)$ are sets of global and temporal constraints parametrized by some focus $H \in F$.

The parametrized constraints allow for formalizing focus-specific regulations. For instance, a constraint on the credit points for a given focus $H \in F$ can be formalized as follows.

$$\{(S_i)_{i=1}^n \subseteq M^n \mid \text{ for all } V \in H : l(V) \leq \textstyle\sum_{m \in (V \cap \overline{S})} c(m) \leq u(V)\}$$

3.2. Modules dependency

Some module descriptions have a prerequisite section which details what modules or competences are required or recommended before participating in that particular module. These prerequisites are represented as hard or soft constraints, respectively. For example, the module "Introduction to Computer Science" may be required before "Data Structures". For capturing this concept in terms of hard constraints between modules, we extend our formalization as follows:

- $D \subseteq M \times M$ relates pairs of dependent modules.

For each $(m_1, m_2) \in D$ (read as m_2 depends on m_1), if m_1 and m_2 are in a study plan, the semester of m_1 must be before that of m_2.

We formalize the constraint as:

$$\{(S_i)_{i=1}^n \subseteq M^n \mid \text{ for all } (m_1, m_2) \in D : m_2 \in S_i \text{ implies } m_1 \in S_j, j < i\}$$

3.3. Blocking modules

Another concept are blocking modules. A study regulation may define a set of elective modules that block each other. For example, either version A or B of a module can be chosen, i.e. selecting the module version A blocks the choice of module version B and vice versa. For capturing this concept, we extend our formalization by the following:

- $B \subseteq M \times M$ relates pairs of modules blocking each other.

For each pair of modules $(m_1, m_2) \in B$, either module m_1 or m_2 can be part of a valid study plan but not both. We formalize the corresponding constraint by:

$$\{(S_i)_{i=1}^n \subseteq M^n \mid \text{ for all } (m_1, m_2) \in B : |\{m_1, m_2\} \cap \overline{S}| \leq 1\}$$

3.4. Examination

In order to address achievements and completion of a module, we depend on examinations. Examinations are bound to a module and each has an unique identifier. We distinguish between *primary examinations* (e.g. passing oral exams) and *secondary examinations* (e.g. passing 50% of exercises). Formally:

- $E_p(m)$ contains sets of possible combinations of primary examinations needed to accomplish module $m \in M$, and
- $E_s(m)$ contains secondary examinations needed to accomplish module $m \in M$.

Let $(E_i)_{i=1}^n$ be a sequence of examinations of length n such that

$$E_i \subseteq \bigcup_{\substack{m \in M \\ X \in E_p(m)}} X \cup \bigcup_{m \in M} E_s(m) \text{ for } 1 \leq i \leq n.$$

Each E_i is a set of examinations achieved at semester i.

Finishing a secondary examination is a requirement of a module to either be allowed to apply for a primary examination of the same module or to complete the module itself. Once a secondary examination is accomplished it persists. Finishing some primary examination is a requirement of a module to get completed. A module is accomplished as soon as all required examinations are satisfied.

For capturing the dependency among examinations of a given module $m \in M$, we extend our formalization by the following:

- $D(m) \subseteq E_s(m) \times E_p(m)$ relates pairs of dependent examinations.

We formalize dependencies among examinations and completion of a module, respectively, by:

$$\{(S_i)_{i=1}^n \subseteq M^n \mid \text{for each } m \in \overline{S} \text{ and each } (e, X) \in D(m) : \\ X \subseteq \textstyle\bigcup_{i=k}^n E_i \text{ implies } \bigcup_{(e', X) \in D(m)} \{e'\} \subseteq \bigcup_{j=1}^k E_j, 1 \leq k \leq n\}$$

$$\{(S_i)_{i=1}^n \subseteq M^n \mid \text{for each } m \in S_i \text{ there exists an } X \in E_p(m) : E_s(m) \cup X \subseteq \textstyle\bigcup_{j=1}^i E_j\}$$

The formalization of the following refinements is left to the future and needed to link examinations to courses. A module defines at least one eligible teaching format for each secondary examination, e.g. exercise, lecture, seminar, tutorial, project, internship, colloquia, etc. In practice, one or more examinations of at least one module is linked to a course with a corresponding teaching format. Each student has three attempts to pass primary examination of a module. If all three attempts are failed, the module cannot be completed.

4. Encoding study regulations

This section presents an ASP approach to represent study regulations and generate valid study plans. As usual, this is divided into two parts: a specific instance and a general encoding. An instance represents the elements of a specific study regulation by a set of facts, while the encoding provides the semantics associated with study regulations. Given an instance that

represents one study regulation, the answer sets of the encoding together with the instance correspond to the study plans for that study regulation.

We try to keep the notation as close as possible to the formalization of Section 2. The program uses the same symbols as before for sets and functions, but always in lowercase, to adapt to the conventions of ASP. For example, the sets $\overline{S}$, S_i and M_w are denoted by the terms `s`, `s(i)` and `m(w)`, respectively. In what follows, we often use the logic programming notation to refer to those sets and functions.

Listing 1 shows the first part of the instance `cogsys.lp` for the *Cognitive Systems* master, that specifies the sets and functions of the study plan. Sets are defined using atoms of the form `in(e,a)` that represent that the element `e` belongs to the set `a`. For example, the atom `in(bm1,b)` expresses that `bm1` $\in$ `b`. Functions are defined similarly, using atoms of the form `map(f,e,v)` that represent that the value of the function `f` applied to the element `e` is `v`. For example, `map(c,bm1,9)` expresses that `c`(`bm1`) $=$ `9`. To define the facts more compactly, we make extensive use of pooling using the operator ';'. For example, the three facts `in(bm1,b).`, `in(bm2,b).`, and `in(bm3,b).` defining the set `b` are captured by the single rule `in((bm1;bm2;bm3),b).` in Line 2.

```
1  % b, f, a, o and p                    % c
2  in((bm1;bm2;bm3),b).                  map(c,(bm1;bm2;bm3),9).
3  in((fm1;fm2;fm3),f).                  map(c,(fm1;fm2;fm3),6).
4  in((am11;am12;am21),a).               map(c,(am11;am12;am21),6).
5  in((am22;am31;am32),a).               map(c,(am22;am31;am32),6).
6  in(E,o) :- in(E,(f;a)).               map(c,(pm1;pm2;pm3),12).
7  in((pm1;pm2;pm3),p).                  map(c,im,15).
8                                        map(c,msc,30).

10 % m                                   % s
11 in(E,m) :- in(E,(b;f;a;p)).           map(s,bm1,w;s,bm2,s;s,bm3,w).
12 in((im;msc),m).                       map(s,(fm1;fm2;fm3),w).
13                                       map(s,(am11;am12;am21),e).
14                                       map(s,(am22;am31;am32),e).
15 % e                                   map(s,(pm1;pm2;pm3),e).
16 in(fm1,e).                            map(s,(im1;msc1),e).

18                                       % l and u
19                                       map(l,b,27;l,o,24;l,p,24;l,m,120).
20                                       map(u,b,27;u,o,24;u,p,24;u,m,120).
```

Listing 1: First part of the instance of the Cognitive Systems master in `cogsys.lp`.

The definitions of the constraints in (14)-(20) provide the conditions that every study plan $(S_i)_{i=1}^n \subseteq M^n$ must satisfy. These conditions usually refer to operations over sets, that we represent in ASP using prefix notation. For example, the condition of (14) refers to the intersection of the sets $\overline{S}$ and F, that is denoted in the logic program by the term `int(s,f)`. Other terms can be used to denote the union, substraction and complement of sets. The regulation constraints could in principle be very diverse, but in our investigation of various study regulations we have found that they can be captured by a few types of general constraints, that we represent in ASP by different predicates. The general encoding gives their semantics, while the specific instance of each study regulation provides facts over those predicates to represent

the corresponding constraints. Listing 2 shows the second part of our example instance, that specifies the constraints of the *Cognitive Systems* master. It uses atoms of the following form, with the associated meaning (where A and B denote sets, F denotes a function, and L and U are integers):

- `empty(A)` means that $A = \emptyset$,
- `equal(A,B)` means that $A = B$,
- `subseteq(A,B)` means that $A \subseteq B$,
- `card(A,leq,U)` means that $|A| \leq U$,
- `sum(A,F,bw,(L,U))` means that $L \leq \sum_{e \in A} F(e) \leq U$, and
- `sum(A,F,geq,L)` means that $L \leq \sum_{e \in A} F(e)$.

The general encoding includes more predicates to represent other relations among sets, like proper subset or superset, and it also allows other types of comparisons within atoms of the predicates `card/3` and `sum/4`. Using these predicates, the global constraints (14)-(16) are captured in Lines 23-25. The first constraint consists of two conditions, and this is accordingly represented by two facts. Line 24 uses pooling to refer to all the sets in {b, o, p, m}, and atoms over `map/3` to capture the values L and U of the functions `l` and `u` applied to those sets. The last rule of the block defines a new set `gc3` that consists of `im` and `msc`, and compares it via `subseteq` with `s`. Temporal constraints are represented in Lines 28 to 31. The first three use atoms of the predicate `empty/1` that refer to `m(w)`, `m(s)`, and the specific sets of modules `s(i)` of each semester `i`. The last one defines the set `tc4` that consists of `msc`, applies to it a new kind of temporal operator called `before`, and uses the resulting term in an atom of predicate `sum/4`. The term `before(tc4)` denotes the set of modules that occur in the study plan before some element of `tc4`; in this case, before the module `msc`. Using our previous mathematical notation, the set `before(tc4)` is $\{m \in M \mid m \in S_i$ and there is some $m' \in$ `tc4` $\cap S_k$ such that $i < k\}$. The general encoding includes other similar operators like `after` or `between`.

```
22 % global constraints
23 card(e,leq,2). equal(int(s,f),e).
24 sum(int(H,s),c,bw,(L,U)) :- H = (b;o;p;m), map(l,H,L), map(u,H,U).
25 in(im,gc3). in(msc,gc3). subseteq(gc3,s).

27 % temporal constraints
28 empty(int(s(I),s(J))) :- I = 1..n, J = 1..n, I < J.
29 empty(int(m(w),s(2*K))) :- K = 1..n, 1 <= 2*K, 2*K <= n.
30 empty(int(m(s),s(2*K-1))) :- K = 1..n, 1 <= 2*K-1, 2*K-1 <= n.
31 in(msc,tc4). sum(before(tc4),c,geq,90).
```

Listing 2: Second part of the instance of the Cognitive Systems master in `cogsys.lp`.

Listing 3 shows the general encoding in `encoding.lp`. It takes as input the constant `n` that gives the length of the study plan. This constant is used by the choice rule in Line 2 to generate the possible study plans, represented by the sets `s(i)` for `i` between 1 and `n`. Then, Line 5 defines the set `s` as the union of all `s(i)`'s, and Line 8 defines the sets `m(w)` and `m(s)`. After this, Lines 11-20 handle the additional sets that may occur in the constraints. The first block of rules identifies the sets that occur as arguments in the constraints. Then, the rules in Lines 16 and 17 recursively

look for the sets occurring inside the operators `int` and `before`. The encoding contains other similar rules for the other operators, but we do not show them here. Once all the new sets have been identified, additional rules provide their definition. Line 19 defines the intersection of two sets, and Line 20 defines the modules occurring before some module of another set. The complete encoding includes further rules for the other operators. The next part of the encoding, in Lines 23-33, enforces the constraints. The first ones about `empty/1`, `subseteq/2` and `equal/2`, use the predicate `in/2` to eliminate the cases that are not consistent with the constraints, while those about `card/3` and `sum/4` rely on cardinality and aggregate atoms for that task. For example, the condition $|\mathtt{A}| \leq \mathtt{U}$ for `card(A,leq,U)` is captured by the cardinality atom `{ in(E,A) } U`, and the condition $\mathtt{L} \leq \sum_{e \in \mathtt{A}} \mathtt{F}(e) \leq \mathtt{U}$ for `sum(A,F,bw,(L,U))` is captured by the aggregate atom `L #sum{ V,E : in(E,A), map(F,E,V) } U`. Finally, the last block of statements in Lines 36 and 37 displays the sets `s(i)`.

```
1  %  generate
2  { in(E,s(I)) } :- in(E,m), I=1..n.

4  % s = s(1) U ... U s(n)
5  in(E,s) :- in(E,s(I)).

7  % m(w) and m(s)
8  in(E,m(X)) :- X = (s;w), in(E,m), map(s,E,X).

10 %  additional  sets
11 set(A) :-       empty(A).
12 set(A) :- subseteq(A,B).     set(A) :-    equal(A,B).
13 set(B) :- subseteq(A,B).     set(B) :-    equal(A,B).
14 set(A) :-    card(A,R,L).    set(A) :- sum(A,M,R,L).
15 %
16 set(A) :- set( int(A,B)).    set(B) :- set(int(A,B)).
17 set(A) :- set(before(A)).
18 %
19 in( E, int(A,B)) :- set( int(A,B)), in(E,A), in(E,B).
20 in(E1,before(A)) :- set(before(A)), in(E1,s(I)), in(E2,A), in(E2,s(J)), I < J.

22 %  constraints
23 :- empty(A), in(E,A).
24 %
25 :- subseteq(A,B), in(E,A), not in(E,B).
26 %
27 :- equal(A,B), in(E,A), not in(E,B).
28 :- equal(A,B), not in(E,A), in(E,B).
29 %
30 :- card(A,leq,U), not { in(E,A) } U.
31 %
32 :- sum(A,F,bw,(L,U)), not L #sum{ V,E : in(E,A), map(F,E,V) } U.
33 :- sum(A,F,geq,    L), not L #sum{ V,E : in(E,A), map(F,E,V) }.

35 %  display
36 #show.
37 #show (M,I) : in(M,s(I)).
```

Listing 3: Meta-encoding for all study regulations in `encoding.lp`.

We can now run the ASP solver *clingo* with the instance for the *Cognitive Systems* master and the general encoding. For `n=4` we obtain, among others, an answer set that corresponds to the admissible study plan S of Example 1:

```
clingo -c n=4 cogsys.lp encoding.lp
...
Answer: 1
(bm1,1) (bm3,1) (fm1,1) (am12,1) (bm2,2) (am21,2) (pm1,2)
(im,3) (am31,3) (pm3,3) (msc,4)
```

5. ASP-driven user interface

In this section, we present our interactive prototype User Interface (UI) for creating study plans in accordance with study regulations. The UI allows users to add or remove modules within a semester and iterate through different study plan configurations based on the selected modules. As the user makes module selections, the system automatically displays the consequences of those choices and limits the available options accordingly.

This prototype was built using the system *clinguin* [4] that defines the UI by a logic program that interacts continuously with *clingo*. *Clinguin* employs a Client-Server architecture, where communication occurs via an HTTP protocol using JSON. In essence, the server is responsible for executing *clingo* and computing the information required to define the UI, namely a *ui-state*. This process occurs in two distinct steps. Firstly, the *domain-state* is computed using the domain-specific encodings (domain files) described in Section 4. This *domain-state* is made out of facts that differentiate between user-selected atoms, potential selections, and inferred atoms. In the second step, the server utilizes an encoding to generate atoms that define the layout, style, and functionality of the interface, collectively referred to as the *ui-state*.

The workflow of our study regulations UI is shown in Figure 2 and can be summarized as follows. The server is started by providing the domain files `cogsys.lp` and `encoding.lp`, as well as the UI file `ui.lp` (Listing 4) as command-line arguments:

```
clinguin server --domain-files cogsys.lp encoding.lp --ui-files ui.lp -c n=4
```

When the client is launched, it requests the *ui-state* from the server. Upon receiving the *ui-state* the client utilizes *tkinter* [5] to render the corresponding UI. Subsequent user interactions with the UI generate new requests to the server, providing information about the selected policy. The available policies are defined by the server, for instance, adding a user selection or getting the next solution. Once the server has completed the policy, it returns the updated *ui-state* to the client to be rendered.

The detailed workflow of the server is as follows: it begins by creating a *clingo* control object using the given domain files (`cogsys.lp` and `encoding.lp`) and grounding the program. It then performs three different solve calls to find the brave consequences (atoms considered as "possible"), cautious consequences (atoms considered as "required"), and the first model. These outputs are combined to represent the *domain-state*, with the brave and cautious consequences appearing in predicates `_b` and `_c`, respectively. To achieve interactivity and handle changes

[4] https://github.com/potassco/clinguin
[5] https://docs.python.org/3/library/tkinter.html

effectively, this control object utilizes multi-shot solving [2], which allows for the continuous solving of logic programs that undergo frequent changes. In *clingo*, this capability is facilitated through its API, enabling the implementation of reactive procedures involving grounding and solving. In the context of interactivity, this allows us to represent the user's selections as assumptions which are interactively modified. These assumptions are passed to the solving procedure without the need for re-grounding, and amount semantically to the addition of integrity constraints. Each assumption `A` is represented in the *domain-state* through a fact `_clinguin_assume(A)`.

Subsequently, a separate control object is employed to generate the *ui-state*. By using the *domain-state* as input, the UI encoding (`ui.lp`) produces a single stable model that encompasses the atoms of the *ui-state*. For this, predicates `element/3`, `attribute/3` and `callback/3` are used to define the layout, style and functionality of the UI, respectively. An element `X` of type `T` inside element `X'` is defined by atom `element(X,T,X')`. The attributes of an element, such as position and style, are specified by `attribute(X,K,V)`, where `K` and `V` denote the name and value of the attribute, respectively. The reactive part of the UI is defined by `callback(X,A,P)`, where performing action `A` on element `X` triggers the policy `P` on the server. These atoms are mapped into Python classes using *clorm*[6], a Python library that provides an Object Relational Mapping (ORM) interface to *clingo*.

In Figure 1, we present three UI snapshots of the study plan creation process. The window is created using the fact in Line 1 from Listing 4. For each semester `I`, Line 2 generates a container for the semester column. Line 3 creates a container for the column header, `header(I)`, which serves as the parent for the label (Line 4) and dropdown menu (Line 5). The dropdown menu items are defined in Lines 6 to 8. The body in these rules is activated if the atom `in(E,s(I))` is part of the brave consequences, indicating that the module `E` could be added to semester `I`. Furthermore, `not _c(in(E,s(I)))` ensures that the module has not been already added. This behavior can be observed in the first image of Figure 1, where the dropdown menu for available modules in the third semester does not include *bm2* because `in(bm2,s(3))` is not part of the brave consequences. Line 7 defines the `label` attribute for the element `ddmi(E,I)` generated in Line 6. Line 8 adds a callback when the element is clicked, triggering the `add_assumption` policy in the server. In the example interaction from Figure 1, when the user clicks on the dropdown menu item *im*, the server adds `in(im,s(2))` as an assumption and generates the *domain-state*. These facts are enriched with `_clinguin_assume(in(im,s(2)))` and are used to generate the *ui-state* corresponding to the second image. In that image, we can see that module *im* has been added to the third semester. Additionally, the `msc` module has been automatically included for the fourth semester because `in(msc,x(4))` is now part of all stable models. Line 9 creates the container to hold the modules assigned to each semester. Lines 10 and 11 define an auxiliary predicate to determine when a module should be displayed. The first line is applicable when it is part of all stable models or selected by the user. The second line shows the stable model only when the `_clinguin_browsing` atom is present in the *domain-state*. This atom is added by the system when the user clicks the *Next* button from the menu bar, triggering the transition from the second to the third image of Figure 1. This menu bar allows users to browse through stable models, select the current model for further editing, and clear all

[6] https://github.com/potassco/clorm

selections [7]. Line 12 generates the gray module container `m(E,I)` for each displayed module `E`. Within this container, Line 13 adds the label, while Lines 14 and 15 include a remove button exclusively for modules selected by the user.

```
1  element(window, window, root).
2  element(s(I), container, window):- I=1..n.
3  element(header(I), container, s(I)):- I=1..n.
4  element(slabel(I), label, header(I)):- I=1..n.
5  element(sdd(I), dropdown_menu, header(I)):- I=1..n.
6  element(ddmi(E,I), dropdown_menu_item, sdd(I)):- _b(in(E,s(I))), not _c(in(E,s(I))).
7  attribute(ddmi(E,I), label, E):- _b(in(E,s(I))), not _c(in(E,s(I))).
8  callback(ddmi(E,I), click, add_assumption(in(E,s(I)))):- _b(in(E,s(I))), not _c(in(E,s(I))).
9  element(sm(I), container, s(I)):- I=1..n.
10 display_m(in(E,s(I))):- _c(in(E,s(I))).
11 display_m(in(E,s(I))):- _clinguin_browsing, in(E,s(I)).
12 element(m(E,I), container, sm(I)):- display_m(in(E,s(I))).
13 element(mlabel(E,I), label, m(E,I)):- display_m(in(E,s(I))).
14 element(mbutton(E,I), button, m(E,I)):- _clinguin_assume(in(E,s(I))).
15 callback(mbutton(E,I), click, remove_assumption(in(E,s(I)))):- _clinguin_assume(in(E,s(I))).
```

Listing 4: Selected lines from the UI encoding `ui.lp`. The complete code can be found in https://github.com/potassco/clinguin/tree/dev/examples/clingo/study_regulations.

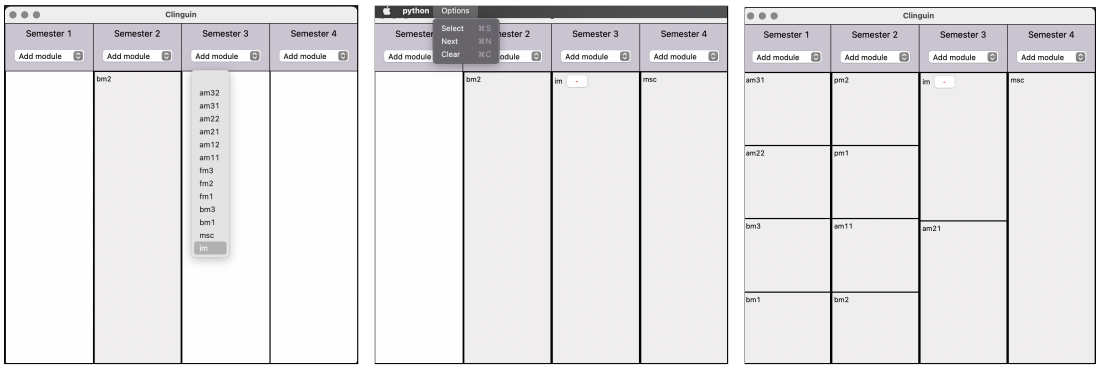

Figure 1: Snapshots of the user interaction with the prototype UI to create a study plan.

6. Related work

ASP, event calculus and process mining techniques were already used in [3] for solving study regulation problems. However, [3] presents an overview of the *AIStudyBuddy* project, and contains neither a formalization of study regulations nor any implementation details. Unlike this, [4] presents a web-based Decision Support System for a degree planning problem along

[7] The menu bar can be automatically added to the UI by including the `--include-menu-bar` parameter in the server's command-line.

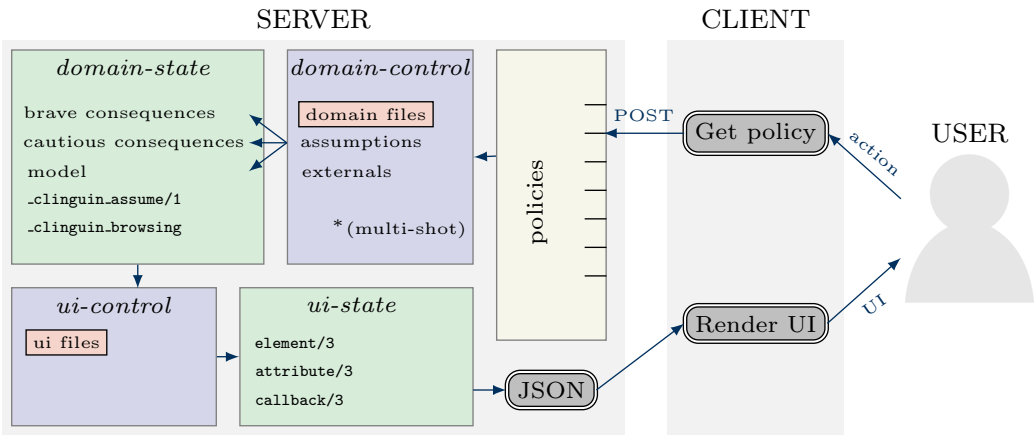

Figure 2: Workflow of the UI using *clinguin*. Input files are shown in pink.

with a mathematical formalization. Degree requirements are mentioned but not formalized. Also, [5] considers educational planning problems. This includes stress and learning effects on students in a personalized study plan generation. It aims at reducing the duration of student plans and is implemented via integer linear programming. Finally, [6] present a data-driven approach for implementing a course recommendation algorithm. A traditional collaborative filtering algorithm is extended to consider additional course path data.

7. Summary

We have introduced a conceptualization of study regulations based on set-based constraints. This formalization is both simple and general to capture a wide range of study regulations. We indicated how the basic formulation easily extends to more complex constructions. This will be further elaborated in future work. The identification of basic principles in study regulations also allowed us to obtain a very general ASP encoding. The building blocks of each study regulation are captured in terms of facts so that the actual encoding is also applicable to a large range of study programs. Finally, we have described an ASP-driven user interface for interactive elaboration of study plans. Again, the interface is designed in a generic way and broadly applicable. Moreover, this case study serves as a nice illustration of *clinguin* and how it can be used for interactive ASP applications.

Finally, study regulations offer a very rich playground for applications of knowledge representation and reasoning techniques. Study plans have a light temporal flavor and resemble finite traces in linear temporal logic [7]. The creation of a study plans amounts to a configuration task, which also brings about interaction and explainability. Finally, we have so far only been concerned with the hard constraints emerging from study regulations but there is so much commonsense knowledge involved, like defaults, preferences, deontic laws, updates, etc.

Acknowledgments. This work was partly funded by DFG grant SCHA 550/11 15 and BMBF project *CAVAS+* (16DHBKI024).

References

[1] V. Lifschitz, Answer set programming and plan generation, Artificial Intelligence 138 (2002) 39–54.

[2] R. Kaminski, J. Romero, T. Schaub, P. Wanko, How to build your own asp-based system?!, Theory and Practice of Logic Programming 23 (2023) 299–361. doi:10.1017/S1471068421000508.

[3] M. Wagner, H. Helal, R. Roepke, S. Judel, J. Doveren, S. Goerzen, P. Soudmand, G. Lakemeyer, U. Schroeder, W. van der Aalst, A combined approach of process mining and rule-based ai for study planning and monitoring in higher education, in: Proceedings of the International Conference on Process Mining (ICPM'22): Process Mining Workshops, Springer-Verlag, 2023, pp. 513–525.

[4] S. Samaranayake, A. Gunawardena, R. Meyer, An interactive decision support system for college degree planning, Athens Journal of Education 10 (2023) 101–116.

[5] J. Baldazo, R. Yasmín, R. Nigenda, Scheduling personalized study plans considering the stress factor, Interactive Learning Environments (2023) 1–20.

[6] Y. Shen, H. Li, Z. Liao, Online education course recommendation algorithm based on path factors, in: Proceedings of the Fifth International Conference on Information Systems and Computer Aided Education (ICISCAE'22), IEEE Computer Society Press, 2022, pp. 257–260.

[7] G. De Giacomo, M. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: F. Rossi (Ed.), Proceedings of the Twenty-third International Joint Conference on Artificial Intelligence (IJCAI'13), IJCAI/AAAI Press, 2013, pp. 854–860.