1 Problem

Let there be k types of values, $n = (n_1, ..., n_k)$ values, which are assumed to be distinct and m *slots* which may be optional and associated with a penalty. Each value and each slot are of exactly one type. A set of constraints may apply that either limit the set of possible solutions or impose a penalty on solutions violating them. The problem considered here consists of finding an assignment of a value to each slot so that both the type requirement and any additional hard constraint is satisfied and the cost incurred by violated soft constraints is minimized.

/*

1.1 Model

Problem

The problem is assigning to each of n assignment slots one of m components. Components have a type and the component needs to fit that of the assignment slot.

```
Problem

    Tags
    Groups
    ComponentTypes

    Assignments
      ComponentSlots
        ComponentType
        Fixed , Component
      Optionality
      Weight

    Components
      ComponentType
      Groups
      Tags

    Rules
      AssignmentSet
      Weight
      Optionality
      TopCondition
        ConditionType
        Subconditions
        Attributes
```

1.1.1 Components

The basic elements of each scheduling problem are the **components**. In a university timetabling problem these could be events, rooms and teachers. Each component has a *component type* $\mathcal{T}_C$ and a finite domain $\mathcal{D}_C$. Component types can be boolean, discrete and finite numeric types (?) or custom types. Components are assumed to be unique: $i \neq j \implies c_i \neq c_j$. The domain of a component is, by default, the set of all values of that type. A problem specifies a set of groups and a set of tags. Each component has for each tag a corresponding value $tag_{j,i} \in \mathbb{N}_0$. For each group $group_j$ and component c_i, it holds that either $group_{j,i}$ or $\neg group_{j,i}$. As an example, a staff scheduling problem could have a component type $\mathcal{T}_{nurse}$ with the

corresponding domain $\mathcal{D}_{nurse} = \{Anna, Ben, Charlie\}$. Component types are called *ordered* if they define a strict weak ordering on their domain.

1.1.2 Assignments

Assignments are sub-problems of the scheduling problem, in which a number of components of different types need to be combined: for example, an assignment of a room, a time, a class and students.

Each assignment a_i defines a sub-problem, in which . It consists of slots $as_{i,1}, ..., as_{i,k}$ that each specify a component type $type_{i,j}$, a number of components to fill that slot $\in \{1, k \in \mathbb{N}, n\} \subset \mathbb{N}$. Additionally it specifies whether that slot is optional $optional_{i,j}$ and, if so, a weight $weight_{i,j}$.

An assignment a_i consists of $k \geq 1$ component slots $as_{i,1}, ..., as_{i,j}$.

Slots can be *fixed*, meaning

In a nurse rostering problem, an assignment a_i could consist of a $type_{i,1}$ = Assignments are called *ordered* if they have at least one component slot of an ordered component type.

In the university course timetabling as presented in ITC19 [**?**], an assignment could consist of a single fixed *course* slot, a single non-optional *roomTime* slot and as many optional *student* slots as the course capacity allows.

1.1.3 Condition

A condition as an object holds information that a . A set of assignments can be evaluated over a condition. The number of assignments depend on the condition type.

A **basic condition** is one that does not

A **composite condition** is a condition the evaluation of which depends on at least one other condition. Such a chain of conditions will always end with basic

1.1.4 Rules

Additionally to the assignments, a problem has **rules**, specifying the relationship between individual assignments.

To apply a condition, they need to be associated with a set of assignments over which they are meant to hold, as well as information necessary for th

A rule connects a chain of conditions to a set of combinations of assignment over which it should be enforced.

When a rule is generated, it must be generated for each viable combination of assignments.

1.1.5 Model

A model is an assignment of components to assignment slots such that no hard rule is violated and the sum of penalties of .

1.2 Conditions

Basic	
ComponentIs(AssignmentID, SlotID, Component)	ComponentIn(AssignmentID, SlotID, Component...)
SameComponent(Assignments..., SlotID)	InGroup(assignment, slotID, groupID)
TagEqual(Assignment, SlotID, TagID, Value)	
TagEqualAbove(assignment, slotID, tagID, value)	TagEqualBelow(assignment, slotID, tagID, value)
Boolean	
Not(Condition)	
And(Condition...)	Or(Condition...)
Implies(Condition, Condition)	Iff(Condition, Condition)
Xor(Condition, Condition)	
Ordered	
Simultaneous(TimedComponent...)	
Consecutive(TimedComponent1, TimeComponent2)	
Counting Unordered	
MaxAssignment(Assignment..., Condition...)	MinAssignment(Assignment..., Condition...)
Counting Ordered	
MaxConsecutive(Assignment..., OrderedComponentSlot, Condition...)	
MinConsecutive(Assignment..., OrderedComponentSlot, Condition...)	
MaxInSequence(Assignment..., OrderedComponentSlot, Condition...)	
MinInSequence(Assignment..., OrderedComponentSlot, Condition...)	
???	
Only(Condition...)	
Exclude(Condition...)	Include(Condition...)

1.2.1 Basic Conditions

ComponentIs(Assignment, Slot, Value) **formal** $x_{a,s} = c_j$ **smt2** $(= \ x_{a,s} \ c_j)$

ComponentIn(Assignment, Slot, Value...) **formal** **smt2**

$$\bigvee_{c_j \in values} s_{a_i} = c_j$$

SameComponent(Assignment..., Slot) For each combination of two assignments $(a_i, a_j), i \neq j$, add the constraint

$$x_{a_i,slot} = component1 \wedge x_{a_j,slot} = component2 \implies component1 \neq component2$$

formal
smt2

InGroup(Assignment, Slot, Group) This condition can be encoded by a constraint that limits the domain of the slot variable to the component variables that fulfil the group condition. Let *comps* be the component variables corresponding to exactly those components

$$\bigvee_{c \in allComponents}$$

TagEqual(assignment a, slot s, tag, value) Like InGroup, this condition can be handled by passing a constraint that limits the domain of assignment variable. Let *comps* be the component variables corresponding to exactly those components c_i where $tag_{c_i} = value$.

$$\bigvee_{c \in comps} x_{a,s} = c$$

TagEqualAbove(assignment, slotID, tagID, value) Let *comps* be the component variables corresponding to exactly those components c_i where $tag_{c_i} \geq value$.

$$\bigvee_{c \in comps} x_{a,s} = c$$

TagEqualBelow(assignment, slotID, tagID, value) Let *comps* be the component variables corresponding to exactly those components c_i where $tag_{c_i} \leq value$.

$$\bigvee_{c \in comps} x_{a,s} = c$$

1.2.2 Composite Conditions

Condition used inside .. signifies the evaluation of the .

Not(Condition)

$$\neg Condition$$

And(Condition...)

$$\bigwedge_{c \in Conditions} c$$

formal
smt2

Or(Condition...)

$$\bigvee_{c \in Conditions} c$$

Implies(Condition1, Condition2)

$$Condition1 \implies Condition2$$

formal
smt2

Xor(Condition1, Condition2)

$$(\neg Condition1 \wedge Condition2) \vee (Condition1 \wedge \neg Condition2)$$

formal
smt2

Iff(Condition1, Condition2)

$$(Condition1 \implies Condition2) \wedge (Condition2 \implies Condition1)$$

formal
smt2

MaxAssignment(Assignment..., Condition...)

MinAssignment(Assignment..., Condition...) Add a bitvector bv_i of the length $\#Assignments$ For each assignment a_k, add a rule of the form

$$\bigwedge_{c \in \text{Condition}} c(a_i) \implies extract(a_k)$$

[**?**] describe a declarative , the number of set bits can be checked by

$$bv = bv\&(bv - 1)$$

If this constraint is violated, the penalty should be multiplied by the Hamming-Weight of the resulting vector.

MaxConsecutive(Assignment..., OrderedComponentSlot, Condition...)

MinConsecutive(Assignment..., OrderedComponentSlot, Condition...) All given assignments must have a slot of the *OrderedComponentSlot* type. This type needs to be an ordered type. The ith bit of the generated bitvector corresponds to the ith assignment with respect to the given order. Assignments that are equivalent with respect to the given order are *collapsed* into a bit that is set if the conditions hold for at least one of the equivalent assignments. Let $Conditions(A)$ denote the conjunction of the given conditions resolved for the assignment A. For every given assignment, add:

$$Conditions(A) \implies bv.extract(i)$$

As described in [**?**], the following equation will hold iff there exists no block of a length more than max:

$$0 = bv\&(bv >> 1)\&(bv >> 2)\&...\&(bv >> max)$$

The same paper describes a three-step algorithm to check minimal block length by first

Simultaneous(Assignment..., OrderedComponentSlot) Each of the given assignments needs to have a slot of the *OrderedComponentSlot* type and that type needs to be ordered.

Consecutive(Assignment..., OrderedComponentSlot)

Only(Assignment, Condition...)

2 Encoding

Let t_i denote the k component types, $c_{,}$ the components of each type. Let $C = \{\}$ denote the set of all components. For all components $c_{k_1}, ..., c_{k_n}$ of each of the k sorts it must be specified that the component variables are distinct. Every component slot in an assignment corresponds to one decision variable. Let $x_{a,s}$ denote these decision variables. Let $\mathcal{M} : x_{a,s} \rightarrow c_i$ denote the model assignment from each assignment slot variable to exactly one component.

2.1 Conditions

ComponentIs(AssignmentID, SlotID, Component) *component* is assigned to the slot $SlotID$ of assignment $AssignmentID$

ComponentIn(AssignmentID, SlotID, Component...) one *component* in the set of $k \geq 1$ components given is assigned to the slot $SlotID$ of assignment $AssignmentID$. The given components need to be of the type required by that slot.

SameComponent(Assignments..., SlotID) The components $c_1, ..., c_k$ of slot $slotID$ of the k assignments given are identical. The given assignments need to have a slot $slotID$.

InGroup(assignment, slotID, groupID) The component c assigned to slot $slotID$ of assignment $AssignmentID$ is in group $groupID$.

TagEqual(Assignment, SlotID, TagID, Value) The tag $TagID$ of the component c assigned to slot $SlotID$ of assignment $AssignmentID$ is equal to *value*.

TagEqualAbove(assignment, slotID, tagID, value) The tag $TagID$ of the component c assigned to slot $SlotID$ of assignment $AssignmentID$ is equal to or greater than *value*.

TagEqualBelow(assignment, slotID, tagID, value) The tag $TagID$ of the component c assigned to slot $SlotID$ of assignment $AssignmentID$ is equal to or smaller than *value*.

Not(Condition)

And(Condition...)

Or(Condition...)

Implies(Condition, Condition)

Xor(Condition, Condition)

Iff(Condition, Condition)

MaxAssignment(Assignment..., Condition...)

MinAssignment(Assignment..., Condition...)

MaxConsecutive(Assignment..., ?Ordering?, Condition...)

MinConsecutive(Assignment..., ?Ordering?, Condition...)

MaxInSequence(Assignment..., ?Ordering?, Condition...)

MinInSequence(Assignment..., ?Ordering?, Condition...)

Simultaneous(TimedComponent...)

Consecutive(TimedComponent1, TimeComponent2)

Only(Condition...)

Exclude(Condition...)

Include(Condition...)

from decision procedures:
6.1.1 Syntax The subset of bit-vector arithmetic that we consider is defined by the following grammar:
atom : term rel term — Boolean-Identifier — term[constant]
rel : ¡ — =
—
We only need the subset of this that is actually used. Really, all that is needed is a type,
SyntacticObject class?
constraints across multiple assignments
forEach(condition) forEach(condition , assignments)
conditions can include placeholders: a , b
inGroup()
or(any("A", not(inGroup(haid1))), any("B", not(inGroup(haid2))) , any(""))
any(A, inGroup1) & any(B, inGroup2)
inGroup(A, Group1) & inGroup(B, Group2) =¿
–
any(name, condition)
–
for(a, b : asgnComb)
add(a.ingroup() & b.ingroup())
Conditions are used to 1. filter assignments, 2. be filled out to be added as rules
Rule: Require(Condition)
These are written out as a language and need to be parsed
By component here I mean the variables in assignments - that should probably be named better.
Basic Condition: Component = Component — Component != Component — InGroup(Component, Group) — InSet(Component, Set) — Not(Condition) — Condition AND Condition — Condition OR Condition — Condition IMPLIES Condition — Condition IFF Condition — Condition XOR Condition
Numeric Condition = Numeric ¡ Numeric — Numeric ¡= Numeric — Numeric ¿ Numeric — Numeric ¿= Numeric — Numeric = Numeric — Numeric != Numeric
Numeric := Int — Tag(Component, Tag) — NumAssigned(Condition) — MaxConsecutive(Condition) — MinConsecutive(Condition) — MaxBreak(Condition) — MinBreak(Condition)
Condition, Component, Group
need: determine whether a thing is a condition or component
2 arguments or a list of arguments?
Operator precedence: NOT AND OR IFF, IMPLIES, XOR ————————
Precedence climbing:
primary := '(' expression ')' — variable
———————— Instead of numeric: AtLeast(Int, Bitvector) — AtMost(Int, Bitvector) — Exactly(Int, Bitvector)
A bitvector is created with a from an ordered set of assignments and set of conditions.
CreateBitvector(set, set)
from assignments or component slots?
—
When a translator receives a rule, it will need to recursively(?) encode each part.
This means that for each relevant combination of components, a sentence needs to be added to the solver.
- build all constraints sequentially or at the same time? but how? -
Example: A1.Shift.StartTime == A2.Shift.StartTime IMPLIES A1.Nurse != A2.Nurse

(A1.Shift.StartTime == A2.Shift.StartTime) IMPLIES (A1.Nurse != A2.Nurse)

A1.Shift.StartTime == A2.Shift.StartTime IMPLIES A1.Nurse != A2.Nurse

Implies condition: only need to add rules for which the premise is true This rule only uses basic logical operations. Instantiate it with all combinations of A1 and A2

Nurses: Patrick Andrea Stefaan Sara Nguyen

InGroup() or

2.2 Rules

Rules define constraints between assignments.

SameTime(A1, A2) =¿ A1.Component("Nurse") != A2.Component("Nurse")

"SameTime(A1, A2)" "=¿" "A1.Component("Nurse")" "!=" "A2.Component("Nurse")"

—SameTime(A1, A2)— —=¿— —A1.Component("Nurse")— —!=— —A2.Component("Nurse")— bool op obj op obj

bool !=(obj, obj) bool =(obj, obj)

bool =¿(bool, bool)

bool SameTime(Ass, Ass)

objct Component()

A rule is a function taking either 2 or n assignments, returning bool

i want to save parts recursively

every function part of a function needs to depend on assignments

LEFT OPERATOR RIGHT

Left and right can be functions or rules

the rule object provides a function for

rule := expression op expression expression := function — rule

Formula: wrapper class for some proper formula or: function to invoke for every translatable unit

information that needs to be kept: assignment between external and internal ID

elements might be added to groups after group-wide constraints are created

- When a create constraint is called, create a constraint object

- Go through all constraint objects and call generate on them all - Create component objects and constraint objects - generate constraints

-

1. create a problem object with a name/id
2. create a class for each type of component (can use using)
3. create a class for each type of assignment (must derive, is abstract)
4. create groups and tags
5. create components, with appropriate groups and tags
6.

2.2.1 Components

Classes

Times

Rooms

Students

2.2.2 Assignments

2.2.3 Rules

Constraints that are evaluated over pairs incur a penalty for every violated pair.

SameStart(Class, Class) Classes must start on the same time slot Defined as:

```
SameStart(class1, class2) :=
```

SameTime(Class, Class)

DifferentTime(Class, Class)

SameDays(Class, Class)

DifferentDays(Class, Class)

SameWeeks(Class, Class)

DifferentWeeks(Class, Class)

SameRoom(Class, Class)

DifferentRooms(Class, Class)

Overlap(Class, Class)

NotOverlap(Class, Class)

SameAttendees(Class, Class)

Precedence(Class, Class)

WorkDay(S, Class, Class)

MinGap(G, Class, Class)

MaxDays(Days ...)

MaxDayLoad(Slots ...)

MaxBreaks(Breaks ...)

MaxBlock(Blocks ...)

2.3 Modelling the ITC2021 Problem

Phased or unphased!

"In this competition we only consider time-constrained double round-robin tournaments with even numbers of teams" (each team plays exactly one game per timeslot)

time-constrained/compact: uses the minimal number of time slots needed otherwise: time-relaxed
timeslots: single timeslot with parallel assignments for multiple games tasks: team vs. team game
2RR: - if (a, b) played in the first half, then (b, a) must play in the second round
create a group for each team create a task for each team x team game add them to the corresponding

2.3.1 Phased

In **phased** competitions, the competition is split into intervals in which all combinations of teams compete once.

conjunction of group requirements
one per phase problem: determining phase length

2.3.2 Capacity Constraints

CA1: teams, max, mode: H for home, A for away, slots, type: HARD or SOFT; teams from cannot play more than max games of mode in timeslots slots (in benchmarks teams only ever contains one team - other benchmarks can be expanded into that format)

CA2: teams1, min, max, mode1: H for home, A for away, HA for both, mode2: GLOBAL ???, teams2: played against teams, slots, type

CA3: teams1, max, mode1, teams2, intp, mode2, type;

2.3.3 Notes reading

- MetaData can be ignored - here: only single league with all teams (encoding should accomodate multiple though) - here: no grouping teams and time slots into time groups (could be achieved easily with groups?) - RobinX is general format? not from ITC? -

1. create timeslots (only ID and name) - calculate max possible number of games per timeslot and create that many instances for each (problem: many symmetric assignments?)
2. create a group for each team
3. (league: id, name)
4. (need group for each league for rules?)
5. (team: id, league, name)
6. create group for each team
7. read structure: for each league:
   (a) create tasks for each team x team combination in a league, depending on round-robin number, add team groups
   (b) set compactness rule (???) if applicable
   (c) set game mode (phased/unphased) rule (???) if applicable
8. The competition instances only use 9 of the possible constraints (these should be implemented here)

2.3.4 Components

Slots: group: each timeslot has a group with its id
Games: name: IDteam1_IDteam2 groups:

1. group: "H" or "A" for home or away for each time
2. group "HA" for both teams
3. "l" league

4. groups with the id of each team

In total 8 groups per game.

Assignment: One fixed timeslot, $\lfloor \#games/2 \rfloor$ optional game slots

Implicit Rules: // a team can only play one game simultaneously for(assignment) itc21.addRule(Unique(gameType));

rules used:

- AssignOnce(component)
- Unique(assignment, slotType)
-

Rules:

CA1: "teams t each play at most max games of mode m in slots s", can be hard or soft "each team in temas triggers a deviation equal to the number of mode games in slots more than max"

components: groups (team), timeslots (), for each team (group): NumAssigned(InComponent(Timeslots ts...) & InGroup(Mode, mode)) ¡= MAX

CA2: "each team in teams1 does not play more than max games of mode m against teams2 during timeslots slots" for each team in

NumAssigned(InTimeslots(slots) & Groups(teams2) & Group(mode)) ¡= MAX

CA3: "Each team in teams1 plays at most max games of mode m against teams2 " For each group in teams1: ???? INTP ???? NumAssigned()

C4: "teams1 play at most max games of mode m against teams2 during time slots slots"

NumAssigned(OR(Group(teams1)) & OR(Group(teams2)) & Group(mode) & In(slots)) ¡= MAX

GA1: (min max meetings slots type) "fixed and forbidden time slot assignments" "at least min and at most max games from meetings = $\{(i_1, j_1), (i_2, j_2), ...\}$ take place in time slots slots"

(if min = max add single equality constraint)

NumAssigned() ¿= min NumAssigned() ¡= max

(Break constraints)

"If a team plays a game with the same home-away status as its previous game, it is called a break"

BR1: (teams intp mode2 slots type) "Each team in teams has at most intp home/away breaks (H,A,HA) during time slots slots"

"BR1 constraints with multiple teams can be broken up into one constrant per teams, therefore ITC2021 only has single team constraints"

NumAssigned(InComponent(Timeslot, slots) & 'prevGame.mode = curGame.mode') ¡= INTP

BR2

—

Assignment:

One assignment per game

Assignment: FIXED: game REQUIRED: time (not exclusive)

same team =¿ not equal times

for each combination of game assignments: implies(sameComponent(slotType), or(notIn(A, HA1), notIn(B,HA1))

not(ingroup(A, HA1) and ingroup(B, HA1) and sameComponent(slot))

3 Modelling The Second International Nurse Rostering Competition (2014)

In short, the scheduling problem presented in [**?**] consists of assigning nurses to shifts in multiple consecutive planning periods. The competition was held in 2015, on the website the benchmarks and a solution validator are still available.

This is an example for a scenario (global part of the problem description) from a test instance:

```
SCENARIO = n005w4

WEEKS = 4

SKILLS = 2
HeadNurse
Nurse

SHIFT_TYPES = 3
Early (2,5)
Late (2,3)
Night (4,5)

FORBIDDEN_SHIFT_TYPES_SUCCESSIONS
Early 0
Late 1 Early
Night 2 Early Late

CONTRACTS = 2
FullTime (15,22) (3,5) (2,3) 2 1
PartTime (7,11) (3,5) (3,5) 2 1

NURSES = 5
Patrick FullTime 2 HeadNurse Nurse
Andrea FullTime 2 HeadNurse Nurse
Stefaan PartTime 2 HeadNurse Nurse
Sara PartTime 1 Nurse
Nguyen FullTime 1 Nurse
```

Only one time unit is necessary: a counter of days since the start of the planning horizon. Create groups: HeadNurse, Nurse, Early, Late, Night, Saturday1, Sunday1, Saturday2, Sunday2, Saturday3, Sunday3, Saturday4, Sunday4 Nurse can be used as an alias for Agent, Shift for TimedTask.

```
using Nurse = Agent<std::string, std::string, std::string>;
using Shift = TimedTask<std::string, std::string, std::string, int>;
```

Create each nurse, add the appropriate groups for roles. When reading from a file, this can be done in a loop; here only examplary for a single nurse:

```
Nurse patrick("Patrick");
patrick.addGroup("HeadNurse");
patrick.addGroup("Nurse");
```

There is only a single type of assignment for this problem. Since the problem is to assign nurses *to* shifts, each shift will be fixed in one assignment.

```
auto &asgn = inrc2.newAssignment(name);
auto &time = inrc2.newComponent(name, timeType);
time.addGroup(skill);
asgn.setFixed(timeSlot, time);
```

Create a TimedTask for every role required for each shift. Add the appropriate groups for role requirements, shift type and day of the week. Add these requirements to all Shifts s:

```
s.Require(shift.inGroup(HeadNurse) IMPLIES nurse.inGroup(HeadNurse));
s.Require(shift.inGroup(Nurse) IMPLIES nurse.inGroup(Nurse));
```

Ideas to model the constraints as described in [**?**] as rules:

H1. Single assignment per day Implies(SameGroup(timeslot, day), not(SameComponent(nurse))) Implies(SameComponent(dayslot), not(SameComponent(nurse)))

internally: timeslots are fixed, so just add illegal combinations directly

H2. Under-staffing This is already enforced by making required tasks non-optional.

H3. Illegal shift type successions For every illegal combination (prec, succ):

InGroup(A1.Shift, prec) AND InGroup(A2.Shift, succ) AND ImmediateSuccessor(A1.Shift, A2.Shift) IMPLIES A1.Nurse != A2.Nurse

Could list out not(SameComponent())

SameComponent()

H4. Missing required skill Enforced by making required role tasks non-optional.

S1. Insufficient staffing for optimal coverage Enforced by adding the appropriate weight (30) as penalty to optional shifts.

S2. Consecutive assignments Define

SameType(A1, A2) := (A1.shift.inGroup(Early) AND A2.shift.inGroup(Early)) OR (A1.shift.inGroup(Late) AND A2.shift.inGroup(Late)) OR (A1.shift.inGroup(Night) AND A2.shift.inGroup(Night))

MinConsecutive(A1.nurse == A2.nurse AND SameType(A1, A2)) ¿= MIN

S3. Consecutive days off

S4. Preferences

S5. Complete week-end

S6. Total assignments (only evaluated at the end of the planning period)

S7. Total working week-ends (only evaluated at the end of the planning period)

4 Program

5 API

5.1 Problem

```
class Problem {

    public:

        void print(std::ostream &) const;

        //Component<ID> &newComponent(const ID &id, const ComponentType<ID> &type);

        Component<ID> &newComponent(const ID &id, const ID &type);
```

```
        Assignment<ID> &newAssignment(const ID &id);


        const std::set<ID> &getAllGroups() const;

        const std::set<ID> &getAllTags() const;

        const std::vector<Component<ID>> &getComponents(const ID &componentType) const;

        const std::map<ID, Assignment <ID>> &getAssignments() const;

        const std::vector<Rule<ID>> &getRules() const;

        //void addRule(const std::string &);

        //void addRule(const Rule<ID> &&);

        //Rule<ID> &addRule(std::unique_ptr<Condition<ID>> c);
        void addRule(const ID& id, std::unique_ptr<Condition<ID>> c, const bool &hard, const int &weigh

        std::vector<ID> getComponentTypes() const;
        const ID addComponentType(const ID &);

        void addGroup(const ID&);
```

5.2 Assignment

```
void setFixed(const ID &name, const Component<ID>&);
        void setFixed(const ID &name, std::vector<Component<ID>>&);

        void setVariable(const ID &name, ID componentType, bool optional);

        const std::map<ID, ComponentSlot<ID>> & getComponentSlots() const;

        void setOptional(bool optional);

        void setWeight(int weight);
```

5.3 Component

```
Component(const ID &id) : id{id} {}
        //virtual const std::string componentType() const = 0;

        const ID getID() const;

        const std::map<ID, int> &getTags() const;
        const std::set<ID> &getGroups() const;

        void addGroup(const ID&);
        void removeGroup(const ID&);

        void setTag(const ID &, const int);
```

5.4 •

6 Toy Example