1 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.

1.1 List of Conditions

```
\hline
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)

Not(Condition)
And(Condition...)
Or(Condition...)
Implies(Condition, Condition)
Xor(Condition, Condition)
Iff(Condition, Condition)

MaxAssignment(Assignment..., Condition...)
MinAssignment(Assignment..., Condition...)
MaxConsecutive(Assignment..., OrderedComponentSlot, Condition...)
MinConsecutive(Assignment..., OrderedComponentSlot, Condition...)
MaxInSequence(Assignment..., OrderedComponentSlot, Condition...)
MinInSequence(Assignment..., OrderedComponentSlot, Condition...)

Simultaneous(TimedComponent...)
Consecutive(TimedComponent1, TimeComponent2)

Only(Condition...)
Exclude(Condition...)
Include(Condition...)
```

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...)

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

1.2 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 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.

/*

2.1 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
```

2.1.1 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.

2.1.2 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.

2.1.3 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

2.1.4 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.

2.1.5 Model

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

2.1.6 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

$$x_{a_i,slot} = \mathit{component}1 \wedge x_{a_j,slot} = \mathit{component}2 \implies \mathit{component}1 \neq \mathit{component}2$$

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 $tag_{c_i} \leq value$.

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

2.1.7 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...)

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.

```
class Shift : public Assignment {

        Expression generate() override;
        Agent nurse;
        const TimedTask shift;
        std::vector<Rule> requirements;

};
```

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 A1.Simultaneous(A2) IMPLIES A1.Nurse != A2.Nurse, or

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

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 Immediate-Successor(A1.Shift, A2.Shift) IMPLIES A1.Nurse != A2.Nurse

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 MaxConsecutive(A1.nurse == A2.nurse AND Same-Type(A1, A2)) ¡= MAX

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)