
UCSP: Implementation

Dmitry K.

May 24, 2016

Abstract

This article proposes a system for generating possible *University Classes Schedules*. It uses multi-agent negotiation to find satisfactory solutions to the problem, while trying to consider *personal preferences* of the represented people and institutions.

Contents

1	Implementation	2
1.1	University Classes	2
1.2	Negotiating Agents	4
1.2.1	Common Goal	5
1.2.2	Messaging	5
1.3	Coherence	5
1.3.1	Information and Relations	5
1.3.2	Contexts	8
1.3.3	Capabilities	10
1.3.4	Beliefs	12
1.3.5	Obligations	15
1.3.6	Preferences	15
1.3.7	External	17
1.3.8	Decision	17
1.4	Agent	17

1 Implementation

1.1 University Classes

A class is an event, that brings together a *group of students*, and a *professor* in certain *classroom* in order to learn/teach the specified *discipline*. It happens periodically, usually weekly, at the established *day of week* and *time*.

A *discipline* should describe an atomic (not dividable) educational activity. For example, if the students are required to take a normal class and also do some specific laboratory practice, then two disciplines should be created, one of them describing the required lab equipment.

```
data Discipline = Discipline { disciplineId           :: String
                             , disciplineMinutesPerWeek :: Int
                             , disciplineRequirements   :: Set Requirement
                             }
deriving (Typeable, Show, Eq, Ord)
newtype Requirement = Requirement String deriving (Show, Eq, Ord)
```

For inner usage, the classes are divided into

- *abstract* — without day and time;
- *concrete* — with full time information.

```
class (Ord c, Show c, Typeable c) =>
  AbstractClass c where classDiscipline :: c -> Discipline
                        classGroup      :: c -> GroupRef
                        classProfessor  :: c -> ProfessorRef
                        classRoom       :: c -> ClassroomRef
                        classNumber    :: c -> Word

class (AbstractClass c, DiscreteTime time) =>
  ConcreteClass c time | c -> time
  where classDay      :: c -> Day
        classBegins  :: c -> time
        classEnds    :: c -> time

data Class      = ∀c time. ConcreteClass c time => Class c
data SomeClass = ∀c      . AbstractClass c      => SomeClass c
```

The “System.Time.Day” is redefined, dropping the “Sunday”.

```
data Day = Monday | Tuesday | Wednesday
        | Thursday | Friday | Saturday
deriving (Eq, Ord, Enum, Bounded, Ix, Read, Show)
```

The classes are negotiated by the interested parties: 1) students / groups, 2) professors, 3) classrooms. Each negotiation participant has a *timetable*, holding a schedule for one week, that repeats throughout the academic period. The

--

timetable is actually a table: the columns represent days of week; the rows — discrete time intervals. Actual timetable structure may vary, as can be seen in figure 1.

	Mon	Tue	Wed	Thu	Fri	Sat
08:30 – 09:00						
09:00 – 09:30						
09:30 – 10:00						
10:00 – 10:30						
10:30 – 11:00						
11:00 – 11:30						
11:30 – 12:00						
⋮						

(a) Timetable without recesses.

	Mon	Tue	Wed	Thu	Fri	Sat
08:30 – 09:10						
09:15 – 09:55						
10:05 – 10:45						
10:50 – 11:30						
11:40 – 12:20						
12:25 – 13:05						
13:15 – 13:55						
⋮						

(b) Timetable with recesses.

Figure 1: Possible *timetable* structures.

```

class (Ord t, Bounded t, Show t, Typeable t) ⇒ DiscreteTime t where
  toMinutes    :: t → Int
  fromMinutes :: Int → t

data SomeTime = ∀t.(DiscreteTime t) ⇒ SomeTime t
someTimeMinutes (SomeTime t) = toMinutes t

instance Eq SomeTime where (≡)    = (≡)    ‘on’ someTimeMinutes
instance Ord SomeTime where compare = compare ‘on’ someTimeMinutes

class (DiscreteTime time) ⇒ Timetable tt e time | tt → time
                                     , tt → e
                                     , e → time

where listEvents :: tt → [e]
      eventsOn  :: tt → Day → [e]
      eventsAt  :: tt → time → [(Day, e)]
      eventAt   :: tt → Day → time → Maybe e

```

One should distinguish the resulting timetables, shown in figure 1 and the timetable, held an agent during the negotiation. The first one is immutable and is the result of agent's participation in the negotiation. The set of such timetables, produced by every the participant, is the **university schedule** for given academic period.

During the negotiation, an agent's inner timetable gets changed on the fly, in order to record agreements made. This means that we are dealing with *side effects*, that need to be explicitly denoted in Haskell. The following definition leaves it free to choose the monad abstraction for those effects.

```
class (DiscreteTime time, Monad m) =>
  TimetableM tt m e time | tt -> time
  , tt -> e
  , e -> time
  where putEvent  :: tt -> e -> m tt
        delEvent  :: tt -> e -> m tt
        ttSnapshot :: (Timetable ts x time) => tt -> m ts
```

1.2 Negotiating Agents

As it was mentioned before, the schedule is formed in a negotiation between *professors*, *groups* and *classrooms*. To distinguish those three types of participants, agent's role is introduced. The role: 1) identifies the kind of person/entity, represented by the agent; 2) defines agent's reaction on the messages received; 3) defines agent's goal.

A *representing agent* is a computational entity, that represents a *real person or object* in it's virtual environment. In current case, it represents one's interests in a *negotiation*. Such an agent must

- (1) pursue the *common goal* — it must consider the common benefits, while being egoistic enough to achieve it's own goal;
- (2) respond to the messages received in correspondence with (1);
- (3) initiate conversations (send messages, that are not responses), driven by (1);
- (4) become more susceptible (less egoistic) with passage of time.

```
data NegotiationRole = GroupRole
  | FullTimeProfRole
  | PartTimeProfRole
  | ClassroomRole
  deriving (Show, Typeable)
```

1.2.1 Common Goal

Agent's own *goal* represents its egoistical interests. They may (and will) contradict another agent's interests, thus creating *incoherence*. The general rule in this case is to strive for solutions, benefiting the whole schedule. Because the schedule doesn't yet exist as a whole during the negotiation, an agent should consider instead the benefits, obtained by itself and the rest of the agents.

The *common goal* is incorporated in the *contexts* mechanism, and is discussed in Section 1.3.7.

1.2.2 Messaging

Is this section really needed?

1.3 Coherence

The coherence mechanism is based on [?]. It uses the *contexts* as means of separating (and further prioritizing) different *cognitive aspects*. The contexts used are based on *BDI* agent architecture.

The *combined coherence* is used as a measure of goal achievement. It's combined of coherence values, calculated by agent's contexts.

1.3.1 Information and Relations

The coherence is calculated over an *information graph*, that represents some aspect of agent's knowledge. The nodes of the graph are some *pieces of information* and the edges represent some *relations* between these pieces.

```
newtype IGraph = IGraph (Set Information)
graphNodes :: IGraph → [Information]
graphNodes (IGraph inf) = Set.toList inf
graphJoin :: IGraph → [Information] → IGraph
graphJoin (IGraph inf) new = IGraph (inf ∪ Set.fromList new)
fromNodes :: [Information] → IGraph
fromNodes = IGraph ∘ Set.fromList
relationOn :: IRelation a → IGraph → RelValue a
relationOn rel (IGraph inf) = ⊥ -- TODO
```

The proposed system makes use of the following information:

1. **Personal knowledge**, known only by one actor.
 - (a) **Capabilites**: information about what an agent can do, what kind of arrangements it can make.
 - (b) **Obligations**: information about *strong restrictions*, imposed over the agent.
 - (c) **Preferences**: information about *weak restrictions*.

2. **Shared knowledge**, obtained in the negotiation.

- (a) **Others' capabilities** — information about the counterpart agents, that are known to be (un-) capable of doing something.
- (b) **Classes proposals**:
 - i. **Abstract** — has no specific time assigned.
Concrete — has a specific time defined.
 - ii. **Complete** — references all three representing agents: a *group*, a *professor* and a *classroom*.
Partial — references less then three representing agents.
- (c) **Classes decisions**:
 - i. **Class acceptance** — a mark for *accepted classes proposals*. Only *complete* proposals can be accepted; all the three mentioned agents must accept it, or none.
 - ii. **Class rejection** — a mark for *ignored classes proposals*, a result of *yield* decision, discussed in Section ??.

```

data InformationScope = Personal | Shared
  -- “Ord” instance is mainly needed to create “Set”s.
class (Typeable i, Eq i, Ord i) => InformationPiece i
  where type IScope i :: InformationScope
class (InformationPiece i, Personal~IScope i) => PersonalInformation i
class (InformationPiece i, Shared~IScope i) => SharedInformation i
  where sharedBetween :: i -> Set AgentRef
  -- -----

instance Eq SomeClass where
  (SomeClass a) == (SomeClass b) = cast a == Just b
  -- TODO
instance Ord SomeClass
instance InformationPiece SomeClass where
  type IScope SomeClass = Shared
instance SharedInformation SomeClass
  -- -----

instance Eq Class
instance Ord Class
instance InformationPiece Class where type IScope Class = Shared
instance SharedInformation Class
  -- -----

data Information = forall i. InformationPiece i => Information i
collectInf :: (Typeable a) => Information -> Maybe a
collectInf (Information i) = cast i

```

```

instance Eq Information where
  (Information  $i_1$ )  $\equiv$  (Information  $i_2$ ) =
    case cast  $i_1$  of Just  $x$        $\rightarrow x \equiv i_2$ 
    -                                $\rightarrow$  False

instance Ord Information where
  (Information  $i_1$ )  $\leq$  (Information  $i_2$ ) =  $\perp$ 
  -- -----

newtype Needs = Needs (Set Discipline)
  deriving (Eq, Ord, Show, Typeable)

newtype CanTeach = CanTeach (Set Discipline)
  deriving (Eq, Ord, Show, Typeable)

instance InformationPiece Needs
instance InformationPiece CanTeach

```

The *binary relations* connect some information pieces, assigning to the edge some value. The *whole graph relations*, on the other side, are applied to the graph as a whole and produce a single value.

The relations used, as well as the information in the graph, depend on the *context*.

```

data RelValBetween a = RelValBetween {
  relBetween    :: (Information, Information)
  ,
  relValBetween :: a
}

type RelValsBetween a = Map (IRelation a) [RelValBetween a]

newtype RelValWhole a = RelValWhole a
unwrapRelValWhole (RelValWhole a) = a

type RelValsWhole a = Map (IRelation a) (RelValWhole a)
-- -----

class InformationRelation r where
  relationName :: r a  $\rightarrow$  String
  coerceRelation :: (Coercible a b)  $\Rightarrow$  r a  $\rightarrow$  r b

class InformationRelation r  $\Rightarrow$ 
  BinaryRelation r where
    binRelValue :: (Num a)  $\Rightarrow$  r a  $\rightarrow$  Information  $\rightarrow$  Information  $\rightarrow$  Maybe a

class InformationRelation r  $\Rightarrow$ 
  WholeRelation r where
    wholeRelValue :: r a  $\rightarrow$  IGraph  $\rightarrow$  a
  -- -----

data IRelation a =  $\forall r$ . BinaryRelation r  $\Rightarrow$  RelBin (r a)
  |  $\forall r$ . WholeRelation r  $\Rightarrow$  RelWhole (r a)
relName (RelBin a) = relationName a

```



```

relName (RelWhole a) = relationName a
instance Eq (IRelation a) where (≡) = (≡) 'on' relName
instance Ord (IRelation a) where compare = compare 'on' relName
coerceIRelation :: (Coercible a b) ⇒ IRelation a → IRelation b
coerceIRelation (RelBin r)    = RelBin (coerceRelation r)
coerceIRelation (RelWhole r) = RelWhole (coerceRelation r)

-- -----
type RelValue a = Either [RelValBetween a] (RelValWhole a)

```

1.3.2 Contexts

In order to use contexts for information *coherence assessment*, the concepts of *context-specific information graph* and *assessed information* are introduced. The context-specific graph holds the information, already known/accepted by the agent, and is relevant for the context in question. The assessed one is *assumed* during the evaluation process.

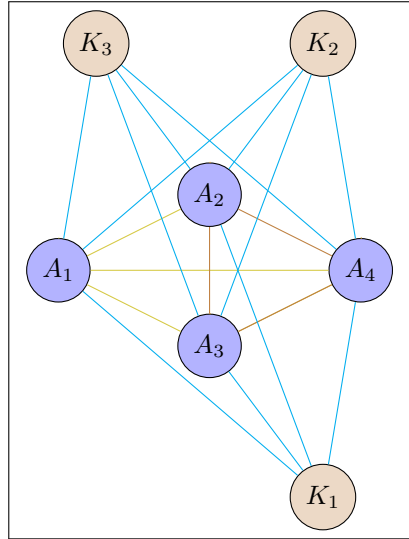


Figure 2: Binary relations within an information graph. One can distinguish the relations between the assessed information pieces and the relations between assessed and the known ones.

To assess some information, it's propagated through the contexts, in the *specified order*, that stands for contexts priority. Each context should have a *coherence threshold* specified; after the assessed information's coherence has been estimated, it's compared against the threshold and either **Success** or **Failure** is returned, along with the evaluated coherence value. The information, that

has successfully passed a context, is propagated further; otherwise the failure is returned.

```

class Context (c :: * → *) a where
  contextName      :: c a → String
  contextInformation :: c a → IO IGraph
  contextRelations  :: c a → IO [IRelation a]
  contextThreshold  :: c a → IO a

  combineBinRels    :: c a → RelValsBetween a → Maybe (CBin a)
  combineWholeRels  :: c a → RelValsWhole a → Maybe (CWhole a)
  combineRels       :: c a → CBin a → CWhole a → a

newtype CBin a = CBin a
newtype CWhole a = CWhole a
data AssessmentDetails a -- TODO
data SomeContext a = ∀c. Context c a ⇒ SomeContext (c a)

-- -----

type AnyFunc1 res = ∀a. a → res a
mapEither :: AnyFunc1 r → Either a b → Either (r a) (r b)
mapEither f (Left a) = Left $ f a
mapEither f (Right a) = Right $ f a
assessWithin' :: (Context c a) ⇒
  [Information]
  → c a
  → IO (Maybe a, AssessmentDetails a)

assessWithin' inf c = do
  contextInf ← contextInformation c
  contextRels ← contextRelations c
  let assumed = contextInf `graphJoin` inf
    (bins, whole) = partitionEithers
      $ (λr → mapEither ((,) r) $ r `relationOn` assumed)
      <$> contextRels
    assessed = do rBin ← c `combineBinRels` Map.fromList bins
      rWhole ← c `combineWholeRels` Map.fromList whole
      return $ combineRels c rBin rWhole
  return (assessed, ⊥)

-- -----

data AssessedCandidate a = AssessedCandidate {
  assessedAt      :: SomeContext a
  , assessedVal   :: Maybe a
  , assessedDetails :: AssessmentDetails a
}
data Candidate a = Success { assessHistory :: [AssessedCandidate a]
  , candidate     :: [Information]

```

--

```

    }
    | Failure { assessHistory :: [AssessedCandidate a]
               , candidate      :: [Information]
               }
    }

-- -----

assessWithin :: (Context c a, Ord a) =>
  Candidate a -> c a -> IO (Candidate a)

assessWithin f@Failure { } _ = return f
assessWithin (Success hist c) cxt = do
  (mbA, details)  <- c 'assessWithin' cxt
  threshold       <- contextThreshold cxt
  let ac = AssessedCandidate (SomeContext cxt) mbA details
  return $ if mbA > Just threshold
    then Success (ac : hist) c
    else Failure (ac : hist) c

```

Some contexts might also be capable of *splitting* information graphs into *valid candidates* – the sub-graphs, that are *valid* at the context. The candidates can be assessed by the rest of the contexts.

```

class (Context c a) => SplittingContext c a where
  splitGraph :: c a -> IGraph -> IO [Candidate a]

```

1.3.3 Capabilities

The capabilities context handles question “Am I able to do it?”. It’s main purpose is to discard immediately any proposal that would never be accepted.

- *Group*: “Am I interested in the discipline?”
- *Professor*: “Am I qualified to teach the disciple?”
- *Classroom*: “Do I suit the disciple?”, “Do I have the capacity required?”

An agent should mark any other agent, that has declined some proposal for *capabilities* reasons, describing the reason. It should further avoid making same kind of proposals to the incapable agent.

```

data family Capabilities (r :: NegotiationRole) :: * -> *
data instance Capabilities GroupRole a = GroupCapabilities {
  needsDisciplines :: [Discipline]
}
data instance Capabilities FullTimeProfRole a = FullTimeProfCapabilities {
  canTeachFullTime :: [Discipline]
}

```

```

-- -----
data CanTeachRel a = CanTeachRel
instance InformationRelation CanTeachRel where
  relationName _ = "CanTeach"
  coerceRelation = coerce
instance BinaryRelation CanTeachRel where
  binRelValue _ a b =
    let v ds c = if classDiscipline c ∈ ds then 1 else 0
    in case collectInf a of
      Just (CanTeach ds) → let
        r1 = case collectInf b of Just (SomeClass c) → Just $ v ds c
        r2 = case collectInf b of Just (Class c) → Just $ v ds c
        in r1 <|> r2
      _ → Nothing
-- -----

data NeedsDisciplineRel a = NeedsDisciplineRel
instance InformationRelation NeedsDisciplineRel where -- TODO
instance BinaryRelation NeedsDisciplineRel where -- TODO
-- -----

-- Every capability must be coherent. 0*X = 0
combineBinRelsStrict _ bRels | null bRels = Nothing
combineBinRelsStrict _ bRels = Just ∘ CBin ∘ product
                                ∘ concatMap (map relValBetween)
                                $ Map.elems bRels

combineWholeRelsStrict _ wRels | null wRels = Nothing
combineWholeRelsStrict _ wRels = Just ∘ CWhole ∘ product
                                ∘ map unwrapRelValWhole
                                $ Map.elems wRels

combineRelsStrict _ (CBin b) (CWhole w) = b * w
-- -----

instance (Num a) ⇒ Context (Capabilities GroupRole) a where
  contextName _ = "Capabilities"
  contextInformation = return ∘ fromNodes ∘ ([:])
                    ∘ Information ∘ Needs
                    ∘ Set.fromList ∘ needsDisciplines
  contextRelations _ = return [RelBin NeedsDisciplineRel]
  contextThreshold _ = return 0

  combineWholeRels = combineWholeRelsStrict
  combineBinRels = combineBinRelsStrict
  combineRels = combineRelsStrict

instance (Num a) ⇒ Context (Capabilities FullTimeProfRole) a where
  contextName _ = "Capabilities"

```

```

contextInformation = return ◦ fromNodes ◦ ([:])
                  ◦ Information ◦ CanTeach
                  ◦ Set.fromList ◦ canTeachFullTime
contextRelations _ = return [RelBin CanTeachRel]
contextThreshold _ = return 0
combineWholeRels = combineWholeRelsStrict
combineBinRels   = combineBinRelsStrict
combineRels      = combineRelsStrict

```

1.3.4 Beliefs

The beliefs is a *splitting* context, that uses as it's internal knowledge: 1) *state of the timetable*, that represents *best* candidate, generated until now; 2) *interesting* proposals, both generated by agent itself and received from the others, that are preserved throughout agent's lifetime.

Assessing yields one of three values

$$\begin{cases} -1 & \text{if two proposals intersect in time} \\ 0 & \text{if both proposals have the same } \textit{abstract} \text{ part} \\ 1 & \text{otherwise} \end{cases}$$

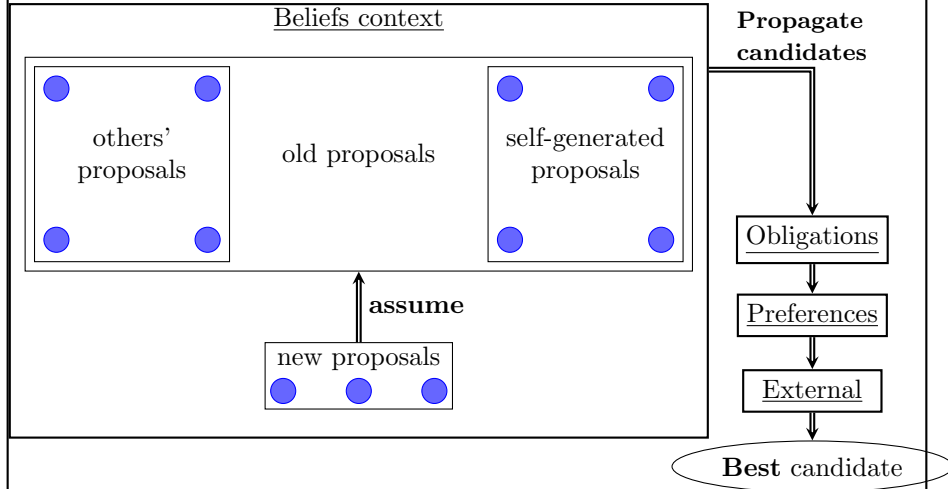


Figure 3: Assessing proposal coherence, starting from *Beliefs* context.

should be written in another place, not in this context

The assessment of *concrete proposals* (containing concrete classes) in the graph consists in

--

1. *assuming* the proposal information;
2. *splitting* the assumed information graph into valid candidates;
3. *propagating* of the candidates through the rest of the contexts;
4. comparing the *best candidate* with the previous *best*.

? The proposal is called *interesting* and is accepted (and the assumed graph becomes the new information graph of *beliefs* context) if it's assumption causes better candidate generation. It's rejected otherwise (and the assumed graph is discarded).

Splitting is a process of extraction of *acceptable* sub-graphs, that compares the coherence values at graph's edges against a threshold. The splitting can be achieved with one of two following strategies:

1. *Joining* proposals while validness is preserved.
2. *Partitioning* of proposals until validness is achieved.

First strategy is used in this project, due to less memory consumption (it doesn't have to generate or store big invalid graphs, that would be present at the first steps of the second strategy).

The splitting is implemented as follows:

Let $C = \{c\}$ be a set of *class proposals*.

$A_i = \{a_i\}$ be a set of *acceptable candidates*, composed of i proposals.

$A = \bigcup_i A_i$ be a set of *acceptable candidates*.

1. Each single candidate is acceptable: $A_1 = \{[c] \mid \forall c \in C\}$.
2. Form A_2 by extending each candidate $[c'] = a_1 \in A_1$ with $c \in C$, if and only if c' and c do not intersect. If $A_1 \neq \emptyset$, then try to form A_2 .
- \vdots
- i. Form A_i by extending each candidate $[c'_1, \dots, c'_{i-1}] = a_{i-1} \in A_{i-1}$ with $c \in C$, if and only if $\forall c' \in a_{i-1}$, c' and c do not intersect. If $A_i \neq \emptyset$, then try to form A_{i+1} .
- \vdots
- n. $A_n = \emptyset \implies$ all the *acceptable candidates* were generated. Done.

```

data Beliefs a = Beliefs { knownProposals :: IORef IGraph }
data TimeConsistency a = TimeConsistency

instance InformationRelation TimeConsistency where
  relationName _ = "TimeConsistency"
  coerceRelation = coerce

instance BinaryRelation TimeConsistency where
  binRelValue _ i1 i2 = do
    Class c1 ← collectInf i1
    Class c2 ← collectInf i2

    let sameParticipant = classGroup c1    ≡ classGroup c2
                        ∨ classProfessor c1 ≡ classProfessor c2
                        ∨ classRoom c1     ≡ classRoom c2

    sameDay = classDay c1 ≡ classDay c2
    timeIntersects x y    = classBegins x ≤ classBegins y
                        ∧ classEnds x    ≥ classBegins y

    sameAbstract = classDiscipline c1 ≡ classDiscipline c2
                  ∧ classGroup c1    ≡ classGroup c2
                  ∧ classProfessor c1 ≡ classProfessor c2
                  ∧ classRoom c1     ≡ classRoom c2
                  ∧ classNumber c1   ≡ classNumber c2

    intersect = sameParticipant
              ∧ sameDay
              ∧ (timeIntersects c1 c2 ∨ timeIntersects c2 c1)

    return $ if sameAbstract then 0
              else if intersect then -1 else 1

instance (Num a) ⇒ Context Beliefs a where
  contextName _      = "Beliefs"
  contextInformation = readIORef ∘ knownProposals
  contextRelations _ = return [RelBin TimeConsistency]
  contextThreshold _ = return 0

  combineWholeRels = combineWholeRelsStrict
  combineBinRels   = combineBinRelsStrict
  combineRels      = combineRelsStrict

instance (Num a) ⇒ SplittingContext Beliefs a where
  splitGraph b gr = do
    iGraph ← readIORef $ knownProposals b
    let cNodes = catMaybes $ collectInf <$> graphNodes gr
        consistent x y = binRelValue TimeConsistency x y ≡ Just 1
    extendCandidate Failure { } = []
    extendCandidate Success { candidate = inf } = do
      c ← cNodes
      [Success { assessHistory = [], candidate = graphNodes gr ++ [c] }
       | all (consistent c) inf]

```

```

    a1 = Success [] ∘ (:[]) <$> cNodes
    return $ fix (λf acc last → let ext = concatMap extendCandidate last
                                in if null ext
                                then acc
                                else f (acc ++ ext) ext
    ) a1 a1

```

1.3.5 Obligations

Obligations determine the rest *strong restrictions* over the classes. Possible obligations might depend on agent's role and are usually determined by the institution. For example: maximum classes per day, lunch recess, lower/upper class time limit, two classes must/cannot follow etc.

The expected values are

- 0 if the obligation is broken;
- 1 otherwise.

All the obligations must comply over a candidate.

```

data Obligations a = Obligations {
    obligationsInfo :: [Information],
    obligationsRels :: [IRelation (ZeroOrOne a)]
}

instance (Num a) ⇒ Context Obligations a where
    contextName _ = "Obligations"
    contextInformation = return ∘ fromNodes ∘ obligationsInfo
    contextRelations   = return ∘ map coerceIRelation ∘ obligationsRels
    contextThreshold _ = return 0

    combineBinRels   = combineBinRelsStrict
    combineWholeRels = combineWholeRelsStrict
    combineRels      = combineRelsStrict

    -- This constructor should be hidden.
    newtype ZeroOrOne a = ZeroOrOne a
    complies = ZeroOrOne 0
    fails    = ZeroOrOne 1

```

1.3.6 Preferences

Preferences determine *weak restrictions*, that are intended to be set by the represented person (the institution in case of the classroom).

The expected value must be inside $[0, 1]$ (unit) interval. They are combined as follows:

Binary:

\forall binary preference relation $\text{pref}_i \implies$

$$P_{\text{bin}}^i = \frac{\sum_{\langle n_1, n_2 \rangle} \text{pref}_i(n_1, n_2)}{|\{\langle n_1, n_2 \rangle\}|}$$

$$P_{\text{bin}} = \prod_i P_{\text{bin}}^i; \quad P_{\text{bin}}^i \in [0, 1]; \quad P_{\text{bin}} \in [0, 1].$$

Whole:

\forall whole graph relation pref_i

$$P_{\text{whole}} = \prod_i \text{pref}_i(\text{graph})$$

Combined: $P = P_{\text{whole}} \times P_{\text{bin}}$

The context should diminish its influence over time to avoid possible over-restrictions due to conflicting personal interests.

```

data Preferences a = Preferences {
  preferencesInfo      :: [Information],
  preferencesRels      :: [IRelation (InUnitInterval a)],
  preferencesThreshold :: IORef a
}

instance (Fractional a) => Context Preferences a where
  contextName _ = "Preferences"
  contextInformation = return o fromNodes o preferencesInfo
  contextRelations   = return o map coerceIRelation o preferencesRels
  contextThreshold   = readIORef o preferencesThreshold

  combineBinRels _ = fmap CBin o combineBinRelsMeansProd'
  combineWholeRels _ = fmap CWhole o combineWholeRelsProd'
  combineRels      = combineRelsProd

  -- -----

  maybeMean [] = Nothing
  maybeMean xs = Just $ sum xs / fromIntegral (length xs)
  combineBinRelsMeansProd' :: (Fractional a) => RelValsBetween a -> Maybe a
  combineBinRelsMeansProd' = foldr f Nothing
  where mean' = maybeMean o map relValBetween
        f xs acc@(Just _) = ((*) <$> acc <*> mean' xs) <|> acc
        f xs _              = mean' xs

```

```

combineWholeRelsProd' mp | null mp = Nothing
combineWholeRelsProd' mp = Just ◦ product ◦ map unwrapRelValWhole
    $ Map.elims mp

combineRelsProd _ (CBin bin) (CWhole whole) = bin * whole

-- -----

newtype InUnitInterval a = InUnitInterval a
inUnitInterval :: (Fractional a, Ord a) ⇒ a → Maybe (InUnitInterval a)
inUnitInterval x |      x ≥ 0
                  ∧      x ≤ 1 = Just $ InUnitInterval x
inUnitInterval _ = Nothing
fromUnitInterval (InUnitInterval x) = x

```

1.3.7 External

External contexts take into account the *opinions* of the agents that are referenced by the solution candidate. It is responsible for *common goal* assessment. The assessment must be *objective* — it must give no preference to agent’s own interests.

1.3.8 Decision

1.4 Agent

Here follows *agents* implementation.

```

class AgentComm ag where
class (AgentComm ag) ⇒ CommAgentRef ref ag where
    agRef    :: ag → ref ag
    agComm :: ref ag → ag
data AgentRef = ∀ref ag. CommAgentRef ref ag ⇒ AgentRef (ref ag)
-- -----

data GroupRef    = GroupRef    String deriving (Show, Eq, Ord)
data ProfessorRef = ProfessorRef String deriving (Show, Eq, Ord)
data ClassroomRef = ClassroomRef String deriving (Show, Eq, Ord)

```