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.

1 Implementation

1.1 University Classes

A class is an en 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*.

For inner usage, the classes are divided into

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

```
class AbstractClass c where classDiscipline :: c → Discipline
                            classGroup      :: c → GroupRef
                            classProfessor  :: c → ProfessorRef
                            classRoom       :: c → ClassroomRef
                            classNumber     :: c → Word
class (AbstractClass c) ⇒ ConcreteClass c time | c → time
  where classDay    :: c → Day
        classBegins :: c → time
        classEnds   :: c → time
data Class time = ∀c      .ConcreteClass c time ⇒ Class c
data SomeClass = ∀c       .AbstractClass c ⇒ SomeClass c
  -- redefined 'System.Time.Day' – no '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.

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

	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 (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.

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 is 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 the 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 theese pieces.

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 arrangments 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. **Complete** – references all three representing agents: a *group*, a *professor* and a *classroom.*
     - ii. **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 **??**.

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.*

```
class (Typeable i) ⇒ InformationPiece i
data Information = ∀i.InformationPiece i ⇒ Information i
  -- ————————————————————————————————
data RelValBetween a = RelValBetween {
    relBetween    :: (Information, Information)
  , relValBetween :: a
  }
type RelValsBetween a = [RelValBetween a]
```

```
newtype RelValWhole a = RelValWhole a
unwrapRelValWhole (RelValWhole a) = a
  -- ------------------------------------------
class BinaryRelation r where
  binRelValue :: r a → Information → Information → Maybe a
class WholeRelation r where wholeRelValue :: r a → IGraph → a
data IRelation a = ∀r.BinaryRelation r ⇒ RelBin (r a)
                 | ∀r.WholeRelation r ⇒  RelWhole (r a)
type RelValue a = Either (RelValsBetween a) (RelValWhole a)
  -- ------------------------------------------
class InformationGraph g where
  graphNodes :: g → [Information]
  graphJoin  :: g → [Information] → g
  relationOn :: IRelation a → g → RelValue a
data IGraph = ∀g.InformationGraph g ⇒ IGraph g
instance InformationGraph IGraph where
  graphNodes    (IGraph g) = graphNodes g
  graphJoin     (IGraph g) = IGraph ∘ graphJoin g
  relationOn r  (IGraph g) = r `relationOn` g
```

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.

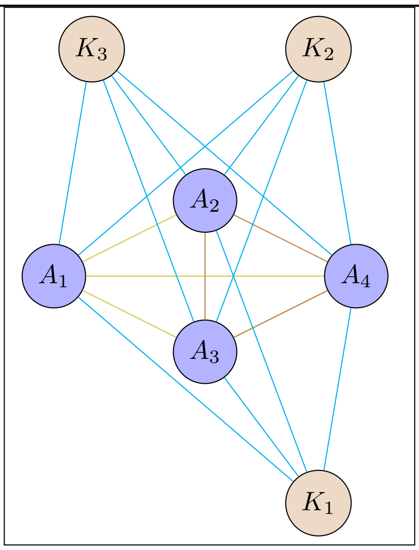

Figure 2: Binary relations within an information graph. One can distinguish the relations between the assessed information pieces and the relations between assessed and the known ones.

To assess some information, it's propagated through the contexts, in the *specified order*, that stands for contexts priority. Each context xshould 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 :: * → *) where
  contextName        :: c a → String
  contextInformation :: c a → IGraph
  contextRelations   :: c a → [IRelation a]
  contextThreshold   :: c a → IO a
  combineBinRels     :: c a → RelValsBetween a    → Maybe (CBin a)
  combineWholeRels   :: c a → [RelValWhole a]     → 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 ⇒ SomeContext (c a)
  -- ────────────────────────────────
assessWithin' ::      (Context c) ⇒
                      [Information]
```

```
    →                  c a
    →                  (Maybe a, AssessmentDetails a)
  assessWithin' inf c = (assessed, ⊥)   -- TODO
    where assumed = contextInformation c `graphJoin` inf
          (bins, whole) = partitionEithers
                          $ (`relationOn` assumed) <$> contextRelations c
          rBinMb = c `combineBinRels` concat bins
          rWhole = c `combineWholeRels` whole
          assessed = flip (combineRels c) rWhole <$> rBinMb
    -- ---------------------------------
  data AssessedCandidate a = AssessedCandidate {
        assessedAt      :: SomeContext a
    ,   assessedVal     :: Maybe a
    ,   assessedDelails :: AssessmentDetails a
    }
  data Candidate a = Success {assessHistory :: [AssessedCandidate a]
                             , candidate    :: [Information]
                             }
                   | Failure {assessHistory :: [AssessedCandidate a]
                             , candidate    :: [Information]
                             }
    -- ---------------------------------
  assessWithin :: (Context c, Ord a) ⇒
                  Candidate a → c a → IO (Candidate a)
  assessWithin f@Failure { } _ = return f
  assessWithin (Success hist c) cxt = do
    let (mbA, details) = c `assessWithin'` cxt
        ac = AssessedCandidate (SomeContext cxt) mbA details
    threshold ← contextThreshold cxt
    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) ⇒ SplittingContext c where
    splitGraph :: c a → IGraph → [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 have been accepted.

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

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

1.3.4 Beliefs

The beliefs is a *splitting* context, that uses as it’s internal knowledge the state of the timetable at the moment.

Assessing yields one of three values

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

The assessment of *concrete proposals* (containing concrete classes) in the graph consists in finding *time coherence* for every possible pair of *different* proposals. If any of the coherence values $\neq 1$, then the graph is invalid and the assessment is -1. In case that all coherence values are (strongly) positive, the result is 1.

1.3.5 Obligations

1.3.6 Preferences

1.3.7 External

1.3.8 Decision

1.4 Agent

Here follows *agents* implementation.