The Erlangen Advanced Timetabling System (EATTS) Unified XML File Format for the Specification of Timetabling Systems

Johannes Ostler · Peter Wilke

Abstract It would be nice if all timetabling problems could be described in a canonical and universal description language with standardized syntax. If such a description would be available algorithms and/or frameworks could be used to interpret and/or compile these descriptions, and solve the problem at stake. And systems, algorithms, problems, etc. could be compared in a simple way.

Here we would like to suggest to step towards a standard, a XML format to define a great number of different problems like Course Timetabling, High School Timetabling, Employee Timetabling et cetera. In addition tools to edit these descriptions and to apply algorithms to solve the described problem.

Keywords EATTS · Erlangen Advanced Timetabling System · XML · Benchmarks · File Format · Standard · Timetabling Systems · Timetabling Problems

1 Introduction

A common and widely accepted description format for timetabling problems and its solution would be a great help when algorithms and/or frameworks are compared. When looking at the literature there are description formats but they are too specific and can't be used for general problems. The narrow focused system evolved due to historic reasons, specific domain properties or efficiency reasons, e.g. very often these description are incomplete because part of the problem is hard coded in the implementation. In the year 2003 our research group published a approach for a unified timetabling framework and format [GWB03]. In the following years we continued this work and would like to suggest the next version of our description format.

Johannes Ostler
Universitaet Erlangen-Nuernberg, Informatik 5, Martensstrasse 3, 91058 Erlangen, Germany
Tel.: +49 (9131) 85-27998
E-mail: Johannes.Ostler@Informatik.Uni-Erlangen.DE

Peter Wilke
Universitaet Erlangen-Nuernberg, Informatik 5, Martensstrasse 3, 91058 Erlangen, Germany
Tel.: +49 (9131) 85-27998
E-mail: Peter.Wilke@Informatik.Uni-Erlangen.DE

2 Existing XML format for High School Timetabling Problems

There are some approaches on standardization of formats to define timetabling problems of a certain problem type. Especially the standardized format for benchmarks in High School Timetabling [PAD$^+$08] is interesting. In that paper a way is shown to give a unified format for High School Timetabling problems of different countries. The purpose is to provide a format for benchmark data interchange between independent groups of researchers. "The students are organized in base groups, and follow lessons. The lessons of a (usually fixed) group of students in the same subject (like math, arts, ...), usually with the same teacher, we will call a course."[PAD$^+$08, p. 3] Students, teachers and rooms are the three main groups of resources. Moreover there are assets, which are organized in "four categories: Time, Courses, Subjects, and Resources"[PAD$^+$08, p. 7]. Every time slot has a sequence number, the day and optional the week is attached. "A course is a collection of events that involves a group of students and a subject. Events that refer to the same course, are called lessons of the course. ... A resource is an entity with time restrictions. The most common resources are the students, teachers, and rooms. ... Events are the basic scheduling object. An event can represent either a single lesson, or a set of lessons, that have to be taught at the same time."[PAD$^+$08, p. 8f] There are a set of standardized constraints, which can be referenced in the XML. Their parameters can be set via XML tags.

The structure of the XML is:

```
<Problem>
  <Instance>
    <Assets> ... </Assets>
    <Events> ... </Events>
    <Constraints> ... </Constraints>
  </Instance>
  <Solutions> ... </Solutions>
</Problem>
```

Listing 1 Structure of High School Timetabling XML [PAD$^+$08, p. 11f]

The `Solutions` tag includes an `Event` tag per event which stores information about the assignments of this event. Information about the costs and/or the constraint violations also can be included.

3 Benefits of a new XML format

The existing formats can only describe a special class of timetabling problems. But the need for flexibility requires to categorize and compare many different problems. One approach would be to adapt our solver so that it can read many input file formats. But if there is a unified format other research groups can use our benchmark files, too.

We decided to create a XML format for the following reasons. First of all XML is a standard of information interchange using internet and good support to handle easily XML data is available in most computer languages. XML is a structured format, the syntax can be defined by a Document Type Definitin (DTD) [wik10] or a XML Schema Definition (XSD) [W3C10a]. At least XML is easy to understand and many software engineers are familiar with it. More information about XML can be found on the websites of W3C® [W3C10b].

4 The EATTS XML File Format for Problem Definitions

4.1 Entities of a Timetabling Problem

Descriptions start with the specification of the entities of a typical Timetabling Problem. In many cases a set of time slots has be assigned to a set of events. Usually this assignment is subject to some constraints. In some cases other resources are not assigned fixed to the events. For example in Course Timetabling exactly one room has be chosen from a set of rooms.

Our approach to describe different Timetabling problems distinguishes between three main entity types: resources (including time slots), events and constraints. The solution of a Timetabling problem is the assignment of resources of different resource types to the events. Solution have assigned costs depending on the violated constraints.

In our approach the description consists of a model part and a data part. While the model takes care of the general structure of the problem at hand the data part takes care of the individual components of the problem. For a School Timetabling Problem the model would contain the resource types like teachers and rooms, while the data part would contain the individual teacher with names, hours per week, subjects, etc.

4.2 RECPlan

In the EATTS XML format a Timetabling Problem is declared by a `RECPlan` tag. REC stands for Resources, Events and Constraints. The typical structure of this tag is shown in the listing below.

```
<RECPlan name="nameOfThePlan" planVersion="0">
  <TimeFrame ... />
  <ResourceTypes> ... </ResourceTypes>
  <ResourceCollections> ... </ResourceCollections>
  <Resources> ... </Resources>
  <Events> ... </Events>
  <EventCollections> ... </EventCollections>
  <Constraints> ... </Constraints>
</RECPlan>
```

Listing 2 Structure of RECPlan

The `TimeFrame` tag defines the time frame of the planning period.

The defintion of resource types starts with the tag `ResourceTypes`. Resources of a resource type can be subdivided into collections which are defined in the tag `ResourceCollections`. The same mechanism is available to describe event types and event collections. To specify the resources the tag `Resources` is used, similar for events the tag `Events`.

The constraints have there own part in the description starting with the tag `Constraints`.

4.3 TimeFrame

The planning period of a Timetabling Problem starts at absolute date *start*. All other points in time and intervals of time are declared relatively to this date. Because of the variety of timetabling problems it is necessary to define a time grid. Typically schedules

are represented as a table. Often the columns represents one day and the lines one hour or parts of one hour. The time which lies between the first point in time of column i and this of column $i+1$ is called the horizontal scaling hs. The vertical scaling vs is the minimal timespan between two vertical lines. The value of the time t is given relatively to $start$, so $t_{abs} = t + start$. A time slot (s, d) starts at the absolute point in time $t_{start} = s + start$ and ends at $t_{end} = t_{start} + d$.
The following condition must be true to ensure that all sub-units can be mapped correctly to the given time line.

$$\{hs, t, s, d\} \subset \{x = n \cdot vs \mid n \in \mathbb{N}\} \tag{1}$$

```
<TimeFrame absoluteStart="1253526334272" horizontalScaling="86400000"
           verticalScaling="900000" cycleLength="7" repeatNCycles="1"/>
```

Listing 3 Structure of RECPlan

The attributes *horizontalScaling* and *verticalScaling* determine the horizontal and vertical scaling. The unit of time values is milliseconds. The absolute start *absoluteStart* is given as a Unix timestamp value. Unix time stamps are widely supported, even on windows systems, e.g. Java VM. The period to be planned is typically a multiple of hs, typical periods are a week respectively seven days (see above). In that case the attribute *cycleLength* has the value seven. For School Timetabling the schedule of a standard week will be valid for the following x weeks. So the attribute *repeatNCycles* will have the value x.

4.4 Resource Types

Every family of Timetabling Problems has the same types of resources. For example in Course Timetabling there are teachers, rooms, classes, students, and subjects, while for rosters the resource types are employees, machines et cetera. The description starts with the definition of resource types and after that the resources of each resource type are defined. Every resource type has a denomination like *Teacher* or *Employee* and a set of attributes. The definition of an attribute includes the name and the type of the attribute's value.

```
<ResourceTypes>
  <Type name="TimeSlot">
    <Attribute name="Slot" typeKind="S" typeDenom="TimeSlot" isList="false"/>
  </Type>
  <Type name="Room">
    <Attribute name="Name" typeKind="S" typeDenom="String" isList="false"/>
    <Attribute name="Size" typeKind="S" typeDenom="Integer" isList="false"/>
  </Type>
  ...
</ResourceTypes>
```

Listing 4 Example of a ResourceTypes tag

The `Type` tag defines a resource type. The attribute *name* specifies its name. This tag includes a set of `Attribute` tags, which represents the attribute definitions of the resource type. The attribute *typeKind* typecasts the type of the attribute value. Possible types are "S" for simple, "G" for group, "R" for resource and "E" for event. The type simple includes elementary values like integers, floats, strings or time slots.

The attribute *typeDenom* specifies this type more precisely. The attribute values can be single values or a list of values. This property will be defined by the attribute *isList*.

4.5 ResourceCollections

Resources of a certain type can be grouped in collections. There are four different kinds of collections.

1. Simple groups are called resource group. A resource group includes a set of resources of a certain resource type. The relationship between a resource and a resource group is n : m. Every resource can be added to any number of collections and vice versa.
2. The union of the elements of a set of collections can be defined by the `Union` tag.
3. The `Intersection` specifies the intersection of the elements of a set of collections.
4. If the set R_{rt} is the set of all resources of the resource type rt and G_{rt} is any resource collection of the resource type rt, then $INV(G_{rt}) = \{r \in R_{rt} | r \notin G_{rt}\}$ is the absolute complement of G_{rt}. The tag `Inverter` defines such a complement of a resource collection.

```
<ResourceCollections>
  <ResourceCollectionsByType typeName="TimeSlot">
    <Group name="MorningSlots"/>
    <Inverter name="AfternoonSlots">
      <CollectionRef name="MorningSlots"/>
    </Inverter>
    <Union name="AllSlots">
      <CollectionRef name="MorningSlots"/>
      <CollectionRef name="AfternoonSlots"/>
    </Union>
  ...
  </ResourceCollectionsByType>
  ...
<ResourceCollections>
```

Listing 5 Example of a ResourceCollections tag

All collections of a plan are enclosed in `ResourceCollections` tag. The `Resource-CollectionsByType` tag is a container for all definitions of collections of a certain resource type. The attribute *typeName* specifies the corresponding resource type by its name.

A simple collection of resources is defined by a `Group` tag. The *name* attribute determines the denomination of the collection. This name must be unique for all collections of the same type. This tag does not include any information regarding which resources are members of the group. A `CollectionRef` tag references a collection by its name. A union or intersection can include any number of other collections while an inverter can include only one.

4.6 Resources

The resource definitions are encapsulated in `Resources` and `ResourcesByType` tags following the collections. The `Resource` tag defines one resource. Every resource has its unique id which is declared in the *id* attribute. Enclosed in a `AttributeValues` tag the `AttributeValue` determines the value of the attribute which is referenced by the value of the *name* attribute. The resource defines to which resource groups it is assigned to.

```
<Resources>
  <ResourcesByType typeName="Class">
    <Resource id="Class0">
      <AttributeValues>
        <AttributeValue name="classRoom">
          <ResourceRef id="Room16" resourceType="Room"/>
        </AttributeValue>
        <AttributeValue name="name">
          <String>1A</String>
        </AttributeValue>
        <AttributeValue name="grade">
          <String>1</String>
        </AttribteValue>
        <AttributeValue name="teacher">
          <ResourceRef id="Teacher7" resourceType="Teacher"/>
        </AttributeValue>
        <AttributeValue name="group">
          <ResourceCollectionRef name="students 1A" resourceType="Student"/>
        </AttributeValue>
      </AttributeValues>
      <ContainingGroups>
        <CollectionRef name="sports male of grade 1"/>
        <CollectionRef name="sports female of grade 1"/>
      </ContainingGroups>
    </Resource>
  </ResourcesByType>
</Resources>
```

Listing 6 Example of `Resources` tag

The values of the attributes are enclosed in tags which depend on the type of the value. Possible Tags and their meaning are listed in the following table.

Value Tag	Type of Value
String	string literals
Integer	integer numbers
Float	floating-point numbers
Boolean	boolean values
Time	relative time values
TimeSlot	relative time spans
ResourceRef	references to resources
ResourceCollectionRef	references to a resource collection
EventCollectionRef	references to a event collection

Table 1 Value types of attributes

4.7 Events

The planning process is all about events. The time table - as a result of the planning process - determines which resource is assigned to which event. Below rt a resource type and e an event of the plan is shown. For every event it can be declared how much resources of a certain resource type have to be assigned as minimal $min_{e,rt}$ or maximal $max_{e,rt}$ value so that the resulting time table is a valid solution. Two additional subsets of resources of this type can be defined: one which is assigned fixed to this event and/or one subset of resources which could be assigned, at least one of them must exist.

Let $F_{e,rt}$ and $O_{e,rt}$ be a subset of R_{rt}. $F_{e,rt}$ is the set of resources of the resource type rt which is assigned fixed to the event e and $O_{e,rt}$ is the set of resources from which the planning component can select. Let s be a result of the timetabling problem, so $s(e, rt)$ is the set of resources of resource type rt which are assigned to the event e and $s(r)$ is the set of events assigned to the resource r.
The following must be true:

$$F_{e,rt} \subset R_{e,rt} \subset F_{e,rt} \cup O_{e,rt} \qquad (2)$$

The following condition should be true to avoid constraint violations.

$$min_{e,rt} \leq |R_{e,rt}| \leq max_{e,rt} \qquad (3)$$

```
<Events>
  <Event name="5A_Maths">
    <ResourceLists>
      <ResourceList resourceTypeName="TimeSlot" max="4" min="4" fixed="true">
        <OptionalGroup name="MorningSlots"/>
      </ResourceList>
      <ResourceList resourceTypeName="Class" max="1" min="1" fixed="true">
        <FixedGroup name="class 5A"/>
      </ResourceList>
    ...
      <ResourceList resourceTypeName="Teacher" max="1" min="1" fixed="true">
        <OptionalGroup name="teachers of 5A_Maths"/>
      </ResourceList>
      ...
   </ResourceLists>
   <ContainingGroups>
    <EventCollectionRef name="Maths Lessons">
    <EventCollectionRef name="Lessons of 5A">
   </ContainigGroups>
  </Event>
</Events>
```

Listing 7 Example of `Events` tag

The `Event` tag defines an event. Its name is set by the *name* attribute. For every resource type this tag includes a `ResourceList` tag, which defines $min_{e,rt}$ by the attribute *min* or $max_{e,rt}$ by *max*. $F_{e,rt}$ is declared by the tag `FixedGroup`, $O_{e,rt}$ by the `OptionalGroup` tag. If one of these group tags is missing the corresponding group is believed to be an empty set, but at least one of them must be not empty. The sets are referenced by the name of the group, which is stored in the *resourceTypeName* attribute. Just like the resources events can also be classified into collections. The `EventCollectionRef` tags in the `ContainingGroups` tag determines to which event group this event is assigned to.

The assignments of resources of a certain type rt to a set of events $\{e_1, ...e_n\}$ can be linked. That means that $s(e_1, rt) = s(e_2, rt) = ... = s(e_n, rt)$. For this purpose the attribute *baseEvent* must be added to the `ResourceList` tags of $\{e_2, ...e_n\}$. The value of this attribute is the name of e_1.

4.8 Constraints in general

In the preceding sections a unified model for defining different Timetabling Problems was described. Now it's time to develop a uniform way to declare constraints. First of all lets have a look at different solutions s_i of the same problem. A constraint measures the quality of a solution from a certain point of view. It defines a cost function $q_c(s_i)$. The codomains of these functions are subsets of $\mathbb{R}_0^+$. Let C be the set of all constraints of a problem. The quality of a solution s_i is defined as $Q(s_i)$.

$$Q(s_i) = \sum_{c \in C} q_c(s_i). \tag{4}$$

The only aspect in which the solutions s_k and s_l can differ are the different assignments of resources to the events. If the costs of a constraint does not depend on these assignments this means that they are fixed costs for all solutions and can't be reduced by any planning algorithm. For simplicity they should not be part of the description, i.e. only variable costs should be defined in the model.

A constraint is defined by its scope $scope_c$ and a function $q_c(s, y)$, whereby s is a solution and y is an element of the scope. It is typical that the constraints differ in weight. So a weight function $w_c : \mathbb{N} \to \mathbb{R}$ must be defined for every constraint c. That way there must be a function $viol_c(s, y) : S \times scope_c \to \mathbb{N}$ which counts the number of violations against constraint c in the solution s on the view of y.

A view of a resource or event can be understood as looking at the solution showing only the parts where this resource or event is involved. E.g. a student wants to see his personal time table and not the whole plan of the entire school.

$$q_c(s, y) = w_c(viol_c(s, y)) \tag{5}$$

$$q_c(s) = \sum_{y \in scope_c} q_c(s, y) \tag{6}$$

```
<Constraints>
  <OperationLists> ... </OperationLists>
  <Selectors> ... </Selectors>
  <Variables> ... </Variables>
  <TimeClashConstraint>
    <ConstraintProperties name="TimeClash for Teacher" type="hard">
      <PenaltyFunction weight="50.0" coefA="1.0" coefB="0.0" exponent="2.0"/>
      <Description>
        No teacher can give more than one lesson at the same point in time.
      </Description>
    </ConstraintProperties>
    <ResourceTuple>
      <ResourceCollectionRef name="Teacher_All" resourceType="Teacher"/>
    </ResourceTuple>
  </TimeClashConstraint>
</Constraints>
```

Listing 8 `Constraints` tag

The definitions of the constraints are enclosed in the `Constraints` tag. To define standardized constraints we use lists of operations, selectors and variables. These entities will be explained in the sections 4.9, 4.10, and 4.12 below. Now the typical structure of a constraint definition is shown using the example of a `TimeClashConstraint` tag. The constraint of the example is violated if one teacher gives more than one lesson at the same time. The `ConstraintProperties` tag declares all properties common to all types of constraints. Every constraint has a denomination given by the *name* attribute and a *type*. In our description language constraints come in three different flavours: the familiar hard and soft constraints and in addition soft hard constraints. In normal mode the latter are considered as hard constraints but in exception mode - when no feasible solution could be found in normal mode - they are considered as soft constraints. This eases the burden on the planning algorithm. Every constraint c must have a weight function w_c. We use a standardized weight function which depends on four float parameters: weight, coefA, coefB and exponent. These parameters are set by the corresponding attributes of the `PenaltyFunction` tag.

$$w_c(x) = \begin{cases} weight \cdot (coefA \cdot x^{exponent} + coefB) & \\ 0 & \end{cases} \quad if \quad \begin{matrix} x \in \mathbb{N}^+ \\ x = 0 \end{matrix} \tag{7}$$

The `Description` offers the opportunity to document additional information.

The `ResourceTuple` tag declares that this constraint must be evaluated for every resource of the resource type *Teacher*. The view of a constraint definition can be one of the following: the view of an event, the view of a relation between events or the view of one resource. An example for the latter is the view of a teacher. So there are three different main types of constraints in the model: event, event relation and resource constraints.

4.9 Event Constraints

An event constraint ec measures a solution s from the view of a single event e. The costs of the event e of ec for the solution s are defined by the function $viol_{ec}(s, e)$. Every event constraint has a scope $Scope_{ec} \subset E$.

Let's have a look at the declaration of $viol(s, e)$. Let $A(s, e)$ be the assigned resources of event e in the solution s. The violation count function $viol(s, e)$ indicates if the constraint is violated in regards to event e in the context of solution s. Typically $viol(s, e)$ is composed of three parts: two operations $evar_i(A(e))$ and $evar_j(A(e))$ and a relational operator *rop*. An event variable *evar* is composed of three parts too: a selection function $sel(A(e)) : A(s, e) \rightarrow s(e, rt_i) \subset R_{rt_i}$, an attribute value selection function $attr : R_{rt_i} \rightarrow V$, and an operation $op : V \rightarrow V'$. Hereby V and V' are sets of lists of attribute values. The function *rop* maps the cross product $V'_l \times V'_r$ into the set of natural numbers.

$$evar(A(s, e)) = op(attr(sel(A(s, e)))) \tag{8}$$

$$viol_{ec}(s, e) = rop(evar_i(A(s, e)), evar_j(A(s, e))) \tag{9}$$

An example of an event constraint if the Room Size Constraint which is violated if the capacity of one of the assigned rooms is less than the number sc of assigned students.

$$viol^*_{ec}(e, s, r) = \begin{cases} 1 & if \quad capacity(r) < sc \\ 0 & else \end{cases} \tag{10}$$

$$viol_{ec}(e, s) = \sum_{r \in s(e, Room)} viol^*_{ec}(e, s, r) \tag{11}$$

```
<Selectors>
  <EventSelector name="Room Capacity Selector" lookAtEveryResourceSingle="
      true">
    <TupleValue type="Room" attribute="Capacity"/>
  </EventSelector>
```

```
    <EventSelector name="Student Count Selector" lookAtEveryResourceSingle="
        false">
      <TupleValue type="Student"/>
    </EventSelector>
  </Selectors>
```

Listing 9 Selectors of Room Size Constraint

The *Room Capacity Selector* returns the capacity values of the assigned rooms. Because *lookAtEveryResourceSingle* has the value *true* the capacity of every assigned room must be compared with the right operand. The *Student Count Selector* collects all assigned students into a list of resources. Because only the count of students matters no attribute value is selected.

```
<OperationLists>
  <OperationList name="List Length">
    <Operation>list_size</Operation>
  </OperationList>
</OperationLists>
  ...
<Variables>
  <EventVariable name="Student Count" typeDenom="Integer">
    <EventSelectorRef name="Student Count Selector"/>
  <OperationListRef name ="List Length"/>
  </EventVariable>
  <EventVariable name="Room Capacity" typeDenom="Integer">
    <EventSelectorRef name="RoomCapacitySelector"/>
  </EventVariable>
</Variables>
```

Listing 10 Variables of Room Size Constraint

The values returned by the selectors will be handled by the variables. *Student Count* returns the size of the students list of the *Student Count Selector*, hereby it uses the operation list *List Length*.

```
  <EventConstraint relOpDenom="geq">
     <ConstraintProperties name="Room size" type="soft" slopeType="split">
       <PenaltyFunction weight="50" coefA="1" coefB="2" exponent="1"/>
       <Description>The room size must be >= student count</Description>
     </ConstraintProperties>
     <EventVariableRef name="Room Capacity"/>
     <EventOperand>
       <EventVariableRef name="Class Capacity"/>
     </EventOperand>
     <CollectionRef name="Events_All"/>
   </EventConstraint>
```

Listing 11 Room Size Constraint

The relational operator $\geq$ is declared by the attribute *relOpDenom*. There is a list of different relational operators which can be referenced by name. The `CollectionsRef` tag defines the scope of the constraints, in this case all events. The event operand can also be a constant.

A special type of event constraints is the `MinMaxConstraint`. The scope of this constraint is a set of events and the constraint is defined for a certain resource type rt. The resource list, which is associated with event e and resource type rt, defines a minimal $min(e, rt)$ and maximal $max(e, rt)$ count of resources giving the interval in which the number of resources assigned to event e should lie.

$$viol_{minmax\_rt}(e) = \begin{cases} min_{e,rt} - |s(e,rt)| & if \quad |s(e,rt)| < min_{e,rt} \\ |s(e,rt)| - max_{e,rt} & if \quad |s(e,rt)| > max_{e,rt} \\ 0 & else \end{cases} \tag{12}$$

```
<MinMaxConstraint type="minmax">
  <ConstraintProperties name="TimeSlot MaxConstraint" type="hard">
    <PenaltyFunction weight="100.0" coefA="1.0" coefB="2.0" exponent="0.0"/>
    <Description/>
  </ConstraintProperties>
  <ResourceTypeRef name="TimeSlot"/>
  <CollectionRef name="All"/>
</MinMaxConstraint>
```

Listing 12 MinMaxConstraint tag

The attribute *type* defines which limit must be checked. Possible values are '*min*' (only the minimal limit), '*max*' (only the maximal limit), and '*minmax*' (check both limits). The tags `ResourceTypeRef` references the resource type rt, and `CollectionRef` defines the scope of the constraint.

4.10 Resource Constraints

The view of a resource constraint is the view of a single resource or a tuple of resources. A special example of the first case is shown above with the *TimeClashConstraint*. If a student *stud* can choose between courses of the same subject *sub*, there will be a constraint necessary to ensure that the student will participate only at one of these courses. One possibility to express this constraint is to define a maximal number of events to which both (*stud* and *sub*) are assigned to.

In this case the view of this constraint are tuples $(stud, sub) \quad whereby \quad stud \in R_{Student} \quad and \quad sub \in R_{Subject}$. Below the definition of a resource constraint by the example of a Workload Constraint is given. Every teacher has an attribute *workload* and this limit should not be exceeded by the generated solution.

A resource tuple selector rts maps the set of all resources to a set of resource tuples. The codomain $Codom(rts)$ of rts is a subset of $T_{rt_1,...,rt_n}$ the set of all tuples of resources of the resource types $rt_1, ..., rt_n$.

$$\begin{gathered} Codom(rts) \subseteq T_{rt_1,...,rt_n} \\ T_{rt_1,...,rt_n} = \{tuple \in R_{rt_1} \times ... \times R_{rt_n} : rt_i \neq rt_j \quad \forall i,j \in [1,n] \wedge i \neq j\} \end{gathered} \tag{13}$$

Let tuple $t = (r_1, ..., r_n)$ be an element of $T_{rt_1,...,rt_n}$. There are two ways to define the associated events, on the one hand the intersection way $IS(t)$, on the other hand the union way $U(t)$.

$$IS(t) = \{e \in E \mid e \in s(r_i) \quad \forall i \in [1,n]\} \tag{14}$$

$$U(t) = \{e \in E \mid (\exists i \in [1,n] : e \in s(r_i))\} \tag{15}$$

A resource selector rs includes a resource tuple selector rts_{rs} and defines whether US or IS will be applied. The resource selector rs_{rc} of a resource constraint rc defines the scope $scope_{rc}$.

$$scope_{rc} = Codom(rts_{rc}) \tag{16}$$

Let $t \in scope_{rc}$ be a tuple.

$$rs(t) = \begin{cases} IS(t) \\ U(t) \end{cases} \tag{17}$$

```
<Selectors>
 <ResourceSelector name="Events of a Teacher" unionType="Union">
  <ResourceTuple>
   <ResourceCollectionRef name="Teacher_All"
                          resourceType="Teacher" type="elem"/>
  </ResourceTuple>
 </ResourceSelector>
 <ResourceSelector name="Student x Subject" unionType="Intersection">
  <ResourceTuple>
   <ResourceCollectionRef name="Student_All" resourceType="Student"
                          type="elem"/>
   <ResourceCollectionRef name="mainSubjects"
                          resourceType="Subject" type="group"/>
  </ResourceTuple>
 </ResourceSelector>
</Selectors>
```

Listing 13 Resources Selectors

Events of a teacher represents the view of one teacher and returns whose assigned Events. The second resource selector $Student \times Subject$ is a little bit more complex. Its view is a tuple of a student and a set of subjects *subjects*. If the attribute *type* is '*group*' the tuples will be generated with a set of all resources of the group. An example is the group of all main subjects *mainSubjects*. So tuples between all students of the collection *Student_ALL* ('*elem*') and the collection *mainSubjects* ('*group*') can be generated. The return values are the events to those the student and at least one subject of *subjects* is assigned to. IS is selected because the *unionType* is set to *Intersection*.

A resource variable rv is based on a resource selector rs_{rv}. It can be expanded by an event variable ev_{rv} and an operation list ol_{rv}. Every resource variable rv_{rc} maps a tuple $t \in scope_{rc}$ to a value $rv_{rc}(t)$. This value is the left operand of $viol_{rc}(t)$.

$$EV_{rv}(x) = \begin{cases} ev_{rv}(x) & \text{if } ev_{rv} \text{ is set} \\ x & \text{else} \end{cases} \tag{18}$$

$$OL_{rv}(x) = \begin{cases} ol_{rv}(x) & \text{if } ol_{rv} \text{ is set} \\ x & \text{else} \end{cases} \tag{19}$$

$$rv_{rc}(t) = OL_{rv}(EV_{rv}(rs_{rv}(t))) \tag{20}$$

```
<Constraints>
  <OperationLists>
    <OperationList name="Duration"> ... </OperationList>
  </OperationLists>
  <Selectors>
    <EventSelector name="timeSlotsSelector">
      <TupleValue type="TimeSlot" attribute="slot"/>
    </EventSelector>
    ...
  </Selectors>
  <Variables>
    <EventVariable name="timeSlots">
      <EventSelectorRef name="timeSlotsSelector"/>
    </EventVariable>
    <ResourceVariable name="Workload of Teacher">
      <ResourceSelectorRef name="Events of a Teacher"/>
      <OperationListRef name="Duration"/>
      <EventVariableRef name="timeSlots"/>
    </ResourceVariable>
  </Variables>
```

Listing 14 Variables of Work Load of a Teacher

The event variable *timeSlots* maps the list of events given by the resource selector *Events of a Teacher* to a list of time slots. The operation list *Duration* computes the sum of durations of these time slots. This is the result of the evaluation of the resource variable *Workload of Teacher*.

A resource constraint *rc* consists of a relational operator *rop*, a resource variable *rv* as left operand and a right operand *right*. The scope of the constraint is defined by rs_{rv}. The violation count function $viol_{rc}(s, tuple)$ is defined by the following equation:

$$viol_{rc}(tuple) = rop(rv(tuple), right(tuple)); \tag{21}$$

$$q_{rc}(s) = \sum_{tuple \in scope_{rc}} w_{rc}(viol_{rc}(s, tuple)) \tag{22}$$

The operand on the right hand side of a resource constraint can be a constant value. This value has one of the types which are available for the definition of attributes. In some cases it is necessary to refer to an attribute value *attr* of a resource *r* of the tuple *tuple*.

```
<ResourceConstraint relOpDenom="DISTTIME">
  <ConstraintProperties name="Work load of a teacher" type="soft">
    <PenaltyFunction weight="50.0" coefA="1.0" coefB="2.0" exponent="1.0"/>
    <Description>A teacher should work the given work load </Description>
  </ConstraintProperties>
  <ResourceVariableRef name="workloadOfTeacher"/>
  <ResourceOperand>
    <TupleValue type="Teacher" attribute="workload"/>
  </ResourceOperand>
</ResourceConstraint>
```

Listing 15 Work Load of a Teacher

4.11 Event Relation Constraints

In some cases the view of one event or one resource tuple is not sufficient to express all constraints involved. An example is the Same Resource Constraint which is violated if for a certain resource type *rt* $s(e_1, rt) \neq s(e_2, rt)$. This relation is postulated for a the resource type *rt* and a set of events E_{rel} with $e_1 \in E_{rel} \wedge e_2 \in E_{rel}$. An event relation constraint *erc* sets two collections of events E_{left} and E_{right} in a relation. The scope of this constraint is $scope_{erc} = E_{left} \times E_{right}$. So the owner of a constraint violation is a tuple of events $s(e_l, e_r)\, with\, e_l \in E_{left} \wedge e_r \in E_{right}$. The parameters of the violation count function $viol_{erc}$ are this tuple and the solution *s*. As said before the tuple consists of two parts e_l and e_r. For each of this parts a reference to an event variable ev_{left} resp. ev_{right} is defined. And a relational operator rop_{erc} must be defined.

For each event of this tuple must be referenced event variables ev_{left} and ev_{right} and a relational operator rop_{erc} must be set.

$$viol_{erc}((e_l, e_r)) = rop_{erc}(ev_{left}(e_l), ev_{right}(e_r)) \tag{23}$$

```
<EventRelationsConstraint relOpDenom="SET_EQUAL">
  <ConstraintProperties name="Sports 1A at Same Time" type="soft">
    <PenaltyFunction weight="10.0" coefA="1.0" coefB="2.0" exponent="1.0"/>
    <Description>The sports lessons of the class 1A should be at the same
        time</Description>
  </ConstraintProperties>
  <EventRelationOperand>
   <EventVariableRef name="timeSlots"/>
   <CollectionRef name="Sports Male 1A"/>
  <EventRelationOperand>
  <EventRelationOperand>
   <EventVariableRef name="timeSlots"/>
   <CollectionRef name="Sports Female 1A"/>
  <EventRelationOperand>
</EventRelationsConstraint>
```

Listing 16 Event Relation Constraint: Same Time

In the example above the left operand returns the assigned time slots of the male sports lessons, the right operand the time slots of the female sports lessons. The relational operator *SET_EQUAL* compares this two lists and returns 0 if there is no difference and a value $x > 0$ otherwise. Another possibility is to declare two events at the same time to link the resource lists of these events.

4.12 Operation Lists

In the sections 11 and 14 we used operation lists without defining this entity. An operation list ol is a sequence of operations $(op_1, ..., op_n)$.

$$\begin{gathered} ol(x) = op_n(...(op_2(op_1(x)) \\ domain(ol) = domain(op_1) \wedge codomain(ol) = codomain(op_n) \\ codomain(op_i) \subset domain(op_{i+1}) \quad \forall\, i \in [1; n-1] \end{gathered} \tag{24}$$

The operations are defined in a XML file. This file of definitions can be used by an editor to create a selection of available operations. The definition of operations must be expressed in a programming language as part of the code of the solver tool. At this point the designer of a timetabling system has to decide whether be would prefer to modify the solver program code for performance reasons or to interpret the specification and to preserve the flexibility. The same approach is used for defining relational operators.

5 EATTS XML File Format for Results

5.1 Requirements to a Result Format

In section 4 we have described the EATTS format to define different timetabling problems. The purpose of planning is to generate a time table of high quality, i.e. with low costs. After a couple of runs of the planning framework there is a list of solutions. Then these results must be compared. In some cases this comparison could be based on the absolute costs of the solution, but in other cases it must be a bit more precise. So a list of constraint violations and their owner would be nice. Let *ec* be an event constraint and $viol(e, s) > 0$. So *ec* is violated on the view of the event *e*. So there is a constraint violation of *ec* with owner *e*. The owner of a resource constraint violation is a resource tuple. The owner of an event relation constraint violation is a tuple of events.

A result file should contain information about the plan it is linked to, certainly the assignments of the solution, the absolute costs and the constraint violations and its owners. But there is another problem. If a plan is accepted it will be used by many people. These users are interested in various views of this plan, but they are usually not interested in the problem definition. So the result file should include all data to generate all the required views on the plan.

5.2 The Structure of EATTS Result Format

The format is also XML based, because of the benefits shown in section 3.

```
<TTResult>
  <Plan/>
  <Events>...</Events>
  <TimeModel>...</TimeModel>
  <Resources>...</Resources>
  <ConstraintViolations>...</ConstraintViolations>
</TTResult>
```

Listing 17 Structure of EATTS Result Format

The `Plan` stores a reference to the problem definition file. A list of all events with name and index is enclosed in the `Events` tag. For visualization it is necessary to have some informations regarding the time model. The common style of timetable visualization is a table. This table has *cycleLength* columns and *r* rows. Every timeslot must be assigned to one cell or a set of cells in this table. So every time slot resource gets a day-index which defines the column, and vertical start and end index, which set the rows. `Resources` includes all resources sorted by resource types. Every `Resource` has a *name* and an *index* attribute and contains a list of the events it is assigned to.

The total costs of the solution is the value of the attribute *totalCost* of the `ConstraintViolations`. This tag includes a list of the constraint violations sorted by constraints in their order of appearance in the XML file. The `ConstraintViolation` tag has information about its cost and its owner.

6 Conversion of High School Timetabling XML into EATTS XML Format

6.1 Conversion of Time, Resources, and Events

In this section we want to show how a problem given in the existing format for High School Timetabling can be described in the EATTS format. `TimeGroups`, including `Week` and `Day`, can be converted into resource groups of the type *TimeSlot*. Resources of this type must be generated automatically so that there is a 1:1 mapping between `Time` elements *telem* and the resources of the type *TimeSlot timeSlot*. As example *Mon_1* will be mapped to a time resource with start time 8 *hours* and duration 1 *lesson*, whereby the start of the planning period is $Monday\ 0:00$ and the duration of one lesson must be set. The membership to time groups of *telem* must be adopted to the corresponding resource groups of type *TimeSlot*.

The concept of resource types, resources and resource groups is very similar in both formats so that a conversion is possible without a hitch. It should be considered that the names of resource types and resource groups in EATTS format must be unique. The name of the resources will be stored in attribute *name* of type *String*.

The conversion of courses and events is a little bit more difficult. In the EATTS concept a course consists of one event or a set of events with their linked time slot resources. In some cases the accumulated working time may differ from the time which should be credited to the work load account, i.e. a premium, a *Properties* resource type with an attribute *workCredit* and a property resource must be assigned fixed to every event. If a class is split into two or more courses, like for sports, for each of these courses a group of the participating students must be declared. If these courses should be at the same time an event relation constraint can be added or the *TimeSlot* resource lists can be linked to ensure that they are assigned to the same time slots.

6.2 Conversion of Constraints

Now will be shown how the constraints can be defined with the concept of EATTS format. There is a list of constraints in the paper [PAD+08, p. 10f]. *AssignTimSlot-Constraint* and *AssignResourceConstraint* can be converted to a `MinMaxConstraint`. The *LinkedEventsConstraint* and the *AvoidSplitAssigmnetsConstraint* can be matched to an event relation constraint, as shown in section 4.11, or the according resource lists can be coupled.

NoResourceClashConstraint is a resource constraint, for every tuple of resources must the count of events, which are assigned to all resources of the tuple, less or equals 1. For the constraints *WorkloadAssignment*, *IdleTimesConstraint* and *Time-SlotAmountConstraint* an operation must be applied to the list of assigned time slots of a certain resource. This operation must compute the duration of these time slots or the count or must measure the idle times.

The selector of a *SubjectSequenceConstraint* selects the assigned events to a combination of a class or student and a collection of subjects. The resource variable selects the time slots of these events and applies an operation on the time slots list which measures the sequences. The $viol(class, subjectCollection)$ function can compare the result of the operation with the allowed sequence length. The *ClusterTimeSlotCon-straint* is similar, the count of events assigned to a resource and a collection of Time-Slots should lie between minimum and maximum. The *CourseSpreadingConstraint* can also be matched to a resource constraint by selecting the event groups over the fixed assigned property resources. In that case there must be a 1:1 relation between property resource and event.

6.3 Conversion of Solutions

The conversion of the solutions needs a framework which computes the constraint violations. In the first step the problem must be read. Then the assignments must be set according to the solution. Now the framework can compute the total cost and the constraint violations. In the last step the data can be written to the solutions file in EATTS format.

6.4 Conversion of EATTS XML format into High School Timetabling XML Format

This way of conversion would in most cases be senseless, at least if the statement of the problem is not concerned with High School Timetabling. But if there is a well defined problem definition for High School Timetabling like the result of the conversion above, an inversion of this process is possible. Let 2^{HST} be the set of all timetabling problems which can be defined with the High School Timetabling format and 2^{EATTS} the set of all timetabling problems which can be described with the EATTS format. Above we have shown that there is a conversion $C : 2^{HST} \to 2^{EATTS}$. Let $Codom(2^{HST})$ be the codomain of 2^{HST} in 2^{EATTS}, so $Codom(2^{HST}) \subsetneq 2^{EATTS}$. The inverse function C' is defined by the following equation.

$$C' : Codom(2^{HST}) \to 2^{HST}; C'(C(hst)) = hst \quad \forall\, hst \in 2^{HST} \tag{25}$$

7 EATTS XML File Format Tools

The EATTS framework provides a set of tools to handle XML files defined in the format described above. First of all there is a web based editor. It can be used to create new timetabling problems, the input mask will be automatically adapted to the user defined resource types with different attributes. There is also a tool, called plan viewer, to visualize timetables from various views like the view of an event or of a resource. Often the generated plans should be changed manually after planning. So an application would be nice which gives support on changing the assignments, whereby the costs and constraint violations must be updated just in time. The work on this will be finished soon. At least the core of an automatic timetabling framework must be a solver for generating as good as possible timetables. Our solver is written in JAVA and provides classes to read and write the EATTS XML format and a set of optimization algorithms which can be applied to all timetabling problems defined by the XML. So the framework can be used to compare different algorithms on many various timetabling problems.

8 Conclusion

Our approach is to define different types of timetabling problems with one format. The existing formats are not flexible enough, because they can only describe timetabling problems of one class. So we declared the EATTS XML format with regard to the possibility of describing as much as possible problems.

Some very special constraints, like counting idle times, are not easy compatible with the EATTS general approach of constraint definition. In these cases must be implemented very special operations in the solver. Often these operations can be used for other problems, too. The XML definition of some constraints could be very inconvenient and need many tags, especially in case of event relation constraints. But this should not be a big problem if the problem definition is automatically generated.

The main future work is to describe many different real world problems with our format. On this way we will adapt the XML format and define a set of operations. Then we want to categorize these problems and test the solver. We think a group of types will need a special set of operators, like neighborhood or genetic operators, to get good time tables. We want to have a special look on reusability and standardization. We like to get problem definitions and suggestions at any time.

Acknowledgements We would like to thank Eugen Kremer for his participation in the EATTS project and especially the XML file format.

References

[GWB03] Matthias Gröbner, Peter Wilke, and Stefan Büttcher. A standard framework for timetabling problems. In *Practice and Theory of AutomatedTimetabling IV*, volume 2740 of *Lecture Notes in Computer Science*, pages 24–38. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-40699-0. ISSN 0302-9743 (Print) 1611-3349 (Online). URL `http://www.springerlink.com/content/2mj0rwlpbp5uvh1v/`.

[PAD+08] Gerhard Post, Samad Ahmadi, Sophia Daskalaki, Jeffrey H. Kingston, Jari Kyngas, Cimmo Nurmi, David Ranson, and Henri Ruizenaar. An xml format for benchmarks in high school timetabling. In *Proceeding of the 7th international conference on the Practice and Theory of Automated Timetabling*. Patat2008, 2008. URL `http://w1.cirrelt.ca/~patat2008/PATAT_7_PROCEEDINGS/Papers/Post-WD2a.p% df`.

[W3C10a] W3C®, January 2010. `http://www.w3.org/XML/Schema`.

[W3C10b] W3C®, January 2010. `http://www.w3.org/XML/`.

[wik10] wikipedia, January 2010. `http://en.wikipedia.org/wiki/Document_Type_Definition`.