



The University of Sydney  
Basser Department of Computer Science

**A STANDARD DATA FORMAT FOR  
TIMETABLING INSTANCES**

*BY EDMUND K. BURKE & JEFFREY H. KINGSTON*

**TECHNICAL REPORT 512**

**FEBRUARY 1997**

**BASSER DEPARTMENT OF COMPUTER SCIENCE**

**UNIVERSITY OF SYDNEY**

Basser Department of Computer Science  
University of Sydney  
Sydney N.S.W. 2006  
Australia

# A Standard Data Format for Timetabling Instances

*Edmund K. Burke*  
(*ekb@cs.nott.ac.uk*)

*Paul A. Pepper*  
(*pap@cs.nott.ac.uk*)

*Jeffrey H. Kingston*  
(*jeff@cs.usyd.edu.au*)

Department of Computer Science  
University of Nottingham  
UK

Basser Department of Computer Science  
University of Sydney  
Australia

## Abstract

Many formats have been developed for representing timetable data instances. The variety of data formats currently in use makes the comparison of research results and exchange of data extremely difficult. As many researchers have observed, the need for a standard data format for timetabling instances is now becoming urgent, so as to allow the sharing of data and testing of algorithms on standard benchmarks. At present this is not possible in research on timetable construction.

In this paper we state the general requirements of a standard data format and present a standard data format and language for use in the evaluation of timetable instances.

## 1. Introduction

Research into computerised timetabling has been carried out since the early years of the computer age. A review of the vast literature on this subject [1, 8, 9, 10, 11] quickly reveals that most of the work was undertaken in ignorance of the work of others, and cases where one researcher has produced results directly comparable with the results of another are extremely rare. Consequently, it is difficult to evaluate the contributions made by practically oriented researchers.

Recently, the advent of a series of conferences devoted to computerised timetabling [2, 3] has brought the timetabling research community together and stimulated much fertile exchange of ideas. There has also been some exchange of university examination timetabling data (e.g.[4]) between different institutions in different countries. But the problem of evaluating practical contributions remains, and will do so until the timetabling systems of different researchers are routinely tested on common data.

Other research communities have solved this problem of sharing data. For example, research into the travelling salesperson problem has been facilitated for many years now by TSPLIB [5], a library of instances of the travelling salesperson problem which is freely available over the internet and universally used as test data for programs which attempt to solve this problem.

A prerequisite for any such library is a standard data format with an agreed meaning. Of course, instances of the travelling salesperson problem are very simply described, as graphs whose edges have numeric weights. Real-world instances of the timetabling problem are much more complex. There are many different constraints, their precise nature varies subtly,

and they are seldom stated explicitly in data files. Describing these constraints is a difficult problem.

The only previous proposal for a standard data format for timetabling known to us is a discussion paper presented by Andrew Cumming at the ICPTAT'95 conference, but not submitted formally to the conference or printed in its proceedings. That proposal was criticised for lack of generality, but it was valuable in stimulating a discussion during which the full difficulty of the problem became apparent.

In this paper we propose a standard data format for instances of the timetabling problem. Section 2 presents the requirements that such a format must satisfy, and Section 3 presents the proposed format itself. Like all standards, it will be successful only if it meets the needs of its users, and accordingly the authors welcome all comments and suggestions.

## 2. Requirements

This section presents the requirements that have guided our design of a standard data format for timetabling.

The first requirement is generality. It must be possible to express any reasonable instance in the format, for otherwise it will be unacceptable to those whose instances are excluded. It is plainly impossible to enumerate every constraint that researchers have encountered or will encounter. Instead, we propose a format based on set theory and logic, in which any computable function may be defined.

The second requirement is that it must be possible to express instances completely. This includes straightforward data expressing the resources and meetings, complex logical constraints, and proposed solutions. It also includes the criteria by which solutions are to be judged: the hard and soft constraints, and the weighting of the soft constraints, must be explicitly given in the instance itself. Otherwise there will be no agreement on the relative quality of proposed solutions.

This requirement offers an opportunity that we intend to exploit. It will be possible to evaluate any proposed solution against the criteria laid down in the instance. We propose to make freely available a computer program to do this, and thus to establish an independent means of evaluating and comparing proposed solutions.

The third requirement is that translation each way between the standard format and the many existing formats used by researchers must be practicable.

The complex set theory and logic expressions required to express constraints precisely cause a double difficulty here. Not only are they themselves non-trivial to parse and translate, they will in general have no analogues in the data formats of individual researchers, since, as remarked earlier, such constraints are rarely made explicit.

Both difficulties may be overcome in practice by a format which permits all the set theory and logic expressions to be confined to library files. An instance file may then begin with an include directive referring to such a library file, and serving as the definition of all the complex constraints, and then proceed to give the more straightforward data describing the resources and meetings of the instance. A timetabling system whose built-in assumptions conform to the definitions in the library file need never read that file.

The types of problem tackled by researchers are often so different that interchange of data is inherently impossible. These distinct types of problem will need distinct, incompatible library files. But we are hopeful that data exchange will be possible between researchers whose types of problem, while not exactly identical, are sufficiently similar; and even without data exchange, the advantages of precise definition and independent evaluation of solutions will remain.

### 3. Proposed Standard Format

In this section we give an outline of the standard data format, including a description of the data types, keywords and the grammatical form of the language. We also outline some further facilities which we intend developing, for use with the language, to simplify the task of the data conversion and testing. Amongst our discussion of the language we demonstrate the composition of some timetabling constraints using our language.

Researchers familiar with the Z specification language [6] will notice that some of the concepts and constructs which we use are similar to those found in Z. This is especially true of the way in which Z uses sets since we felt that it meets our requirements very well. We would therefore like to acknowledge the designers of the Z specification language.

#### 3.1 Outline of the Format

The language which we propose is functional in nature, allowing us to express the timetable problem, in its mathematical form, as simply as possible. Data types catered for in the format include the following; class, function, set, sequence, integer, float, boolean, char, and string.

For the data types integer, float, boolean, char and string the usual operations are defined (e.g. addition, subtraction, multiplication and division for the integer data type).

Classes are provided to allow new abstract data types to be created. Variables are placed within the body of a class along with functions (which are likely to be forms of constraint) and should be provided to operate on those variables. The following is a simple example which we shall use to illustrate the class construct within the language.

```
class Meeting
{
  Students      : set of student;
  Room          : room;
  TimeSlot     : time;

  function SufficientSeating() : boolean;
};
```

Here we see that the data types `student`, `room` and `time`, which would also have been defined using the class construct, are aggregated within a new abstract data type `Meeting`. We could now go on to construct an instance of `Meeting` in the following manner:

```
Maths1 : Meeting;
Maths1.Students = {Greaves, Watt, Dickens};
```

Notice that the `.` (dot) operator is used to access the components within the `Maths1` object. In this case no values are assigned to the `Room` and `TimeSlot` components since these would be used to hold the values which determine a solution to the timetable problem instance (this would be carried out by some form of timetabling application).

Alternatively we could have assigned values to the component objects within the `Maths1` object in a single expression using square brackets to contain the assignments:

```
Maths1[ Students = {Greaves, Watt, Dickens},
        Room = ?,
        TimeSlot = ? ];
```

The ? (question mark) operator is used to indicate that no value has yet been assigned to the Room and TimeSlot variables.

Although the example we have chosen to use is a simplification of how a real meeting might be modelled it would be possible to extend our example definition using single inheritance. This would involve using the `inherit` keyword in the following manner:

```
class ExamMeeting inherit Meeting
{
    ExamId : integer;
}
```

The inheritance mechanism will provide an important feature in allowing standard definitions within the format to be extended where necessary. All standard definitions using the `class` construct will be placed within library files - a standard meeting class would be an obvious candidate for inclusion within the libraries. These will then be included by a timetabling instance where the `inherit` keyword may be used to extend any of the classes contained therein.

Our new abstract data type may also require new member functions and the prototypes for these functions could also have been added to our definition of the `ExamMeeting` using the notation shown in our `Meeting` for the function `SufficientSeating()`. The prototype in this case tells us that no parameters are passed in to the function and that its evaluation will result in a boolean. The implementation of `SufficientSeating` would use the following syntax:

```
function SufficientSeating() : boolean
{
    Room.Capacity >= Count(Students);
}
```

The main body of the function is placed between braces, in the case of `SufficientSeating()` above the returned value will be true if the capacity of `Room` is greater than or equal to the number of elements within the set `Students`. If parameters are to be passed into a function then these should appear between the parentheses placed after the function name. If an integer parameter is passed in to the function then it would take the following form:

```
function SufficientSeating( Unavailable : integer ) : boolean
```

Where multiple function parameters are used, they should be separated by commas and if a reference to a member function's calling object instance is required then the `self` keyword should be used.

Sets will be another very useful data type within the language since many of the components that timetabling instances deal with involve groups of objects (groups of students representing, say, classes of students, groups of classes, etc.) and viewing these groups as sets allows us to express many of the constraints encountered within timetabling problems clearly and simply.

Sets within the language are defined in their conventional mathematical form whereby all the elements of any given set are unique and the usual operations on sets of objects are defined - we also treat internal data types as sets, the member elements of which would be their literal values. We do, however, restrict any object of type `set` to contain objects of only one data type.

In addition to the commonly encountered set operations - `member`, `union`, `intersect`, `subset`, `-` (set exclusion) and `count` - we also include the operators `forall`, `exists`, `sum` and `prod`. The `forall` and `exists` operators have the same meaning as their abstract mathematical operators  $\forall$  and  $\exists$ , respectively, whilst the `sum` and `prod` operators are used to evaluate the sum and product, respectively, of numeric values within a set. An example implementation of the *No-Clashes* constraint shows the syntax of the `forall` operator (the syntax of the `exists` operator is defined in exactly the same format) and illustrates its expressive power.

```
function NoClashes( Meetings : set of Meeting ) : boolean
{
  forall m1 in Meetings
  (
    forall m2 in Meetings | m1 != m2
    (
      ( m1.Time intersect m2.Time == {} ) or
      ( m1.Resource intersect m2.Resource == {} )
    );
  );
}
```

In this example the variable `m1` is bound within the scope of the first instance of the `forall` operator (i.e. between the operators associated pair of parentheses) and similarly for the variable `m2` though this is also bound within the expression following the vertical bar. The vertical bar is optionally used to restrict the set of elements over which `forall` will operate and in the above example we have used it to restrict comparisons between non-identical meetings. If we now look at the abstract mathematical form of the `NoClashes` function we can see that the transformation to an implementation can be achieved relatively easily.

$$\begin{aligned} \text{NoClashes}(\textit{Meetings}) = & \\ & \forall m_1 \in \textit{Meetings} \text{ and } \textit{time}_1, \textit{resource}_1 \in m_1, \\ & \forall m_2 \in \textit{Meetings} \text{ and } \textit{time}_2, \textit{resource}_2 \in m_2, \\ & m_1 \neq m_2 \Rightarrow (\textit{time}_1 \cap \textit{time}_2 = \emptyset) \vee (\textit{resource}_1 \cap \textit{resource}_2 = \emptyset) \end{aligned}$$

The syntax of the `sum` and `prod` operators is defined in a similar manner to `forall` and `exists`. As an example usage of the `sum` operator we may obtain the sum value of all soft constraints imposed by a timetabling instance as follows:

```
sum m in Meetings (m.Soft())
```

Here `Meetings` is the set of all meetings and `Soft()` is a member function defined within the class definition of a meeting. The type system would guarantee that `Soft()` is defined for all instances of the meeting class and this could then be used to evaluate any associated soft constraints, as shown in this example.

Difficulties can arise when defining equality and equivalence (identity) within a language. The implementation of the `NoClashes` function above provides an example for us to illustrate the rules which we define to distinguish between the equivalence and the equality of variables.

When two variables of a user defined type (the type being created using the `class` feature) are compared using either of the operators `==` or `!=`, we define the operation to be a

comparison of equivalence. Although reassignment is not permitted we use the `==` operator to avoid any confusion with the assignment operator, `=`. In the `NoClashes` example above a comparison of the identities of the variables `m1` and `m2` is carried out. If a programmer wishes to compare the contents of two user defined objects, of the same type, then he should define a comparison function within the body of the class definition to carry this out for him. It is only when an object instance of a user defined type is compared with a literal value that the contents of that object are compared. This kind of comparison can be seen in the `NoClashes` implementation. An object of type `set` of `Time` is compared with the empty set literal value, `{}`, using the `==` operator. The comparison of object instances of built in types (such as `integer`, `float`, `char`, etc.) using the operators `==` and `!=` will involve the comparison of the contents of those instances (i.e. checking the equality/inequality of the objects).

The `seq` data type is used to model sequences within the language. It is similar to the `set` data type in that most of the valid operations for `set` objects have their analogue for `seq` objects. The usual exceptions to this would be the `head` and `tail` sequence operators which are included in the language. The literal values of the `seq` data type may be expressed in the same format as `set` literal values. Chevrons are used to delimit the `seq` literal value where braces would be used to delimit the literal values of the `set` data type.

In order to express conditionals we use the `if` construct and illustrate its syntax as follows:

```
if count Maths1.Students > Rooms.MainHall then
  // Take action
else
  // Take alternative action
end
```

We also include the `let` keyword in the language to allow commonly used expressions to be abbreviated by a user defined synonym within an expression. The syntax of this will be defined as shown in the following simple example:

```
let Penalty = (Meeting.Room.SeatingCapacity - Count(Meeting.Students)) *
0.1
in
  // Penalty is now used within this block as a variable
end
```

As can be seen in many of the examples provided within this section the double forwards slash, `//`, is used to indicate the start position of a comment on a single line.

### 3.2 Other Facilities

In addition to the language and standard data format we will also develop a library of standard functions. These will consist of the most commonly encountered constraints found in timetabling and general utility functions likely to be used within timetabling. [7] provides a good starting point for the selection of such constraints. All the standard functions will be implemented using the language and will comply with the standard format. The `No-Clashes` constraint, which was used as an example in the previous section, would be one such constraint to have its implementation appear in the library. Examples of other constraints which might also appear in the library would include `Sufficient-Seating` (a hard constraint which states that there should be sufficient seats available within a given set of rooms for a meeting or a selected number of meetings), `Share-Period` (a constraint stating that a given set

of meetings must share the same time period), Restrict-To-Periods (constraining a meeting, or set of meetings, to a given set of time periods).

Programs will be necessary to convert data in a non-standard format into the standard data format. In order to encourage the development of consistent conversion programs we will publish a standard interface. This might then be used to interface with a common conversion application which would be used as the standard front-end to the conversion utility. Although there is no necessity to make conversion programs publicly available we would encourage the contribution of data expressed in the standard format and would hope to provide assistance with the conversion process where required.

The conversion of data into the standard data format will allow us to build up a pool of timetabling data, incorporating the variety of constraints encountered within this area. A large range of data may (eventually) be selected from this pool and converted into a researcher's own format for use in testing the effectiveness and robustness of their methods of timetable solution.

Use of a standard data format will also facilitate objective testing and comparison of results between researchers using different data formats. Data would be taken from the data pool and converted into a researcher's own format for use in validating their timetabling methods. Subsequent conversion back to the standard data format would allow objective comparison of the results of the researcher's methods against other solutions. We aim to develop benchmarking and evaluation applications to assist and promote the consistency of such testing.

#### **4. Conclusion**

We hope that our work will promote an open exchange of timetabling data between researchers and the objective comparison of research results (the exchange process will be simplified once conversion programs are written). Just as the success of the Travelling Salesman Problem Library has shown, these objectives are achievable.

We will be looking to place all the facilities outlined in this paper on the internet once they are available. Allowing open access to our work, via the development of a timetabling library internet site, will be important if we are to promote the exchange of ideas with respect to the solution of timetabling problems.

Many formats used by timetabling applications will have been designed with concerns for minimising data storage requirements or to facilitate the fast processing of data. This means that such formats would not be suitable as a standard format since generality or readability of data (by humans) would be forfeited. Our aim, however, is to provide generality; allowing the maximum number of data instances to be represented as is practicable. Since we do not expect our format to be used internally by timetabling applications this aim can be met.

An initial investment in time will be necessary from potential beneficiaries of the standard data format. This and the initial delay in the development of inertia in the growth of the timetable data pool may cause a hesitation in some to become involved, though we think the benefits to be gained will encourage researchers sufficiently.

Suggestions regarding the development and improvement of the standard data format and associated facilities are welcomed by the authors. Moreover, a great deal of the success of the standard data format is reliant upon feedback from those who will make use of it and gain the resulting benefits (e-mail may be addressed to any of the authors with the text "TTLIB comments" placed in the *subject* header section).

## 5. References

- [1] Jeffrey H. Kingston, *A Bibliography of Timetabling Papers*, URL "<ftp://ftp.cs.su.oz.au/jeff/timetabling/>".
- [2] E.K. Burke and P. Ross (eds.), *The Practice and Theory of Automated Timetabling: Selected papers from the First International Conference on The Practice and Theory of Automated Timetabling*, Lecture Notes in Computer Science, Vol. 1153, Springer 1996.
- [3] Proceedings of the 2<sup>nd</sup> International Conference on the Practice and Theory of Automated Timetabling, Toronto, August 1997, to be published by Springer.
- [4] E.K. Burke, J.P. Newall, R.F. Weare, *A Memetic Algorithm for University Exam Timetabling*, in [2], pp. 241-250.
- [5] Bob Bixby and Gerd Reinelt, *TSLIB - A Library of Travelling Salesman and Related Problem Instances*, September 1992. URL "<http://nhse.cs.rice.edu/softlib/catalog/tslib.html>".
- [6] Ben Potter, et. al., *An Introduction to Formal Specification and Z*, Prentice Hall, 1991
- [7] E.K. Burke, et. al., *Examination Timetabling in British Universities - A Survey*, in [2], pp. 76-90.
- [8] M.W. Carter and G. Laporte, *Recent Developments in Practical Examination Timetabling*, in [2], pp. 3-21.
- [9] V.A. Bardadym, *Computer-Aided Lessons Timetables Construction. A Survey*. USIM (Management Systems and Computers, 8, pp. 119-126, 1991.
- [10] M.W. Carter, *A Survey of Practical Applications of Examination Timetabling Algorithms*, in Operations Research, 34, pp. 193-202, 1986.
- [11] W. Junginger, *Timetabling in Germany - a Survey*, Interfaces, 16, pp. 66-74, 1986.

**ISBN 1 86451 280 6**