An Extensible Modelling Framework for the Examination Timetabling Problem

David Ranson[1] and Samad Ahmadi[2]

[1] Representational Systems Lab, Department of Informatics,
University of Sussex, Falmer, UK
d.j.ranson@sussex.ac.uk
[2] School of Computing, De Montfort University,
The Gateway, Leicester, LE1 9BH, UK
sahmadi@dmu.ac.uk

Abstract. A number of modelling languages for timetabling have been proposed to standardise the specification of problems, solutions and their data formats. These languages have not been adopted as standard due to not simplifying the modelling process, lack of features and offering little advantage over traditional programming languages. In contrast to this approach we propose a new language-independent modelling framework for general timetabling problems based on our experience of modelling the examination timetabling problem (ETP) using STTL. This framework is a work in progress but demonstrates the possibilities and convenience such a model would afford.

1 Introduction

In this paper, the rationale for proposing a new modelling framework for the ETP is discussed in relation to existing languages designed for timetabling. The timetabling problem itself is described followed by a brief survey of the existing languages. A model for the ETP in STTL is presented as a case study, examining some of the underlying problems with the existing approaches. A standard model for timetabling is then presented which addresses some of these issues.

Timetabling can be described as the general problem of "sequencing events subject to various constraints" [1]. This is typically, as the name suggests, assigning timeslots to events in order to create a feasible solution for a given problem. This is a complex task and the general timetabling problem is known to be NP-Hard.

Examination timetabling problem (ETP) is a significant special case of the general timetabling problem. Production of exam timetables is a practical challenge faced by almost all academic institutions on at least one occasion every year. The most important characteristics of the exam timetabling problem are the constraints that describe the problem.

The most important constraint violation for the ETP is the "clash" (or first degree student conflict) constraint which states that a student cannot be timetabled to sit more than one exam at the same time. This is an example of a hard constraint as it may not

E. K. Burke, H. Rudová (Eds.): PATAT 2006, pp. 281–292. ISBN 80-210-3726-1.

be violated in finding a feasible solution. Other examples of hard constraints are duration and room capacity constraints; e.g. exams cannot be scheduled into time periods with durations shorter than that of the exam.

The "consecutive exams" constraint is an example of a *soft* constraint. A violation of this constraint exists when a student is timetabled to sit more than one exam in immediate succession. This constraint exists in most instances of the exam timetabling problem, but may not be universal. Institutions may also add their own unique constraints such as not mixing language exams on the same day[2]. As different institutions use very different constraints it is hard to generalize the problem in such a way that it is applicable to all cases. Any universal model for the ETP must therefore have some flexibility in the constraints which are specified.

The goal in exam timetabling is to minimize the number of violations of these constraints over a solution. Normally a cost is assigned to each type of constraint, with the hard constraints having much higher associated costs than the soft constraints. The total cost for a solution is then given as the sum of the costs for all the violations found.

There are many different and varying approaches to solving the exam timetabling problem being used at institutions and by researchers. A recent survey shows that these approaches include Sequential methods, Clustering approaches, Case-based reasoning and a number of Heuristic approaches[3]. This wide variety of the algorithms and software applications use different models and data formats adding to the cost of implementation due to handling of the model and the data.

The data published by Carter [4] (and other publicly available data) has been used for some benchmarking but can relate to instances of the problem over a decade old since when many Universities have seen expansion in their numbers of students and courses, especially modular courses where students take exams from many different departments.

The need for a modelling standard and a standard data format has been recognised for some time and the requirements of such a standard have been discussed in detail [5]. These properties include generality, completeness, and easy translation with existing formats.

It is the authors' belief that other research areas where standard formats have become the norm have benefited from increased corporation between researchers and better benchmarking resources have lead to advances in research. Examples of this in practice are the Travelling Salesman Problem (TSPLIB) [6, 7] and the MPL (Mathematical programming language), MPS (Mathematical programming standard.

2 Progress Towards a Standard Format

There have been at least three attempts at creating modelling languages and standard data formats for timetabling problems since the proposal by Burke, Kingston and Pepper in 1998 [5]. These are the: Standard TimeTabling Language (STTL) [8, 9], TimeTabling Mark-up Language (TTML)[10] and UniLang [11].

STTL is a complete object oriented functional language designed to be suitable for modelling timetabling problems using set theory. STTL specifies the problem being modelled as well as the evaluation function for the model, instance data and solutions.

TTML is based on MathML which is an XML application for modelling maths formulae. The goal was to create a language with the functionality of STTL but using the fashionable XML. The TTML learning curve is steeper than that of STTL and again seems overly complicated, especially for specifying the complex logic involved in these problems.

UniLang is another language with similar aims to STTL. It attempts to be a simple language easily understandable by humans as well as machines, modelling the problem by identifying subclasses of the problem and using this to guide their design. In the first aim it has largely been superseded by languages such as XML. Whilst demonstrated to be capable for its purpose, UniLang does not seem as expressive as STTL or TTML.

We are unaware of any of these data formats, or any other format, being used to share timetabling data. The known exception to this is the publicly available datasets on the University of Melbourne Timetabling Problem Database website[4].

Perhaps, the main reason these languages have not been adopted as standard is that they offer no advantages to the user over any traditional programming language. These idealistic languages do not simplify the modelling process, and can even be restrictive in that they do not have all the features of a modern programming language, are overly complicated or appear cumbersome.

3 A Case Study: Modelling the Exam Timetabling Problem in STTL

An STTL model for the exam timetabling problem has been created and used as the data format for a working application[12, 13]. This model was based on the model Kingston [8] has created for the High School timetabling problem.

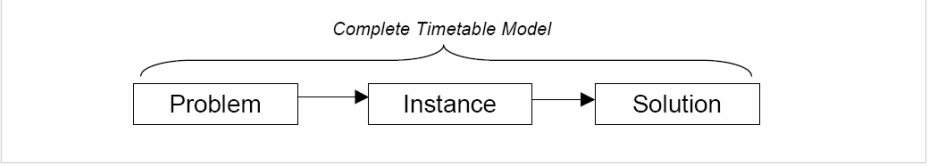

Fig. 1. The components of an STTL Model. A complete model is made up of the Problem, Instances of that problem and finally Solutions for the Instances

Each STTL problem is made up of three components, normally split into three different files. As its name suggests the Problem file contains the STTL code for modelling the problem, the constraints, and the evaluation function. The Instance file contains concrete data for instantiating a particular instance of the Problem and finally the Solution file contains values for the *solution variables* found in the Problem.

To model this problem we need to specify a problem file. The following two class diagrams show how we will model the entities and constraints found in our Exam Timetabling model:

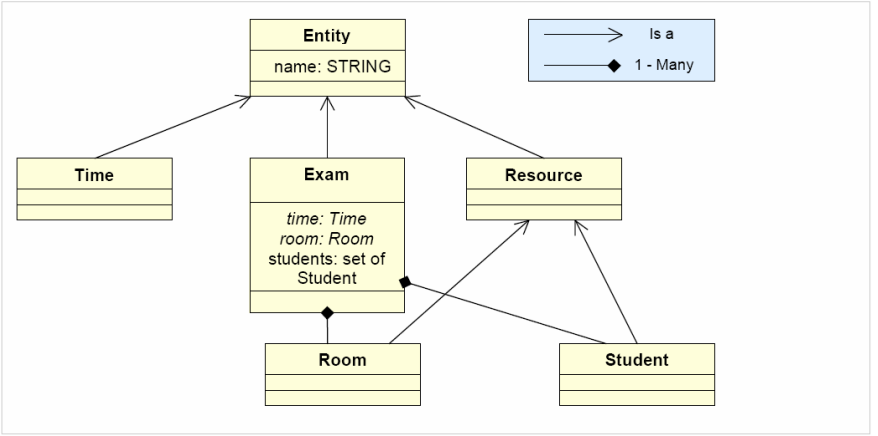

Fig. 2. The classes that make up our ETP model

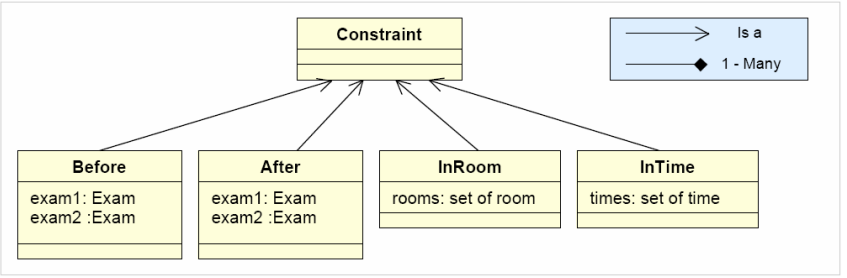

Fig. 3. The Constraints that are added to our ETP problem as Classes extending the Constraint class

This problem has been fully coded in STTL and is available for inspection and use online at: `http://www.informatics.sussex.ac.uk/users/djr23/STTL/`

The evaluation function can be used to evaluate existing solutions (in STTL format) demonstrating the functionality of STTL using the publicly available interpreter.

4 Limitations of this Model

This model was successfully used within a timetabling application[13], with existing data being converted to the STTL Instance file format. However this experience made

apparent some limitations of the model due to both our design approach and issues with STTL itself.

The STTL language involves a learning curve and, probably because of its nature as an Object Oriented Functional language, is quite complicated to use. Although set theory is one good way of specifying these kinds of problems it might not be the best way from a purely modelling point of view. The code fragment below, part of the STTL ETP evaluation function, illustrates the complexity of this language.

```
violations:SEQ[Violation] = (createViolation
(roomViolationExist, name + " should not be scheduled
in this room") + createVioion(timeViolationExist, name
+ " should not be scheduled in this time") +
createViolation(clashExist(all Exam), "Room Clash in "+
room.name + " at "+ time.name))
```

The design also introduces other complexities, distinct from those created due to the syntax; in the model presented above time is represented as a "Time" class which inherits directly from the "Entity" class however it seems that *time* and *room* are very similar classes. For consistency in design we suggest that in our Extensible Model these should inherit the properties of a *container* class (itself a resource) which is used to contain sets of other resources.

There are also inconsistencies in the way that constraints are modelled. In some cases constraints are modelled as classes, containing all the functions for finding violations, however in other cases constraints are modelled as functions inside arbitrary classes. For example, Fig. 3 shows all the constraints we modelled apart from the clash constraint which is implemented as a function in the Exam class. It would be nice if all the constraints were modelled in the same way as this would allow all existing constraints to be extended and for all constraints to be handled in the same way by a single evaluation function.

Due to its design the STTL interpreter can be quite slow compared to other languages; the application we were creating was highly interactive it needed to be very responsive. The STTL interpreter proved to be too slow for our purposes and so the evaluation function was re-implemented in Java using the STTL simply as the data format for input and output.

From this experience we found that STTL was of most use as a data format for specifying instances and solutions precisely, whilst the evaluation functions and problem specifications were largely extraneous. It was found to be a relatively simple task to translate data from different formats into STTL.

5 Designing a Flexible Model

The experience of using STTL and modelling timetabling problems suggested that a new, maybe simpler, approach to modelling these problems should be examined. Rather than proposing a new timetabling language we propose the idea of a standard model for timetabling problems building on the ideas found in STTL but also making use of the functionality, standardization and ease of use provided by modern Object Oriented languages.

Our goal is to create a small and simple subset of Classes which are required to model the examination timetabling problem, but that can be extended or added to model other timetabling problems. The model will be based on the structure of the problem domain and its solution rather than considering any particular approach to solving the problem or any particular implementation language. We intend to exploit the features of Object Oriented programming and the UML modelling language to achieve this. Such a model would still need to conform to the requirements set out in [6] summarised as:

- Generality
- Completeness of problem
- Ease of translation

This can be augmented with the additional requirement, *Ease of modelling*. These two properties provide an actual incentive for adopting this flexible model over other formats which exist. Ease of modelling suggests that this framework will actually make it easier to model timetabling problems than using a general language and is achieved in two ways:

1. Defined hierarchical framework
2. Reusable components

This framework will define model for the exam timetabling problem but can be extended to model other timetabling problems. It may well be that it won't be the most suitable framework for *every* timetabling problem but our aim is to make it suitable for the vast majority of applications.

We choose an object oriented approach as this allows us to use a subset of the well defined UML language to specify our framework and use the standard inheritance mechanism to create the flexibility we require. In the examples and terminology below the Java language is assumed but there is no reason that the design cannot be implemented in another language.

An ontology for constructing scheduling systems is proposed in [14]. The ontology proposed is structured around a constraint satisfaction model where activities are assigned resources subject to constraints. This is a good basis for modelling the timetabling problems and this approach is also taken in our model described below.

Based on all these ideas we propose an extensible model based on the constraint satisfaction problem built up in three layers:

1. Constraint Satisfaction Problem
2. General Timetabling Problem
3. University Examination Timetabling Problem

Each layer builds upon the previous layer adding problem specific resources and constraints. Once the lower layers have been implemented they can be re-used for different timetabling problems with a minimal amount of work. The functionality available at each of the lower layers is always available at the highest abstraction

level, for example a constraint specified in the General Timetabling Problem, can also be applied to the ET problem.

5.1 The Constraint Satisfaction Problem layer:

The lowest level we consider is the constraint satisfaction problem, of which timetabling is an example. This problem simply consists of constraints that need to be satisfied, a 'Resource' class is added representing anything that is not a constraint.

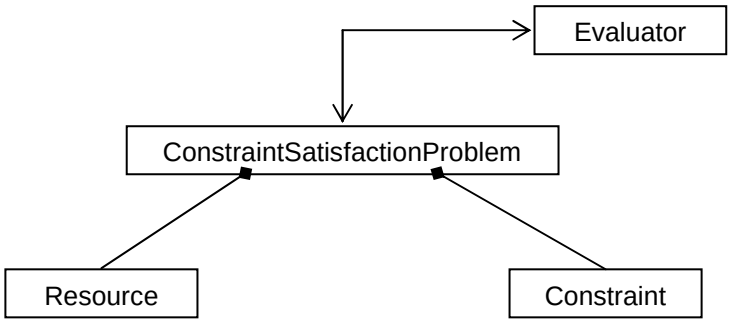

Fig. 4. The classes present in the Constraint Satisfaction Model

Constraints are modelled as functional classes. Each Constraint implements the methods shown in **Table 1**. The getViolationCount() method contains the logic for specifying the Constraint.

Table 1. Description of the Constraint class

Constraint Class	
getViolationCount()	Returns the number of violations of this Constraint found in the problem.
getWeight()	Returns the weight to be applied to violations of this constraint to calculate the cost of this solution.
isHard()	Returns true only if this is a hard constraint.

By storing attributes for the weight assigned to violations of this constraint and whether or not the constraint is *hard* or *soft* each Constraint class becomes responsible for evaluating itself. An overall evaluation function in an "Evaluator" class can then aggregate all these evaluations into the global evaluation function for the entire problem.

The final class introduced here is the Evaluator which is responsible for evaluating instances of this abstract Constraint Satisfaction problem.

Table 2. Description of the Evaluator Class

Evaluator Class	
Evaluate()	Calculates the cost of the current problem.
isFeasible()	Returns true only if no hard constraints have been violated.

In our instance, the actual evaluate 'function' is very simple, and can be implemented in few lines in Java:

```
public int evaluate(){
          int cost = 0;
          for (Constraint constraint:
          problem.getConstraints()) {
            cost += constraint.getViolationCount() *
             constraint.getWeight();
          }
          return cost;
}
```

5.2 The General Timetabling Problem

This model can then be extended for the abstract General Timetabling Problem, as illustrated below in figure 5:

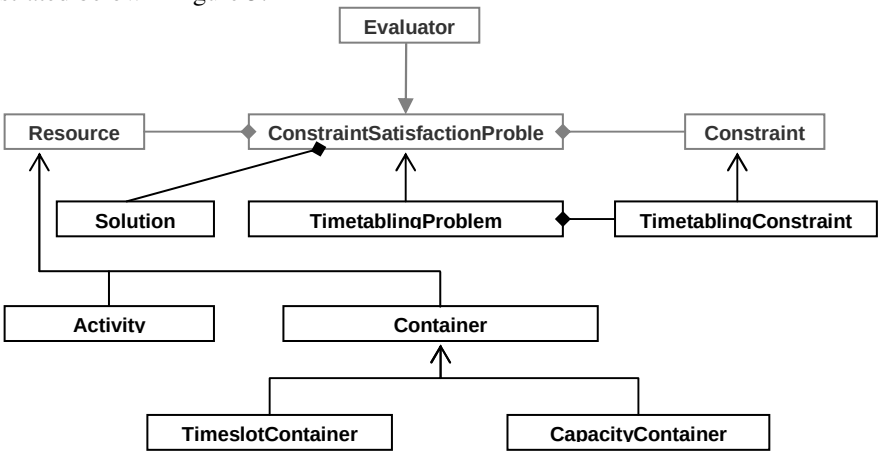

Fig. 5. The Classes in the General Timetabling Model

The representation of time is one of the most difficult design decisions to make in a model such as this. As Time is not a Constraint we choose to model Time as a sequence of Timeslots, implemented using our Container interface to which Activities can be assigned. Each TimeslotContainer is specified with a duration and an order,

this simple representation could easily be extended with more information such as day/week information or whether a break exists beforehand.

The solution is represented by a completely new Solution class which stores the assignment of Activities to Containers.

Table 3. Description of the Classes found in the General Timetabling Problem

Class	Description
Activity	Any activity that is to be timetabled.
Solution	Stores the container each activity has been timetabled to
TimetablingConstraint	Constraints that can access the Timetabling resources
Container	A container where an activity can be timetabled
CapacityContainer	A container with a limit to the number of resources that can be added
TimeslotContainer	An ordered container with a specified duration

5.3 The University Exam Timetabling Problem layer

With the lower layers taken care of the ETP layer can be modelled relatively easily. Note that no work is needed to change the default Evaluator or Solution classes. In fact the only classes introduced here are those that directly map the abstract Timetabling problem to the real world Exam Timetabling application. It is envisioned that further timetabling problems can be modelled using this framework with similar ease.
The following classes and constraints are introduced to the model to implement the ETP:

Table 4. Resources in the Exam Timetabling Problem

Resource	Description
Exam	Models exam activities and their enrolments. Enrolments are lists of students taking this exam. As the activity resource is extended the name and duration attributes are already implemented.
Student	Models a student as a resource.
Room	A Container Class in which exam activities can be scheduled.

A working prototype of this model was built using Java and shown to work with our existing STTL data using a simple parser. As our design only specifies the interface of the model we were able to build in a number of optimizations to make our model

efficient at handling large data sets. The complete API specification for our model can be found online at:
`http://www.informatics.sussex.ac.uk/users/djr23/emdocs`

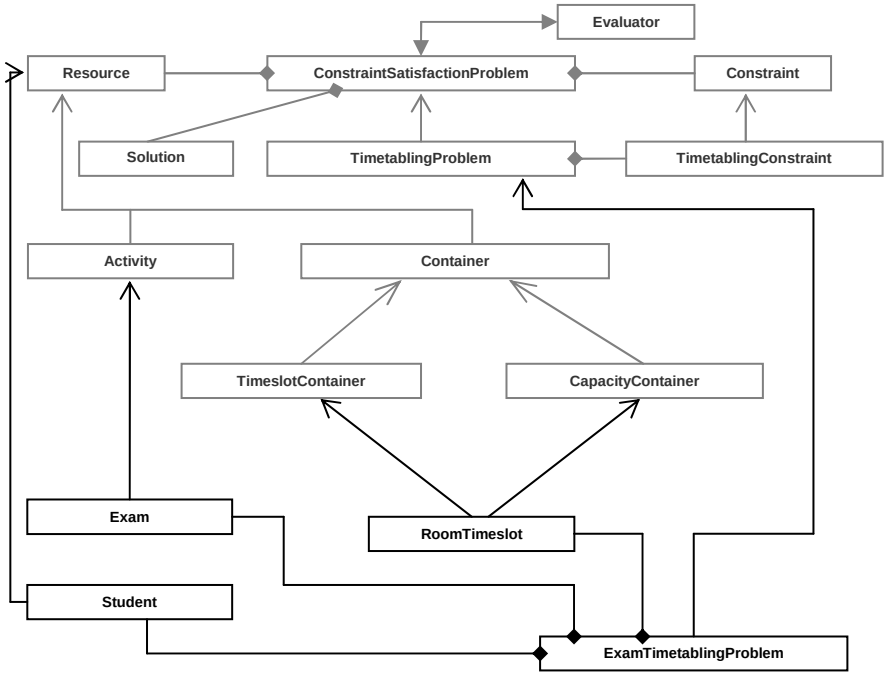

Fig. 6. Classes in the complete Exam Timetabling Problem model

Some of the Constraint classes register event listeners with the Solution class so they are notified of any changes to the Solution. This allows an incremental approach to counting the violations of each constraint and much improved performance.

6 Future Work

In this paper an attempt to design an "extensible" modelling framework, with the aim of simplifying the modelling process for many timetabling problems, is reported and applicability of this approach is demonstrated to model the Examination Timetabling problem. However, to demonstrate the extensibility of the modelling framework, it will be necessary to show that the model works for other timetabling applications and that the same design consistency can be applied across different problems in this domain. One possibility is to set up an online repository where these different applications of the model (documentation, implementations and problem data) can be accessed. We welcome any use of this model, especially in real world systems or applications to other timetabling problems.

An alternative to the use of STTL for data format for the Timetabling models is to store data as a simple XML document containing the information needed to instantiate each Class in the model. The logic and specification of the actual problem would remain in the implementation language but the instance data could be exchanged in this format, regardless of what language the model was implemented in. It would also be useful to create parsers for reading and saving to other data formats such as the Carter data format.

7 Concluding Remarks

The aim of this paper has partly been to reignite discussion on the issue of "Standard Timetabling Languages" but mainly to promote our ideas on a different approach to this topic and how these problems could be modelled inline with modern programming paradigms.

Unlike other approaches we have deliberately shied away from advocating a particular programming language (apart from for the purposes of demonstrating our exam timetabling model) as we believe this is best decided by the capabilities of the user. All mainstream languages are capable of modelling problems in this domain. Trying to form consensus around a standardized language is always difficult but focusing on this when such a language is not required can cause discussion to stagnate and limit progress.

References

1. Fisher, J.G. and R.R. Shier, *A Heuristic Procedure for Large-Scale examination scheduling problems.* Congressus Numerantium, 1983. **39**: p. 399-409.
2. Burke, E.K., D. Elliman, P. Ford, and R. Weare. *Examination Timetabling in British Universities - A Survey*. in *PATAT*. 1995.
3. Gaspero, L.D. and A. Schaerf, *Tabu Search Techniques for Examination Timetabling* in *Selected papers from the Third International Conference on Practice and Theory of Automated Timetabling III* 2001 Springer-Verlag. p. 104-117
4. The Timetabling Problem Database, 2003, retrieved on 30/01/2006 from http://www.or.ms.unimelb.edu.au/timetabling.html
5. Burke, E.K., J.H. Kingston, and P.A. Pepper, *A Standard Data Format for Timetabling Instances* in *Selected papers from the Second International Conference on Practice and Theory of Automated Timetabling II* 1998 Springer-Verlag. p. 213-222
6. Burke, E.K. and J.H. Kingston, *A Standard Format for Timetabling Instances.* Lecture Notes In Computer Science, 1997. **1408**.
7. TSPLIB-A library of travelling salesman and related problem instances, 1995, retrieved on January 2006 from http://softlib.rice.edu/tsplib.html
8. Kingston, J.H., *Modelling Timetabling Problems with STTL* in *Selected papers from the Third International Conference on Practice and Theory of Automated Timetabling III* 2001 Springer-Verlag. p. 309-321
9. A user's guide to the STTL Timetabling Language, retrieved on January 2006 from http://www.it.usyd.edu.au/~jeff/ttsttl1.ps

10. Ozcan, E., *Towards an XML based standard for Timetabling Problems: TTML*, in *Multidisciplinary Scheduling: Theory and Applications: 1st International Conference, Mista '03 Nottingham, UK, 13-15 August 2003. Selected Papers*, G. Kendall, et al., Editors. 2005, Springer-Verlag.
11. Reis, L.s.P. and E. Oliveira, *A Language for Specifying Complete Timetabling Problems*, in *Selected papers from the Third International Conference on Practice and Theory of Automated Timetabling III*. 2001, Springer-Verlag. p. 322-341.
12. Ranson, D., *Interactive Visualisations for the Generation, Evaluation and Analysis of heuristics in scheduling: Thesis Progress Report 2004*. 2004, Progress Report, University of Sussex.
13. Ranson, D. and P.C.-H. Cheng. *Graphical Tools for Heursitic Visualization.* in *Multidisciplinary International Conference on Scheduling: Theory and Applications*. 2005. New York, USA.
14. Smith, S. and M. Becker, *An Ontology for Constructing Scheduling Systems*, in *Working Notes of 1997 AAAI Symposium on Ontological Engineering*. 1997, AAAI Press.