Declarative Configuration Applied to Course Scheduling

by

Vincent S. Yeung

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

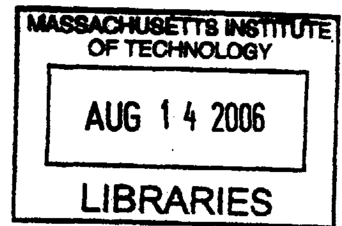
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

[June 2006]
May 2006

Copyright 2006 Vincent S. Yeung. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.



Author ....
Department of Electrical Engineering and Computer Science
May 26, 2006

Certified by ....
Daniel Jackson
Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

ARCHIVES

Declarative Configuration Applied to Course Scheduling

by

Vincent S. Yeung

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2006, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes a course scheduling system that models planning as a satisfiability problem in relational logic. Given a set of course requirements for a degree program, our system can find a schedule of courses that will complete these requirements. It supports a flexible XML format for expressing course requirements and also handles additional user-specific constraints, such as requirements that certain courses be taken at particular times. Various optimizations were included in the translation to relational logic to improve the performance of our system and the quality of its results. We ran experiments on our system using degree programs from the Department of Electrical Engineering and Computer Science at MIT as input, and found that our approach is competitive with conventional planners.

Thesis Supervisor: Daniel Jackson
Title: Professor

Acknowledgments

I thank my advisor, Prof. Daniel Jackson, for supervising my research at LCS/CSAIL in the past four years, and for introducing me to the fascinating area of formal methods. I thank Emina Torlak for being an excellent mentor in this and previous research, and Derek Rayside for helping me clear numerous technical hurdles.

I thank my living group, Next House Third West, for the camaraderie that I will very much miss, and fellow MEng students Philip Guo and Jelani Nelson, among many others, for sharing the joy of thesis writing.

Finally, I thank my parents, who have always been supportive of me, and my brother, who has guided me when I needed his guidance.

Contents

1 Introduction **15**

1.1 Motivation . . . 15

1.2 System Overview . . . 16

1.2.1 Setting up the Problem . . . 17

1.2.2 Converting to Relational Logic . . . 18

1.2.3 Solving with Kodkod . . . 20

1.2.4 Viewing Solution and Adding Constraints . . . 20

1.3 Contribution . . . 21

2 Data Model and Representation **23**

2.1 Relational Model . . . 23

2.1.1 Discussion . . . 25

2.2 Requirement Translation . . . 26

2.2.1 Using Partial Instance Functionality . . . 27

2.2.2 Global Constraints . . . 27

2.2.3 Prerequisite Translation . . . 28

2.2.4 Degree and Additional Constraint Translation . . . 29

3 System Architecture **33**

3.1 Overview . . . 33

3.2 Key Data Abstractions . . . 33

3.2.1 Semester, Course, Attribute . . . 34

3.2.2 Schedule . . . 34

3.2.3 CourseGrouping 34
3.2.4 Requirement 34
3.2.5 DegreeProgram 35
3.2.6 PrereqMap 35
3.2.7 Problem, Solution 35
3.3 Translation Architecture 35
3.4 Remote Execution Architecture 37
3.4.1 Alternative Design 38

4 Evaluation **41**
4.1 Performance 41
4.1.1 Minimum-size Subset Requirements 42
4.1.2 Minimality 43
4.2 Limitations 48
4.2.1 Efficiently Using Completed Courses 48
4.2.2 Inexact Requirements 49
4.2.3 Course Scope 50
4.2.4 Numerical Constraints 50

5 Related Work **51**
5.1 Planning as Propositional Logic 51
5.2 Planning as Temporal Logic 53
5.3 Other Course Planners 53

A Input Specification Format **55**
A.1 Course Requirement Specification 56
A.1.1 Course Definitions 56
A.1.2 Prerequisites 57
A.1.3 Degree Requirements 58
A.2 Partial Schedule Specification 60
A.3 Additional Constraints 61

A.3.1 Time Requirement 61
A.3.2 Never Schedule Requirement 62

B Requirement Knowledge Acquisition **63**
B.1 Prerequisite Information Acquisition 63
B.2 Degree Requirements Acquisition 64

C User Manual **67**
C.1 Quick Start Guide 67
C.2 Detailed Guide 67
C.2.1 Invoking the System 67
C.2.2 GUI Overview 68
C.2.3 Loading Input 68
C.2.4 From a Degree Program 69
C.2.5 From a Problem File 69
C.2.6 Creating a Partial Schedule from a Grade Report 69
C.2.7 Saving a Query 69
C.3 Generating a Schedule 70
C.3.1 Viewing and Tweaking a Solution 70

D Complete Example **73**

E PDDL Example **77**

List of Figures

1-1 Data flow in course scheduling system 16
1-2 Example of generated solution, with an additional requirement 21
2-1 Object model 24
3-1 Requirement translation dependency diagram. 36
3-2 Local execution architecture. 37
3-3 Remote execution architecture. 38

List of Tables

2.1 Supported requirements. 30

2.2 Mandatory course requirement translation 30

2.3 Minimum-size subset requirement translation (general case) 31

2.4 Minimum-size subset requirement translation (all single-course case) . 31

2.5 No overlap requirement translation 31

2.6 Time requirement (before) requirement translation 32

2.7 Time requirement (after) requirement translation 32

2.8 Never schedule requirement translation 32

Chapter 1

Introduction

1.1 Motivation

Course requirements for an academic degree program are often complex, with long prerequisite chains and layered requirements. Nonetheless, a student is faced with the problem of finding a schedule of courses for future semesters that satisfies the necessary requirements. Since course requirements are usually electronically accessible, there is much incentive to build an automated schedule generator, which would be of great help to students organizing course schedules to complete their degrees.

Course scheduling can be generalized as a *planning* problem. Although planning has long been a focus of artificial intelligence, from a usability and efficiency standpoint, general-purpose planners are not necessarily ideal for generating course schedules. From a usability standpoint, general-purpose planners often use input formats that are either awkward to use for representing degree requirements (e.g. the Planning Domain Definition Language [9]) or overly complex for a user not familiar with discrete mathematics (e.g. temporal logic input). From an efficiency standpoint, we discuss in Chapter 4.1 how some general-purpose planners may have trouble in the course scheduling domain.

We have developed a system for creating course schedules that aims to be both usable and sufficiently efficient for this problem domain. Our system uses a general-

izable technique for solving planning problems via translation into relational logic[1]. Additionally, domain-specific features, such as the ability to input partially complete schedules and a simple syntax for expressing degree requirements, aid in making the system more usable by the general student body.

1.2 System Overview

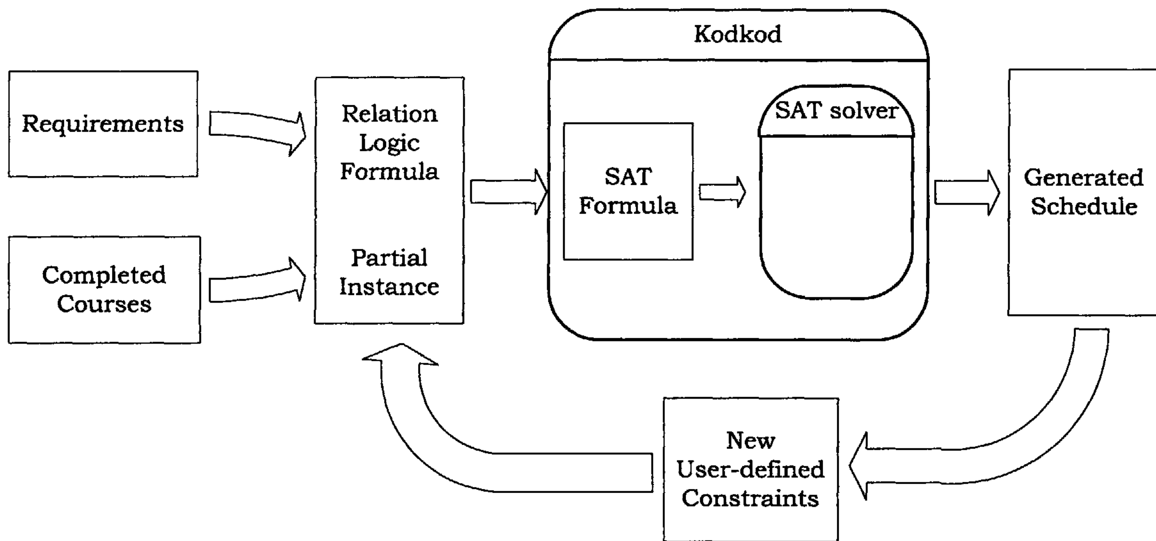


Figure 1-1: Data flow in course scheduling system

Figure 1-1 gives an overview of our system's functionality. The targeted user of the system is a student looking to generate a schedule for future semesters. This user supplies a set of courses he has completed and selects a set of requirements for the desired degree. The requirements may be supplied by the student himself or another person; the system has been packaged with requirements for several degree programs at the Massachusetts Institute of Technology. The requirements are used to generate a relational formula, while the set of completed courses is used to construct a *partial instance*. A partial instance consists of mappings from relations to tuples that they are known to contain, and aids in reducing the number of unknowns in the analysis.

[1]In this thesis, we use the definition of relational logic as presented in [2]—that is, first-order logic with the extension of transitive closure.

Next, the system executes a solver, Kodkod [10], to find a *solution* to the formula. A solution is an assignment of relations to values allowed by their bounds that conforms to the original partial instance, and for which the formula evaluates to true.

Kodkod converts the formula into SAT and executes a general-purpose SAT solver, such as ZChaff [7]. The solution returned by the SAT solver is translated back into a schedule and presented to the user. Through the interactive interface of our system, the user can proceed to add additional constraints to optimize the generated solution according to personal preferences; these constraints are conjoined to the original relational formula, and a new solution is generated.

Our strategy formalizes course scheduling as a satisfiability problem. Indeed, much research has been done to model planning as a satisfiability problem [5], and to automatically construct propositional SAT formulas from planning problems [3, 1]. Our approach differs in that the translation is done to relational logic, and the ability of Kodkod to accept partial instances is used whenever possible to reduce the number of variables in the generated formulas.[2]

1.2.1 Setting up the Problem

In this section, we will give an example of the system in action. To begin, the user selects a degree program whose requirements will be loaded into the system. The requirements are represented in an XML format (see Appendix A). Suppose the user has chosen the requirements for the Electrical Engineering and Computer Science (EECS) Bachelor of Science program at the Massachusetts Institute of Technology. These requirements encapsulate information about:

- Required core classes and electives for the degree. For instance, they specify that the EECS core classes 6.001, 6.002, 6.003, and 6.004, must be taken, and that at

[2] The amount of time needed to find a solution to a SAT formula is strongly affected by the number of Boolean variables and propositional clauses in the formula. When translating relational logic into SAT, a relation is decomposed into multiple Boolean variables. For instance, a binary relation from a domain of m atoms to a range of n atoms is represented by a matrix of $m \times n$ variables; a variable x_{ij} evaluates to true if the ith atom in the domain and jth atom in the range are related. When Kodkod is supplied with a partial instance, it replaces such variables with constant Boolean values. If this were not possible, Kodkod would have to not only include extraneous variables but also add propositional clauses to constrain the variables' values.

least two of the three computer science "header" classes (introductory subjects for the three concentrations: Artificial Intelligence, Systems, and Theoretical Computer Science) must be completed.

- Prerequisite relationships between individual classes. For example, the prerequisites for 6.004 (computer architecture) are 6.001 (computer programs) and 6.002 (circuits). Certain courses' prerequisite requirements may be satisfied in more than one way. For instance, 6.002's prerequisites are one of 18.03/18.06 (differential equations/linear algebra) and one of 8.02/8.022 (variations of electricity and magnetism offerings).

In addition to course requirements, the user gives the system a set of completed courses. These courses can be used to satisfy prerequisite or degree requirements. Completed courses may be entered manually or imported from an HTML grade report file produced by the MIT Student Information System (http://student.mit.edu).

1.2.2 Converting to Relational Logic

To represent the course scheduling problem, we define a set of relations and constrain the values that can be bounded to each relation with partial instances and formulas. The details of this data model are explained in Chapter 2. Here, we will instead present a simple example that involves the following relations in the model:

- `Semester`: unary relation containing the set of semesters.
- `Course`: unary relation containing the set of all possible courses.
- `sCourses`: binary relation whose domain is `Semester` and range is `Course`. `sCourses` contains a tuple (s, c) if c is scheduled for the semester s.

As an example, consider the requirement that the core courses 6.001, 6.002, 6.003, and 6.004 must be taken. To express this requirement, we write the following relational formula (expressed in Alloy syntax [4]):

```
(6.001 + 6.002 + 6.003 + 6.004) in Semester.sCourses
```

The plus sign (+) and dot (.) between relations (i.e. not including a dot within a course number) denote union and join operations, respectively, while the keyword

`in` denotes subset of. Note that singleton unary relations must be created for each of 6.001, 6.002, etc. in order to be able to reference individual courses in a formula.

For each relation that is created, Kodkod requires that we specify an *upper bound* and optionally a *lower bound* of tuples. The upper bound of a relation consists of all tuples that *may* appear in the relation; the lower bound consists of all tuples that *must* appear in the relation. A partial solution can thus be set with the lower bound; knowledge that a course c must be scheduled in semester s can be handled by including the tuple (s, c) in the lower bound of `sCourses`. All courses that have already been completed are mapped in this manner to a special semester called `PastSemesters`—this is done to distinguish courses that the user has already taken from courses generated by the system and is helpful because we do not check if prerequisite relationships are satisfied for the former (the user might have obtained special permission from the department).

Also note that lower bounds are used to completely define some relations; this happens when the lower and upper bounds are identical. Completely defining a relation eliminates the need for the external solver to determine its value from the formulas. For instance, `Semester` and `Course`, as well as the singleton course relations mentioned earlier, are set in this manner.

Prerequisites

To handle prerequisite relations, we define additional binary relations to map courses to their prerequisites. We then set the lower and upper bounds of these relations with the prerequisite information defined in the input. For example, we must bound the appropriate relations with the information that 6.001 and 6.002 are prerequisites for 6.004.

Once the appropriate relations are pre-set with prerequisite relationships, the system can include a single formula to require that for any course appearing in `Semester.sCourses`, the course's prerequisites must be scheduled for a semester before the one for which the course has been scheduled.

It should be noted that the system must be able to handle the fact that prerequisite

requirements for some courses can be satisfied in multiple ways. The details of how this is treated are discussed in Chapter 2.

1.2.3 Solving with Kodkod

Once the scheduling system has created a relational formula and bounds information, it executes Kodkod to generate a solution to the formula. If no solution is found, the user is alerted that there is no way to complete the degree (because the requirements are inconsistent, or because there are not enough semesters left to complete the degree, etc.). Otherwise, the solution is mapped back to the original planning problem; that is, we extract information from the values of the relations. The most important relation, as one might guess, is `sCourses`, which enumerates the generated schedule. In Chapter 2, we will illustrate how our data model allows further interesting information to be extracted; for instance, if a particular requirement, such as the computer science header requirement mentioned earlier, can be satisfied by two of three possible courses, it is possible to easily determine which two courses were actually chosen to satisfy this requirement without having to examine the entire generated schedule.

1.2.4 Viewing Solution and Adding Constraints

Figure 1-2 is a screenshot of the user interface for presenting a generated schedule. Courses are grouped by the semester to which they are assigned (for instance, 18.06 and 6.002 have been scheduled for Fall 2006), and the various buttons allow the user to add additional constraints to adjust the solution according to their personal preferences.

The user can, for example, require a certain course to be omitted from the schedule or to be scheduled only before/after a certain time. In the figure, such a requirement appears at the bottom panel; any new solution that is generated will include the course 6.UAT in a semester after Spring 2007. Detailed instructions for these and other features are given in the User Manual (Appendix C).

Such additional constraints fit in well with our translation scheme. They are

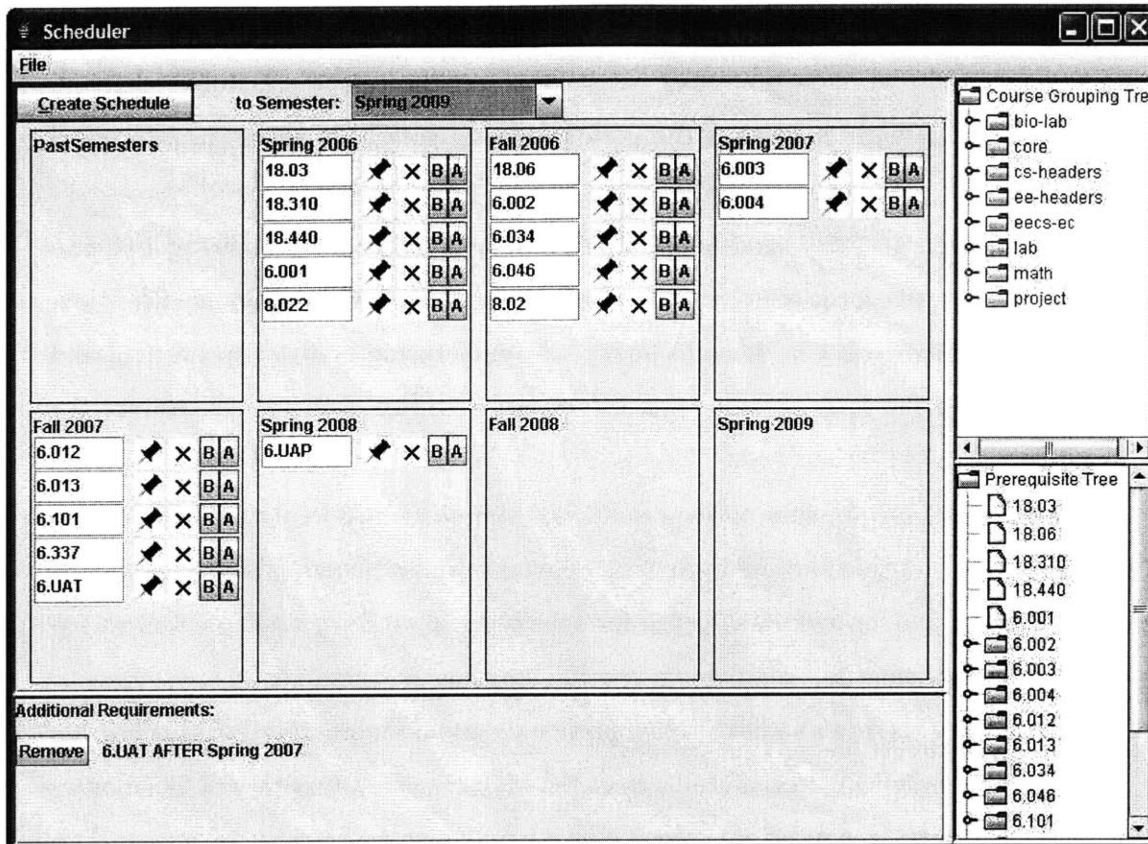


Figure 1-2: Example of generated solution, with an additional requirement

translated into relational logic and conjoined to the original formula created from course requirements. The solver is then re-run, and a new schedule is returned.

1.3 Contribution

This thesis presents a method for modeling planning problems as a relational formula with a partial instance. To our knowledge, although planning has often been modeled as satisfiability problems in other logics (e.g. propositional [5, 3, 1] and temporal [8]), the particular approach we use is uncommon. Using a relational model to represent our data allows our system to easily extract more useful information from the relational solution than just the generated schedule; for instance, the system can explain how particular requirements were satisfied in the generated schedule simply

by evaluating the appropriate relations. In addition to the representation benefits of our approach, we show empirical results that, in the course scheduling domain, it is competitive with or exceeds the performance of general-purposes propositional solvers.

This method of translation is applicable not only to course scheduling but also for providing a platform to solve other planning problems with minor modification. Planning problems exist in domains such as manufacturing and transportation, and may appear in the form of order precedence and resource capacity constraints, for example.

This thesis also presents an expressive but simple syntax for formalizing course requirements. For this particular domain, general-purpose planning languages can be overwhelming and/or awkward to use for specifying input by a user not well-versed in discrete mathematics, so a domain-specific language is desirable.

Finally, this thesis embodies a fully functional system for creating course schedules that will hopefully be of use to students at the Massachusetts Institute of Technology and wherever else it may be adopted.

Chapter 2

Data Model and Representation

In this chapter, we discuss our relational logic representation of course scheduling. We begin with an overview of the model and subsequently detail the translation mechanism for different types of requirements.

To illustrate our data model, we will provide examples from the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology. Most MIT courses are numbered in the format $X.Y$ where X is the department number (e.g. '6' for EECS) and Y is the subject number.

2.1 Relational Model

Figure 2-1 depicts the relations used in our representation. The boxes represent unary relations (i.e. sets) while the labeled arrows represent binary relations between sets. The open-ended arrows represent a subset relationship. Finally, the multiplicity markers ?, !, and + denote *zero or one*, *exactly one*, and *at least one*, respectively.

The following is a hierarchical description of the relations.

- `Attribute`—an attribute, such as `Spring` or `Even` (year, e.g. 2006), that can be assigned to a particular `Course` or `Semester`.
- `PrereqSet`—a set of courses, such as `{6.001,6.002}`, that together can fulfill the prerequisite requirements of a particular course. Note that `PrereqSet` contains the empty prerequisite set `{}`.

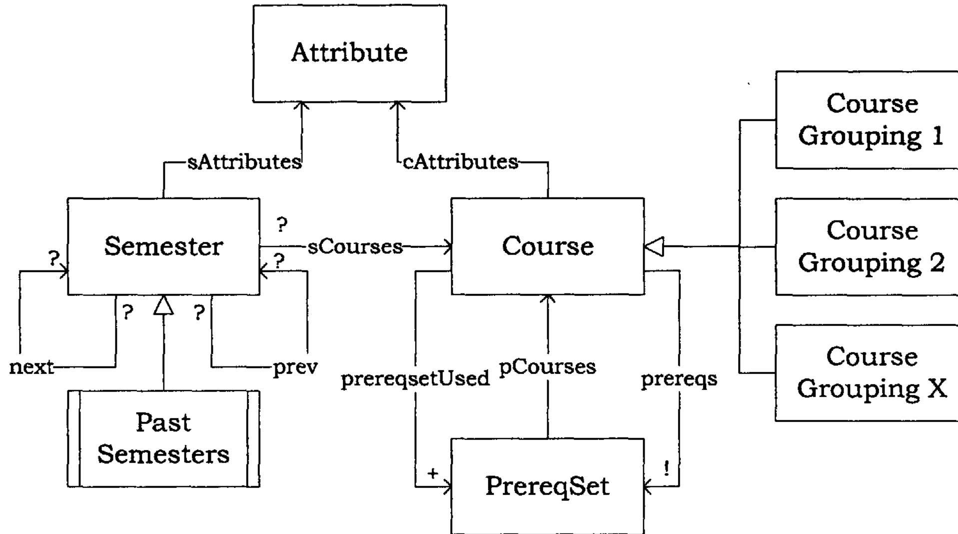


Figure 2-1: Object model

  - pCourses maps a PrereqSet to the courses it contains. For instance, a prerequisite set P that represents the courses 6.001 and 6.002 will be mapped by the pCourses tuples (P,6.001) and ((P,6.002).

- Course—represents a single course, such as 6.002.
  - cAttributes maps a course to its attributes, with tuples such as (6.033,Spring), which indicates that 6.033 is only offered in spring semesters.
  - prereqs indicates which PrereqSet(s) may be used to satisfy a course's prerequisite requirements. Courses that do not have prerequisites are mapped to the empty set {}.
  - prereqsetUsed represents the one prerequisite set that is used to satisfy a course's requirements in the generated schedule; this is useful in the event that a course is mapped to multiple possible PrereqSets by prereqs.
  - *Course Groupings* are subsets of Course that are created during translation for writing more expressive constraints. A course grouping may or may not be a singleton set. See Section 2.2 for details.

- Semester—denotes a single semester. PastSemesters is a singleton subset of Semesters and is used to represent the amalgam of all semesters in the past.
  - prev and next denote the ordering of elements of Semester. For instance, prev might have the tuple (Fall2006,Spring2006), and next would then

contain the tuple `(Spring2006,Fall2006)`.

– `sCourses` maps a semester to the set of `Courses` scheduled for it.

2.1.1 Discussion

Attributes

Attributes were designed as a simple way of indicating that not all courses are offered every semester. If a course has a specific attribute, such as `Spring`, then our constraints will require that it be scheduled only to a semester that also possesses this attribute (see Section 2.2.2). This mechanism is also used to handle the common case of a course that is offered every other year, and can also be used for one-time course offerings (by defining a unique attribute).

Course Groupings

A *course grouping* is a unary relation that represents some arbitrary set of courses. It is created during translation into relational logic to allow formulas to directly reference this arbitrary set. When the user defines a course grouping, he must also provide a set of requirements to constrain the values that the set can take. For example, consider the requirement that at least two of the three MIT computer science "headers" (6.033, 6.034, 6.046) must be taken. The user can define a course grouping named "cs-headers" and include the requirement that two of three courses must be in the course grouping, and the system will appropriately generate a relational formula from the requirement to constrain the relation `cs-headers`. The purpose of creating such a grouping is so that the user can easily refer to the set of *selected* computer science headers when writing other requirements.

For instance, suppose in addition to the header requirement above, the student is required to take two elective courses. The elective courses may be chosen from the headers and a selection of intermediate and advanced courses, but the same course may not be used to fulfill both the header and elective requirements. Thus, it is useful to be able to write a constraint that can refer to the two course groupings that satisfy

each of these requirements and require that they do not intersect.

Single-course course groupings are a special case. These are simply course groupings that are created to represent not an arbitrary set of courses, but instead exactly one course. For instance, consider the constraint that the set of scheduled courses must contain 6.001. When our system translates this into relational logic, it becomes `6.001 in Semester.sCourses`. This requires a singleton unary relation named `6.001` that contains only the course 6.001. Thus, for every course, we create such "single-course" relations.

Prerequisite Sets

The notion of *prerequisite sets* was introduced to allow for courses whose prerequisite requirements can be satisfied in multiple ways. Usually, the department denotes this as a conjunction of disjunctions; for instance, in MIT's Electrical Engineering and Computer Science curriculum, the circuits course 6.002's prerequisites are one of 18.03/18.06 (differential equations/linear algebra) and one of 8.02/8.022 (variations of electricity and magnetism offerings). In contrast, our representation is a disjunction of conjunctions. The same requirement can be rewritten as one requiring that any one of the four sets {18.03,8.02}, {18.03,8.022}, {18.06,8.02}, and {18.06,8.022} must be taken. As we will see in section 2.2.3, this representation allows for a simple way of expressing prerequisite constraints in relational logic.

2.2 Requirement Translation

In this section, we will describe the details of translating user-specified scheduling constraints into the above representation.

Translation has the following steps:

1. Allocate course grouping relations depending on the requirements.
2. Create bounds (including partial instances, if applicable) for all relations.
3. Generate from the requirements logical formulas over the relations.

2.2.1 Using Partial Instance Functionality

Kodkod, the relational logic constraint solver used in our system, supports the notion of a *partial instance*. For each relation, we must specify an *upper bound* and optionally a *lower bound* of tuples. The upper bound of a relation consists of all tuples that *may* appear in the relation; the lower bound set consists of all tuples that *must* appear in the relation. A partial instance can thus be set with a non-empty lower bound.

Furthermore, we can completely define the value of a relation by setting identical lower and upper bounds. Doing so eliminates the need for the external solver to determine the relation's value from the formulas.

The following relations are completely defined prior to solving:

- The "Types": `Attribute`, `Course`, `PrereqSet`, `Semester`.
- `PastSemesters`.
- Single-course course groupings.
- `prereqs`, `pCourses`.
- `next`, `prev`.
- `cAttributes`, `sAttributes`.

If the user has entered a set of courses that he has taken in the past or wants to fix parts of his future schedule (e.g. to require a specific course to be scheduled for a certain time), we specify these constraints by setting a non-empty lower bound for `sCourses`.

2.2.2 Global Constraints

Matching Attributes

As mentioned in Section 2.1.1, we constrain the the model so that a course is only ever assigned to a semester that has the course's attributes. The formula is as follows:

```
pred attributesMatch() {
  all s : Semester - PastSemesters |
      s.sCourses.cAttributes in s.sAttributes
}
```

Preventing Schedule Overlap

The relation sCourses needs to be constrained so that a course is not assigned to multiple semesters. This can be done as follows:

```
pred noOverlap() {
  all s1 : Semester | all s2 : (Semester - s1) |
      no (s1.sCourses & s2.sCourses)
}
```

No semester skipping

Finally, the system should not generate a schedule that contains "skipped" semesters, i.e. an empty semester before a non-empty one, unless that semester is the special PastSemesters, whose courses are user-specified. The constraint can be written as follows:

```
pred noSemesterSkipping() {
  all s1 : (Semester - PastSemesters) |
      all s2 : (s1.^prev - PastSemesters) |
          some s1.sCourses => some s2.sCourses
}
```

where ^ denotes the transitive closure operation.

2.2.3 Prerequisite Translation

The first step of prerequisite translation is completely defining the values of the relations prereqs and pCourses, to represent the user-provided prerequisite relationships between courses. This, as mentioned previously, is done by setting identical lower and upper bounds for these relations.

Next, we need to bound the prereqsetUsed relation. Its upper bound is the same as that of prereqs. In addition to that, we add a logical formula to constrain that preresetUsed is a *function* with domain Semester.sCourses and range PrereqSet. Thus, for any scheduled course, prereqsetUsed will "choose" one of its valid prerequisite sets.

The next step is to generate the logical formulas to constrain `sCourses` so that for every scheduled course, the courses in its chosen prerequisite set, as specified in `prereqsetUsed`, are scheduled to be taken in a previous semester. This can be expressed in relational logic as follows:

```
pred prereqConstraint() {
  prereqsetUsed.pCourses in ~sCourses.^prev.sCourses
}
```

where ~ denotes the transpose operation.

2.2.4 Degree and Additional Constraint Translation

As discussed in Appendix A, our system will translate, in addition to prerequisites, degree requirements and also additional constraints not specific to the degree program (such as user preferences).

Degree requirements are specified as a set of conjoined requirements that constrain the special course grouping corresponding to the expression `Semester.sCourses`, as well as requirements to constrain any other course groupings that are defined and referenced. Additional requirements are a conjoined set of standalone requirements.

Modularity of Translation

A major benefit of course groupings (discussed in Section 2.1.1) is that they allow for the modular translation of requirements. Although the requirements that constrain a course grouping C_1 may reference another course grouping C_2, the translation of these requirements do not depend on the requirements that constrain C_2; when we reference the relation corresponding to C_2, we can assume that this relation will be properly constrained when C_2's requirements are translated. Thus, we can group translation of requirements by the course groupings they constrain. Standalone requirements can be translated independently.

Type	Requirement
Course grouping	Mandatory course Minimum-size subset
Standalone	Time Never schedule
Both	No overlap

Table 2.1: Supported requirements.

Translation by Requirement Type

Table 2.1 summarizes the set of requirements we support. Certain requirements must occur in the context of a course grouping definition, while others must be standalone. When a requirement is part of a course grouping's definition, we call that grouping the requirement's *subject.*

Mandatory course requirement

A mandatory course requirement indicates that a desired set of course groupings must appear in the subject; this can be written as a simple subset formula.

Input:	Course groupings $c_1, c_2, \ldots, c_n$ must appear in subject S.
Formula:	`c1+c2+...+cn in S`

Table 2.2: Mandatory course requirement translation

Minimum-size subset requirement

A minimum-size subset requirement indicates that at least some number (k) of a selection of course groupings must appear in the subject. There are a number of ways to translate this. Assuming the set of allowed course groupings is of size n, one approach is to enumerate all $\binom{n}{k}$ subsets of size k, write a subset formula for each of them, and take the logical disjunction.

As we will discuss in Section 4.1, this approach is inefficient for values of n and k sometimes encountered in practice (e.g. $n = 80, k = 2$). However, assuming that *all* n *choices of course groupings are single-course groupings* (defined in Section 2.1.1), we

Input:	At least k of the n course groupings $c_1, c_2, \ldots, c_n$ must appear in subject S.
Formula:	`(K1 in S) || (K2 in S) || ... || (Kx in S)`, where each K_i is a unique union of k members of the set of course grouping choices, and $x = \binom{n}{k}$.

Table 2.3: Minimum-size subset requirement translation (general case)

can instead use a substantially more efficient translation with an existential formula. Note that whereas the number of clauses in the general translation above is linear in $\binom{n}{k}$, in the translation below it is linear in $\binom{k}{2}$, which in practice is much smaller.

Input:	At least k of the n course groupings $c_1, c_2, \ldots, c_n$ must appear in subject S.
Formula:	`some v1,v2,...,vk: (c1+c2+...+cn) | (v1+v2+...+vk) in S && (no I1) && (no I2) &&...&& (no Ix)`, where each I_j is a unique pairwise intersection of the quantified variables v_i (e.g. `(v1&v2)`, `(v2&vk)`), and $x = \binom{k}{2}$.

Table 2.4: Minimum-size subset requirement translation (all single-course case)

No overlap requirement

A no-overlap requirement simply states that none of the course groupings it specifies can have overlapping values; that is, the same course must not appear multiple times. If the requirement is part of a grouping's definition, that grouping will be included in the constraint. The translation is a conjunction of empty pairwise intersection constraints.

Input:	The course groupings $c_1, c_2, \ldots, c_n$, and the subject S, if applicable, must not have overlapping values.
Formula:	`(no P1) && (no P2) && ... && (no Px)`, where each P_i is a unique pairwise intersection (e.g. `(c1&c2)`, `(c1&cn)`), and $x = \binom{n}{2}$.

Table 2.5: No overlap requirement translation

Time requirement

A time requirement indicates that a course must be scheduled before, at, or after a specified semester. A before/after requirement is written as a formula in relational logic. Note that unlike the requirements described earlier, the input to a time requirement is a course, not a course grouping.

Input:	The course c must be scheduled to be taken *before* semester S.
Formula:	`c in S.^prev.sCourses`

Table 2.6: Time requirement (before) requirement translation

Input:	The course c must be scheduled to be taken *after* semester S.
Formula:	`c in S.^next.sCourses`

Table 2.7: Time requirement (after) requirement translation

Notice that this translation requires that c be scheduled (as opposed to only requiring that the timing constraints be satisfied when the course is in fact scheduled). This is consistent with our definition of the requirement as described in Appendix A. While it is possible to also translate the requirement that a course be taken exactly at a particular time into a relational formula, we instead take advantage of the partial instance functionality of Kodkod and do something simpler—include the (*semester*, *course*) tuple in lower bound of the relation `sCourses`.

Never schedule requirement

We can require that a course not be scheduled to any semester with a simple formula. Again note that the input to this requirement is a course, not a course grouping.

Input:	The course c must not be scheduled to be taken in any semester.
Formula:	`not (c in Semester.sCourses)`

Table 2.8: Never schedule requirement translation

Chapter 3

System Architecture

3.1 Overview

Our scheduler system was written in approximately 6500 lines of Java 1.5 code. It uses the relational constraint solver Kodkod, which is also Java-based, and various external Java libraries for HTML parsing and graphical user interface layout.

There are two versions of the system: a web service and a standalone application; the web service performs the resource-intensive processing on a server. In both versions, the user interacts with the system through a common Swing graphical user interface. In the case of the web service, the interface is part of a Java applet within a web page.

3.2 Key Data Abstractions

We now describe the key data types in the system. Note that many of these types can be serialized into XML. This simplifies remote execution as most code from the standalone version can be reused, and the only things that need to be added are wrappers that serialize, transfer, and deserialize XML (see Section 3.4).

3.2.1 Semester, Course, Attribute

These basic data types are implemented as Java classes and are instantiated to represent atoms in our system's relational data model (described in Section 2.1). The data model defined a fourth basic type—`PrereqSet`. In the implementation, we simply represent a `PrereqSet` atom as a parameterized Java `Set` of `Course` objects.

3.2.2 Schedule

This is a simple hashtable data structure that stores the set of courses allocated to each semester and an ordering over the set of semesters. The class is used to represent both an initial schedule of completed courses provided by the user and a final schedule generated by the system.

3.2.3 CourseGrouping

This class represents the concept of a course grouping in our relational model, as defined in 2.1.1. Each instance of `CourseGrouping` corresponds to one arbitrary set in the data model, and contains its name and the set of `Requirements` that define (constrain) the course grouping.

3.2.4 Requirement

`Requirement` is an interface that must be implemented by all requirement types, whether it be standalone or part of a course grouping definition. The interface contains a single method, `accept()`, for accepting *visitors*, as is common when using the common Visitor design pattern. Each instance of `Requirement` should represent a concrete constraint specified by the user. For instance, the requirement that a course 6.001 must be taken is represented as an instance of the class `MandatoryCourseReq`, which implements `Requirement`.

Different types of requirements should be implemented as distinct subtypes of `Requirement`. For instance, each requirement type listed in the translation rules in

Section 2.2.4 is implemented as a separate class in our system. This allows for a modular requirement translation mechanism, as described in Section 3.3.

3.2.5 DegreeProgram

A `DegreeProgram` is an abstraction for the requirements needed to complete a degree and contains a set of `CourseGrouping` objects (each of which contains a set of `Requirements`). `DegreeProgram` must include a special `CourseGrouping` instance that corresponds to the set of all scheduled courses (the expression `Semester.sCourses` in our relational model).

3.2.6 PrereqMap

This map-like data structure stores the prerequisite sets for each course. Prerequisite sets may be added to a course and retrieved for one in constant expected time.

3.2.7 Problem, Solution

A `Problem` object packages all the requirements (regarding the degree program, prerequisite relationships, and any additional standalone constraints) that will be passed to the backend. Thus, an XML serialization of this type is self-sufficient for archiving a query.

`Solution` is an interface for representing a solution generated by the system and contains methods for accessing the generated schedule and extracting values of individual course groupings and information from other relations (e.g. `prereqsetUsed`—see Section 2.1.).

3.3 Translation Architecture

Translation involves two steps: the allocation of relations and the translation of requirements into the relational model. The output at the end of the translation process

is a relational formula object created using Kodkod's API, as well as upper and lower bound information for the relations that are referenced in the formula.

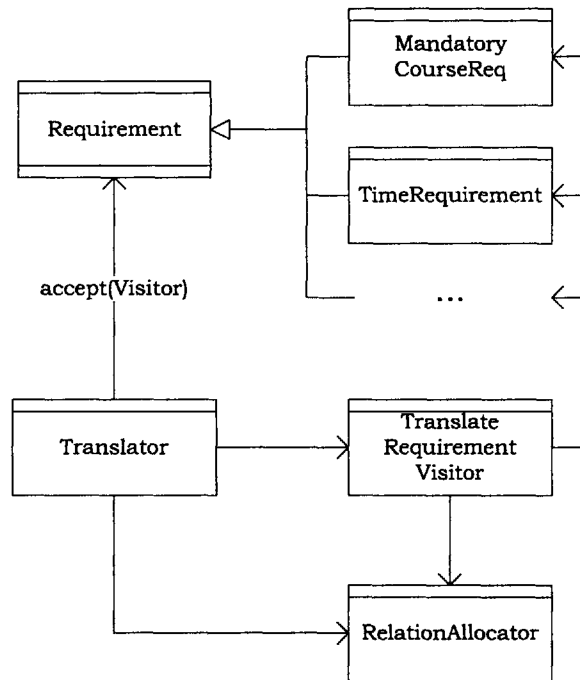


Figure 3-1: Requirement translation dependency diagram.

Figure 3-1 shows a dependency diagram of the classes that perform the translation. The `Translator` class is the central controller for translation and invokes the `RelationAllocator` to allocate relation objects for use in the relational formula. After allocation, the `Translator` creates upper and possibly lower bounds for each relation, in the manner described in Section 2.2.1. Finally, it invokes the class `TranslateRequirementVisitor` to translate each requirement that is encountered, either in a standalone context or in the definition of a `CourseGrouping`; the formula translations of the requirements are conjoined to the global constraints (described in Section 2.2.2) and returned as a single formula. Note that `TranslateRequirementVisitor` depends on `RelationAllocator`, for retrieving references to the allocated relation objects (corresponding to a particular course grouping, for instance).

`TranslateRequirementVisitor` implements the `RequirementVisitor` interface, which contains overloaded `visit()` methods to accept each subtype of `Requirement` as an argument and return a value, in this case a relational formula object representing

the visited requirement.

For instance, the aforementioned `MandatoryCourseReq` requiring 6.001 to be taken would be handled by a method with signature `visit(MandatoryCourseReq)`, which would examine the requirement object, and return the appropriate subset formula.

To support a new requirement type, a developer would simply need to augment `RequirementVisitor` and `TranslateRequirementVisitor` to contain a `visit()` method for the new type. The rest of the translation system can remain unchanged.

3.4 Remote Execution Architecture

The web service version of our system delegates the resource-intensive processing to a remote server; specifically, the translation of a `Problem` into relational logic and the execution of Kodkod run remotely. We use Java *servlets*, hosted on the Apache Tomcat web server, to support remote execution in our system. A Java servlet is a class on the server that can listen to an HTTP connection and process GET/POST requests. The advantage of Java servlets is that they are in pure Java—servlets can easily be integrated with other parts of the system and invoke the same code as a local Java application. This allows the standalone and web service versions of our system to share not only essentially all front-end (user interface) code, but also much of the translation and solving back-end.

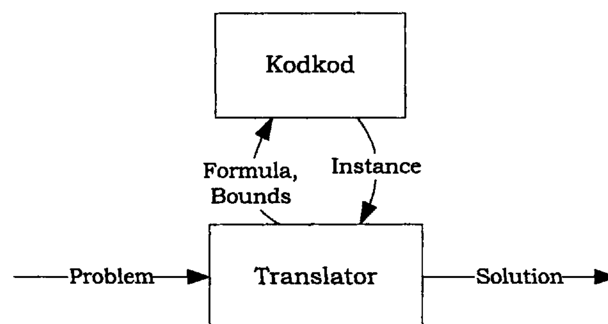


Figure 3-2: Local execution architecture.

To see the differences between the two versions of the system, first consider the architecture for the standalone version (Figure 3-2). A `Problem` object is passed to

the `Translator` class, which invokes Kodkod and converts the returned instance (that is, a satisfying model of tuple assignments to relations) into a `Solution`.

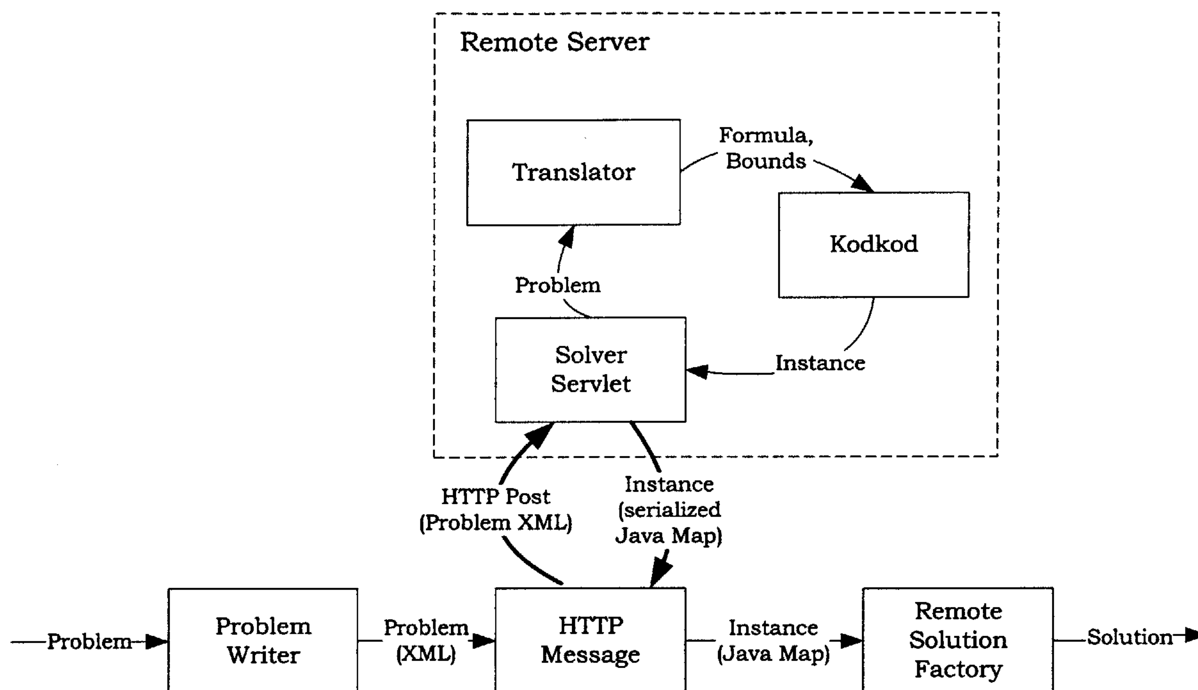


Figure 3-3: Remote execution architecture.

The analogous architecture for the web service is shown in Figure 3-3. The `Problem` object is first converted into XML, and then passed to a HTTP request sender called `HTTPMessage`. An HTTP/POST request is sent to the server and handled by the `SolverServlet` class, which invokes the `Translator` and Kodkod in the same manner as before. The instance returned by Kodkod is converted into a String-based Java Map (that maps from relation names to its tuples) and sent back as an HTTP response. `RemoteSolutionFactory` is run locally on this response and wraps it in the `Solution` interface.

3.4.1 Alternative Design

It would have arguably been more uniform to develop an XML serialization for the `Solution` interface and, for the HTTP response, simply construct an instance of `Solution` remotely and return its XML serialization. The primary disadvantage of

this approach is that the efficiency of a data structure like Java's Map or Kodkod's Instance, which are suitable for handling many retrieval requests, is lost. On the other hand, having XML functionality for `Solution` would have the advantage of enabling the user to record the system's output, but this loss is not very significant because we already support XML serialization of the generated `Schedule`, which is the most important part of a `Solution`.

Chapter 4

Evaluation

4.1 Performance

As our system is meant for practical usage by students, it is imperative that it shows reasonable performance, both in its speed and in the quality of results it returns.

We performed empirical tests with two example degree programs, one for the Bachelor of Science (SB) degree in Electrical Engineering and Computer Science from MIT, and the other for the Master of Engineering (MEng) degree in the same department. Despite its name, the latter is actually a combination degree that includes the former's coursework as a subset, but additionally contains substantially more complicated logic for requiring electives from varying concentrations. With the optimizations described in this section (some of which actually sacrifice running time for more desirable output), the system finds a schedule for the SB program in 4.9 seconds and for the MEng program in 8.6 seconds on a 2.8GHz Intel Pentium 4 machine with 1GB of RAM.[1] Our system can be expected to run in similar times for most other degree programs.

[1]All reported computation times in this chapter were run with a scope of 8 semesters and *no completed courses* on Sun's Java 1.5 JDK for Windows XP. Interestingly, running with a set of completed courses does *not* seem to affect the speed by a noticeable amount.

The JVM was run with a maximum heap size of 128MB. The standalone version of the application was used, but the remote execution overhead for the web version is usually no more than a second. Results were averaged across 5 trials.

4.1.1 Minimum-size Subset Requirements

A big hurdle that we faced was the poor performance of the original minimum-size subset requirement translation, and various alternatives were considered. As discussed in Section 2.2.4, the general case translation for this type of requirement creates a clause for each of the $\binom{n}{k}$ subsets of size k from the n choices. This requirement type is often used to express the notion of "elective courses" in a degree (i.e. courses that are optional individually but of which at least a fixed number must be taken). A typical elective constraint may have n and k of 80 and 2, respectively, and generating $\binom{80}{2} = 3160$ clauses in the formula and translating them into SAT takes a prohibitive amount of time (> 60 seconds for the MEng degree program; the exact range depends on what other optimizations are used in combination.). The translation also requires more memory than usual; both the Java heap and stack sizes needed to be increased to handle the deeply nested conjunction.

As we described previously, if all the choices allowed by the requirement were single-course course groupings, we could use a far more efficient translation involving existential quantifiers. The efficiency of this approach relies on the fact that existentially quantified formulas can be "skolemized" [2]. Normally, when a relational formula is translated into propositional logic for use in a SAT solver, quantified formulas are "grounded out," which is a method used in bounded first order logic analysis to convert a quantified formula into a conjunction (for a universal quantification) or disjunction (for an existential quantification) of propositional clauses. For instance, consider the existential quantified formula `some x:X | P(x)`. Assuming X has an upper bound consisting of the atoms x_0 and x_1, then the quantified formula can be grounded out into `P(x0) || P(x1)`.

Skolemization involves recognizing that we can express the same requirement by explicitly defining a singleton unary relation x' to be a subset of X and including the single propositional clause $P(x')$. If the model finder (solver) can find a model (solution) that satisfies this formula, then the value of x' must be an element of X that satisfies P, and proves the existential formula. If the solver cannot find any

satisfying solution, then it proves the existential formula to be false for the given bounds of X. One can see that the skolemization is constant in n (the size of X) and far more efficient than grounding out, which is linear in n.

Assuming the "solution pruning" method for minimality (described in the next section) is used, changing from the general translation to the existential translation for the single-course case reduces the running time of our system for the SB program from 7.0 seconds to the 4.9 seconds stated above, and the MEng program from essentially unsolvable to the 8.6 seconds from above.

However, note that if an existential quantified formula appears in the context of a negated formula, then it can no longer be skolemized and instead must be grounded out. Local minimality constraints, described in Section 4.1.2, do in fact negate a requirement's translation, so the effect of grounding out is an issue there. We will see that the aforementioned "solution pruning" method attempts to resolve some of these issues.

Finally, instead of existential formulas, if our solver supported primitive handling of set cardinality expressions, then minimum-size subset requirements would be trivial to translate, and should scale just as well as the skolemized existential formulas, without the danger of needing to be grounded out.

4.1.2 Minimality

The issue of minimality proved to be one of the most challenging parts of our system's development. Since we model course scheduling as a satisfiability problem, our system may return a solution that, while satisfying the course requirements, is less desirable than other possible solutions. A common, and the most serious, such type of undesirable solution is one that contains too many courses. In fact, if we do not explicitly try to restrict the size of the generated schedule in our relational formula, our system will almost always return a schedule that has more courses than is practical (e.g. for instance, more than ten per semester).

There are at least four ways in which we could control this problem:

1. Assuming our backend solver allowed us to write expressions involving set car-

dinalities, we could set a hard limit on the size of the set of courses that are scheduled.

2. Without built-in cardinality expressions, we could try to emulate their support with relations (see description below).
3. We could add an optimality constraint to the relational formula so that any generated instance must satisfy the constraint.
4. We could prune an oversized solution to get rid of extraneous courses.

Using Set Cardinality

When we encountered this problem, the first option was out of the question because Kodkod did not yet have support for cardinality expressions. A prototype of the second option was attempted, but, not surprisingly, it proved to be impractical (running for minutes without halting). We created a unary relation of positive integers and a ternary relation of type `Semester->Course->Integer` to index each course scheduled for a semester. We also created a binary `inc` relation between integers to represent their natural ordering. The idea is that we can determine the number of courses assigned to each semester by seeing how "far" its indexing goes. Even if we cap the maximum possible index to a small number like ten, with more than 250 courses and a typical number of 6 semesters, the relation size blows up, and quantified formulas to constrain the indexing relation in useful ways become a bottleneck to translate into SAT. Moreover, even simple expressions containing that relation would be more time consuming to translate than most other expressions.

There is another problem with both of the set cardinality approaches. When given a cap on the number of courses that can be scheduled, it is probable that the system would return solutions that have exactly or close to the maximum size. Thus, depending on what we specify the maximum to be, the solution would still potentially have a significant number of extraneous courses, or worse yet, if we set the maximum to be too small, the system would not be able to find a solution.

An alternative would be to begin with a low maximum value, run the solver, and if a solution could be found, increment the maximum, rerun the solver, and repeat

until one was returned (or use some other searching mechanism, e.g. binary search, to find the correct maximum instead of fixed incrementation). This approach has two primary problems: first, the expected number of searches must be small (e.g. 3) to be comparable in speed to another approach we will describe below, and second, the method must be able to cope with the fact that the lack of a solution can indicate an overly low maximum or instead an overconstraint in other parts of the input (e.g. an unsatisfiable inconsistency in the requirements). The simplest way to handle the latter would be to have the system give up after a fixed number of retries, but that would be undesirable in speed and potentially in correctness.

Using Optimality Constraints

A well-written optimality constraint has the advantage of scheduling few or no extraneous courses. The approach we take is to require *local minimality.* The idea is to ensure that the generated solution cannot be reduced in size by removing any single course in it and still satisfy all the specified requirements. In relational logic, for each non-single-course course grouping G which must satisfy a predicate $P(G)$ (from its definition), we conjoin the following formula to $P(G)$:

```
all c: G | not P(G-c)
```

For the special course grouping corresponding to the entire set of scheduled courses, `Semester.sCourses`, we must do something slightly different because the minimality formula must be aware of the prerequisite constraints put on the schedule denoted by the `sCourses` relation. Assuming the user-defined requirements on `Semester.sCourses` are translated into relational logic as the predicate P, we conjoin the following formula to P(`Semester.sCourses`):

```
all c: (Semester-PastSemesters).sCourses |
  not (P(Semester.sCourses-c) &&
      ((Semester-PastSemesters).sCourses-c).prereqsetUsed.pCourses
        in (Semester.sCourses-c))
```

There are two caveats. First, a local minimality constraint of this type does not always generate minimally-sized schedules. For instance, consider a solution that

used a course X to satisfy a certain degree requirement, when Y could have been used instead. Suppose X had a prerequisite X', and Y had no prerequisites. Then, even though the generated solution was not minimal (X and X' could have been replaced by Y), it would still satisfy local minimality constraints in the above form because removing X would violate the mandatory course requirement, and removing X' would violate the prerequisite requirement of X.

Secondly, the modularity of course grouping translation (as discussed in Section 2.2.4) actually poses a problem in this approach because each course grouping's local minimality is considered independently. A course grouping generally represents a specific requirement in the degree program; often times, two requirements could be satisfied in a synergistic way, but considering the local minimality of the corresponding course groupings separately would not take this possible synergy into account. For instance, suppose there are two course groupings A and B, each of which could be satisfied in a number of ways, but one configuration allowed A's courses to be used as prerequisites for B's courses, thus eliminating the need to schedule extra courses to satisfy B's prerequisites. Our approach would not be aware of this and would only select courses to locally minimize the sizes of A and B.

In spite of these deficiencies, this approach is satisfactory in generating reasonably-sized schedules in practice. It does, however, increase processing time by a significant amount. The reason is that a university quantified formula for local minimality may take a substantial amount of time to ground out. Consider the local minimality formula for `Semester.sCourses` presented above. The expression `(Semester-PastSemesters).sCourses` has an upper bound that is the range of the relation `sCourses` (i.e. the set of all courses). For a typical analysis, there are over 250 courses, and grounding out becomes a significant bottleneck in our analysis. The increase in processing time by adding the minimality constraints is particularly egregious when the existential translation for single-course minimum-size subset requirements is used since, as suggested earlier, the existential formulas would no longer be skolemizable in the context of a negation and thus would take a lot of time to ground out. In fact, whereas running the system for the SB program with no opti-

mizations takes 3.7 seconds, with local optimality constraints, it takes 20.1 seconds; for the MEng case, where there are even more existential quantifiers, the running time increases from 5.4 seconds with no optimizations to over 70 seconds with the local minimality constraints added. It is interesting to note that, once grounded out, the resulting propositional formula takes a negligible amount of time to solve using a SAT solver. (This means that if we were able to use cardinality expressions to translate minimum-size subset requirements, then we would not expect to experience as significant of an increase in running time, as there would be less grounding out needed.)

Pruning an Existing Solution

The next method we propose extensively uses the partial instance functionality of Kodkod and improves in speed compared to the local minimality method above while retaining most of the latter's benefits. The idea is to first run the solver without minimality constraints to receive a bloated solution and then prune this solution.

Clearly, the pruning process must be fast. Pruning is done by executing the solver a second time, now with the local minimality constraints in place for non-single-course course groupings, as above. This may seem counterintuitive, as the whole point of pruning was to avoid the expensive quantification brought about by local minimality constraints. However, the reason we can now reduce the solving time is because the first solution will be used as a partial instance for the second run. Specifically, the tuples of relations corresponding to course groupings and of the binary relation `sCourses` will be used as the upper bounds of those relations in the second trial. In practice, this dramatically reduces the time needed for grounding out because the universe of quantification goes from over 250 courses to approximately 80 for the range of `sCourses` and single-digit sizes for course grouping relations (the latter of which may be the range of an existential quantifier for a minimum-size subset requirement). In fact, the reduction in solving time is so dramatic that with the combination of this pruning technique and the existential translation for minimum-size subset requirements, we achieve the user-friendly solving times noted at the

beginning of this chapter.

There is one subtle drawback of this approach, but we feel that it is outweighed by the benefits. If the input to the system contained standalone requirements (i.e. those that are not part of a course grouping definition), such requirements are not factored into minimality constraints. Consequently, it is conceivable in rare cases that given a partial instance (from the first solution) that satisfies the additional requirements, there is *no way* to reduce all course groupings to locally minimal values (with respect to the groupings' definitions) and still satisfy the standalone requirements. In this case, the system will fail to find a solution, and we would have to either restart from the first step or simply return the unpruned solution.

4.2 Limitations

4.2.1 Efficiently Using Completed Courses

The user of our system can enter a set of courses he has already completed prior to the analysis. Ideally, the system would use these courses to the fullest extent to satisfy requirements.

Unfortunately, the exact behavior of the system is difficult to control, and in fact, when given the choice, the system will sometimes schedule an extra course to satisfy a requirement even when there is an unused completed course that is also applicable.

We could try to add optimality constraints to the relational formula to use completed courses whenever possible, but this turned out to be difficult to do due to two reasons. First, not all completed courses can be used to satisfy requirements, so the constraints must take this into account. Second, the fact that one completed course can be used in multiple ways and that multiple completed courses can be used to satisfy the same requirement make it hard to write optimality constraints in a modular fashion, as we did for minimality constraints.

Fortunately, there are some ways to get around this problem. First, our system allows the user to add new standalone constraints after seeing a solution and can

regenerate a new solution with those constraints included. As described in Section 2.2.4, these include constraints to disallow a course from being scheduled, so it is possible, for instance, for the user to manually exclude extraneous courses this way.

Alternatively, if the user knows exactly how he wishes to satisfy a particular requirement, he can edit the degree requirements input directly and add a requirement to the corresponding course grouping saying that it must be satisfied in a certain way.

4.2.2 Inexact Requirements

In the process of acquiring official MIT degree requirements detailed in Appendix B, we encountered some difficulty in translating certain requirements precisely.

The first difficulty was the imprecise prerequisite specification format on the MIT Student Information System web site. As discussed in Appendix B, the HTML pages generated by that system are marred by inconsistent usage of semicolons, commas, and words such as "or." In order to cope with these inconsistencies, our tool for extracting knowledge from these pages uses a heuristic for decisions in the case of ambiguities. Related to this issue is the notion of "joint" courses; MIT offers many courses that are denoted as interdisciplinary, and such a course will have different course numbers in different departments. For instance, an introductory discrete mathematics course for computer science students has both a computer science department course number (6.042J) and a mathematics department course number (18.062J). Notice that both course numbers end with the letter "J" (for "joint"); this is officially the case for all interdisciplinary subjects, but the online database inconsistently refers to such courses with and without the suffix, and occasionally even requires the same course twice (under different names) in a prerequisite requirement. Although these issues can be dealt with (at least heuristically) in the code after they are discovered, the size of the course catalogue makes discovery difficult, and one cannot be assured that heuristics developed today will continue to work well when different inconsistencies are introduced in the future.

A more serious issue occurs when experimental course offerings are considered. Many departments periodically introduce new experimental courses that are taught

under a temporary course number for that semester; if the course is received well, it will be added to the curriculum and assigned a permanent course number. As a result of this, departments often reuse temporary course numbers for vastly different experimental courses. Then, because courses sharing a number may satisfy a mutually exclusive set of requirements, it is impossible to properly consider such a course in our system unless the user creates a special name for it both in the requirement input and when indicating it as a course that he has taken.

4.2.3 Course Scope

As mentioned previously, the performance of the system is strongly affected by the number of courses that are considered.

In the requirement input files packaged with the system, we have only included the prerequisite relationships for courses in the same department as the degree to be satisfied. This limits the courses considered for scheduling to the department's courses and any course mentioned as prerequisites of the department's courses or in a degree requirement. Thus, using the default input files, the system can potentially schedule a course from another department without also scheduling that course's prerequisites to be taken beforehand.

4.2.4 Numerical Constraints

Our system does not support requirements that involve numerical properties. This includes, for instance, a requirement that a certain number of units must be completed prior to graduation. The ability to handle such constraints would require support of integers in the relational backend; this can be emulated with the creation of an integer type and binary relations from and to the type to denote standard arithmetic operations. However, this approach becomes infeasible when we must deal with expressions that relate an arbitrary set of courses to their numerical properties because the number of tuples in such a relation would be very large (the product of the number of courses and the number of possible integers enumerated by the data model).

Chapter 5

Related Work

Due to its many applications, planning has long been a focus of computer science and artificial intelligence research. Traditionally, planning problems have been solved via deduction (e.g. resolution theorem proving). In the past decade, however, there has been a trend to study planning as a satisfiability problem. In this chapter, we compare our work with other systems that reduce planning to satisfiability and also with another course scheduling system, and show that it demonstrates reasonable performance and usability.

5.1 Planning as Propositional Logic

Planning as satisfiability was arguably popularized by the work of Kautz and Selman [5], who proposed a method of encoding planning problems in propositional logic and showed its feasibility. With the advent of increasingly high-performing SAT solvers, this approach gained popularity, and a number of tools (e.g. Blackbox [6] and MEDIC [3]) have since been created to automatically generate propositional formulas from planning problems specified in a standard format, the most common of which is STRIPS/PDDL [9].

Our system converts a scheduling problem into relational logic, which Kodkod translates into propositional logic, and solves the resulting formula using a SAT solver. However, it differs in two main ways from conventional planners. First, as we had seen

in Chapter 2, using a relational model allows us to easily extract more information than just the schedule from a solution; for instance, we were able to see how a particular requirement was satisfied and which courses were used to satisfy a course's prerequisite simply by evaluating the relevant relations using Kodkod; while it is possible to retrieve the same information from a PDDL plan, one would have to actually look at individual steps of the plan. Secondly, it is more natural to write degree requirements using our system's specification format than it is to write them in PDDL. To illustrate, while creating our experimental PDDL input, we found it difficult to describe prerequisites in such a way that a course could have an arbitrary number of prerequisite courses; instead, we had to limit the size of prerequisite sets to fixed values (e.g. 1, 2, and 3). We also found expressing the notion of semesters to be awkward as a PDDL problem inherently already carries a notion of time.

How does the performance of our system compare to that of a tool like Blackbox?[1] To test this, we created a PDDL problem for the Bachelor of Science degree in Electrical Engineering and Computer Science at MIT using the same data as the counterpart packaged with our system. We modeled taking a course and satisfying a specify requirement as actions in the domain. The details of the PDDL model can be found in Appendix E.

Initially, we found the experience to be more straightforward than expected. We began by ignoring the notion of semesters and simply allowed Blackbox's built-in time step handling to combine parallelizable actions into a single time step whenever possible; each time step can then be interpreted as a semester. This works nicely because if a course A depends on a prerequisite B, then the actions of taking A and B would not be parallelizable, and thus would be performed in different time-steps and be interpreted by us to be in different semesters, as expected. With the no-semester simplification, Blackbox was able to find highly optimal (small in the number of courses) solutions in just 2 seconds, whereas our system took around 5

[1] Blackbox was the immediate predecessor (2003) to the award-winning SatPlan2004 tool at the 14th International Conference on Automated Planning and Scheduling; we had problems executing the latter on our machine, and do not expect the general trend of results we describe in this section to differ if the latter were used.

seconds (Chapter 4.1).[2]

However, as soon as we included the notion of semesters in the problem (in preparation for expressing the course/semester attribute constraints handled by our system), Blackbox's performance dropped immediately. It aggressively allocated memory (up to around 500MB within a few seconds) and took roughly 23 seconds to find a solution. Thus, at least for the specific domain of planning problems we are trying to solve, our system is competitive with or outperforms Blackbox.

5.2 Planning as Temporal Logic

Instead of translating into propositional logic, some systems model planning as a satisfiability problem in linear time logic and use temporal logic model finders to search for solutions. PADOK [8] is one such system and claims to be competitive with the best propositional planners in most domains. However, its performance appears to rely on the use of external "hints" (that is, constraints, e.g. global invariants, that do not affect the satisfiability of the problem but facilitate the search by eliminating bad paths), so the tool may not be suitable for our problem domain. Nonetheless, a benefit of using temporal logic is that we expect to be able to express course prerequisite and semester constraints naturally in PADOK, unlike in PDDL.

5.3 Other Course Planners

There has been at least one other effort to build a course planning system for MIT students. Ross Glashan and John Rebula's *Classifier* website (http://classifier.mit.edu) allows users to add courses to a schedule and see which degree requirements are satisfied as a result. On the surface, Classifier has some similarities to our system, but in fact, they perform two different tasks. Classifier verifies if a given schedule satisfies requirements, while our system focuses on generating a schedule to satisfy requirements.

[2]Computations were performed on a 2.8 GHz Intel Pentium 4 machine with 1GB of RAM, running cygwin/Microsoft Windows XP.

Appendix A

Input Specification Format

In this appendix, we will describe the manner in which requirements are specified as input to our system. We could have designed this in at least three different ways: using a general-purpose planning language like PDDL [9], arbitrary first-order logic formulas, or a domain-specific language for course requirements. Arbitrary first-order logic constraints (as allowed in [1]) may require too much expertise from the user of our system, who may be unfamiliar with discrete mathematics. On the other hand, general-purpose formats for specifying planning problems can be cumbersome to use for this problem domain (see Appendix E, in which we attempt this).

Our system accepts and translates both user-independent and user-defined requirements. User-independent requirements are the official course requirements specified by the school or department. Our system also handles additional user-defined constraints; for instance, the user can indicate that he has already taken certain courses or that a specific course should be scheduled at a particular time. In the following sections, we will describe the format for specifying the following requirement categories:

1. Official course requirements (required)
2. User-specified partial schedule (optional)
3. Other user-specified constraints (optional)

The specification of official course requirements appears in a `<degreeProgram>` XML element and is a required input to our system. A partially defined schedule

appears in a `<schedule>` XML element, while any other constraints appear in a `<additionalReqs>` element. This format separates official requirements from user-defined ones. The three categories can appear in separate XML files or be packaged in one combined file under a `<problem>` element, which must contain the three sub-elements in the given order.

The XML Document Type Definition (DTD) for our input format is available at http://people.csail.mit.edu/vshyeung/problem.dtd.

A.1 Course Requirement Specification

We divide official course requirements into two categories: course prerequisites and the set of courses required for a degree; we will shorten the latter as "degree requirements." All of this information is stored in the `<degreeProgram>` element, whose DTD is as follows:

```
<!ELEMENT degreeProgram (courses, prereqs, groupings)>
```

`<courses>` contains the definitions of courses (name and attribution information) referred to in this degree program. `<prereqs>` contains all the prerequisite relationships among the courses, and `<groupings>` contains the course groupings that are used to model the degree requirements.

A.1.1 Course Definitions

Before specifying the course requirements, however, courses must be defined. A course definition provides its name and any *attributes* the course contains. Valid attributes include `Odd` (for an odd numerical year, e.g. 1991, 2005), `Even`, `Fall` (for a Fall semester), and `Spring`. Additional attributes may be defined if the user creates a custom schedule file (Section A.2) and defines them in its semesters. The DTD for course definition-related tags are as follows:

```
<!ELEMENT courses (courseDef)*>
<!ELEMENT courseDef (attrib)*>
```

```
<!ATTLIST courseDef name CDATA #REQUIRED>
<!ELEMENT attrib (#PCDATA)>
```

A <courses> element contains all the course definitions. Each <courseDef> element defines a single course (by giving its name in the required name XML attribute of the element) and lists its course attributes, which are strings taking on one of the four values above (or a custom attribute defined in the schedule).

The following is an example of a course with two attributes. This definition states that 6.856 is offered only in the semesters Spring 2005, Spring 2007, etc.

```
<courses>
  <courseDef name="6.856">
    <attrib>Spring</attrib>
    <attrib>Odd</attrib>
  </courseDef>
<courses>
```

If during later processing (e.g. while parsing requirements), the system encounters a course not defined in the <courses> element, it will create a course with that name with no attributes.

A.1.2 Prerequisites

Each course has zero or more prerequisite sets. A prerequisite is a set of courses that together can satisfy the parent's prerequisite requirement. A prerequisite set can have one or more members.

The DTD for the prerequisite-related tags are as follows:

```
<!ELEMENT prereqs (prereq)*>
<!ELEMENT prereq (course, prereqSet+)>
<!ELEMENT prereqSet (course)+>
<!ELEMENT course (#PCDATA)>
```

The <prereqs> element contains all prerequisite requirement specifications. Each course's prerequisite specification is contained on a <prereq> element. An individual prerequisite set's members are enumerated in a prereqSet requirement. A course is an element identified by name.

For example, consider the course 6.002, which the prerequisites of 8.02 or 8.022, and 18.03 or 18.06. This can be written as follows:

```
<prereqs>
  <prereq><course>6.002</course>
    <prereqSet><course>8.02</course><course>18.03</course></prereqSet>
    <prereqSet><course>8.02</course><course>18.06</course></prereqSet>
    <prereqSet><course>8.022</course><course>18.03</course></prereqSet>
    <prereqSet><course>8.022</course><course>18.06</course></prereqSet>
  </prereq>
<prereqs>
```

A course with no prerequisites should not have a `<prereq>` entry.

A.1.3 Degree Requirements

Central to our input specification format for degree requirements is the notion of a "course grouping." As detailed in Section 2.1.1, a course grouping is an arbitrary set of courses whose exact value is constrained by set of requirements in the grouping's definition.

A course grouping is often used to represent the courses used to satisfy one particular degree requirement. For instance, consider the requirement that two of the three computer science headers (6.033, 6.034, and 6.046) must be taken. We can define a course-grouping named "cs-headers" and constrain the value of the set with a minimum-size subset requirement, described below, such that it must include two of those three classes. The DTD for course groupings is as follows:

```
<!ELEMENT grouping (mandatoryCourseReq | maxSizeSubsetReq |
minSizeSubsetReq | noOverlapReq)* >
<!ATTLIST grouping name CDATA #REQUIRED>
```

A course grouping is defined in a `<grouping>` element which has a required `name` attribute. The course grouping can have zero or more requirement elements in its definition.

For instance, the example mentioned above can be written as follows:

```
<grouping name="cs-headers">
  <minSizeSubsetReq>
    <size>2</size>
    <subset>
      <member>6.033</member>
      <member>6.034</member>
      <member>6.046</member>
    </subset>
  </minSizeSubsetReq>
</grouping>
```

Mandatory Course Requirement

A mandatory course requirement states that specific courses and/or course groupings must appear in the subject course grouping (i.e. the one that this requirement defines). The DTD for a mandatory course requirement is as follows:

```
<!ELEMENT mandatoryCourseReq (member)+>
<!ELEMENT member (#PCDATA)>
```

A `<mandatoryCourseReq>` element must contain one or more `<member>` elements. Each `<member>` must contain as its text the string name of a course or a previously defined grouping. Any unrecognized name is assumed to be a course with no attributes.

The following is a simple example requiring the courses 6.001 and 6.002.

```
<mandatoryCourseReq>
  <member>6.001</member>
  <member>6.002</member>
</mandatoryCourseReq>
```

Minimum-Size Subset Requirement

This requirement is used to require that at least a certain number of choices from a selection of course groupings and/or courses must be present in the subject. We saw an example of this in the header requirement above.

The DTD for a minimum-size subset requirement is as follows:

```
<!ELEMENT minSizeSubsetReq (size, subset)>
<!ELEMENT size (#PCDATA)>
```

where the `<member>` element is the same as for mandatory course requirements, and the `size` element must contain as its text a positive integer no larger than the number of `<member>` elements in this requirement.

No Overlap Requirement

A no overlap requirement states that the course groupings and/or courses it specifies must not overlap with each other or with the subject. Its DTD is as follows:

```
<!ELEMENT noOverlapReq (member)+>
```

This requirement is often used to model the notion that two requirements cannot be satisfied by the same course. For instance suppose that in addition to the `cs-headers` grouping defined earlier, there is an elective course requirement that can be satisfied by a header course or one of several other subjects, but the same course cannot be used to satisfy both the header and elective requirements.

Then, we might define the `cs-elective` grouping as follows:

```
<grouping name="cs-elective">
  <noOverlapReq>
    <member>cs-headers</member>
  </noOverlapReq>
  <minSizeSubsetReq>
    <size>1</size>
    <subset>
      <member>6.033</member>
      ... (other headers and electives)
    </subset>
  </minSizeSubsetReq>
</grouping>
```

A.2 Partial Schedule Specification

Our system also optionally accepts as input a partial schedule—that is, a pre-defined assignment of courses to past and/or future semesters. All past semesters are com-

bined as one "PastSemesters" abstraction, as the system need not know exactly when a course was completed in the past.

A partial schedule can also serve a second purpose—to define semester attributes. If a partial schedule is not specified, the requirement writer is limited to the default attributes specified in A.1.1. The DTD for schedule-related tags is as follows:

```
<!ELEMENT schedule (semester)+>
<!ELEMENT semester ((attrib)*,(course)*)>
<!ATTLIST semester name CDATA #REQUIRED>
```

A `schedule` element contains a series of `semester` elements, each of which has a required `name` attribute, and zero or more attributes (in the same `<attrib>` element format as before) and courses (in the same `<course>` format as before). Note that the order in which the `<semester>` elements are specified will determine the time ordering of the semesters. The special semester name `PastSemesters` should be used only for that purpose and, if present, must appear as the first semester specified.

A.3 Additional Constraints

Additional constraints that are not part of a grouping definition may be specified. These are generally constraints added by the user to tweak the output of the system.

A.3.1 Time Requirement

A time requirement is used to require that a course be scheduled at, before, or after a semester. Note that this implies that the specified course must be scheduled. The DTD for a time requirement is as follows:

```
<!ELEMENT timeReq EMPTY>
<!ATTLIST timeReq course CDATA #REQUIRED>
<!ATTLIST timeReq operator (BEFORE|AT|AFTER) #REQUIRED>
<!ATTLIST timeReq semester CDATA #REQUIRED>
```

The requirement is specified with a `<timeReq>` element, which must contain as attributes the name of the course that is to be scheduled, one of the three operators

BEFORE, AT, or AFTER, and the name of the semester to which this operator is relative.

Here is an example of a time requirement, which requires that the course 6.UAT be scheduled after Fall 2006 (exclusive).

```
<timeReq course="6.UAT" operator="AFTER" semester="Fall 2006" />
```

A.3.2 Never Schedule Requirement

A never schedule requirement does precisely as its name implies—it will force the system to not schedule a particular course. The DTD for a never schedule requirement is as follows:

```
<!ELEMENT neverScheduleReq EMPTY>
<!ATTLIST neverScheduleReq course CDATA #REQUIRED>
```

The requirement is specified by a simple `neverScheduleReq` element with a required attribute containing the name of the course to exclude.

For example, to exclude the course 6.011, one can write:

```
<neverScheduleReq course="6.011" />
```

Appendix B

Requirement Knowledge Acquisition

In this chapter, we describe the method used for extracting official MIT course requirements. The extracted requirements have been encoded in our system's input specification format (Appendix A).

B.1 Prerequisite Information Acquisition

MIT posts its course catalogue at the Student Information System website (http://student.mit.edu). The course catalogue has entries such as the following:

```
6.002 Circuits and Electronics
------
Undergrad (Fall, Spring) Rest Elec in Sci & Tech
Prereq: 8.02 or 8.022; 18.03 or 18.06
Units: 4-2-9
Lecture: TR11 (10-250) Lab: TBA Recitation: WF9 (26-302) or WF10
(26-302, 4-265) or WF11 (26-310, 4-265) or WF12 (26-210, 26-310) or
WF1 (26-210) or WF2 (34-302) or WF3 (34-302) +final
------
Fundamentals of the lumped circuit abstraction. Resistive elements and
networks; independent and dependent sources; switches and MOS
devices; digital abstraction; amplifiers; and energy storage elements.
Dynamics of first- and second-order networks; design in the time and
frequency domains; analog and digital circuits and applications.
```

```
Design exercises. Alternate week laboratory. 4 Engineering Design
Points.  A. Agarwal, J. H. Lang
```

We wish to automatically extract the prerequisite relationships from the "Prereq:" line in each entry. We have written a Java class `WebsisPrereqHTMLParser` to do this. The format in which this line is written, however, is inconsistent. While the delimiters "," and "or" generally imply disjunction and the delimiters ";" and "and" generally stand for conjunction, sometimes "," is used for conjunction, as in "X or Y, Z." In addition, some entries are written in plain English, with the use of words such as "either," "recommended," or even "permission required." Our parser ignores unrecognizable words as conjunctive delimiters.

There is also an inconsistent standard for referring to "joint" courses, i.e. an interdisciplinary course that has multiple course numbers under multiple departments. Some prerequisite requirements will list all versions of a joint course (often using the "/" delimiter), while others do not. Some will include the joint course suffix "J" at the end of the course number (e.g. 6.042J), while others do not. The parser currently strips off all suffixes, and simply treats "/" as a disjunctive delimiter.

Another issue concerns introductory mathematics and natural sciences subjects. At MIT, multiple versions of freshman calculus, physics, chemistry, and biology are offered, and they have different course numbers (e.g. 18.024 versus 18.02). Officially, these courses are interchangeable with one another for fulfilling most requirements, but listed prerequisite requirements usually only include a subset of the possible choices. Our system takes the input as is and does not attempt to infer that there are additional possible choices.

B.2 Degree Requirements Acquisition

Requirements for different degree programs can be obtained from the MIT bulletin (http://web.mit.edu/catalogue). Unfortunately, the wide discrepancy between requirements for different majors makes automatic extraction difficult, and we find it

best to simply encode the degree requirements by hand. Worse yet, some majors have a large number of elective courses, and the aforementioned website does not usually explicitly list the set of possible electives, and we have to search for them on the department's own page.

For electives in the Electrical Engineering and Computer Science department, we located the sets of electives in the different concentrations (e.g. Theoretical Computer Science, Artificial Intelligence as comma-delimited lists at the department's website, and wrote a simple tool that converts the lists into XML tags for our system's input format.

Sometimes the translation can be tricky. For instance, the MEng degree program in EECS requires students to take headers (introductory subjects) and electives in three different concentrations. For one of these three concentrations, the so-called "big header" concentration, at least two electives must be taken (in addition to the header), while for each of the other two concentrations, labeled as "small header," one elective must be taken. In addition to these seven courses, two other electives or headers from any concentration must be taken. We model this program using multiple course groupings to represent the different possible ways (concentrations) to satisfy a big and small header requirement, and then compose them in a minimum-size subset requirement with "chosen-big-header" and "chosen-small-headers" groupings. The remaining two electives are stored in a separate course grouping and is simply required to not overlap with the former two.

In short, the translation of degree requirements entail more manual effort than prerequisite relationships, but the task is generally straightforward.

Appendix C

User Manual

C.1 Quick Start Guide

Visit our system's home page at http://optima.csail.mit.edu/scheduler. Click on the top link to invoke the Java applet. You will need to have Java 1.5 installed. When prompted for security permissions, select "Yes"—this enables the system to load and save XML input files on your file system.

The simplest way to generate schedules in our system is the following:

1. Select the time scope of the analysis by selecting a semester in the "to Semester" combo box.
2. Press Ctrl+L to load one of the included degree programs.
3. If you wish to include a set of completed courses, press Ctrl+R to load a saved HTML copy of your WebSIS grade report.
4. Click on "Create Schedule" (or press "Return" on your keyboard).

C.2 Detailed Guide

C.2.1 Invoking the System

Our system requires Java 1.5 or above. The web version has no practical memory requirement, as the core processing is done on a server, but the standalone version needs roughly 128MB of memory to function properly.

Web Version

Follow the link on http://optima.csail.mit.edu/scheduler to invoke the Java applet, and select "Yes" when prompted for security permissions.

Standalone Version

Follow the link in http://optima.csail.mit.edu/scheduler to invoke the application via Java Web Start, and select "Yes" when prompted for security permissions. The appropriate JAR files will automatically be downloaded to your system.

C.2.2 GUI Overview

Both versions of our system have the same user interface, which consists of the following parts:

- A menu bar at the top.
- A toolbar below the menu bar.
- A main schedule panel, which contains a box for each semester included in the analysis.
- An additional requirements panel at the bottom for displaying additional user-defined constraints.
- A course grouping panel at the upper right, which can be used to view the course groupings defined in a degree program and the groupings' values after a solution has been found.
- A prerequisite viewing panel, which shows the prerequisite relationships between the classes defined in the degree program.

C.2.3 Loading Input

To begin an analysis, you must first load a degree program. Once one has been loaded, the course grouping panel will show the course groupings defined in the program, and the prerequisite panel will show the prerequisites of all defined courses in a tree structure.

C.2.4 From a Degree Program

To load from a degree program, press Ctrl+L or select File→Load Degree Program from the menu. In the option dialog that appears, you can select a pre-packaged degree program file (whose contents are available on our homepage), or select "Load custom..." from the list to load your own XML file. Press OK to proceed.

C.2.5 From a Problem File

A problem file contains a degree program, a partially completed schedule (which also indicates the semester time scope of the analysis, thus bypassing the "to Semester" option on the GUI), and any additional constraints specified by the user. It allows the most control over the analysis that is performed.

To load from a problem file, press Ctrl+P or select File→Load Complete Problem from the menu.

C.2.6 Creating a Partial Schedule from a Grade Report

MIT students can access their internal grade report at https://student.mit.edu/cgi-bin/shrwsgrd.sh. Our system can automatically load in the HTML output from this page (which you will need to save onto a file) to fill in the completed courses set in "PastSemesters."

Before you load a grade report, you must have already loaded a degree program (or a problem file, in which case its "PastSemesters" input would be discarded). Then, press Ctrl+R or select File→Load Grade Report from the menu and select a file containing your HTML grade report.

C.2.7 Saving a Query

All information about a query (including all additional user-specified constraints, but NOT including the results) can be saved in a problem file, so that the exact same query can be executed again at a later time.

To save a query, press Ctrl+S or select File→Save Problem from the menu.

C.3 Generating a Schedule

Once input has been successfully loaded, you will see that the course grouping panel at the upper right and the schedule panel will be populated. Now, to generate a schedule, press "Return" or click on the "Create Schedule" button below the menu bar.

C.3.1 Viewing and Tweaking a Solution

Once a solution is generated, each course appears as a row in its respective semester's box on the schedule panel. The course grouping panel will show the value of each course grouping if you expand the corresponding node on the tree. Expanding the tree in the prerequisite panel will allow you to see how a particular course's prerequisites were satisfied.

Keeping a Course

If you like the semester for which a particular course has been scheduled, you can force the system to keep the course there on subsequent reruns by clicking the pin-like button next to the course.

Moving a Course

If you want to change when a scheduled course should appear on subsequent reruns, press the course's label (be sure to put you mouse pointer over the course number text), and drag the course to another semester's box. Note that courses cannot be moved to "PastSemesters." The moved course will automatically be "pinned."

Requiring a Course to Appear Before/After a Semester

You can force a course to be scheduled before or after a certain time by clicking the "B" or "A" button, respectively, next to the course. A dialog will appear to allow you to choose the semester that the course should appear before/after. This constraint will appear in the additional requirements panel at the bottom.

Excluding a Course

A course can be explicitly excluded from future schedules by clicking the cross mark next to it. This constraint will appear in the additional requirements panel at the bottom.

Removing an Additional Requirement

You can remove an additional requirement appearing in the requirements panel at the bottom simply by clicking the "Remove" buttons next to it. You can remove a requirement to keep a course simply by clicking the pin-like button again. Likewise, you can remove a requirement to exclude a course by clicking the cross mark again.

Appendix D

Complete Example

We include in this appendix a full model of relational formulas generated by our system, presented in Alloy-like syntax. The translated requirements are those for the Bachelor of Science program in Electrical Engineering and Computer Science at MIT. The translation mechanism used to create the model is described in Chapter 2. The minimality constraints from Chapter 4 were not included in this model, as they are not used during the system's first execution.

Many of the relations in the model were pre-set using a partial instance, which is not evident by looking at the formulas. The complete list of such relations can be found in Section 2.2.1.

Single-course course groupings are prefixed with `cg`, and the dot in the course number is removed to avoid ambiguity with the binary join operator (e.g. `cg6001` for 6.001). In Alloy, a signature would need to be created for each course grouping before it can be used, but we will ignore that detail here.

```
module scheduler

sig Attribute {}

sig PrereqSet {
  pCourses : Course
}

sig Course{
```

```
  prereqs : PrereqSet,
  prereqsetUsed : one PrereqSet,
  cAttributes : Attribute
}

sig Semester {
  sCourses : Course,
  next, prev : Semester,
  sAttributes : Attribute
}


/**********************
Translated requirements
***********************/

sig math in Course {}{
  some v1 : (cg6041+cg6042+cg18440) |
    v1 in math
}

sig core in Course {}{
  cg6001 in core && cg6002 in core &&
  cg6003 in core && cg6004 in core &&
  cg1803 in core
}

sig project in Course {}{
  cg6UAT in project && cg6UAP in project
}

sig csHeader in Course {} {
  some v1, v2 : (cg6033+cg6034+cg6046) |
    (no v1&v2) && ((v1+v2) in csHeader)
}

sig eeHeader in Course {} {
  some v1, v2 : (cg6011+cg6012+cg6012+cg6021) |
    (no v1&v2) && ((v1+v2) in eeHeader)
}

sig bioLab in Course {}{
  cg6021 in bioLab && cg6022 in bioLab
}
```

```
sig lab in Course {}{
  cg6101 in lab || cg6111 in lab || cg6115 in lab ||
  cg6121 in lab || cg6131 in lab || cg6142 in lab ||
  cg6151 in lab || cg6161 in lab || cg6163 in lab ||
  cg6170 in lab || cg6171 in lab || bioLab in lab
}

sig elective in Course {} {
  no (csHeader & elective) &&
  no (eeHeader & elective) &&
  no (csHeader & elective) // extraneous, but in our translation rules

  some v1 : (cg6.011+cg6.012+cg6.013+cg6.021+cg6.033+cg6.035+cg6.805+
             cg6.821+cg6.823+cg6.824+cg6.826+cg6.827+cg6.828+cg6.829+
             cg6.830+cg6.831+cg6.846+cg6.857+cg6.872+cg6.883+cg6.884+
             cg6.893+cg6.894+cg6.034+cg6.801+cg6.803+cg6.804+cg6.807+
             cg6.825+cg6.831+cg6.833+cg6.834+cg6.836+cg6.837+cg6.838+
             cg6.839+cg6.863+cg6.866+cg6.867+cg6.868+cg6.869+cg6.871+
             cg6.872+cg6.873+cg6.874+cg6.877+cg6.881+cg6.882+cg6.891+
             cg6.892+cg6.946+cg6.046+cg6.045+cg6.251+cg6.336+cg6.337+
             cg6.338+cg6.339+cg6.840+cg6.841+cg6.844+cg6.851+cg6.852+
             cg6.854+cg6.855+cg6.856+cg6.859+cg6.875+cg6.885+cg6.895+
             cg6.896+cg18.433) |
    v1 in elective
}

pred sCoursesPred() {
  math in Semester.sCourses &&
  csHeader in Semester.sCourses &&
  eeHeader in Semester.sCourses &&
  project in Semester.sCourses &&
  lab in Semester.sCourses
  elective in Semester.sCourses
}

/*****************
Global constraints
*****************/

pred attributesMatch() {
  all s : Semester - PastSemesters |
    s.sCourses.cAttributes in s.sAttributes
}

pred noOverlap() {
```

```
  all s1 : Semester | all s2 : (Semester - s1) |
    no (s1.sCourses & s2.sCourses)
}

pred noSemesterSkipping() {
  all s1 : (Semester - PastSemesters) |
    all s2 : (s1.^prev - PastSemesters) |
      some s1.sCourses => some s2.sCourses
}

pred prereqConstraint() {
  prereqsetUsed.pCourses in ~sCourses.^prev.sCourses
}

fact {
  sCoursesPred() && attributesMatch() && noOverlap &&
  noSemesterSkipping && prereqConstraint()
}
```

Appendix E

PDDL Example

PDDL input is specified in two files: a domain file that describes the problem and a problem file that instantiates the objects in a particular problem and states the starting and ending conditions.

The following is the domain file after we added the notion of semesters (as discussed in Section 5.1), which turned out to perform rather poorly. Removing the predicate `current-semester` from the preconditions of the `Take-Course-X` actions will speed up the analysis drastically. Because we could not practically handle the notion of semesters, semester and course attribute constraints were not included in the PDDL model.

Overview of model:

- Taking a course is modeled as an action whose precondition is that the necessary prerequisites are fulfilled. Multiple course taking actions have been created because a course's prerequisite requirement can potentially be satisfied with any number of prerequisite courses.
- Simple mandatory course requirements are stated as part of the end goal in the problem.
- Satisfying a non-trivial requirement (e.g. ones that required minimum-size subset logic) is modeled as an action in the domain. When all of these are satisfied, the predicate `degree-finished` can be changed to true.

```
(define (domain course)
```

```
(:requirements :strips :equality :typing)
(:types course person semester)
(:predicates (noprereq ?c - course)
             (prereq1 ?p1 ?c - course)
             (prereq2 ?p1 ?p2 ?c - course)
             (prereq3 ?p1 ?p2 ?p3 ?c - course)
             (prereq4 ?p1 ?p2 ?p3 ?p4 ?c - course)
             (ee-header ?e - course)
             (take-ee-headers ?e1 - course ?e2 - course ?p - person)
             (cs-header ?c - course)
             (take-cs-headers ?c1 - course ?c2 - course ?p - person)
             (math ?c - course)
             (take-math ?c - course ?p - person)
             (eecs-elective ?c - course)
             (take-eecs-elective ?c - course ?p - person)
             (degree-finished ?p - person)
             (taken ?c - course)
             (current-semester ?s - semester)
             (next-semester ?s1 ?s2 - semester)
             )

(:action Take-Course-0
   :parameters (?c - course ?s - semester)
   :precondition (and (noprereq ?c)
      (current-semester ?s))
   :effect (and (taken ?c)))

(:action Take-Course-1
   :parameters (?p1 ?c - course ?s - semester)
   :precondition (and (prereq1 ?p1 ?c)
                         (current-semester ?s)
                         (taken ?p1)
      )
   :effect (taken ?c))

(:action Take-Course-2
   :parameters (?p1 ?p2 ?c - course ?s - semester)
   :precondition (and
                         (prereq2 ?p1 ?p2 ?c)
                         (current-semester ?s)
                         (taken ?p1)
                         (taken ?p2)
      )
   :effect (taken ?c))
```

```
(:action Take-Course-3
   :parameters (?p1 ?p2 ?p3 ?c - course ?s - semester)
   :precondition (and
                               (prereq3 ?p1 ?p2 ?p3 ?c)
                               (current-semester ?s)
                               (taken ?p1)
                               (taken ?p2)
                               (taken ?p3)
      )
   :effect (taken ?c))


(:action Take-Course-4
   :parameters (?p1 ?p2 ?p3 ?p4 ?c - course ?s - semester)
   :precondition (and
                               (prereq3 ?p1 ?p2 ?p3 ?p4 ?c)
                               (current-semester ?s)
                               (taken ?p1)
                               (taken ?p2)
                               (taken ?p3)
                               (taken ?p4)
      )
   :effect (taken ?c))


(:action Take-EE-Headers
   :parameters (?e1 ?e2 - course ?p - person)
   :precondition (and (taken ?e1)
                               (taken ?e2)
                               (ee-header ?e1)
                               (ee-header ?e2))
   :effect (and (take-ee-headers ?e1 ?e2 ?p)))

(:action Take-CS-Headers
   :parameters (?c1 ?c2 - course ?p - person)
   :precondition (and (taken ?c1)
                               (taken ?c2)
                               (cs-header ?c1)
                               (cs-header ?c2))
   :effect (and (take-cs-headers ?c1 ?c2 ?p)))

(:action Take-EECS-Elective
   :parameters (?c - course ?p - person)
   :precondition (and (taken ?c)
                               (eecs-elective ?c))
```

```
    :effect (and (take-eecs-elective ?c ?p)))

  (:action Take-Math
    :parameters (?c - course ?p - person)
    :precondition (and (taken ?c)
                       (math ?c))
    :effect (and (take-math ?c ?p)))


  (:action Finish-Degree
    :parameters (?c1 ?c2 ?e1 ?e2 ?el ?m  - course ?p - person)
    :precondition (and (take-ee-headers ?e1 ?e2 ?p)
                       (take-cs-headers ?c1 ?c2 ?p)
                       (take-eecs-elective ?el ?p)
                       (take-math ?m ?p))
    :effect (degree-finished ?p))

  (:action Next-Semester
    :parameters (?s1 ?s2 - semester)
    :precondition (and (next-semester ?s1 ?s2)
                       (current-semester ?s1))
    :effect (and (not (current-semester ?s1))
                 (current-semester ?s2)))
)
```

The problem file instantiates the objects and specifies start and end conditions. The following are excerpts from the problem file we used:

```
(define (problem prob01)
    (:domain course)
    (:objects

spring06 - semester
fall06 - semester
spring07 - semester
fall07 - semester
spring08 - semester
fall08 - semester
spring09 - semester

6.002 - course
6.003 - course
6.004 - course
```

```
6.011 - course
[... removed for brevity]

bob - person
)

(:init
(ee-header 6.011)
(ee-header 6.012)
(ee-header 6.013)
(ee-header 6.021)

(cs-header 6.033)
(cs-header 6.034)
(cs-header 6.046)

(math 6.041)
(math 6.042)
(math 18.440)

(current-semester spring06)
(next-semester spring06 fall06)
(next-semester fall06 spring07)
(next-semester spring07 fall07)
(next-semester fall07 spring08)
(next-semester spring08 fall08)
(next-semester fall08 spring09)

(eecs-elective 6.011)
(eecs-elective 6.012)
(eecs-elective 6.013)
(eecs-elective 6.021)
(eecs-elective 6.033)
[... removed for brevity]

(prereq2 8.02 18.03 6.002)
(prereq2 8.02 18.06 6.002)
(prereq2 8.022 18.03 6.002)
(prereq2 8.022 18.06 6.002)
(prereq1 6.002 6.003)
(prereq2 6.001 6.002 6.004)
(prereq2 6.003 6.041 6.011)
(prereq2 6.003 18.440 6.011)
(prereq3 8.02 18.03 6.003 6.012)
(prereq1 6.003 6.013)
```

```
(prereq3 2.003 8.02 18.03 6.021)
(prereq3 6.002 8.02 18.03 6.021)
(prereq3 6.071 8.02 18.03 6.021)
(prereq3 10.301 8.02 18.03 6.021)
[... removed for brevity]
(noprereq 8.02)
(noprereq 18.03)
(noprereq 18.06)
[... removed for brevity]
)

(:goal (and
    (taken 6.001)
    (taken 6.002)
    (taken 6.003)
    (taken 6.004)
    (taken 18.03)
    (taken 6.UAP)
    (taken 6.UAT)
    (degree-finished bob)
))
)
```

Bibliography

[1] Marco Baioletti, Stefano Marcugini, and Alfredo Milani. An Extension of SAT-PLAN for Planning with Constraints. In *AIMSA*, pages 39–49, 1998.

[2] Daniel Jackson. Automating First-Order Relational Logic. In *Foundations of Software Engineering*, pages 130–139, 2000.

[3] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic sat-compilation of planning problems. In *IJCAI*, pages 1169–1177, 1997.

[4] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A Micromodularity Mechanism. In *ESEC / SIGSOFT FSE*, pages 62–73, 2001.

[5] Henry A. Kautz and Bart Selman. Planning as Satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[6] Henry A. Kautz and Bart Selman. Unifying SAT-based and Graph-based Planning. In *IJCAI*, pages 318–325, 1999.

[7] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An Efficient SAT Solver. In *SAT (Selected Papers)*, pages 360–375, 2004.

[8] Marta Cialdea Mayer, Andrea Orlandini, Giulio Balestreri, and Carla Limongelli. A Planner Fully Based on Linear Time Logic. In *AIPS*, pages 347–354, 2000.

[9] Drew McDermott. PDDL — the Planning Domain Definition Language, 1998.

[10] Emina Torlak and Daniel Jackson. The Design of a Relational Engine. In *Foundations of Software Engineering*, 2006. To appear.