Formal Methods of Software Development

Approach to the CB-CTT Problem Using SMT

Estevo Cordal and Francisco Adrian Garcia
UVT

February 6, 2021

Contents

1. Motivation
2. Solving the problem
   - 2.1 Creating the variables
   - 2.2 Encoding the constraints
3. Solution
4. Final Thoughts

Motivation

An on-going problem

Many high schools and universities **still** make their timetables manually.

The **CB-CTT** (Curriculum-Based Course Timetabling) is a widely-known problem, many schools, universities and other services have to yearly deal with how to organize their schedules for students and employees. Hence, it is a big challenge for us to get a closer solution to this matter. We will use the knowledge acquired during Formal Methods in Software Development course for this purpose, trying to solve it using a **SMT solver**, specifically the Microsoft Z3 Solver.

Solving the problem
Basics before starting.

Goal of the assignment

Create a functioning program that provides us with a **timetable** that could be used in a real-life scenario.

What tools are we using?

We decided that the best option would be to use **Python** along with the Z3 Solver for easier and faster coding.

How are we going to do it

We thought about creating a variable that represents **each possible state** in a timetable, and letting the Z3 solver do its work to make a valid schedule

CREATING THE VARIABLES

Creating the variables

We decided on the following format to represent each state: 1.2.3.4.6.5.

- In this example **1** represents year, **2** represents course, **3** is teacher, **4** is the id of the room assigned and **6** represents the time period for day **5**.

We had the following approach to coding the `variables`.

1. First, the information would be introduced manually in `dict` files (Python dictionary files).
2. Second, we would gather that information and introduce the id's in lists.
3. Then, using a few `for loops` we would create every possible variable and save each string on a list.
4. Finally, we would convert each of them to Z3 Ints, save them on another list and add them to the solver.

Why do we have two lists with all of the variables?

The first list is full of **strings**, so we can access them easily, but we need to work with the Z3 Ints, so we would use the first one to select information and give us the **index** for the second one, because they were in the **same order**.

Creating the variables

This is an example of one of the dict files:

```
rooms = {
  "1": "A1.0 80",
  "2": "A1.1 25",                          //The format is the following:
  "3": "A1.2 140",                         //"ID": "RoomName CapacityNo"
  "4": "A2.0 60"
}
```

This dictionary corresponds to **rooms**, but there are more for courses, teachers, time periods and week days.
The information that goes in our variable is the "ID", so when you have the full variable it is very easy to gather all of the information by checking it in the dict files.

Creating the variables

Here's how we would **initialize** a list:

```
def initialize_list(dict1, list1):
    length = len(dict1)
    for x in range(1, length + 1):
        x = str(x)
        list1.append(x)
```

There would be one of these lists for every ID in every dictionary, so now to form **a whole variable**:

```
def create_variables(solver):
    initialize_courses()
    for course in course_list:          //Course includes 'Year.Course.Teacher'
        for room in room_list:
            for period in periods_list:
                for day in days_list:
                    name = course + '.' + room + '.' + period + '.' + day
                    var_list.append(name)        //Appends the string list
                    int_obj = Int(name)
                    bools.append(int_obj)        //This is the z3 variable list
                    solver.add(int_obj >= 0)
```

ENCODING THE CONSTRAINTS

Encoding the constraints

To get a working timetable we had to give a value 0 to the variables that couldn't or preferably **shouldn't be possible**. This is what constraints are for, here are the hard constraints used for this problem:

1. **A teacher can only give one course at a time**, so if a teacher gives a lecture at one time period, the other variables with that teacher and time period should equal 0.
2. **One room can't hold two courses simultaneously**.
3. **Courses of the same year can't be scheduled in parallel**, that is, in the same time period.
4. **Each class has to have 4 weekly lectures exactly.**

Encoding the constraints

Now let's take a look at the **soft** constraints used on our approach:

1. The **capacity** of the room should be, at least, equal to the number of enrolled students for the lecture which will take place there.
2. There should not be **time spaces** between same year classes for each day.
3. **Room stability** should be ensured, which means that for a given subject its classes may always be in the same room.
4. We would like to ensure that classes that are divided into **subgroups have the same time period assigned.** Although in different rooms and with different teachers.

Problem

We can't select each statement that we don't like one by one, we have to give a formula to the Z3 Solver somehow. Right below we will explain the approach that we came up with.

Encoding the constraints

This is a **hard constraint** that ensures that each teacher can only **teach one class at a time**.

```
def teacher_one_at_a_time(solver):
    for i in range(1, len(teachers) + 1):
        for j in range(1, len(time_periods) + 1):
            for k in range(1, len(week_days) + 1):
                xor_list = []
                for m in range(1, len(courses) + 1):
                    data = courses.get(str(m)).split(" ")
                    s = data[0]
                    if data[2] != list(teachers.keys())[list(teachers.values())
    .index(str(i))]:
                        continue
                    for l in range(1, len(rooms) + 1):
                        var = str(s) + '.' + str(m) + '.' + str(i) + '.' + str(
    l) + '.' + str(j) + '.' + str(k)
                        pos = var_list.index(var)
                        xor_list.append(bools[pos])
            solver.add(Sum(xor_list) <= 1)
```

Encoding the constraints

- For *each teacher in a time period of a week day*, a list (xor-list) is created and then we add every single *course* and *room* possible at that period of time to it.
- The first thing to note about this code is the order of the **for loops** and the place where the **list** is reset/added to the solver.
- Of these variables, **only one** can equal 1, hence the list is added to the solver as a sum of every variable (in the list), with the sum having to be **equal to 0** (the teacher could simply not have any classes at that time) or **equal to 1** (it has one specific class at that time), that is shown in the final line.

In addition

If the for loops were in different order, the variables added to the list would be completely different, and the constraint wouldn't express what was intended.

Encoding the constraints

Here is one of the lists that is **added to the solver** by this constraint, let's debunk it:

1.1.1.1.6.5 + 1.1.1.2.6.5 + 1.1.1.3.6.5 + 1.1.1.4.6.5 + 2.4.1.1.6.5 + 2.4.1.2.6.5 + 2.4.1.3.6.5 + 2.4.1.4.6.5 <= 1

- The **fixed variable codes** are the third one (value 1, that represents teacher 1) and the fifth and sixth one (time period 6 for day 5). These are all of the **possible combinations** that this particular teacher could have at that specific time; of these, **only one** (or none) can happen, so the sum of all of these variables has to be less or equal to 1, pretty simple.

THE SOLUTION

Solution

After adding all of the constraints to the solver, we get the list of variables that are **equal to 1**, which essentially makes up the timetable, here is the result:

1.1.1.1.2.2 - 1.1.1.1.6.2 - 1.1.1.1.6.3 - 1.1.1.1.8.3 - 1.2.4.1.2.4 - 1.2.4.1.3.1 - 1.2.4.1.3.4 - 1.2.4.1.4.2 - 2.3.9.1.2.3 - 2.3.9.1.6.1 - 2.3.9.1.8.2 - 2.3.9.1.8.4 - 2.4.11.3.1.1 - 2.4.11.3.1.5 - 2.4.11.3.4.1 - 2.4.11.3.4.4 - 1.5.2.3.2.5 - 1.5.2.3.6.1 - 1.5.2.3.6.5 - 1.5.2.3.8.4 - 2.6.10.4.4.2 - 2.6.10.4.4.5 - 2.6.10.4.7.2 - 2.6.10.4.9.4 - 2.7.7.4.3.2 - 2.7.7.4.6.3 - 2.7.7.4.8.1 - 2.7.7.4.9.1 - 1.8.3.1.1.1 - 1.8.3.1.3.2 - 1.8.3.1.4.5 - 1.8.3.1.7.2 - 1.9.5.1.4.1 - 1.9.5.1.7.3 - 1.9.5.1.7.5 - 1.9.5.1.9.3 - 2.10.6.3.1.2 - 2.10.6.3.6.2 - 2.10.6.3.7.4 - 2.10.6.3.8.3 - 2.11.13.3.3.3 - 2.11.13.3.3.4 - 2.11.13.3.7.1 - 2.11.13.3.9.2

- We now need to change this to a **more readable format**, we know these are the states that actually take place, that means that from 1.1.1.1.2.2, we know the subject from year 1, which has course id 1, will be taught by teacher 1 at room 1 at the time period 2 of day 2. Right below there is a timetable of the first year (first semester).

	Monday	Tuesday	Wednesday	Thursday	Friday
08:00 - 09:00	Basic Finances A1.0 Ramon Minones				
09:00 - 10:00		Discrete Maths A1.0 Gilberto Perez		Calculus A1.0 Jose Rodriguez	Programming I A1.2 Cristian Munteanu
10:00 - 11:00	Calculus A1.0 Jose Rodriguez	Basic Finances A1.0 Ramon Minones			
11:00 - 12:00	Computer Basics A1.0 Guillermo Roca	Calculus A1.0 Jose Rodriguez			Basic Finances A1.0 Ramon Minones
12:00 - 13:00					
13:00 - 14:00	Programming I A1.2 Cristian Munteanu	Discrete Maths A1.0 Gilberto Perez	Discrete Maths A1.0 Gilberto Perez		Programming I A1.2 Cristian Munteanu
14:00 - 15:00					
15:00 - 16:00		Basic Finances A1.0 Ramon Minones	Computer Basics A1.0 Guillermo Roca		Computer Basics A1.0 Guillermo Roca
16:00 - 17:00			Discrete Maths A1.0 Gilberto Perez	Programming I A1.2 Cristian Munteanu	
17:00 - 18:00			Computer Basics A1.0 Guillermo Roca		

Solution

Why are there so many blank spaces between subjects?

You have to remember that this problem was thought taking into account that multiple classes from different years would be taught **at once**, this means that only a few rooms were available for various years.

- Obviously this code isn't perfect, and we could only run 2 years and a few rooms in these time periods, but overall it was a pretty big challenge and our solution took **1072 seconds** (almost 18 minutes), so even with this small sample we could only check if the solution was correct and there were no errors every 18 minutes, this made it an extremely **time-consuming** task. In our solution, every subject is taught 4 times per week, and because of the behaviour of Z3, the solver gave **more priority** to it being taught always in the same class (one of the soft constraints), where it could assure that the class had the number capacity to **hold every student** that had selected that course (another soft constraint) than assuring there weren't any blank spaces between lectures of the same year.

FINAL THOUGHTS

Final thoughts

Overall, this has been a very pleasant experience, we have learned a lot about **Z3** and the way **Python** deals with the solver in general, our programming skills have improved from this and we have really understood the magnitude of the problem that we had in our hands, one that we did not know beforehand.

We now know from this experience that it is **very difficult** to make a perfect timetable because the amount of constraints that need to be added and, especially, checked, make it a very **demanding** problem for any computer, with many calculations and checks back and forth to maximize the solution.