NUSMods Planner: Constraint-based Timetable Planner

Bay Wei Heng[1] **Raynold Ng Yi Chong**[1]

[1] National University of Singapore

{e0014026, e0032363}@u.nus.edu

Abstract. *Timetable planning is a cognitively demanding task due to its combinatorial complexity. We present NUSMods Planner, an augmented version of NUSMods that ships with an automatic timetable planner that takes in a list of user supplied modules, the desired workload and additional constraints (i.e. free days, no lessons during certain time slots) and returns a possible timetable then meets desired workload and constraints if it exists. The server uses z3 to transform the timetable query into a first order logic formula in SMTLIB2 input format which the client then solves with BoolectorJS to extract the timetable that satisfies the constraints. NUSMods Planner has been deployed live and has experienced considerable traffic with favourable reviews, demonstrating the feasibility of using SMT solvers to implement a timetable planning web application.*

1. Introduction

Timetable planning is the activity of assigning objects to time and space such that all constraints are simultaneously met. It has always been a mentally taxing task due the inherent combinatorial complexity of timetable planning. While NUSMods provides students a user-friendly way to plan their school timetable, much of the work is still manual. The fact that the constraints and requirements of a desired timetable can be expressed in first order logic points to the possibility of using an off the shelf constraint solver to implement a fast and easy to use automatic timetable builder.

2. Project Scope and Purpose

We present NUSMods Planner, a web application built on top of NUSMods that offers an automatic timetable generator based on the list of modules and options that one might want in a timetable. Such features include:

- A free day: a day without any lessons
- Lunch Break: at least one hour of no lessons during lunch hours
- Compulsory and Optional modules
- Specific Lesson Slots: user specified lesson slots will appear in timetable, great for students that want to synchronize timetables with their friends
- No lessons before or after certain time

The purpose of this paper is provide a technical discussion of NUSMods Planner, our constraint based reasoning approach to the timetable planning problem, implementation decisions and future work.

Lectures

Class No	Week Type	Day	Start Time	End Time	Venue
1	Every Week	Tuesday	1400	1600	i3-Aud
2	Every Week	Thursday	1600	1800	i3-Aud

Tutorials

Class No	Week Type	Day	Start Time	End Time	Venue
1	Every Week	Wednesday	1200	1300	COM1-B103
2	Every Week	Wednesday	1300	1400	COM1-B103
3	Every Week	Thursday	0900	1000	COM1-0210
4	Every Week	Thursday	1900	2000	COM1-B103
5	Every Week	Friday	1600	1700	COM1-B103
6	Every Week	Friday	1700	1800	COM1-B103
7	Every Week	Friday	1900	2000	COM1-B103
8	Every Week	Friday	1500	1600	COM1-B103

Figure 1. Timetable the module CS3230: Design and Analysis of Algorithms

3. Problem Representation in First Order Logic

3.1. SMT Solvers

Satisfiability Modulo Theories (SMT) solvers take as input a (quantifier-free) first order logic formula F over a background theory T and return:

- `sat (+ model)` if F is satisfiable
- `unsat` if F is unsatisfiable

The rise of efficiency of SMT solvers has created numerous uses for them in software verification, program synthesis, etc[1]. In all of these applications, SMT solvers are used to generate satisfying assignments or proving unsatisfiability. For NUSMods Planner, we use a SMT solver to generate a timetable that satisfies the given constraints.

4. Problem Representation

In our general formulation of timetable planning, we specify 3 main entities: list of modules (compulsory and optional), workload (number of modules to read) and additional options. A module is defined by the following components:

- Module code that identifies the module
- Workload: lectures, tutorials, sectionals etc.
- For each workload type, the class number: SL1, SL2, etc
- For each class number, lesson hours

See figure 1 for a sample module.

A timetable is defined as follows: given a list of modules, for each module selected, a class number from each workload type of the module is in the timetable. A timetable is valid if none of the classes clash with each other. There cannot be more than 1 lesson at any point of time. A sample valid timetable is given in figure 2.

We represent a 2 week timetable as one hour blocks zero indexed from 12 a.m. of the first Monday. Each hour block is an integer value corresponding to the identification number of the workload at that time. If there is no lesson at that time, it is a free variable.

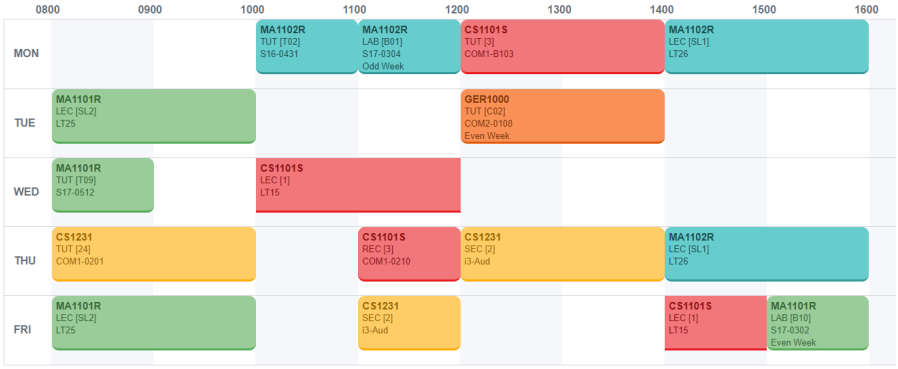

Figure 2. Sample valid timetable

4.1. Valid Timetable

To see how we might translate the aforementioned constraint into first order logic, consider the following: suppose we are deciding which class number from a tutorial (T01, T02) and lecture to choose from (SL1, SL2) for the module m. The lessons hours are in figure 3. Observe that a possible assignment is SL1 and T02.

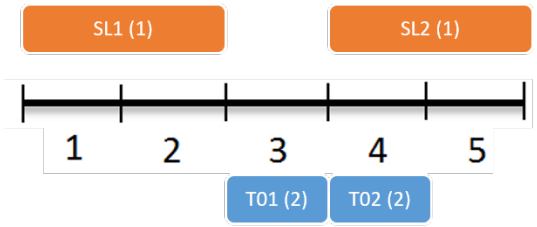

Figure 3. Hours for each class number, the ID is in parenthesis

H_i stores the ID number of the workload. The ID is a function of the module code and workload. We also define free variables SL and T which act as selector variables, they determine which class number for that workload is chosen. So if SL1 is chosen then we have $SL = 1$ and $H_1 = 1, H_2 = 1$. These constraints is expressed in ϕ.

$$\begin{aligned}\phi_1 =&(SL = 1 \rightarrow (H_1 = 1 \wedge H_2 = 1)) \wedge \\ &(SL = 2 \rightarrow (H_4 = 1 \wedge H_5 = 1)) \wedge \\ &(T = 1 \rightarrow (H_3 = 2)) \wedge \\ &(T = 2 \rightarrow (H_4 = 2))\end{aligned}$$

The clauses $(SL = i)$ and $(T = i)$ are known as guard variables as the corresponding H_i is asserted when that class number is chosen.

Next we have to ensure that one class number of each workload type is chosen. This is asserted by getting a disjunction of all possible values for each selector variable workload type.

$$\phi_2 = (SL = 1 \vee SL = 2) \wedge (T = 1 \vee T = 2)$$

We can succinctly assert for no clashes by observing that any H_i can only take on one value. Thus a satisfying assignment to:

$$\begin{aligned}\phi_m =&\phi_1 \wedge \phi_2 \\ =&(SL = 1 \rightarrow (H_1 = 1 \wedge H_2 = 1)) \wedge (SL = 2 \rightarrow (H_4 = 1 \wedge H_5 = 1)) \wedge \\ &(T = 1 \rightarrow (H_3 = 2 \wedge H_3 = 2)) \wedge (T = 2 \rightarrow (H_4 = 2 \wedge H_5 = 2)) \wedge \\ &(SL = 1 \vee SL = 2) \wedge (T = 1 \vee T = 2)\end{aligned}$$

represents a valid timetable for our example with no clashes. As mentioned previously, a possible timetable is to take SL2, and T02. One may check that assigning $SL = 2$ and $T = 2$ satisfies ϕ_m

In the general case, given M, list of modules, for each module, $m \in M$:

$$\begin{aligned}\phi_m = \bigwedge_{w \in \text{Workload}(m)} \Bigg\{ \bigwedge_{c_i \in \text{classNums}(w)} (Sel_w = i) \rightarrow \left(\bigwedge_{h \in \text{Hours}(c_i)} h = H(m, w) \right) \\ \wedge 1 \leq Sel_w \wedge Sel_w \leq |\text{classNums}(w)| \Bigg\}\end{aligned} \tag{1}$$

Where H is the ID generator. Putting everything together, we get:

$$F = \bigwedge_{m \in M} \phi_m$$

4.2. Compulsory and Optional Modules

All compulsory modules will be in the timetable whereas a subset of optional modules will be added to the timetable to meet the user-specified workload. This closely follows the usual student work flow in planning timetables. There are pre-allocated modules or modules that the students definitely wants in the timetable, and a list of optional modules where the remaining modules are picked from to fulfill the n modules workload requirement.

Given an array $M = Comp \cup Opt$, we define set X of size n where each $x_i \in X$ is the index of the module in M if it is chosen. For each $m_i \in M$, ϕ_m, as defined in equation 1, is guarded by $(\bigvee_{j \in [1,n]} x_j = i)$, i.e if the module's index i is in X.

$$F' = \bigwedge_{m_i \in M} \left\{ \left(\bigvee_{j \in [1,n]} x_j = i \right) \rightarrow \phi_m \right\} \tag{2}$$

We assert that n distinct modules are chosen by imposing that X is strictly increasing and in the range $[1, n]$:

$$\psi_1 = \left(\bigwedge_{i \in \{1, n-1\}} x_i < x_{i+1} \right) \wedge 1 \leq x_1 \wedge x_n \leq n$$

For simplicity we also assert that:

$$\psi_2 = \bigwedge_{i \in [1, |Comp|]} x_i = i$$

since all compulsory modules will be in the timetable. We will than have to pick $n - |Comp|$ modules from Opt. This is expressed naturally as the remaining $x_{|Comp|+1}, \ldots, x_n$ are free variables.

Combining everything, we obtain:

$$F'' = \psi_1 \wedge \psi_2 \wedge F'$$

where F' is as defined in equation 2. F'' is then passed into a SMT solver.

4.3. Constructing the Timetable

If F'' is `sat`, the solver will return a model $\mathcal{M}$ that contains the satisfying assignments of the variables in F''. The procedure to extract the timetable is outlined in Algorithm 1. It takes as input, $\mathcal{M}$, the model, M, list of modules of structure as defined in Section 4 and n, the number of modules to read. It returns T, an array of class numbers that are in the timetable.

Algorithm 1 Timetable Extraction Algorithm

```
1:  function TIMETABLEFROMMODEL(M, M, n)
2:      T ← ∅
3:      for all i ∈ [1, n] do
4:          modIndex ← M[x_i]
5:          m ← M[modIndex]
6:          for all w ∈ Workload(m) do
7:              classIndex ← M[Sel_w]
8:              T ← T ∪ {classNums(w)[classIndex]}
9:          end for
10:     end for
11:     return T
12: end function
```

4.4. Free day

Asserting for a free day can be trivially added by including a compulsory mock module that consists of one workload type where each of the class numbers take up one day of the week. However, a minor implementation detail that will be explained in section 4.5 is that ID space of the mock module must be higher and disjoint from that possible with actual mods. Let the lower bound of ID space of the mock module be k.

4.5. No Lessons Before or After

It is not possible to declare another mock module to block out hours from the timetable like in Section 4.4 as the two mock modules will conflict with each other and result in

an unsatisfiable formula. Instead, given a set B of hours that we want to block out of the timetable, we assert:

$$\phi_{block} = \bigwedge_{b \in B} H_b > k$$

It follows that no classes will be appear in the blocked hours since all workload IDs are less than k.

4.6. Lunch Breaks

A lunch break is defined as having at least one hour free during lunch hours (11 a.m, 12 and 1 p.m) every day. This can be achieved by asserting that at least one hour from the lunch hours of each day is greater than k.

4.7. Locked Lesson Slots

Students may want a particular lesson slot in their timetable. This can be trivially achieved by removing all class numbers except for the locked one from the module timetable.

5. Implementation

We shall briefly discuss the implementation details our NUSMods Planner, the software stack and critical design decisions that we made. The software stack is presented in figure 4.

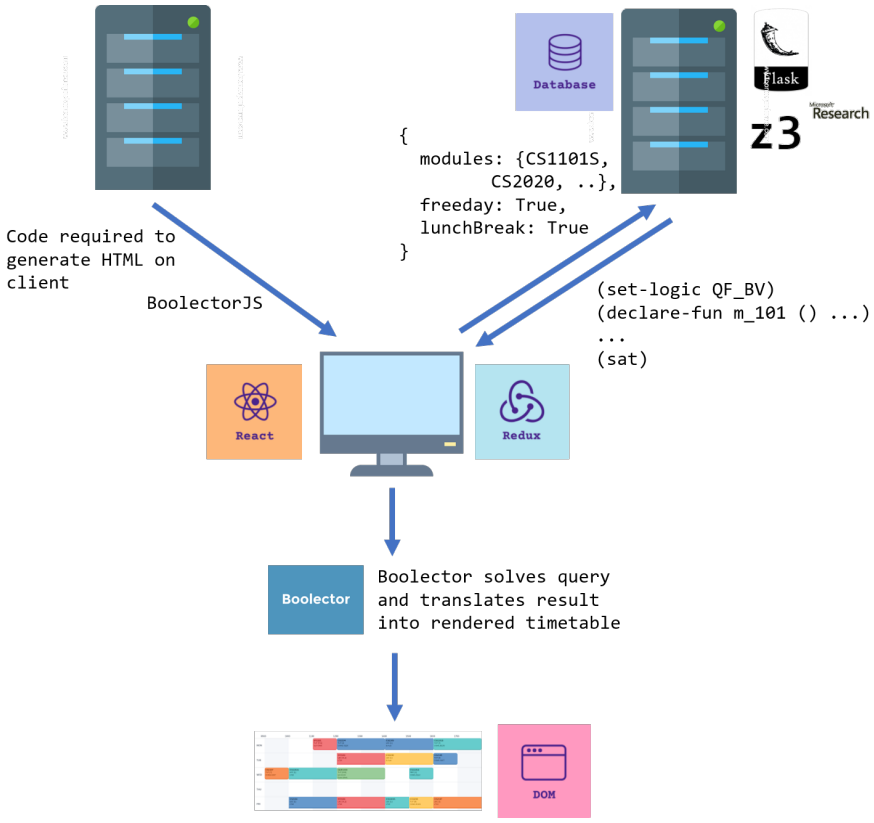

Figure 4. Software Stack for NUSMods Planner

When the client requests for the NUSMods Planner website, an NGINX server will serve the static website files along with BoolectorJS script. When the client requests

for a timetable, parameters such as compulsory and optional modules, and other selected options are sent to a server app which responds by transforming the query parameters into a first order logic equation in the SMTLIB2 input language. SMTLIB2 is a common language and set of benchmarks against which to test and compare solvers. This transformation is done with z3 [2] due its rich and expressive Python API. The client then invokes BoolectorJS to solve the SMTLIB2 input. If it is satisfiable, the timetable is extracted from the model as outlined in algorithm 1. The timetable is then rendered and presented to the user. If the query is unsatisfiable, an error message is reported.

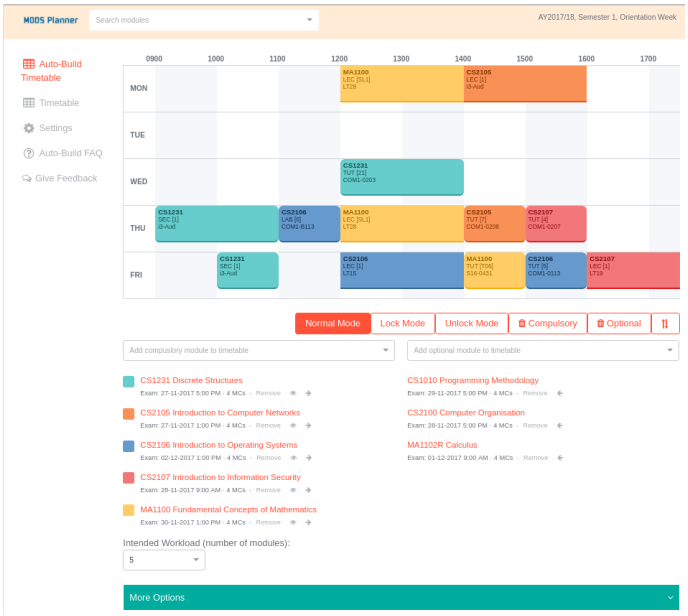

Figure 5. Screenshot of NUSMods Planner

5.1. Delegating SMT Solving to the Client

In our initial implementation, the server was tasked with transforming the query into a first order logic formula and then solving it, and would reply with the extracted timetable. However, preliminary benchmarks on a Digital Ocean instance revealed that the server would take too long to respond resulting in unbearable waiting times on heavier server loads. Hence we searched for a way to delegate the solving, the most computationally expensive part of the whole process, to the client instead.

We decided on BoolectorJS, which to the authors' knowledge is the only SMT solver written in JavaScript. BoolecterJS [3] is a JavaScript version of Boolector SMT solver [4] built with with Emscripten. However, Boolector only supports bit-vectors and arrays so the formula has to be translated from integer arithmetic (QF_LIA) to the theory of unquantified bit-vectors and arrays (QF_BV).

This would allow the SMTLIB2 input to be solved at the client side by invoking its local copy of BoolectorJS. Querying the server usually takes about 0.5s and solving the SMTLIB2 input takes about another 0.5s. This provides for a fast and responsive user experience.

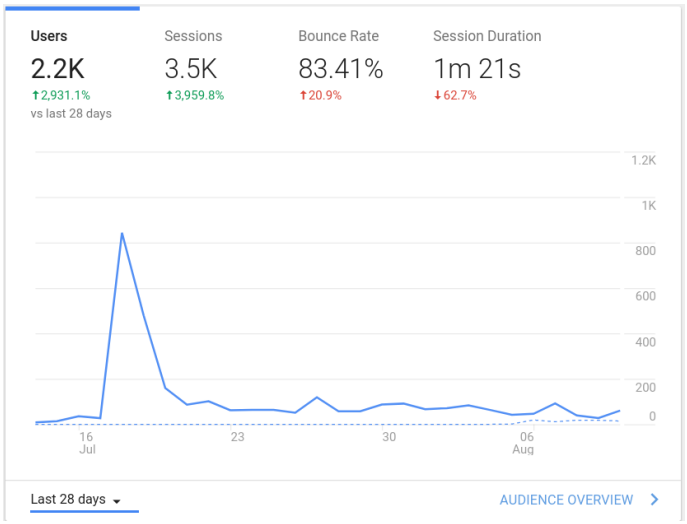

Figure 6. Google Analytics for NUSMods Planner

5.2. Worker Thread to Prevent UI Blocking

As discussed in Section 5.1, the time taken from clicking the generate timetable button to the actual rendering is about 1 second. The UI blocking during this process would be obvious so we migrated the whole process to a worker thread to prevent it from locking the UI.

5.3. Local Mirror of NUSMods API

We noticed that a significant portion of the server response time comes from the server having to query NUSMods API the timetable for every module in the query. Since timetable is unlikely to change often, we decided to mirror the API database on our own server so that it can grab data from its local database.

6. User Analytics and Feedback

We pushed NUSMods Planner into production on 12th July 2017, and till the time of writing at 11th August 2017, there has been a total of 2.3K users and 2.7K sessions. Apart from minor software bugs that were promptly fixed, we did not encounter any server load issues, reflecting the scalability of our approach of delegating SMT solving to the client.

Feedback has been largely positive too, as evident in figure 7. NUSMods Planner can be found at `https://modsplanner.tk/autobuild`.

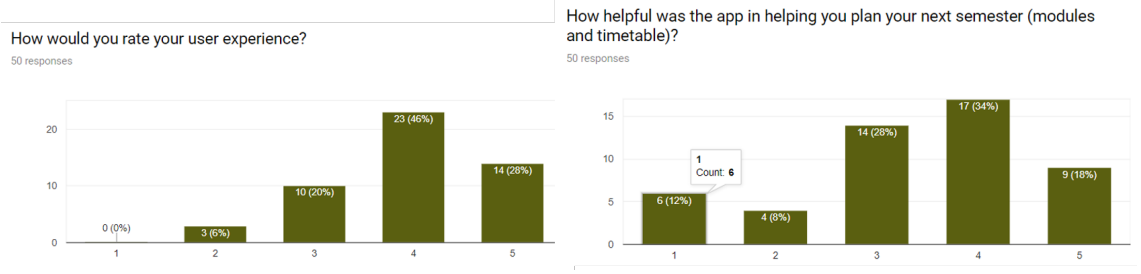

Figure 7. Survey Response for NUSMods Planner

7. Further Work

Much of the immediate future work will be merging NUSMODS Planner with the official NUSMODS project. We wanted to support more features, such as the ability to group optional modules such that at least one is chosen from each group. This is great for students who want to take modules from each ”basket” of modules. While the constraint can be easily expressed in first order logic, we had difficulty exposing this feature in a clean and user friendly manner and decided against supporting it for now.

Another feature that briefly experimented with was optimal timetable planning, such as maximising compactness and minimizing distance travelled between lessons. While there has been work done in solving optimization problems with SMT constraints [5], Boolector does not support such features at the time of writing. While we could implement approximate optimization functionality with iterative queries and binary search, we decided against it due to time constraints.

8. Acknowledgments

We would like to thank our mentors Li Kai and Zhi An for their guidance on the NUSMods codebase. We would also like to thank our advisor Kenneth Lu for his invaluable advice on UI/UX and software engineering.

References

[1] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic optimization with smt solvers. *SIGPLAN Not.*, 49(1):607–618, January 2014.

[2] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Joel Galenson. research.js. `https://github.com/jgalenson/research.js/tree/master/boolector`, 2014.

[4] Robert Brummayer and Armin Biere. *Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays*, pages 174–177. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[5] Feifei Ma, Jun Yan, and Jian Zhang. *Solving Generalized Optimization Problems Subject to SMT Constraints*, pages 247–258. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.