Eindhoven University of Technology
Department of Mathematics and Computer Science
Automated reasoning - 2IMF25 - Assignment group: 100

Marco Chasco Barry
m.chasco.barry@student.tue.nl
2131536

Luigi Ferrara
l.ferrara@student.tue.nl
2218046

Eindhoven, October 2024

Table of Contents

1. Model Checking

1.1 Problem description

We have a set of configurable systems, that in this case represent the Linux Kernel. These configurable systems are usually provided by a propositional logic formula in conjunctive normal form (CNF). It is important to note, that for each of the configurable systems, there are specific feature orders given. We are also asked to discuss the results for the numerical order. So, we present both results.

1.2 Results

We have used the Buddy [1] Python wrapper to do this exercise. We present some explanations and the results for each of the files in the DIMACS, folder.

1.2.1 (A) First Answer

This answers the total number of **valid configurations**. For computing this, we convert the CNF given into a Binary Decision Diagram (BDD) and then look for the satisfiable assignments with the *satcountln*() method from Buddy. For transforming the CNF to a BDD we make use of the apply method that the library implements and the variable creations. We use the extension ln (logarithmic) since this method is faster and optimized for larger files, instead of the flat *satcount*() method. So the results we present are **ln-based**. Of course, we pass the manager and the variable order to the library and it handles it. Take into account that for these results for each of the files, the left value represents the result for the given variable order, and the right value is the one for the numerical variable order **(given order - numerical order)**.

- buildroot: 525.8496250072116 - 525.8496250072116
- busybox: 646.2279065537504
- embtoolkit: 1128.531177019617
- toybox: 57.00874883358935 - 57.00874883358934
- uClinux: 303.0 - 303.0

As we can see, the results for both variable ordering are the same. This makes sense since the number of satisfiable configurations in a BDD does not depend on the variable ordering itself. However, this variable ordering impacts the size of the BDD and the efficiency of the operations. This is why we do not get a solution for busybox and embtoolkit when changing the order to the numerical one.

1.2.2 (B) Second Answer

This answers the lengths of the permissive configurations. For the following results and subsections, as we did in the previous section, we represent the result for the given variable order on the left side, and the result for the numerical order on the right side (given variable order - numerical order).

i)

For **selecting a feature whenever there is a choice**, we prioritize the true branch (setting the feature to 1) for each of the current nodes if it can be still satisfiable with true or false. Being still satisfiable with both means that we have a choice, then we set the feature to 1. The results for the given variable order are the following:

- buildroot: 70 - 66
- busybox: 385
- embtoolkit: 1403
- toybox: 544 - 544
- uClinux: 1850 - 1850

ii)

For **deselecting a feature whenever there is a choice**, we prioritize the false branch (setting the feature to 0) for each of the current nodes if it can be still satisfiable with true or false. Being still satisfiable with both means that we have a choice, then we set the feature to 0. The results for the given variable order are the following:

- buildroot: 558 - 551
- busybox: 776
- embtoolkit: 2051
- toybox: 544 - 544
- uClinux: 1850 - 1850

iii)

For **selecting** a feature if and only if this decision would cover **more valid configurations** than with deselecting the feature, we compute the number of satisfiable assignments for both setting the feature to 1 and 0. If setting it to true gives more valid configurations, then we set it to true. The results for the given variable order are the following:

- buildroot: 61 - 60
- busybox: 432
- embtoolkit: 1457
- toybox: 544 - 544
- uClinux: 1850 - 1850

iv)

For **deselecting** a feature if and only if this decision would cover **more valid configurations** than with selecting the feature, we compute the number of satisfiable assignments for setting the feature to 1 and 0. If setting it to false gives more valid configurations, we set it to false. The results for the given variable order are the following:

- buildroot: 492 - 74
- busybox: 437
- embtoolkit: 1456
- toybox: 544 - 544

- uClinux: 1850 - 1850

For the above **four stepwise** configurations, we can see that there are no results for the numerical order for both the busybox and the embtoolkit files. This makes sense since changing the variable order now changes the efficiency of our logic, and these are very big files. For the other files, they are both pretty similar, and in the cases of toybox and uClinux they are always the same because these two files have the peculiarity of having most of their clauses just formed by one literal.

1.2.3 (C) Third Answer

This answers a **most permissive configuration**. We are looking for the shortest permissive configuration and the one with the smallest sum of orders. For each literal, we check the satisfiable configurations for both setting it to 1 and 0, and take the largest. If we try to satisfy the largest number of valid configurations for each literal we will finally end with the shortest permissive configuration. In case both sizes of the branches true and false are equal, we take the one that has the minimum sum of indexes to satisfy the second requirement of the statement. The results for the given variable order are the following

- buildroot: 492 - 74
- busybox: 437
- embtoolkit: 1456
- toybox: 544 - 544
- uClinux: 1850 - 1850

As we can see, regarding the results we have the same behavior as the previous sections.

For implementing an **undo functionality** to go back to the last state, we would suggest copying the BDD node just before going to the next node. If we do this just before using the cofactors, once we are in the next node of the original BDD (after applying the cofactors), we could be able to know which node was before the cofactor since we stored it.

2. Circuit Verification

2.1 Problem description

Sometimes in the optimization process of some circuit companies, it can happen that circuits are optimized. This means that the number and the type of the internal gates of a circuit and its optimized version can change. When this happens we need to be sure that the optimized circuit represents the same function as the previous one. To this end, we implemented a tool for circuit verification using Binary Decision Diagrams (BDDs) and the OxiDD [2] Python Library.

The circuits were defined in different files with the BENCH format. To create the tool introduced above, we created a script for finding the propositional formula that describes the circuit under discussion.

Consequently, we exploited a bottom-up approach to build a BDD for each non-input gate (including the output gates) of the circuit taken into consideration.

Finally, we could see when the circuit represented the same function simply by computing the logic implication between all the output gates of the standard circuit and all the output gates of the second optimized circuit.

2.2 Formal Definition

Each gate in the BENCH format is defined as:

$$gate_i = op(input_1,\ input_2, \cdots input_{n^i})$$

where n^i is the total number of input of $gate_i$. Each $gate_i$ has a correspondent propositional formula as follows:

$$\varphi_i = input_1\ op\ input_2\ op \cdots input_{n^i}$$

where the $input_i$ can either be another gate propositional formula or one of the initial inputs of the circuit.

For each of the output gate $gate^j_{out}$ of a circuit, there exists a correspondent propositional formula in the same fashion as the ones previously seen, defined as:

$$\varphi_j = input_1\ op\ input_2\ op \cdots input_{n^j}$$

where n^i is the total number of input of $gate^j_{out}$.

In the end, we compare all of the outputs of the standard circuit and the outputs of the optimized version to see if the circuits are equivalent, i.e., if they implement the same function. To this end, we compute the following:

$$\Theta = \bigwedge_{k=1}^{K} (\varphi_k \Leftrightarrow \varrho_k)$$

Where K represents the number of output gates (it must be the same for both the 2 compared circuits), φ_k is k-th output gate function of the standard circuit, and ϱ_k is k-th output gate function of the optimized version circuit. We assume the outputs are placed in the right order, that is, output number 1 of the standard circuit corresponds to output number 1 of the optimized circuit of the pair taken into account.

If Θ is valid, the 2 circuits are equivalent.

2.3 Results

We picked and calculated the **equivalence** of the circuits: *1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 16, 17, 18* and we obtained the following results:

Pair of circuits	**Validity of Θ**
1	True
2	True
3	False
4	False
5	False
6	True
7	True
8	False
9	False
10	True
11	False
13	False
16	False
17	False
18	False

It is worth noticing that the circuits *16, 17, 18* are **false** since they differ in the number of output gates. The tool can be improved in the feature by adding for example the **Fenin** heuristic for circuits, thanks to which we could change the order of variables needed to build the gates and subsequently, come up with a solution with a BDD with a fewer number of nodes. The improvement can result significant

3. Finite State Automata Representation

3.1 Problem description

For this problem, we aim to create a tool that creates Finite State Automata (FSA) representations, considering the initial state of the FSA under discussion, its transitions, and its accepting states. An FSA can be represented using Binary Decisions Diagrams.

We take into account several FSAs defined using the BA format. Each of the FSA has a similar way to picture its states.

An example of a state:

$$[0\ 255|255|255][0][0][0][0]$$

An example of a transition:

$$[0\ 255|255|255][0][0][0][0] \rightarrow [0\ 2|255|255][0][1][0][0]$$

3.2 Variables and Characteristic Functions

To implement the tool described above, we need to find a suitable representation for the states of the FSAs.

To this end, we noticed that a state can be composed of only 12 different types of **atoms**. To encode the states with the least possible number of boolean variables we encode each atom with 4 bits. Since with 4 bits, we can represent 16 elements, 4 representations will not be needed.

It can be stated that this is not the optimal way to encode such a system. However, we focused on making a tool that exploited a generic encoding process for the different states a finite state machine could be composed of.

If a state is composed of 8 atoms, it will have in the end 32 representation **variables**.

3.2.1 Atoms encoding

The atoms are separated by either a space, a bar, a parenthesis, and/or other non-number elements that we don't consider so that we can keep our encoding as compact as possible.

The following is how we encoded each of the atoms:

$$1 \rightarrow 0001$$
$$2 \rightarrow 0010$$
$$3 \rightarrow 0011$$
$$4 \rightarrow 0100$$
$$5 \rightarrow 0101$$
$$6 \rightarrow 0110$$
$$7 \rightarrow 0111$$
$$8 \rightarrow 1000$$
$$9 \rightarrow 1001$$

Furthermore, we have atoms like:

$$255 \rightarrow 1100 \quad \text{and} \quad \text{'empty cell'} \rightarrow 1011$$

Finally, we have a more particular state that is shown sometimes while scanning the transitions. The state is (ERR). When coming up with (ERR) we simply encode it as:

$$(ERR) \rightarrow 1111 * n$$

Where n is the number of atoms of the state with the **maximum length** in the final state machine we are considering.

For each FSA, we take into account the transition composed of the largest number of atoms and then we encode it. Then, if in the same FSA a second transition is represented with fewer atoms, we will fill that with *empty cell*, so that every transition is encoded with the same number of variables.

In conclusion, the encoding of the example transition above is:

$$'0000110011001100000000000000000000000000010110011000000000100000000'$$

Where each 0 and 1 is represented by a boolean variable. For instance, if the third number of the encoding is 0, we have $\neg x_2$ (or $\overline{x}_2$)

3.2.2 Characteristic functions

The characteristic function of the **transitions** specifies all (and only) the valid possible transitions of the FSA. It is defined as follows:

$$\begin{aligned}\Delta'\left(x_0, x'_0, x_1, x'_1, \ldots, x_{n-1}, x'_{n-1}\right) = {} & \overline{x}_0 \cdot x'_0 \cdot x_1 \cdot x'_1 \cdots x_{n-1} \cdot x'_{n-1} \\ & + x_0 \cdot x'_0 \cdot \overline{x}_1 \cdot \overline{x}'_1 \cdots x_{n-1} \cdot x'_{n-1} \\ & \vdots \\ & + x_0 \cdot x'_0 \cdot x_1 \cdot \overline{x}'_1 \ldots \overline{x}_{n-1} \cdot x'_{n-1}\end{aligned}$$

Where n is the number of variables needed to represent a transition in the FSA under discussion and x' stands for the successor variable of x. The characteristic functions for the **initial state** and the **accepting states** are respectively:

$$\Delta''\left(x_0, x_1, ..., x_{(n/2)-1}\right) = x_0 \cdot x_1 \cdots x'_{n-1}$$

$$\begin{aligned}\Delta'''\left(x_0, x_1, \cdots, x'_{(n/2)-1}\right) = {} & \overline{x}_0 \cdot x_1 \cdots x_{n-1} \\ & + x_0 \cdot \overline{x}_1 \cdots x_{n-1} \\ & \vdots \\ & + x_0 \cdot \overline{x}_1 \cdots \overline{x}_{n-1}\end{aligned}$$

It is worth noticing that in our functions, we only take into consideration the existing transitions of the FSA.

3.3 Results

We used the OxiDD [2] Python Library to implement such a tool. The following are the answers we were asked to present.

3.3.1 (A) First Answer

The variable order of the shared BDD has been created following the heuristic discussed during the lectures, i.e., placing related variables close to each other. Hence, the variable order is as follows: x_0, x'_0, x_1, x'_1, and so on, up to the last variable, x'_{n-1}. We remember once again that if x_i is a variable in the source state of a transition, then, x'_i is its correspondent variable into the accepting state of the transition we are taking into account.

To have a better understanding of the impact of such a heuristic, we show in 3.3.3 the difference between the number of nodes of the BDD for the characteristic function for transitions when using the heuristic and when not.

3.3.2 (B) Second Answer

In this section, we report the total number of nodes of the shared BDD for each of the FSA analyzed.

FSA	**Number of nodes**
bakery.1.c.ba	12893
bakery.2.c.ba	9073
fischer.2.c.ba	31969
fischer.3.1.c.ba	7046
fischer.3.2.c.ba	9536
fischer.3.c.ba	7046
mcs.1.2.c.ba	23866
NFA_hard_1.ba	18661
phils.1.1.c.ba	2492
phils.2.c.ba	12469

3.3.3 (C) Third Answer

Now, we outline the number of satisfying assignments and BDD nodes for presenting the characteristic functions of all (i) the initial state, (ii) the transition relations, and (iii) the accepting states.

FSA (i)	Number of nodes	# Satisfying assignments
bakery.1.c.ba	42	1
bakery.2.c.ba	42	1
fischer.2.c.ba	34	1
fischer.3.1.c.ba	34	1
fischer.3.2.c.ba	34	1
fischer.3.c.ba	34	1
mcs.1.2.c.ba	54	1
NFA_hard_1.ba	46	1
phils.1.1.c.ba	34	1
phils.2.c.ba	46	1

FSA (ii)	Heuristic Status	Number of nodes	# Satisfying assignments
bakery.1.c.ba	Not used	48715	2697
bakery.1.c.ba	Used	11641	2697
bakery.2.c.ba	Not used	35802	2085
bakery.2.c.ba	Used	8094	2085
fischer.2.c.ba	Not used	313858	65458
fischer.2.c.ba	Used	29393	67590
fischer.3.1.c.ba	Not used	13708	1401
fischer.3.1.c.ba	Used	6760	1401
fischer.3.2.c.ba	Not used	25826	3856
fischer.3.2.c.ba	Used	8632	3856
fischer.3.c.ba	Not used	13737	1400
fischer.3.c.ba	Used	6760	1400
mcs.1.2.c.ba	Not used	314625	21509
mcs.1.2.c.ba	Used	21551	21509
NFA_hard_1.ba	Not used	237553	25452
NFA_hard_1.ba	Used	17539	25452
phils.1.1.c.ba	Not used	5174	464
phils.1.1.c.ba	Used	2228	464
phils.2.c.ba	Not used	33275	2350
phils.2.c.ba	Used	11350	2350

It is worth noticing, that in the table above, we can also see the difference between when we use the heuristic that we pointed out previously and when we do not. The difference in terms of the number of nodes is significant. With the heuristic, we can cut off even more than **90% of the**

nodes as it happens for *FSA mcs.1.2.c.ba*

FSA (iii)	**Number of nodes**	**# Satisfying assignments**
bakery.1.c.ba	1210	196
bakery.2.c.ba	937	204
fischer.2.c.ba	2542	6866
fischer.3.1.c.ba	252	29
fischer.3.2.c.ba	870	431
fischer.3.c.ba	252	29
mcs.1.2.c.ba	2261	1843
NFA_hard_1.ba	1076	2850
phils.1.1.c.ba	230	81
phils.2.c.ba	1073	295

3.3.4 Further Reduction

We could potentially further reduce the BDDs by following dynamic variable-reordering-based techniques. An instance of such a technique is the **Sifting** [3] algorithm, which finds the locally optimal variable ordering in a BDD.

4. Football Matches Scheduling

4.1 Problem description

We want to **minimize** the total amount of days spent running a football tournament to cut the maintenance costs for the tournament association. To this end, we assume that each match of the tournament will last the same time as it lasted last year (a heuristic suggested by some experts in the field). Each match has a specific **priority**, depending on which phase of the tournament it needs to be scheduled. A lower priority means that the match needs to be played before, and a higher priority means the opposite. Lastly, the association has a specific number of fields available and a specific number of days to fully split all the matches.

For this problem, we use Z3-solver [4] to implement it. We decided to use this library because it was very useful during the first assignment and it is one of the suggested tools.

4.2 Variables and Constraints

We declare the following variables for the described problem:

- M_i is the i-th match in the tournament, where $i \in I = \{1, ..., 32\}$
- M_i^d represents the day d in which a match i is scheduled to be played
- M_i^f is the field f where a specific match i is played, where $f \in F = \{1, 2, 3\}$
- M_i^p is the priority p of match i, where $p \in P = \{1, 2, 3, 4, 5, 6\}$,
- $M_i^{duration}$ is the duration $duration$ of match i,
- M_i^{start} is the timestamp (in minutes) when match i starts,
- M_i^{end} is the timestamp (in minutes) when match i end.

Now we declare the constraints:

The start of a match has to be earlier than its end time.

$$\bigwedge_{i=1}^{|M|} (M_i^{start} < M_i^{end})$$

Every match must begin after lunch, to ensure there is enough audience.

$$\bigwedge_{i=1}^{|M|} (M_i^{start} \geq 14 * 60)$$

Every match must finish before midnight.

$$\bigwedge_{i=1}^{|M|} (M_i^{end} \leq 24 * 60)$$

Two matches cannot be in the same field and on the same day if they overlap in terms of timing.

$$\bigwedge_{i=1}^{|M|} \bigwedge_{j=i+1}^{|M|} ((M_i^d = M_j^d \wedge M_i^f = M_j^f) \implies (M_j^{start} > M_i^{end} \wedge M_j^{end} < M_i^{start}))$$

All the matches of a priority need to be finished before the next priority matches start.

$$\bigwedge_{i=1}^{|M|} \bigwedge_{j=1}^{|M|} ((M_i^p < M_j^p) \implies (M_i^d < M_j^d))$$

where $j \neq i$

These **priorities** are known and they are the following

- Groups-phase: priority 1
- Round of 8: priority 2
- Quarter-finals: priority 3
- Semi-finals: priority 4
- 3rd place: priority 5
- Finals: priority 6

A match is assigned to one of the three available fields.

$$\bigwedge_{i=1}^{|M|} ((M_i^f \geq 1) \wedge (M_i^f < 3))$$

Then, we define a final constraint to ensure that all the matches are played before a variable d_{max}, i.e., the last day a match can be played.

$$\bigwedge_{i=1}^{|M|} (M_i^d \leq d_{max})$$

The final formula now consists of the **conjunction** of the constraints above.

Finally, we declare the **goal function** where we want to **minimize** the last day the matches are played.

$$min(d_{max})$$

4.3 Results

In this section, we present the **results** given by our program to solve the stated problem.

1. Match 1, Day 1, Start 16:11, End 18:02, Field 1, Priority 1
2. Match 2, Day 1, Start 18:03, End 19:41, Field 2, Priority 1
3. Match 3, Day 1, Start 21:52, End 24:00, Field 1, Priority 1
4. Match 4, Day 2, Start 18:05, End 20:08, Field 1, Priority 1
5. Match 5, Day 1, Start 20:09, End 21:43, Field 2, Priority 1
6. Match 6, Day 2, Start 20:09, End 22:02, Field 1, Priority 1
7. Match 7, Day 1, Start 19:42, End 21:51, Field 1, Priority 1
8. Match 8, Day 2, Start 16:11, End 18:04, Field 2, Priority 1
9. Match 9, Day 2, Start 22:12, End 24:00, Field 1, Priority 1
10. Match 10, Day 2, Start 14:00, End 16:10, Field 2, Priority 1
11. Match 11, Day 1, Start 18:03, End 19:41, Field 1, Priority 1
12. Match 12, Day 1, Start 14:00, End 15:30, Field 2, Priority 1
13. Match 13, Day 1, Start 21:52, End 24:00, Field 2, Priority 1
14. Match 14, Day 1, Start 22:12, End 24:00, Field 3, Priority 1
15. Match 15, Day 1, Start 14:21, End 16:10, Field 1, Priority 1
16. Match 16, Day 2, Start 14:00, End 16:01, Field 1, Priority 1
17. Match 17, Day 3, Start 22:18, End 24:00, Field 1, Priority 2
18. Match 18, Day 3, Start 22:02, End 24:00, Field 2, Priority 2
19. Match 19, Day 3, Start 15:35, End 17:42, Field 1, Priority 2
20. Match 20, Day 3, Start 14:00, End 15:34, Field 1, Priority 2
21. Match 21, Day 3, Start 18:27, End 20:25, Field 1, Priority 2
22. Match 22, Day 3, Start 20:26, End 22:01, Field 1, Priority 2
23. Match 23, Day 3, Start 15:34, End 17:15, Field 2, Priority 2
24. Match 24, Day 3, Start 17:43, End 19:40, Field 2, Priority 2
25. Match 25, Day 4, Start 15:47, End 17:57, Field 1, Priority 3
26. Match 26, Day 4, Start 20:02, End 21:46, Field 2, Priority 3
27. Match 27, Day 4, Start 14:00, End 15:46, Field 1, Priority 3
28. Match 28, Day 4, Start 17:58, End 20:01, Field 1, Priority 3
29. Match 29, Day 5, Start 16:03, End 18:06, Field 1, Priority 4
30. Match 30, Day 5, Start 14:00, End 16:02, Field 2, Priority 4
31. Match 31, Day 6, Start 14:00, End 15:49, Field 1, Priority 5
32. Match 32, Day 7, Start 14:00, End 16:05, Field 1, Priority 6

The final solution we get is then: **7 days** are needed as minimum to play all the matches with their respective constraints.

As we can see, all the matches are splitted and all the constraints are satisfied properly. So we get that final solution for the stated problem. Each line in the result represents the following pattern:

- Match number, day the match is played, start time of the match in *hh:mm*, end time of the match in *hh:mm*, field where the match is played and priority of the match.

5. Additional Information

We have made use of ChatGPT [5] for testing purposes and sample generation. Furthermore, we have used Grammarly [6] for checking our spelling errors in the Overleaf report.

References

[1] BuDDy. Buddy python wrapper. https://github.com/jgcoded/BuDDy. 3

[2] OxiDD. The next-gen decision diagram framework. https://oxidd.net. 6, 10

[3] Sifting algorithm. Slides 24-26, from the course slides "Variable Ordering". 12

[4] Microsoft Research. Z3-solver. https://github.com/Z3Prover/z3. 13

[5] OpenAI. Chatgpt. https://chatgpt.com/. 17

[6] Grammarly. Grammarly spelling assistant. https://app.grammarly.com/. 17