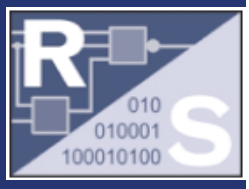
Binary encoding using Dichotomies

Tim Burkert and Laurenz Kamp

TECHNISCHE
UNIVERSITÄT
DARMSTADT



Contents

1 Binary Encoding Using Dichotomies

Dichotomies can be used to find a binary encoding for a state machine. A dichotomy is a pair of sets called L and R. Both sets can contain an arbitrary number of symbols. In the case of state machines, a symbol represents a state.

The first step necessary for binary encoding using dichotomies is to generate a constraint matrix A. It is derived from a minimal cover of the state machine. The constraint matrix contains all combined states that combine multiple states, but not all states of the state machine.

Before explaining the algorithm to find a binary encoding, some terms have to be defined:

- **Root dichotomy of a row a^T:** A root dichotomy of a row a^T of A is a dichotomy where L contains all symbols that have a 1 in a^T and R contains one symbol that has a 0 in a^T.
- **Compatibility:** (L_1, R_1) and (L_2, R_2) are compatible if one of the following statements holds true:
  - $L_1 \cap R_2 = \emptyset$ and $R_1 \cap L_2 = \emptyset$
  - $L_1 \cap L_2 = \emptyset$ and $R_1 \cap R_2 = \emptyset$
- **Coverage:** (L_1, R_1) covers (L_2, R_2) if one of the following statements holds true:
  - $L_1 \supseteq R_2$ and $R_1 \supseteq L_2$
  - $L_1 \supseteq L_2$ and $R_1 \supseteq R_2$
- **Prime dichotomy:** A prime dichotomy is a dichotomy that can not be covered by a compatible dichotomy.

The algorithm to find an exact binary encoding works in the following way:

1. Compute all prime dichotomies.
2. Compute all root dichotomies for the constraint matrix A.
3. Generate a table with one column for every root dichotomy and one row for every prime dichotomy. Every cell of the table is set to 1 if the prime dichotomy covers the root dichotomy. Otherwise, the cell is set to 0.
4. Find a minimal set of prime dichotomies that cover all root dichotomies.
5. Generate the encoding matrix from the L set of the found prime dichotomies.

1.1 Task

The task given was to implement exact binary encoding using dichotomies. A framework was provided that can read `.kiss` files. Those `.kiss` files describe a state machine. The framework has to be extended by the algorithms to generate the dichotomies and to find a binary encoding of all states. The results should be written to a `.blif` file that can then be read by `abc` to map the design to lookup tables. Finally an evaluation should be done to assess the performance of the program.

2 Implementation

2.1 Construction of Constrain Matrix

Before solving the input encoding problem the given problem specification must be transformed into a constrain matrix. This constrain matrix is further called A. Each row i of $A_{i,j}$ describes a constrain on the resulting encoding. Furthermore, each column j of $A_{i,j}$ assigned to a encoded symbol. For example the row $(1,1,0,0)$ means that symbols $3,4$ can be encode together so that the resulting super cube does not include symbols $1,2$. This constrain would be equal to the row $(0,0,1,1)$. The reason for this that a dichotomy describes a bipartition of a set, but for encoding problems only the relation between the elements in both sets are important. Therefore, the partition $A_0 : 1,2 A_1 : 3,4$ and the partition $B_0 : 3,4 B_1 : 1,2$ are inverse partitions but the relations are all equal. In A and B the elements $1,2$ are combined in a set and are not combined with $3,4$. A row describes also a bipartition of all symbols.

The given problem specification used symbolic names for state and binary notation including don't cares for input and output vectors. The constrain matrix A is a result of the minimal symbolic cover. Because of the binary notation of the input and output vectors we transformed the symbolic problem into a binary coded cover problem. We used the positional cube notation to deal with input and output binary notation including don't cares. For the symbolic state names we used a one hot positional encoding.

A covering super cube can for a set of positional cube notation vectors is computed by AND-conjunction all vectors. The resulting cube must be tested for validity. This is given when all cubes are valid (not equals $2'b00$).

2.1.1 Minimal Cover Algorithm

For implementation we implemented an iterative approach. The algorithm terminates when no further improvements are possible. This is when all combination of entries are tested. When a optimization is possible the two vectors that are combined are removed and a new vector covering both is included. When this happens all combinations of entries are tried again until termination.

From the set of resulting super cubes for the states constrain matrix A can be constructed. Only the entries that actual constrain the problem are for interested, therefor entries including all symbols as symbolic implicant or entries that include only one symbolic literal as implicant can be removed. Only entries that portion the symbolic state into a relation of one symbol can be combined with others not include other symbol, like the given example, are for interested.

2.2 Generation of Root Dichotomies

The generation of all root dichotomies is straight forward implementation of a sequential generator. This generator uses all rows of the constrain matrix and computes all resulting root dichotomies for each row. The number of root dichotomies are equal to the number of symbols assigned a zero in the constrain row.

2.3 Generation of Prime Dichotomies Candidates

As stated before to find a exact solution all candidates prime dichotomies must be checked. A candidate is a dichotomy where a symbols are portioned in both sets. We used a long vector with positional symbol notation where each bit is assigned to a symbol. Just by iterating all possible values all prime dichotomies candidates are reached. The inverse property of dichotomies regarding encoding is used to reduce set of candidates to the half. Because all values in the lower half have a value inverse equal in the upper half. For example a positional symbol notation vector for 5 five symbols of $2'b00110$ is equal to $2'b11001$.

2.4 Prime Dichotomies Coverage Table

Before we solve the encoding problem we need to compute which prime dichotomy covers which root dichotomies. For this we generate a table of lookup vectors. Each vector describes the cover relation between a prime dichotomy and all root dichotomies, by using again a positional notation where a bit describes the coverage.

2.5 Cover Root Dichotomies with Primes

Using the before created table it is now possible to look for a minimal set of prime dichotomies that cover all root dichotomies. We start with the small set possible of one prime dichotomy. When no solution is found the search space is increased by one additional element. The combination of elements for the minimal set of prime dichotomies is generator that iterate all combination without repetition. For the k round $\binom{n}{k}$ for n table entries are possible.

2.6 Encoding Correction

Its is possible that the chosen prime dichotomies yield a contradicting encoding. Then the resulting encoding is not for each state unique. That handle we by extending the encoding with bits until the result is unique for each state. We implemented this by using Hashset and a counter initialized to $2'b1$. When a encoding is already contained in the hash set then binary value of the counter is added in front of the encoding. For example the encoding $2'b010$ creates a collision the value is changed to $2'b1_010$. Should this was also be already contained the counter is incremented and the encoding changed to $2'b10_010$. During this iterative approach the maximum counter value for all encoding is stored. The resulting bit length for the state encoding is equal to the number of prime dichotomies used plus the first bit position of the maximum counter value. This approach guaranties that all encode symbols are unique and the amount of bits used is minimal.

2.7 Optimizations

The complexity to find an exact solution scales with $\binom{n}{k}$, where n is the size of the coverage table and k is the amount of lines that are combined to find a solution. One way to shorten runtime is to reduce the table size. Another way is to find a faster algorithm to search the table, but this comes at the cost of non-exact solutions. It is also desirable to parallelize the process of searching the coverage table to utilize multiple processors.

2.7.1 Coverage Table Minimization

Some entries in the coverage table can be removed without lowering the result quality. A line l_1 in the table is eligible for removal if it is covered by another line l_2 in the table. l_1 is covered by l_2 if all bits

that are set in l_1 are also set in l_2. Reducing the table size in this way takes additional time and is more expensive for large tables, but it can reduce table sizes drastically. Evaluation has shown that large coverage tables can be reduced by a factor of 2.

2.7.2 Parallelization

The search for a solution in the coverage table can be parallelized by partitioning the set of all possible combinations of table entries. After the partitions are formed they are assigned to a thread that combines the table entries and checks whether a valid solution was found. When all combinations in the partition have been checked the thread requests another partition that is then checked for a solution.

2.7.3 Non Exact Solution

Large problems that result in big coverage tables take a long time to find an exact solution. The runtime becomes so long that it is impractical to search for an exact solution. Because of this we also implemented an algorithm that searches for a non exact solution. It works in the following way:

1. Select the entry from the coverage table that has the most bits set. If a valid solution is already found abort the search.

2. Try every entry from the coverage table by combining it with the previously selected entry/entries and save the number of additional bits that are set after combining the entries.

3. Select the entry that adds the most set bits to the result. If a valid solution is found abort the process. If not repeat step 2 and 3 until a solution is found.

This non exact approach results in drastically reduced runtime but will typically generate worse results than the exact approach. Both algorithms are compared in chapter 3.

2.8 Implementation Limits

The implementation sets some limitations to the problems that can be solved:

- **Number of states:** States are internally represented as a one hot encoded long variable. This limits the number of states to a maximum of 64. State machines with more states can not be processed, however the runtime required to find an exact solution for such complex problems would be extremely high anyway.

- **Size of coverage table:** Large state machines result in a large coverage table. This poses two problems: The host machine may run out of available memory and the Java garbage collector will be responsibly for most of the execution time. This makes the program less efficient and will eventually result in its termination by the Java VM.

- **Runtime:** Finding an exact solution can take very long for complex state machines. It is possible to use the non exact algorithm, but this will result in worse results.

3 Evaluation

This evaluation analysis the performance and drawbacks of an exact binary state encoding. To measure the result quality we used the ABC a tool for sequential synthesis and verification. Our implementation exports a complete finite state machine using the before computed encoding solution. As required by the minitasked abc was used to mapped the state machine onto LUTs. For each benchmark the execution time was limited to 1 h. One exception was made for benchmark *results_exact/015_dk512.kiss2.blif*, which ran for about ≈ 16 h.

3.1 Runtime

Overall the runtime seems to be in a relation to the number of states, as seen in table 3.1. When comparing our exact and non exact approach more benchmarks are completed in a feasible time. To exclude a faulty implementation the benchmark *results_exact/015_dk512.kiss2.blif* was run until completion. This benchmark was the first benchmark that didn't finish in 1 h. The non exact solution for this benchmark is much quicker but also equal performance measured in Area, Delay and Latches. But this is not for all cases true. For example, benchmark *results_exact/011_train11.kiss2.blif* yield a worse result regarding these criteria.

The two benchmark *results_exact/020_tma.kiss2.blif* and *results_exact/024_pma.kiss2.blif* are finished fast, because of the empty constraint matrix A. An empty constraint matrix result into a binary encoding.

3.2 Mapping Result

As shown in figure 3.1, the non-exact solution performs for all benchmarks worse or equal, excluding the benchmark *results_exact/008_ex6.kiss2.blif*. For some benchmarks up to twice the fpga resource usage. The chosen target fpga technology isn't ideal for comparing exact or near exact solutions, because small differences in the transition function complexity usually yield the same result.

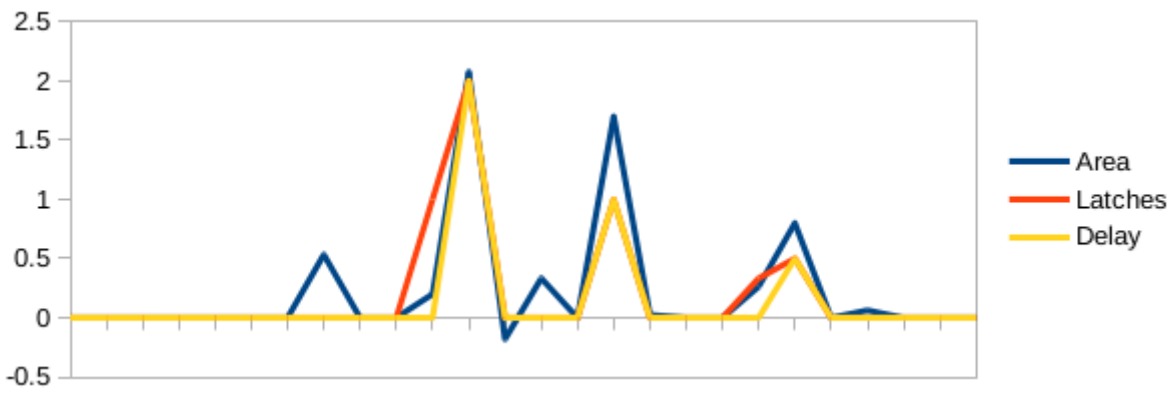


Figure 3.1: Difference between non- and exact solution compared by area, latches and delay.

3.3 Verdict

As seen in the mapping result compared to one hot encoding binary state encoding using dichotomies delivers worse result for the target technology FPGA, when area consumption is compared. For this technology a reduction for boolean function below 5 inputs doest reduce the used area, as seen in the lut description.

LUT description

For example, when using a gate array target technology the amount of used logic gates and latches is reduced compared to a one hot encoding. Reference to result table with latches

To summaries, an exact encoding algorithm should only be used if the target technology can utilize the improved solution and when the resulting enormous runtime increase is acceptable.

Results Data

State Machine	Area	Delay	Latch	Runtime
results_exact/004_dk15.kiss2.blif	8.0	1.00	8	0
results_exact/004_lion.kiss2.blif	1.5	1.00	3	0
results_exact/004_mc.kiss2.blif	3.5	1.00	7	0
results_exact/004_tav.kiss2.blif	8.0	2.00	7	1
results_exact/004_train4.kiss2.blif	1.5	1.00	3	0
results_exact/005_s8.kiss2.blif	7.0	2.00	4	0
results_exact/006_bbtas.kiss2.blif	5.0	1.00	6	0
results_exact/006_s27.kiss2.blif	7.5	2.00	5	1
results_exact/007_beecount.kiss2.blif	10.0	2.00	8	0
results_exact/007_dk14.kiss2.blif	8.0	1.00	8	0
results_exact/007_dk27.kiss2.blif	2.5	1.00	5	0
results_exact/008_dk17.kiss2.blif	6.5	1.00	7	1
results_exact/008_ex6.kiss2.blif	30.5	3.00	12	0
results_exact/008_shiftreg.kiss2.blif	1.5	1.00	4	0
results_exact/009_ex5.kiss2.blif	5.0	1.00	6	1
results_exact/009_lion9.kiss2.blif	5.0	1.00	5	0
results_exact/010_bbara.kiss2.blif	21.0	3.00	7	1
results_exact/010_ex3.kiss2.blif	6.0	1.00	6	1
results_exact/010_ex7.kiss2.blif	5.0	1.00	6	0
results_exact/010_opus.kiss2.blif	19.0	3.00	10	1
results_exact/011_train11.kiss2.blif	10.0	2.00	6	13
results_exact/012_modulo12.kiss2.blif	2.0	1.00	5	1
results_exact/014_ex4.kiss2.blif	16.0	2.00	14	6
results_exact/015_dk512.kiss2.blif	6.5	1.00	8	60000
results_exact/020_tma.kiss2.blif	39.5	3.00	11	0
results_exact/024_pma.kiss2.blif	88.5	4.00	13	1

Table 3.1: Results of Exact Encoding Approach Using Dichotomies

State Machine	Area	Delay	Latch	Runtime
results_nonexact/004_dk15.kiss2.blif	8.0	1.00	8	0
results_nonexact/004_lion.kiss2.blif	1.5	1.00	3	0
results_nonexact/004_mc.kiss2.blif	3.5	1.00	7	0
results_nonexact/004_tav.kiss2.blif	8.0	2.00	7	0
results_nonexact/004_train4.kiss2.blif	1.5	1.00	3	0
results_nonexact/005_s8.kiss2.blif	7.0	2.00	4	1
results_nonexact/006_bbtas.kiss2.blif	5.0	1.00	6	0
results_nonexact/006_s27.kiss2.blif	11.5	2.00	5	0
results_nonexact/007_beecount.kiss2.blif	10.0	2.00	8	0
results_nonexact/007_dk14.kiss2.blif	8.0	1.00	8	0
results_nonexact/007_dk27.kiss2.blif	3.0	1.00	6	0
results_nonexact/008_dk17.kiss2.blif	20.0	3.00	9	0
results_nonexact/008_ex6.kiss2.blif	25.0	3.00	12	0
results_nonexact/008_shiftreg.kiss2.blif	2.0	1.00	4	0
results_nonexact/009_ex5.kiss2.blif	5.0	1.00	6	1
results_nonexact/009_lion9.kiss2.blif	13.5	2.00	6	0
results_nonexact/010_bbara.kiss2.blif	21.5	3.00	7	0
results_nonexact/010_ex3.kiss2.blif	6.0	1.00	6	0
results_nonexact/010_ex7.kiss2.blif	5.0	1.00	6	0
results_nonexact/010_opus.kiss2.blif	24.0	3.00	11	0
results_nonexact/011_train11.kiss2.blif	18.0	3.00	7	0
results_nonexact/012_modulo12.kiss2.blif	2.0	1.00	5	0
results_nonexact/013_s386.kiss2.blif	42.0	3.00	12	0
results_nonexact/014_ex4.kiss2.blif	17.0	2.00	14	1
results_nonexact/015_dk512.kiss2.blif	6.5	1.00	8	0
results_nonexact/015_mark1.kiss2.blif	46.0	4.00	26	0
results_nonexact/016_bbsse.kiss2.blif	32.0	3.00	12	1
results_nonexact/016_cse.kiss2.blif	60.5	4.00	13	0
results_nonexact/016_kirkman.kiss2.blif	43.0	4.00	11	1
results_nonexact/016_sse.kiss2.blif	48.5	4.00	13	0
results_nonexact/018_s208.kiss2.blif	61.0	4.00	8	2
results_nonexact/018_s420.kiss2.blif	44.0	4.00	7	2
results_nonexact/019_ex2.kiss2.blif	19.0	3.00	7	1
results_nonexact/019_keyb.kiss2.blif	65.0	5.00	8	4
results_nonexact/020_ex1.kiss2.blif	77.5	4.00	26	4
results_nonexact/020_s1a.kiss2.blif	109.0	4.00	14	6
results_nonexact/020_s1.kiss2.blif	119.0	5.00	13	11
results_nonexact/020_tma.kiss2.blif	39.5	3.00	11	0
results_nonexact/024_donfile.kiss2.blif	19.5	3.00	6	66
results_nonexact/024_pma.kiss2.blif	88.5	4.00	13	0
results_nonexact/025_s820.kiss2.blif	153.5	6.00	27	393
results_nonexact/025_s832.kiss2.blif	158.0	6.00	28	452
results_nonexact/027_dk16.kiss2.blif	60.0	3.00	10	1243

Table 3.2: Results of Non-Exact Encoding Approach Using Dichotomies