An Approach for Dynamic Selection of Synthesis Transformations based on Markov Decision Processes

Tobias Welp[1] Andreas Kuehlmann[1,2]

[1] University of California at Berkeley, CA, USA
[2] Coverity, Inc., San Francisco, CA, USA

Abstract

Modern logic synthesis systems apply a sequence of loosely-related function-preserving transformations to gradually improve the circuit with respect to certain criteria such as area, performance, power, etc. For the quality of a complete synthesis run, the application order of the transformations for the individual steps are critical as they can produce vastly different outcomes. In practice, the transformation sequences is encoded in synthesis scripts which are derived manually based on experience and intuition of the tool developer. These scripts are static in the sense that transformations are applied independently of the result of previous transformations or the current status of the design. Despite the importance of obtaining high quality scripts, there are only a few attempts to optimize them. In this paper, we present a novel method to select transformations dynamically during the synthesis run leveraging the theory of Markov Decision Processes. The decision to select a particular transformation is based on transition probabilities, the history of the applied synthesis steps, and expectations for future steps. We report experimental results obtained from an implementation of the approach using the logic synthesis system ABC.

1 Introduction

Logic synthesis systems offer a range of equivalence-preserving transformations targeting the gradual improvement of circuits. Examples for these transformations are SAT-sweeping [1] and circuit rewriting [2]. A complete synthesis flow is composed of a sequence of these transformations.

The composition of a good synthesis sequence is complex because of three problems: (1) Even though the impacts of single transformations are most often well understood and analyzed, the interactions between different transformations are mostly unknown. (2) There is no universally good sequence, i.e., a good transformation sequence for one circuit is often suboptimal for other circuits [3]. (3) The quality of a sequence is dependent on optimization criteria targeted by the designer. In different situations, the designer will be interested in a different synthesis flow.

The traditional solution of this problem are synthesis scripts, which are manually created by the tool developers based on experiments and domain experience. This is an unsatisfactory solution for designers: Firstly, the generic goals for the design of a synthesis script are mostly unknown, hence the designers have no possibility to evaluate the suitability of a given script for their purposes. Secondly, even if the script is suitable in terms of the underlying optimization goal, it likely leads to suboptimal results due to problem (2) described above. Designers usually lack sufficient domain expertise to manually generate or update scripts of satisfactory quality on their own.

This paper proposes a novel approach based on Markov Decision Processes (MDPs) to generate synthesis scripts dynamically during the actual synthesis run. The framework attempts to solve all three problems reported above: Firstly, it captures the interactions between different transformations probabilistically using knowledge from a large set of experimental synthesis runs. Secondly, it selects the next synthesis transformation dynamically after having analyzed the response of the circuit to the application of previous transformations rather than statically fixing all transformations upfront. This allows the algorithm to accommodate the specific behaviour of a circuit. Thirdly, our framework allows the specification of optimization targets in the form of reward functions, enabling designers to generate good scripts with respect to their individual demands automatically without expert knowledge about the idiosyncrasies of individual synthesis transformations and their interactions.

2 Prior Art

2.1 Synthesis Script Optimization

Historically, synthesis scripts are tuned manually using experimentation and domain experience [4]. This process is time consuming and is unlikely to yield good results as the interaction of different synthesis procedures are generally not well understood.

A first automated process for the generation of synthesis scripts was presented in [3]. In this work, the space of all possible synthesis scripts is represented as a formal grammar and a genetic algorithm is used for optimization. In contrast to this work, our approach is dynamic, i.e. the decision for a synthesis transformation is dependent on the design responses to previously applied synthesis transformations.

2.2 Markov Decision Processes

MDPs were originally introduced in [5] to solve the following problem: We are given a set of states S and a set of actions A. In each state $s \in S$, an action $a \in A$ is chosen. Upon execution of this action, the state changes. However, the actual state transitions are non-deterministic. Given we are in state s and choose action a, the probability for a transition to state s' is given by $p(s'|a,s)$. With each transition, we associate a reward $R : (s,a,s') \to \mathbb{R}$. Given a policy $\pi : s \to a$, which maps each state s to an action a, we can calculate the expected overall reward R_o of the infinite sequence of transitions from an initial state $s^{(0)} \in S$ as

$$R_o\left(s^{(0)}\right) = E\left\{\sum_{t=0}^{\infty} \gamma^t R\left(s^{(t)}, a^{(t)}, s^{(t+1)}\right) | a^{(t)} = \pi(s^{(t)})\right\}$$

The constant $\gamma \in (0,1)$ is a discount factor which weights immediate rewards higher than those in the future.

978-3-9810801-7-9/DATE11/©2011 EDAA

Our objective is to find the optimal policy π^* among all policies such that the expected overall reward $R_o(s^{(0)})$ is maximized, i.e.

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \; R_o\left(s^{(0)}\right)$$

To solve this problem, we associate a utility $U(s)$ with each state s which corresponds to the expected overall reward when starting at s and choosing actions according to the optimal policy π^*. Assuming that we have calculated these utilities, it is possible to determine the optimal policy $\pi^*(s)$ in state s: We choose action a which maximizes the sum of expected immediate reward and discounted utility of the expected next state. Formally, we have

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} p(s'|a,s)\left(R(s,a,s') + \gamma\, U(s')\right)$$

The utilities $U(s)$ can be described by the following system of recurrences

$$U(s) = \max_a \sum_{s'} p(s'|a,s)\left(R(s,a,s') + \gamma\, U(s')\right) \quad \forall s \qquad (1)$$

The intuition behind these formulae is the following: In state s, the optimal policy π^* will select action a with highest sum of expected immediate reward and discounted utility of the next state. Hence, the utility of the state is the value associated with this action.

To solve the system of nonlinear equations (1), several methods have been proposed, among them value iteration [5], policy iteration [6], and an approach based on linear programming [7].

Assuming the Markov Property, i.e. that the probability $p(s'|a,s)$ of the next state given the current state and the chosen action does not depend on the past states, π^* as defined above can be proved to be the solution to the given optimization problem [5].

3 Dynamic Selection of Transformations

We consider a sequence of synthesis transformations $t^{(1)}, t^{(2)}, \ldots$ where $t^{(i)}$ is the transformation executed at step i. While executing a transformation $t^{(i)}$, we can observe a relative change $\delta^{(i)} = (\delta_a^{(i)}, \delta_d^{(i)}, \delta_p^{(i)})$ to the circuit, where $\delta_a^{(i)}$ is the relative change in area, $\delta_d^{(i)}$ the relative change in delay, and $\delta_p^{(i)}$ the relative change in power consumption of the circuit. We assume that we are given probabilities such as $p(\delta^{(i)}|t^{(1)},\ldots,t^{(i)},\delta^{(1)},\ldots,\delta^{(i-1)})$ which is the probability that $\delta^{(i)}$ is observed assuming that transformation $t^{(i)}$ is executed and that we previously executed the sequence of transformations $t^{(1)},\ldots,t^{(i-1)}$ and observed the changes $\delta^{(1)},\ldots,\delta^{(i-1)}$. Additionally, we define a reward function $R : (t,\delta) \to \mathbb{R}$, which maps a reward to executing transformation t and observing change δ of the circuit.

We attempt to solve the problem of finding a sequence of transformations $t^{(1)}, t^{(2)}, \ldots$ such that the expected value of the overall reward $R_o = R(t^{(1)},\delta^{(1)}) + R(t^{(2)},\delta^{(2)}) + \ldots$ is maximized. For this purpose, we propose to use MDPs to chose at each step i the best next transformation $t^{(i)}$. In this setup, the transformations take the role of the actions. As states we use the most recent k transformations with their respective impact on the quality of the circuit

$$s_k^{(i)} = \left(t^{(i-k)},\ldots,t^{(i)},\delta^{(i-k)},\ldots,\delta^{(i)}\right)$$

Using this mapping, we obtain the following equations for our MDP

$$U(s_k^{(i)}) = \max_t \sum_\delta p(\delta|t,s_k^{(i)})\left(R(t,\delta) + U(s_k^{(i+1)})\right) \qquad (2)$$

$$t^{(i+1)} = \underset{t}{\operatorname{argmax}} \sum_\delta p(\delta|t,s_k^{(i)})\left(R(t,\delta) + U(s_k^{(i+1)})\right) \qquad (3)$$

The parameter k controls how much of the synthesis history is taken into account. For $k \to \infty$, we consider the complete synthesis history. Then, the Markov property holds which implies that the chosen synthesis transformations will be the optimal ones given the available information. Note, however, that this approach is infeasible as the statespace would grow exponentially with the length of the synthesis sequence. A finite k reduces the state space by merging states with equal recent synthesis history. This violates the Markov property, invalidating the optimality result of MDPs. To alleviate this problem, we augment the state with easily captured context information from the circuit under synthesis, such as the current area of the circuit relative to the initial circuit size.

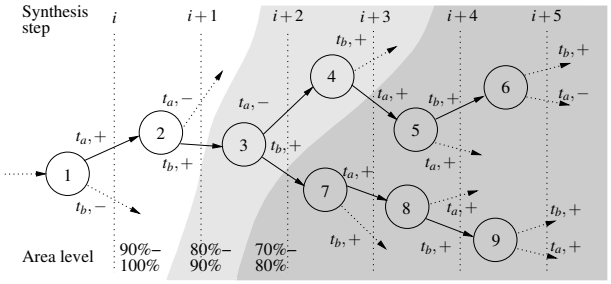

Figure 1: Illustration of applied state space reduction.

To illustrate these ideas, consider the example in Figure 1, which depicts a tree of possible synthesis flows starting at step i. Herein, we assume that $k = 2$ and that there are only two synthesis transformations t_a, t_b which can either improve $(+)$ or deteriorate $(-)$ the quality of the circuit. We augmented the state with the area level quantized in 10% steps. In this example, states $6, 9$ would be merged, as they have equal recent history of transformations and associated changes $(t_a,+),(t_b,+)$ and because they are on the same area levels.

Even when we augment the k-state with context information, the bounded approach is not guaranteed to deliver optimal results. However, we conjecture that the last k steps combined with context information capture the most relevant information and that the solution of the problem on the reduced state space gives a good approximation for the optimal solution. Our experimental results in Section 5 validate this conjecture.

4 Description of Prototype Implementation

To test the applicability of the described theory in a real synthesis framework, we implemented a prototype into the synthesis tool ABC [8]. The implementation requires three phases: Firstly, we estimate the transition probabilities $p(\delta|t,s_k)$. Secondly, we calculate the utilities $U(s_k)$. Thirdly, we execute the actual logic synthesis choosing transformations adhering to the optimal policy π^*. In the following, we describe the implementation details of each phase.

4.1 Estimation of Transition Probabilities

We use a Monte Carlo approach for the estimation of the probabilities $p(\delta|t,s_k)$. Therefore, we exercise designs with transformations selected uniformly at random. After the application of a transformation, we record its impact on the circuit along with the sequence of past transformations. Denoting the number of times change δ is observed running transformation t in state s_k with $c(\delta|t,s_k)$, probability $p(\delta|t,s_k)$ can be estimated by using

$$\hat{p}(\delta|t,s) = \frac{c(\delta|t,s_k)}{\sum_i c(\delta_i|t,s_k)} \qquad (4)$$

where the sum in the denominator iterates over all possible changes. The total number of probabilities to be estimated is bounded by $|T|^{k+1}|D|^{k+1}|C|$, where T is the set of possible transformations, D the set containing all possible changes, and C the set of possible contexts the circuit can be in.

For our prototype implementation, we selected 6 technology independent transformations offered by ABC. Further, we quantized area δ_a and delay changes δ_d into five bins each $\{++,+,+-,-,--\}$, $++$ standing for big improvement ($\geq$ 1.5%), $-$ for a small improvement ($\geq 0.5\%$), $+-$ for basically no change ($< 0.5\%$), $-$ for a small deterioration ($\geq 0.5\%$) and $--$ for a big deterioration ($\geq 1.5\%$) We chose not to consider δ_p as ABC lacks a suitable way of measuring the power consumption. Consequently, D has $5 \times 5 = 25$ elements. We augmented the state with the current area/delay relative to the initial area/delay of the circuit, quantized into 20 bins each. Hence, C contains 20^2 elements.

In our experimentation, we selected $k = 2$, which means that the number of probabilities to be calculated is at most 1.35G. This, however, is a vast over-approximation. Many probabilities are known or can be conjectured to be zero. For instance, we know that the probability of improvement by repetitive, subsequent application of SAT-sweeping is zero. [1]. We also validated this result in our experimentation: After generating about 8M samples, we found that less than 140k probabilities have been nonzero. This sparsity has two fortunate consequences: It reduces the storage requirements dramatically and speeds up the convergence of the estimator (4).

4.2 Calculation of State Utilities

The values of the utilities depend on the transition probabilities and the reward function $R(t,\delta)$. In the following, we first describe guidelines for the definition of reward functions and then the actual utility calculation via value iteration.

4.2.1 Definition of Reward Function

The reward functions allow designers to specify their requirements towards synthesis. Depending on context, they will be interested in defining the reward function differently: For instance, in the early design phase when synthesis is used to obtain rough area and delay estimates, designers will be interested in running fast synthesis runs without thorough optimization. Later, when the design is synthesized for manufacturing, focus will be on quality of results (QOR) even if the synthesis takes substantially longer.

Our synthesis framework allows designers to freely specify the reward function according to their needs. The class of reward functions which has been utilized in our experimentation is a weighted sum of the factors area change δ_a, delay change δ_d, and running time $r(t)$ of operation t:

$$R(t,\delta) = f_a\delta_a + f_d\delta_d - f_t\mathrm{r}(t) \tag{5}$$

The constants f_a, f_d, f_t are the weighting factors for area, delay, and running time, respectively.

4.2.2 Calculation of State Utilities

For the calculation of the utilities, we use value iteration as coded in Algorithm 1. The function NEXTSTATE(s_k,t,δ) predicts the next state contingent on the current state s_k, the chosen transformation t, and the change in area and delay δ. Note that this cannot be calculated exactly due to the applied quantification. In our implementation, we assumed that a small/big change in area/delay leads to 1%/2% change of the area/delay of the circuit. In our experimentation, this prediction turned out sufficiently precise.

Algorithm 1 VALUEITERATION

```
initialize all U'(s_k) = 0
repeat
  for all s_k do
    U(s_k) ← U'(s_k)
  for all s_k do
    U'(s_k) ← max_t Σ_δ p(δ|t,s_k)
                 ×(R(t,δ)) + γ U(NEXTSTATE(s_k,t,δ))
    d̃ ← |U(s_k) − U'(s_k)|
    if d̃ > d then
      d = d̃
until d < ε(1−γ)/γ
```

With $R_{\max}$ being the maximum possible reward of a transition, Algorithm 1 is guaranteed to terminate after

$$N = \left\lceil \left(\log \frac{2R_{\max}}{\varepsilon(1-\gamma)} \right) \left(\log(\gamma^{-1}) \right)^{-1} \right\rceil$$

steps with an error bound of $|U(s_k) - U^*(s_k)| < \varepsilon$ for all states s_k where $U^*(s_k)$ is the exact utility of state s_k [5].

In practice, value iteration has the appealing property that it can be interrupted at any time in order to obtain approximate results. In our experimentation, we were able to obtain approximations which led to good synthesis results after few iterations (< 30).

4.3 Running Logic Synthesis based on MDPs

The procedure describing the actual synthesis process is given in Algorithm 2. At each step, transformation t with highest value v is selected depending on s_k using formulae (2) and (3). In case the selected transformation t has positive value v and no cycle was detected, the transformation will be executed, otherwise the algorithm terminates.

Cycle detection is required to assure termination of the algorithm. Otherwise, speciously promising transformations can be repetitively selected infinitely often. Futile transformations can appear promising because of the applied state space reduction and imprecisely estimated probabilities. In our implementation, a cycle is detected if a state with no recent improvement is visited twice. With the help of this necessary criterion for a cycle, the implemented algorithm is guaranteed to terminate as there is only a finite number of states and the circuit cannot improve infinitely often without deteriorating eventually. In this case, cycle detection will interrupt the loop.

Algorithm 2 MDPBASEDSYNTHESIS

```
Initialize s_k
while true do
  (v, t) = SELECTTRANSFORMATION(s_k)
  if v > 0 and not CYCLEDETECTED(s_k) then
    s_k ← RUNTRANSFORMATION(t, s_k)
  else
    return
```

5 Experimental Results

We performed two experiments to evaluate our approach. Firstly, we compare the QOR of synthesis runs using our approach with those obtained by running the standard script delivered as part of the ABC tool. Secondly, we show that our approach is capable of generating synthesis scripts for specific user requirements.

Benchmark	# Nodes	Δ Nodes	Δ Depths	Δ Time	Δ R_o
ac97_ctrl	10262	2.4%	0.0%	2.2%	-1.24
aes_core	20451	2.7%	5.0%	-51.8%	3.00
des_area	4413	-0.2%	-3.7%	-30.9%	2.26
des_perf	74453	0.7%	0.0%	-37.3%	0.75
ethernet	56206	0.6%	-3.7%	172.6%	5.09
i2c	936	-3.5%	9.1%	12.5%	3.21
mem_ctrl	8564	-16.4%	3.6%	23.6%	0.71
pci_bridge	16407	0.4%	18.2%	14.0%	-0.44
pci_conf	84	2.4%	0.0%	-20.0%	3.43
pci_spoci	821	-6.2%	-6.7%	65.4%	3.17
sasc	552	2.2%	0.0%	-28.6%	6.75
simple_spi	758	1.3%	0.0%	-20.5%	8.5
spi	3214	-1.1%	7.1%	10.5%	0.84
ss_pcm	394	1.0%	16.7%	-26.7%	0.88
stepper	155	-1.3%	0.0%	-4.6%	0.36
systemcaes	9930	-3.4%	-7.9%	43.2%	2.89
systemcdes	2446	-4.0%	4.4%	-10.9%	0.95
tv80	7264	-3.1%	7.3%	9.1%	1.49
usb_funct	13332	-0.3%	17.4%	26.0%	-10.29
usb_phy	357	0.3%	-11.1%	0.0%	0.64
vga_lcd	88818	0.2%	-5.6%	143.8%	4.74
wb_conmax	40522	-4.4%	0.0%	-34.7%	8.14

Table 1: Change in QOR of MDP-based synthesis over *resyn2*.

5.1 Comparison with Standard Script in ABC

Herein, we compare the performance of synthesis sequences generated by our tool (MDP) with that of the ABC standard script *resyn2* on the designs in the subset of the IWLS benchmark set [9] originating from the OpenCores repository. We estimated the probabilities using a set of roughly 8M samples, generated by an equal number of random walks as discussed in Section 4.1 on each of the available benchmark circuits. As reward function, we used template (5) with parameters $f_a = f_d = f_t = 1$. Intuitively, this means that area and delay improvements are weighted equally and that a small improvement $(+)$ in area or delay justifies the running time of a transformation with effort comparable to that of a rewrite operation. After calculating the utilities, we synthesized each benchmark using *resyn2* and our tool.

The results are listed in Table 1. Column 2 gives the sizes of the respective benchmark listed in column 1 and columns 3, 4, and 5 contain the changes in QOR and running time of using our approach instead of using *resyn2*. For instance, the entry in column 3 for *ac97_ctrl* means that synthesizing the design with our approach yields a design with 2.4% larger area than using *resyn2*. The last column contains the absolute difference in terms of the defined reward function by using our approach instead of *resyn2*.

Comparing the respective results in terms of area, delay, and running time, one can observe that the synthesis sequence generated by our tool has similar performance as *resyn2*. A direct comparison is difficult because a better result of one approach in one category (e.g. area) is often accompanied with a worse result in another category (e.g. running time). In terms of the total reward R_o (column 4), our approach yields better results in almost all cases.

The results are quite remarkable, noting that our tool generated the synthesis sequences without any domain knowledge and that we are comparing against a well-tuned script for this benchmark set.

5.2 Optimization for Different Reward Functions

Next, we investigated the impact of different reward functions on the synthesis results. Specifically, we changed the parameters of template (5) to $f_a = 1$ and $f_d = 0$. This specifies that the synthesis focuses only on area improvement. For f_t, we ran our tool with values $0.05, 0.10, 0.25, 0.5, 1, 2, 4, 8, 16$. As explained earlier,

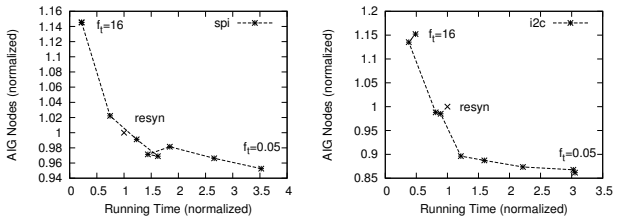

Figure 2: Impact of reward function on QOR.

$f_t = 1$ specifies that a small improvement in area $(+)$ justifies the computational effort of a transformation with running time comparable to that of a rewrite operation. Other values for f_t increase or decrease this proportion linearly, e.g. $f_t = 2$ would put a twice as big weight on running time.

Figure 2 shows the results of the experiment for a choice of 2 benchmarks. Note that we ran the experiments for all benchmarks in the benchmark set with similar results. Each connected point in the plot correspond to one setting of f_t. The plot is normalized with respect to the performance of the standard script *resyn* (indicated by an $\times$).

The plots clearly show that our tool is able to accommodate user requests represented in the reward function.

6 Conclusions and Future Work

The composition of good synthesis sequences is complex and static synthesis scripts are no satisfactory solution. Herein, we presented a novel, dynamic approach based on MDPs. We implemented a prototype of our approach into the logic synthesis system ABC and reported results indicating that our approach performs well and accommodates user requests for different optimization goals.

To improve the efficiency of the described approach, it might be worthwhile to investigate competing approaches for the calculation of the utilities. Our literature study suggests that we can expect an efficiency improvement in using policy iteration or linear programming instead of value iteration. Further, it would be interesting to investigate how the MDP-based approach performs in other synthesis domains. For instance, we conjecture that our approach might also be useful in the area of layout synthesis.

References

[1] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *Digest Tech. Papers IEEE/ACM Int'l Conf. Computer-Aided Design*, (San Jose, California), pp. 50–57, Nov. 2004.

[2] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *Design Automation Conference, 2006 43rd ACM/IEEE*, pp. 532 –535, 0-0 2006.

[3] A. Kuehlmann and L. Van Ginneken, "Grammar-based optimization of synthesis scenarios," in *Computer Design: VLSI in Computers and Processors, 1994. ICCD '94. Proceedings., IEEE International Conference on*, pp. 20 –25, oct 1994.

[4] M. Pipponzi, "MDRIVER: A strategy generation for multiple-level optimization," in *Proceedings of the M.C.N.C. Workshop on Logic Synthesis*, April 1991.

[5] R. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

[6] R. A. Howard, *Dynamic Programming and Markov Processes*. The M.I.T. Press, 1960.

[7] F. d'Epenoux, "A probabilisitic production and inventory problem," in *Management Sciene*, pp. 10:98–108, 1963.

[8] Berkeley Logic Synthesis and Verification Group, "A system for sequential synthesis and verification," Release 70930.

[9] C. Albrecht, *IWLS benchmark library*. http://www.iwls.org/iwls2005/benchmarks.html, 2005.