International Core Journal of Engineering Volume 11 Issue 6, 2025
ISSN: 2414-1895 DOI: 10.6919/ICJE.202506_11(6).0050

A Review on Meta-Heuristic and Reinforcement Learning for VLSI Floor Planning

Qingye Wei
University Road, Southampton, SO17 1BJ, United Kingdom

Abstract

Floor planning is a foundational step in the physical design of very-large-scale integration (VLSI) circuits, directly influencing chip area utilization, interconnect length, and overall performance. Owing to the NP-hard nature of both two-dimensional (2D) and three-dimensional (3D) floor planning problems, exact optimization methods become computationally prohibitive as design complexity increases. This paper provides a comprehensive survey of two major solution paradigms: classical meta-heuristic algorithms and modern reinforcement learning techniques. We first review widely adopted meta-heuristics-including simulated annealing, genetic algorithms, particle swarm optimization, and ant colony optimization-highlighting their operational principles, strengths in global search, and practical challenges related to parameter tuning and convergence. Next, we explore reinforcement learning frameworks that cast floor planning as a sequential decision-making task, with deep reinforcement learning agents capable of learning placement policies in high-dimensional spaces. Comparative analysis reveals that while meta-heuristics excel in adaptability and ease of implementation, reinforcement learning offers potential for automated, data-driven optimization with reduced manual intervention. Finally, we discuss emerging hybrid approaches that integrate meta-heuristic exploration with learned policy refinement, pointing toward future research directions aimed at achieving efficient, high-quality layouts for next-generation VLSI designs.

Keywords

VLSI Floor Planning; Meta-Heuristic Algorithms; Deep Reinforcement Learning.

1. Introduction

Floor planning is a critical step in the physical design of very-large-scale integration (VLSI) circuits, directly impacting chip area, wirelength, and overall performance [1] [8]. As technology scales and design complexity grows, both two-dimensional (2D) and three-dimensional (3D) floor planning problems have become increasingly challenging [2]. These problems are known to be NP-hard [13], meaning that finding exact optimal solutions within reasonable time bounds is infeasible for large-scale instances. Consequently, researchers have turned to approximate methods-particularly meta-heuristic algorithms-to navigate the vast solution space and produce high-quality layouts in a practical timeframe .

Meta-heuristic approaches such as simulated annealing, genetic algorithms, particle swarm optimization, and ant colony optimization have demonstrated strong global search capabilities and adaptability across diverse optimization landscapes. Simulated annealing, one of the earliest techniques applied to VLSI floor planning, mimics the physical process of cooling to escape local optima, while genetic algorithms leverage evolutionary principles of selection, crossover, and mutation to evolve populations of candidate solutions. Particle swarm and ant colony methods,

ISSN: 2414-1895 DOI: 10.6919/ICJE.202506_11(6).0050

inspired by social behaviors in nature, further enrich the toolkit by balancing exploration and exploitation through collective intelligence mechanisms.

Meanwhile, the advent of reinforcement learning (RL) has opened new avenues for automated layout design. By framing floor planning as a sequential decision-making problem, RL agents can learn placement strategies through trial-and-error interactions with a simulated environment. Deep reinforcement learning, in particular, combines the representational power of neural networks with the trial-based optimization of RL, enabling the handling of high-dimensional state and action spaces inherent in modern VLSI design. This survey explores both classical meta-heuristic algorithms and contemporary reinforcement learning techniques, comparing their strengths, limitations, and potential synergies for future research in floor planning.

2. Meta-heuristic Algorithms

Because the floor planning problems of both 2D integrated circuits and 3D integrated circuits are NP-hard problems [8], researchers usually use meta-heuristic algorithms, such as simulated annealing algorithm, to solve the floor planning problem. Next, we will introduce some commonly used meta-heuristic algorithms in floor planning problem.

2.1 Simulated Annealing Algorithm

Simulated Annealing (SA) algorithm is one of the earliest and most popular algorithms used in various stages of VLSI physical design [15]. The floor planning algorithm based on simulated annealing is easy to implement. Many researchers have achieved good results in floor planning problems by combining sequence pairs, cornea block sequences, B * -tree and other floor planning representation methods.

The core idea of simulated annealing algorithm is to gradually reduce the energy of the system by simulating the solid heat annealing process, in order to expect to find a better solution in the search space. The algorithm simulates the process of solid matter from high temperature to low temperature, allowing large oscillations and accepting the probability of inferior solutions at high temperatures, and reducing the amplitude of oscillations and the probability of accepting inferior solutions at low temperatures.

The specific steps of the simulated annealing algorithm are as follows:

Step one: Initialization. At the beginning of the algorithm, set the initial temperature as the starting point of annealing, and randomly set an initial solution as the current solution.

Step two: generate a neighborhood solution by perturbation, and then in each step, generate a neighborhood solution by applying a random perturbation to the current solution, and calculate the corresponding objective function value.

Step three: Sample neighborhood solutions according to the Metropolis rule. If the objective function value of the neighborhood solution is better, that is, closer to the optimal solution, then the neighborhood solution is taken as the current solution; if the objective function value of the current solution is better, then the probability of accepting the neighborhood solution is calculated according to the Metropolis rule, and whether to accept the neighborhood solution as the current solution is determined according to the probability [3]. The probability calculation formula is as follows:

$$P(s,s',t_k)=\begin{cases}1 & \text{if } f(s') \le f(s) \\ \exp\left(\dfrac{-(f(s')-}{t_k}\right. & \text{if } f(s') > f(s)\end{cases} \tag{1}$$

where f(s) is the objective function, t_k is the temperature of the kth iteration, and s and s′ are the current solution and neighborhood solution, respectively.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202506_11(6).0050

Step four: Temperature update, gradually reduce the temperature according to the preset cooling schedule. Common cooling methods include linear cooling and exponential cooling.

Step five: Iterative repetition, repeat steps two to four until the temperature drops to the set termination temperature. At each iteration, an attempt is made to find a better solution in the search space, and as the temperature decreases, the probability of accepting a worse solution decreases gradually.

Simulated annealing algorithm is widely used in optimization problems in many fields, it has better global search ability and adaptability, and has certain advantages in solving complex problems and non-convex problems. The key lies in the probability of accepting inferior solutions and the setting of temperature, reasonable configuration can balance the ability of global search and local search, avoid falling into local optimal solution. In addition, the convergence speed of the algorithm is also related to the characteristics of the problem itself, so it is necessary to carefully adjust the parameters in practical applications to obtain better results.

2.2 Genetic Algorithms

Genetic Algorithm (GA) is a population-based evolutionary optimization technique inspired by the principles of natural selection and genetic inheritance. It operates on a population of candidate solutions, represented as chromosomes, and evolves them over successive generations through biologically-inspired operations such as selection, crossover, and mutation. Each individual in the population encodes a potential solution to the optimization problem and is evaluated using a fitness function that quantifies its performance with respect to the objective. Individuals with higher fitness are more likely to be selected for reproduction, thereby propagating advantageous traits to subsequent generations [4]. The crossover operation recombines genetic information from pairs of selected parents to produce offspring with potentially superior characteristics [9], while mutation introduces random alterations to maintain genetic diversity and avoid premature convergence. Over iterative updates, the population evolves toward increasingly optimal regions of the solution space. GA is well-regarded for its global search capability, robustness, and parallelism, making it effective in tackling high-dimensional and multimodal optimization problems. However, its performance is highly dependent on parameter settings-such as population size, mutation rate, and crossover probability-as well as the encoding and fitness evaluation strategies. Consequently, careful calibration and adaptive control of parameters are often required to ensure convergence efficiency and solution quality in practical applications.

2.3 Particle Swarm Optimization Algorithms

Particle Swarm Optimization (PSO) is a heuristic optimization algorithm that simulates the behavior of bird flocks or animal swarms. In the particle swarm algorithm, each individual is called a particle, and these particles form a population. The particle swarm optimization algorithm starts from random examples, and each example flies in the solution space at a certain speed and is updated based on its own and its neighbor's experience. The execution process of the particle swarm optimization algorithm is as follows:

Step one: Initialize the particle swarm by randomly generating a set of initial positions and velocities for the particles. The initial position of each particle represents a candidate solution, and the velocity determines the direction and distance of the particle's movement.

Step two: Evaluate the fitness of each example. Evaluate the fitness of each particle based on the specific objective function of the problem. The fitness value represents the quality of the particle's solution.

Step three: Update the velocity and position of each particle. Each example updates its velocity and position based on its own historical best position, which is the local optimal solution, and the historical best position of the entire population, which is the global optimal solution (Fbest), as well as some random factors. The update formula is as follows:

ISSN: 2414-1895 DOI: 10.6919/ICJE.202506_11(6).0050

$$V_i = w * V_{i-1} + c_1 * r_1 * (P_{best} - X_{i-1}) + c_2 * r_2 * (G_{best} - X_{i-1})$$
$$X_i = V_i + X_{i-1} \quad (2)$$

where w is the inertia weight, c1 and c2 are the learning factors of individuals and groups, and r1 and r2 are random numbers between [0,1].

Step four: Update the global optimal solution, compare the fitness of each example, and update the global optimal solution of the population.

Step 5: Repeat iterations, repeating steps 2 through 4 until the stopping criteria are met, such as reaching a predetermined number of iterations or finding a satisfactory fitness solution.

The key in the particle swarm optimization algorithm is the velocity update and position update of the particles. Velocity update involves the particle's retention of its own historical optimal solution and its attraction to the global optimal solution, and balances individuality and collectivity by introducing self-experience terms and global experience terms into velocity. Position update updates the position of particles in solution space based on the new velocity value.

The advantage of the particle swarm optimization algorithm lies in its simplicity and ease of implementation, which can perform a fast global search in the solution space and has strong convergence and adaptability [6]. However, there are also some challenges in the particle swarm optimization algorithm, such as the difficulty of falling into the local optimal solution and the sensitivity of parameter setting, so it is necessary to carefully select and adjust the algorithm parameters in practical applications.

2.4 Ant Colony Optimization Algorithms

Ant Colony Optimization (ACO) is a nature-inspired meta-heuristic algorithm that emulates the pheromone-mediated foraging behavior of ants to solve complex combinatorial optimization problems. In ACO, the solution space is conceptualized as a graph in which artificial ants iteratively construct candidate solutions by traversing nodes based on a probabilistic transition rule. This rule is governed by two main factors: the intensity of pheromones deposited on paths, which encodes the collective experience of the colony, and problem-specific heuristic information, which guides ants toward locally promising regions. During each iteration, ants explore the solution space independently and release pheromones along their paths. These pheromones undergo dynamic updates—subject to evaporation to prevent convergence stagnation and reinforcement on high-quality paths to bias future exploration. Through repeated interactions, the algorithm promotes convergence toward optimal or near-optimal solutions as paths with higher pheromone concentrations become increasingly attractive to the ant population. ACO has demonstrated robust performance in solving NP-hard problems such as the Traveling Salesman Problem (TSP), benefitting from its inherent parallelism, adaptability, and global search capability. However, the algorithm's efficacy is sensitive to parameter settings, including evaporation rate, pheromone influence, and heuristic weighting, and may exhibit slow convergence or premature stagnation if improperly configured [7]. Therefore, successful deployment of ACO in practice often necessitates careful parameter tuning and, in many cases, hybridization with local search techniques to enhance convergence stability and solution quality.

3. Reinforcement Learning

3.1 Introduction to Reinforcement Learning

Reinforcement learning (RL) has emerged as a prominent machine learning paradigm alongside the rapid advancement of artificial intelligence. At its core, RL is an interactive learning framework composed of four main elements: agent, environment, action, and reward. The agent interacts with the environment by taking actions, receives feedback in the form of rewards, and adjusts its strategy accordingly through trial-and-error to maximize long-term gains. The ultimate goal is to enable

ISSN: 2414-1895 DOI: 10.6919/ICJE.202506_11(6).0050

autonomous agents to learn optimal behaviors and improve performance through continuous interaction with their surroundings.

One of the long-standing objectives in AI is to develop fully autonomous systems capable of adaptive learning in dynamic environments. RL provides a mathematical foundation for such self-directed, experience-driven learning. While traditional RL methods have shown success in various domains, they often struggle with scalability and are limited to low-dimensional tasks due to challenges such as high memory, computational, and sample complexity.

The rise of deep learning has significantly enhanced RL by introducing powerful function approximation through deep neural networks. Deep learning facilitates automatic feature extraction from high-dimensional inputs such as images, text, and audio, effectively addressing the "curse of dimensionality" through hierarchical and inductive representations. The integration of these capabilities has led to the emergence of deep reinforcement learning (DRL), a field that combines the strengths of both frameworks to tackle more complex and high-dimensional decision-making problems.

3.2 Markov Decision-making Process

Reinforcement learning can be described as MDP. MDP can be expressed by (S, A, P, R, γ):

1) State space (S): A set of possible environment states.

2) Action space (A): The actions that a set of agents can take.

3) State transition probability (P): indicates the probability that the state s changes to s 'when the agent performs the action a.

4) Reward (R): A reward signal for environmental feedback.

5) Discount factor (γ): discount factor.

MDP is usually based on the Markov hypothesis, so the probability distribution of future states depends only on the current state, i.e [14]. the past sequence of states does not affect the environment. Figure 2.16 shows the interaction process between the agent and the environment. At t, the agent observes the state st from the environment and interacts with the environment by taking action at. When the agent takes action, based on the current state st and action at, the environment rewards the agent rt as feedback. Move to a new state st+1. In episodic tasks, this process continues until the agent reaches the termination state, and then the state is reset. Discount cumulative returns are defined as follows:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3}$$

Where $\gamma \epsilon (0,1)$ represents the discount factor and k represents the time step. The goal of reinforcement learning is to maximize the expected return from each state.

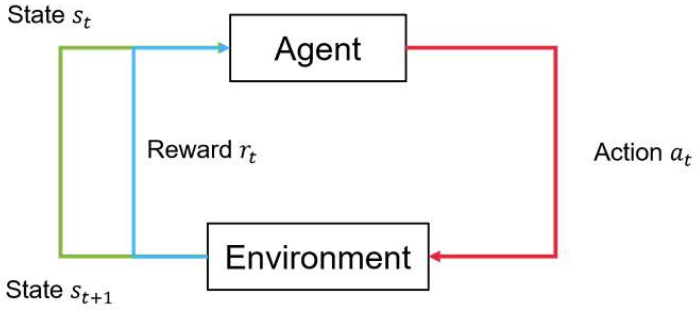

Figure 1. The interaction process between the agent and the environment

ISSN: 2414-1895 DOI: 10.6919/ICJE.202506_11(6).0050

3.3 Value Function

Value function Value function is an index used by reinforcement learning to evaluate the expected return, including state value function and action value function. The state value function $V\pi(s)$ represents the expected return of the state s obtained by the strategy π, which can be used to evaluate the status of the agent. The definition is as follows:

$$V_{\pi}(s) = E_{\pi}\left[G_t | S_t = s\right] \tag{4}$$

The $V\pi(s)$ equation is recursively obtained:

$$V_{\pi}(s_t) = E_{\pi}\left[r_t + \gamma V_{\pi}(s_{t+1})\right] \tag{5}$$

The optimal state-value function $V^*(s) = \max\pi V\pi(s) = \max a Q^*(s, a)$ represents the maximum state value achievable under any policy for state s. $V^*(s)$ can be obtained through a recursive approach, yielding its Bellman equation:

$$V^*(s_t) = E\left[r_t + \gamma Q^*(s_{t+1}, a_{t+1})\right] \tag{6}$$

The action-value function $Q\pi(s, a)$ represents the expected return obtained by executing action a in state s under policy π, which can be used to estimate the goodness or badness of the corresponding actions of the intelligent agent in this state, defined as follows:

$$Q_{\pi}(s, a) = E\left[G_t | S_t = s, A_t = a\right] \tag{7}$$

Obtain its Bellman equation through recursive means for $Q\pi(s, a)$:

$$Q_{\pi}(s_t, a_t) = E_{\pi}\left[r_t + \gamma Q_{\pi}(s_{t+1}, a_{t+1})\right] \tag{8}$$

The optimal action-value function $Q^*(s, a) = \max\pi Q\pi(s, a)$ represents the maximum action value that can be obtained by executing action a in state s under any policy, and $Q^*(s, a)$ is obtained recursively by its Bellman equation:

$$Q^*(s_t, a_t) = E\left[r_t + \gamma Q^*(s_{t+1}, a_{t+1})\right] \tag{9}$$

3.4 Temporal Difference Method

The solution methods for reinforcement learning include methods based on dynamic programming, Monte Carlo methods, and temporal difference methods. Temporal difference (TD) is the core of reinforcement learning, which uses differences over time steps for learning and updating. Traditional reinforcement learning algorithms such as the SARSA algorithm and the Q-learning algorithm are both based on temporal difference learning [10].

Temporal difference learning learns the state value function $V(s)$ based on the TD error, with the update rule as follows:

ISSN: 2414-1895 DOI: 10.6919/ICJE.202506_11(6).0050

$$V(s_t) \leftarrow V(s_t) + \alpha[r + \gamma V(s_{t+1}) - V(s_t)] \tag{10}$$

Here, α represents the learning rate, and $r + \gamma V(st+1) - V(st)$ is the TD error. Temporal difference methods employ bootstrapping, meaning that the update of the value function depends on the value of the current action or state and the value of the next action or state. Temporal difference methods form the basis for both the SARSA algorithm and the Q-learning algorithm. The SARSA algorithm is a policy (on-policy) algorithm that updates the action values using actions sampled from the current policy. Its update mechanism can be defined by the equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{11}$$

Q-learning is a model-free algorithm that uses a Q-table to store state-action values and selects the action that yields the maximum reward in the environment. Its update can be defined by the equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] \tag{12}$$

3.5 DNQ

Deep Reinforcement Learning (DRL) is a technique that combines Deep Learning with Reinforcement Learning. It enables the handling of complex problems with high-dimensional state and action spaces by utilizing deep neural networks [11].

In Deep Reinforcement Learning, the agent employs a deep neural network as a value function approximator or policy approximator to learn strategies that make optimal decisions in a given environment. The multi-layer architecture of deep neural networks provides powerful function approximation capabilities, allowing the agent to autonomously extract and learn useful features from raw or high-dimensional inputs.

DQN (Deep Q-network; Deep Q Network) is a core algorithm of deep reinforcement learning, which combines the reinforcement learning algorithm Q-Learning with deep neural network models to achieve learning and decision-making in complex environments. DQN solves the problem of unstable learning by employing techniques such as experience replay and target networks to stabilize the learning process.

The experience replay mechanism stores each step of experience in the form of a quadruplet (s_t, a_t, r_{t+1}, s_{t+1}) into the experience pool [12]. During training, it uses uniformly randomly sampled experience samples instead of continuously training with samples generated in real-time, reducing the correlation of samples during the training process and enhancing the stability of training. At the same time, experience samples can be resampled multiple times for training, improving the efficiency of data utilization.

The target network is another key technology used by DQN. DQN employs two neural networks for learning, namely the evaluate network and the target networ [13]. The evaluate network is responsible for selecting actions based on the current state and estimating the Q-value of actions; its parameters are updated in each training step. The target network is used to compute the target Q-value, and its network architecture is identical to that of the evaluate network, with its parameters updated by periodically copying those of the evaluate network. For each set of samples drawn from the experience pool, the evaluate network computes the Q-value of the current state, while the target network computes the maximum Q-value of the next state. The TD error is calculated to update the weights of the evaluate network. Since the parameters of the target network do not change frequently, the estimated target Q-value does not fluctuate dramatically. This helps to reduce instability during the training process, making the network more likely to converge. The evaluate network is used to predict the Q-value of each action in the current state, while the target network is used to compute

ISSN: 2414-1895 DOI: 10.6919/ICJE.202506_11(6).0050

the target Q-value. By separating the target network from the online network, the problem of self-influence during the update of target values is avoided.

DQN approximates the optimal action-value function $Q(s, a; \theta) \approx Q^*(s, a)$ using a neural network, where θ represents the parameters of the neural network. The loss function is defined based on the Bellman equation:

$$L(\theta_i) = E_{s,a\sim\rho(\cdot)}\left[\left(y_i - Q(s,a;\theta)\right)^2\right] = E_{(\cdot)}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s,a;\theta)\right)^2\right] \quad (13)$$

Where $\rho(\cdot)$ is the probability distribution with respect to the state s and action a, and yi is the TD target. The model is updated using the backpropagation algorithm.

4. Conclusion

This survey has examined two principal strategies for addressing the NP-hard VLSI floor-planning problem: classical meta-heuristic algorithms and modern reinforcement-learning methods. Meta-heuristics-such as simulated annealing, genetic algorithms, particle-swarm optimization, and ant-colony optimization-offer robust global search capabilities and have been widely adopted due to their flexibility and ease of implementation. Deep reinforcement learning reframes floor planning as a sequential decision task, enabling automated policy learning in high-dimensional design spaces [5]. While meta-heuristics excel in reliability and require relatively modest computational resources, reinforcement learning promises reduced manual tuning and the potential to adapt dynamically to new design rules. Future research should explore hybrid frameworks that combine the exploratory power of meta-heuristics with the adaptive learning strengths of reinforcement learning, aiming to deliver efficient, high-quality layouts for increasingly complex VLSI technologies.

References

[1] Nayak P. A study of technology roadmap for application-specific integrated circuit[D]. Rice University, 2021.

[2] Shalf J. The future of computing beyond Moore's Law[J]. Philosophical Transactions of the Royal Society A, 2020, 378(2166): 20190061.

[3] Leiserson C E, Thompson N C, Emer J S, et al. There's plenty of room at the Top: What will drive computer performance after Moore's law?[J]. Science, 2020, 368(6495): eaam9744.

[4] Burg D, Ausubel J H. Moore's Law revisited through Intel chip density[J]. PloS one, 2021, 16(8): e0256245.

[5] Ding B, Zhang Z H, Gong L, et al. A novel thermal management scheme for 3D-IC chips with multi-cores and high power density[J]. Applied thermal engineering, 2020, 168: 114832.

[6] Huang C Y, Dewey G, Mannebach E, et al. 3-D self-aligned stacked NMOS-on-PMOS nanoribbon transistors for continued Moore's law scaling[C]//2020 IEEE International Electron Devices Meeting (IEDM). IEEE, 2020: 20.6. 1-20.6. 4.

[7] Kang K, Benini L, De Micheli G. Cost-effective design of mesh-of-tree interconnect for multicore clusters with 3-D stacked L2 scratchpad memory[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2014, 23(9): 1828-1841.

[8] Ben Abdallah A, Ben Abdallah A. 3D integration technology for multicore systems on-chip[J]. Advanced Multicore Systems-On-Chip: Architecture, On-Chip Network, Design, 2017: 175-199.

[9] Hu X, Stow D, Xie Y. Die stacking is happening[J]. IEEE micro, 2018, 38(1): 22-28.

[10] Ghaderi Z, Alqahtani A, Bagherzadeh N. AROMa: aging-aware deadlock-free adaptive routing algorithm and online monitoring in 3D NoCs[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 29(4): 772-788.

[11] J. Cong, C. Liu, and G. Luo, "Quantitative studies of impact of 3D IC design on repeater usage," in Proc. Int. VLSI/ULSI Multilevel Interconnection Conf., 2008, pp. 344–348.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202506_11(6).0050

[12] Kim D H, Athikulwongse K, Healy M B, et al. Design and analysis of 3D-MAPS (3D massively parallel processor with stacked memory)[J]. IEEE Transactions on Computers, 2013, 64(1): 112 125.

[13] Talbi E G. Metaheuristics: From Design to Implementation[M]. John Wiley & Sons, 2009.

[14] Mnih V, Kavukcuoglu K, Silver D, et al. Human-level control through deep reinforcement learning[J]. Nature, 2015, 518(7540): 529–533.

[15] Cong J, Huang Y, Yuan B, et al. A topology-driven floorplanner with preplaced modules and soft blocks for 3D ICs[J]. ACM Transactions on Design Automation of Electronic Systems (TODAES), 2011, 16(4): 1–20.