Evaluation and Validation

Jian-Jia Chen
(Slides are based on Peter Marwedel)
TU Dortmund, Informatik 12
Germany

2017年 01 月 10/11 日

These slides use Microsoft clip arts. Microsoft copyright restrictions apply.
Structure of this course

2: Specification
3: ES-hardware
4: system software (RTOS, middleware, …)

Design repository

6: Application mapping
7: Optimization
5: Evaluation & validation (energy, cost, performance, …)

Application Knowledge

Design
8: Test

Numbers denote sequence of chapters
Validation and Evaluation

Definition: **Validation** is the process of checking whether or not a certain (possibly partial) design is appropriate for its purpose, meets all constraints and will perform as expected (yes/no decision).

Definition: Validation with mathematical rigor is called **(formal) verification**.

Definition: **Evaluation** is the process of computing quantitative information of some key characteristics of a certain (possibly partial) design.
How to evaluate designs according to multiple criteria?

Many different criteria are relevant for evaluating designs:

- Average & worst case delay
- Power/energy consumption
- Thermal behavior
- Reliability, safety, security
- Cost, size
- Weight
- EMC characteristics
- Radiation hardness, environmental friendliness, ..

How to compare different designs?
(Some designs are “better” than others)
Definitions

- Let \( X \): \( m \)-dimensional **solution space** for the design problem. Example: dimensions correspond to # of processors, size of memories, type and width of busses etc.

- Let \( F \): \( n \)-dimensional **objective space** for the design problem. Example: dimensions correspond to average and worst case delay, power/energy consumption, size, weight, reliability, …

- Let \( f(x) = (f_1(x), \ldots, f_n(x)) \) where \( x \in X \) be an **objective function**. We assume that we are using \( f(x) \) for evaluating designs.
Pareto points

- We assume that, for each objective, an order \( < \) and the corresponding order \( \leq \) are defined.

- **Definition:**

  Vector \( u = (u_1, \ldots, u_n) \in F \) dominates vector \( v = (v_1, \ldots, v_n) \in F \) if:
  
  \[
  u_i \in \{1, \ldots, n\} : u_i \leq v_i \land \exists i \in \{1, \ldots, n\} : u_i < v_i
  \]

  \( u \) is “better” than \( v \) with respect to one objective and not worse than \( v \) with respect to all other objectives:

- **Definition:**

  Vector \( u \in F \) is indifferent with respect to vector \( v \in F \) if:
  
  \( \iff \) neither \( u \) dominates \( v \) nor \( v \) dominates \( u \)
Pareto points

- A solution \( x \in X \) is called **Pareto-optimal** with respect to \( X \) \( \iff \) there is no solution \( y \in X \) such that \( u = f(x) \) is dominated by \( v = f(y) \). \( x \) is a **Pareto point**.

- **Definition**: Let \( S \subseteq F \) be a subset of solutions. \( v \in F \) is called a **non-dominated solution** with respect to \( S \) \( \iff v \) is not dominated by any element \( \in S \).

- \( v \) is called **Pareto-optimal**
  \( \iff v \) is non-dominated with respect to all solutions \( F \).

- A **Pareto-set** is the set of all Pareto-optimal solutions.

Pareto-sets define a **Pareto-front** (boundary of dominated subspace)
Pareto Point

Objective 1 (e.g. energy consumption)

better

indifferent

worse

Pareto-point

(Assuming minimization of objectives)
Pareto Set

Objective 1 (e.g. energy consumption)

Pareto set = set of all Pareto-optimal solutions

(Assuming minimization of objectives)

Objective 2 (e.g. run time)
One more time …

Pareto point

O₂ (e.g. memory space)

- indifferent
- dominated by (1) (inferior designs)
- dominating (1) (superior designs)

min

min

O₁ (e.g. energy)

Pareto front

O₂ (e.g. memory space)

- dominated design points

min

min

O₁ (e.g. energy)
Design space evaluation (DSE) based on Pareto-points is the process of finding and returning a set of Pareto-optimal designs to the user, enabling the user to select the most appropriate design.
How to evaluate designs according to multiple criteria?

Many different criteria are relevant for evaluating designs:

- Average & worst case delay
- Power/energy consumption
- Thermal behavior
- Reliability, safety, security
- Cost, size
- Weight
- EMC characteristics
- Radiation hardness, environmental friendliness, ..

How to compare different designs?  
(Some designs are “better” than others)
Average delays (execution times)

- **Estimated** average execution times:
  Difficult to generate sufficiently precise estimates;
  Balance between run-time and precision

- **Accurate** average execution times:
  As precise as the input data is.

We need to compute **average** and **worst case** execution times.
Worst case execution time (1)

Definition of worst case execution time:

WCET_{EST} must be
1. safe (i.e. ≥ WCET) and
2. tight (WCET_{EST} - WCET ≪ WCET_{EST})
Worst case execution times (2)

Complexity:
- in the general case: undecidable if a bound exists.
- for restricted programs: simple for "old" architectures, very complex for new architectures with pipelines, caches, interrupts, virtual memory, etc.

Approaches:
- for hardware: requires detailed timing behavior
- for software: requires availability of machine programs; complex analysis (see, e.g., www.absint.de)
Structure of this course

2: Specification & modeling
3: ES-hardware
4: System software (RTOS, middleware, ...)

Design repository
6: Application mapping
7: Optimization
5: Evaluation & validation (energy, cost, performance, ...)

Application Knowledge

8: Test

Numbers denote sequence of chapters

[]“Appendix”: Standard Optimization Techniques
Integer linear programming models

Ingredients:
- Cost function
- Constraints
  \[ C = \sum_{x_i \in X} a_i x_i \text{ with } a_i \in \mathbb{R}, x_i \in \mathbb{N} \] (1)

Constraints:
  \[ \forall j \in J : \sum_{x_i \in X} b_{i,j} x_i \geq c_j \text{ with } b_{i,j}, c_j \in \mathbb{R} \] (2)

Def.: The problem of minimizing (1) subject to the constraints (2) is called an integer linear programming (ILP) problem.

If all \( x_i \) are constrained to be either 0 or 1, the ILP problem is said to be a 0/1 integer linear programming problem.
Example

\[ C = 5x_1 + 6x_2 + 4x_3 \]

\[ x_1 + x_2 + x_3 \geq 2 \]

\[ x_1, x_2, x_3 \in \{0,1\} \]

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(x_3)</th>
<th>(C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
</tr>
</tbody>
</table>

Optimal
Remarks on integer programming

- Maximizing the cost function: just set $C' = -C$
- Integer programming is NP-complete.
- Running times depend exponentially on problem size, but problems of >1000 vars solvable with good solver (depending on the size and structure of the problem)
- The case of $x_i \in \mathbb{R}$ is called linear programming (LP). Polynomial complexity, but most algorithms are exponential, in practice still faster than for ILP problems.
- The case of some $x_i \in \mathbb{R}$ and some $x_i \in \mathbb{N}$ is called mixed integer-linear programming.
- ILP/LP models good starting point for modeling, even if heuristics are used in the end.
- Solvers: lp_solve (public), CPLEX (commercial), ...
Petri Net: Matrix $N$ describing all changes of markings

$$t(p) = \begin{cases} 
- W(p,t) & \text{if } p \in {}^\ast t \setminus {}^\ast \cdot \\
+ W(t,p) & \text{if } p \in {}^\ast \cdot \setminus {}^\ast t \\
- W(p,t) + W(t,p) & \text{if } p \in {}^\ast \cdot \cap {}^\ast t \\
0 & \text{otherwise}
\end{cases}$$

Def.: Matrix $N$ of net $N$ is a mapping

$$N: P \times T \rightarrow \mathbb{Z} \text{ (integers)}$$

such that $\forall \ t \in T : N(p,t) = t(p)$

Component in column $t$ and row $p$ indicates the change of the marking of place $p$ if transition $t$ takes place.

For pure nets, $(N, M_0)$ is a complete representation of a net.
Petri Net: Place – invariants

Standardized technique for proving properties of system models

For any transition \( t_j \in T \) we are looking for sets \( R \subseteq P \) of places for which the accumulated marking is constant:

\[
\sum_{p \in R} t_{-j}(p) = 0
\]

Example:
Characteristic Vector

\[ \sum_{p \in R} t_j(p) = 0 \]

Let:

\[ c_R(p) = \begin{cases} 1 & \text{if } p \in R \\ 0 & \text{if } p \notin R \end{cases} \]

\[ \Rightarrow 0 = \sum_{p \in R} t_j(p) = \sum_{p \in P} t_j(p) c_R(p) = t_j \cdot c_R \]

Scalar product
Condition for place invariants

\[ \sum_{p \in R} t_j(p) = \sum_{p \in P} t_j(p) c_R(p) = t_j \cdot c_R = 0 \]

Accumulated marking constant for all transitions if

\[ t_1 \cdot c_R = 0 \]

\[ \ldots \ldots \ldots \]

\[ t_n \cdot c_R = 0 \]

Equivalent to \( N^T c_R = 0 \) where \( N^T \) is the transposed of \( N \)
More detailed view of constraints

\[
\begin{pmatrix}
  t_1(p_1) & \ldots & t_1(p_n) \\
  t_2(p_1) & \ldots & t_2(p_n) \\
  \vdots & & \vdots \\
  t_m(p_1) & \ldots & t_m(p_n)
\end{pmatrix}
\begin{pmatrix}
  c_R(p_1) \\
  c_R(p_2) \\
  \vdots \\
  c_R(p_n)
\end{pmatrix}
= \begin{pmatrix}
  0 \\
  0 \\
  0 \\
  0
\end{pmatrix}
\]

System of linear equations.
Solution vectors must consist of zeros and ones.
Put Things Together: Maximum Place Invariants

\[
\text{maximize } \sum_{p \in P} x_p \\
\text{such that } \sum_{p \in P} t_j(p) x_p = 0 \\
x_p \in \{0, 1\}, \forall p \in P
\]
Another Example: Knapsack Problem

Example IP formulation:
The Knapsack problem:
I wish to select items to put in my backpack.

There are \( m \) items available.
Item \( i \) weights \( w_i \) kg,
Item \( i \) has value \( v_i \).
I can carry \( Q \) kg.

Let \( x_i = \begin{cases} 1 & \text{if I select item } i \\ 0 & \text{otherwise} \end{cases} \)

\[
\begin{align*}
\text{max} & \quad \sum_{i} x_i v_i \\
\text{s.t.} & \quad \sum_{i} x_i w_i \leq Q \\
& \quad x_i \in \{0,1\}
\end{align*}
\]
Variance of Knapsack Problem

- **Given** a set of periodic tasks with implicit deadlines
  - Task $\tau_i$: period $T_i$,
    - Options: Execution without/with scratchpad memory (SPM)
    - Without SPM: Worst-case execution time $C_{i,1}$
    - With SPM: required $m_i$ scratchpad memory size and Worst-case execution time $C_{i,2}$
    - $U_{i,1} = C_{i,1}/T_i$, $U_{i,2} = C_{i,2}/T_i$

- **Objective**
  - Select the tasks to be put into the SPM
  - Minimize the required SPM size
  - The task set should be schedulable by using EDF

\[
\begin{align*}
\text{min} & \quad \sum_i x_i m_i \\
\text{s.t.} & \quad \sum_i x_i U_{i,2} + \sum_i (1 - x_i) U_{i,1} \leq 1 \\
& \quad x_i \in \{0,1\}
\end{align*}
\]
Evolutionary Algorithms (1)

- **Evolutionary Algorithms** are based on the collective learning process within a population of individuals, each of which represents a search point in the space of potential solutions to a given problem.

- The population is arbitrarily initialized, and it evolves towards better and better regions of the search space by means of randomized processes of
  
  - **selection** (which is deterministic in some algorithms),
  
  - **mutation**, and
  
  - **recombination** (which is completely omitted in some algorithmic realizations).

  [Bäck, Schwefel, 1993]
Evolutionary Algorithms (2)

- The environment (given aim of the search) delivers a quality information (fitness value) of the search points, and the selection process favours those individuals of higher fitness to reproduce more often than worse individuals.

- The recombination mechanism allows the mixing of parental information while passing it to their descendants, and mutation introduces innovation into the population.

[Bäck, Schwefel, 1993]
Evolutionary Algorithms

Principles of Evolution

1 Selection

2 Mutation

3 Cross-over
An Evolutionary Algorithm in Action

max. $y_2$

min. $y_1$

hypothetical trade-off front
Issues in Multi-Objective Optimization

1. How to maintain a diverse Pareto set approximation?
2. Density estimation
3. How to prevent non-dominated solutions from being lost?
4. Environmental selection
5. How to guide the population towards the Pareto set?
6. Fitness assignment
A Generic Multiobjective EA

population

evaluate
sample
vary

new population

archive

update
truncate

new archive

© Thiele
**Example: SPEA2 Algorithm**

<table>
<thead>
<tr>
<th>Step 1:</th>
<th>Generate initial population $P_0$ and empty archive (external set) $A_0$. Set $t = 0$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 2:</td>
<td>Calculate fitness values of individuals in $P_t$ and $A_t$.</td>
</tr>
<tr>
<td>Step 3:</td>
<td>$A_{t+1} =$ nondominated individuals in $P_t$ and $A_t$. If size of $A_{t+1} &gt; N$ then reduce $A_{t+1}$, else if size of $A_{t+1} &lt; N$ then fill $A_{t+1}$ with dominated individuals in $P_t$ and $A_t$.</td>
</tr>
<tr>
<td>Step 4:</td>
<td>If $t &gt; T$ then output the nondominated set of $A_{t+1}$. Stop.</td>
</tr>
<tr>
<td>Step 5:</td>
<td>Fill mating pool by binary tournament selection.</td>
</tr>
<tr>
<td>Step 6:</td>
<td>Apply recombination and mutation operators to the mating pool and set $P_{t+1}$ to the resulting population. Set $t = t + 1$ and go to Step 2.</td>
</tr>
</tbody>
</table>
Summary

Integer (linear) programming
- Integer programming is NP-complete
- Linear programming is faster
- Good starting point even if solutions are generated with different techniques

Simulated annealing
- Modeled after cooling of liquids
- Overcomes local minima

Evolutionary algorithms
- Maintain set of solutions
- Include selection, mutation and recombination
Simulated Annealing

- General method for solving combinatorial optimization problems.
- Based the model of slowly cooling crystal liquids.
- Some configuration is subject to changes.
- Special property of Simulated annealing: Changes leading to a poorer configuration (with respect to some cost function) are accepted with a certain probability.
- This probability is controlled by a temperature parameter: the probability is smaller for smaller temperatures.
Simulated Annealing Algorithm

procedure SimulatedAnnealing;
var i, T: integer;
begin
  i := 0; T := MaxT;
  configuration := <some initial configuration>;
  while not terminate(i, T) do
    begin
      while InnerLoop do
        begin
          NewConfig := variation(configuration);
          delta := evaluation(NewConfig, configuration);
          if delta < 0
            then configuration := NewConfig;
          else if SmallEnough(delta, T, random(0,1))
            then configuration := Newconfiguration;
        end;
      T := NewT(i, T); i := i + 1;
    end;
end; end;
Explanation

- Initially, some random initial configuration is created.
- Current temperature is set to a large value.
- Outer loop:
  - Temperature is reduced for each iteration
  - Terminated if \((\text{temperature} \leq \text{lower limit})\) or \((\text{number of iterations} \geq \text{upper limit})\).
- Inner loop: For each iteration:
  - New configuration generated from current configuration
  - Accepted if \((\text{new cost} \leq \text{cost of current configuration})\)
  - Accepted with temperature-dependent probability if \((\text{cost of new config.} > \text{cost of current configuration})\).
Behavior for actual functions

130 steps

200 steps

[people.equars.com/~marco/poli/phd/node57.html]

http://foghorn.cadlab.lafayette.edu/cadapplets.fp/fpIntro.html
Performance

- This class of algorithms has been shown to outperform others in certain cases [Wegener, 2005].
- Demonstrated its excellent results in the TimberWolf layout generation package [Sechen]
- Many other applications …