Pseudo-Exhaustive Testing, Part I

Joseph Morgan, Principal Research Statistician

Testing Complex Engineered Systems

What is Testing?

"*Testing is the process of executing a … system with the intent of finding errors.*[1]"

[1]G. Myers, *The Art of Software Testing*, Wiley, 1979

Testing Complex Engineered Systems

What is Testing?

"Testing is the process of executing a … system with the intent of finding faults.[1]*"*

[1] G. Myers, *The Art of Software Testing*, Wiley, 1979

Testing Complex Engineered Systems

Where are the bugs?

"*Bugs lurk in corners and congregate at boundaries.*[1]"

[1]B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, 1983

Testing Complex Engineered Systems

Where are the faults?

"Faults lurk in corners and congregate at boundaries.[1]"

[1]B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, 1983

Testing Complex Engineered Systems

The Testing Challenge

Given a complex engineered system and a testing budget:

1. **Selection problem**: How do you select test cases from the input space of the system so that the chance of finding faults, while staying within budget, is maximized?
2. **Quality problem**: Can you make informed assertions about "*fitness for use*" as testing unfolds?
3. **Oracle problem**: How do you determine expected behavior for each test case?

Testing Complex Engineered Systems

Combinatorial Testing

A family of test case selection strategies used to test complex engineered systems.

For a complex engineered system, with m inputs, such as a software system, a strength t covering array will ensure that all possible combinations of the values for any set of $t \leq m$ inputs will appear in the derived test suite at least once.

Testing Complex Engineered Systems

Combinatorial Testing

A family of test case selection strategies used to test complex engineered systems.

For a complex engineered system, with *m* inputs, such as a software system, a strength *t* covering array will ensure that all possible combinations of the values for any set of $t \leq m$ inputs will appear in the derived test suite at least once.

"*Faults lurk in corners and congregate at boundaries.*[1]"

[1]B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, 1983

Testing Complex Engineered Systems

Combinatorial Testing

Partly addresses the testing challenge.

1. **Selection problem**: How do you select test cases from the input space of the system so that the chance of finding faults, while staying within budget, is maximized?
2. **Quality problem**: Can you make informed assertions about "*fitness for use*" as testing unfolds?
3. **Oracle problem**: How do you determine expected behavior for each test case?

Testing Complex Engineered Systems

Alessandro Orso & Gregg Rothermel[1]

Two questions

1. What do you think are the most significant contributions to testing since 2000?
2. What do you think are the biggest open challenges and opportunities in this area?

[1]Orso, Alessandro, and Gregg Rothermel. “Software testing: a research travelogue (2000–2014).” *Future of Software Engineering Proceedings*. 2014. 117-132.

Testing Complex Engineered Systems

Alessandro Orso & Gregg Rothermel[1]

Research Contributions

Automated Test Input Generation

Dynamic Symbolic Execution
Search-based Testing
Random Testing
Combined Techniques

Testing Strategies

Combinatorial Testing
Model-Based Testing
Mining/Learning from Field Data

Empirical Studies & Infrastructure

Regression Testing

Practical Contributions

Frameworks for Test Execution
Continuous Integration

Challenges & Opportunities

Testing Real-World Systems
Oracles
Probabilistic Approaches
Testing Non-Functional Prop.
Domain-Based Testing
Leveraging Cloud and Crowd

[1]Orso, Alessandro, and Gregg Rothermel. "Software testing: a research travelogue (2000–2014)." *Future of Software Engineering Proceedings*. 2014. 117-132.

Testing Complex Engineered Systems

Alessandro Orso & Gregg Rothermel[1]

Research Contributions

Automated Test Input Generation

Dynamic Symbolic Execution
Search-based Testing
Random Testing
Combined Techniques

Testing Strategies

Combinatorial Testing
Model-Based Testing
Mining/Learning from Field Data

Empirical Studies & Infrastructure

Regression Testing

Practical Contributions

Frameworks for Test Execution
Continuous Integration

Challenges & Opportunities

Testing Real-World Systems
Oracles
Probabilistic Approaches
Testing Non-Functional Prop.
Domain-Based Testing
Leveraging Cloud and Crowd

[1]Orso, Alessandro, and Gregg Rothermel. "Software testing: a research travelogue (2000–2014)." *Future of Software Engineering Proceedings*. 2014. 117-132.

Covering Arrays

An introduction

Covering arrays - Preliminaries

Definition

A covering array $\mathbf{CA}_\lambda(N; t, (v_1 \cdot v_2 \cdot \ldots \cdot v_m))$ is an $N \times m$ array such that the i-th column contains v_i distinct symbols. If the array has the property that, for any t column projection ($t \geq 1$), all $\prod_{k=1}^{t} v_{f(k)}$ combinations of symbols exist at least λ times ($f(k)$ the k^{th} column of the projection), then it is a t-covering array.

Note: Exponential notation is usually used for the symbol specification (i.e. $(3 \cdot 2^3)$ instead of $(3 \cdot 2 \cdot 2 \cdot 2)$).

Covering arrays - Preliminaries

Definition

The size of a covering array is referred to as the covering array number $\mathbf{CAN}_\lambda(t, (v_1 \cdot v_2 \cdot \ldots \cdot v_m))$,

$$\mathbf{CAN}_\lambda(t, (v_1 \cdot v_2 \cdot \ldots \cdot v_m)) = \min\{N: \exists \mathbf{CA}_\lambda(N; t, (v_1 \cdot v_2 \cdot \ldots \cdot v_m))\}.$$

Covering arrays - Preliminaries

Definition

The size of a covering array is referred to as the covering array number $\mathbf{CAN}_\lambda(t, (v_1 \cdot v_2 \cdot \ldots \cdot v_m))$,

$$\mathbf{CAN}_\lambda(t, (v_1 \cdot v_2 \cdot \ldots \cdot v_m)) = \min\{N: \exists \mathbf{CA}_\lambda(N; t, (v_1 \cdot v_2 \cdot \ldots \cdot v_m))\}.$$

N	4	5	6	7	8	9	10	11	12	13	14
m	3	4	10	15	35	56	126	210	462	792	1716

$$\mathbf{CA}(N; 2, 2^m),\ m \leq 1716$$

Covering arrays - Preliminaries

Notes:

1. Consider the covering array $\mathbf{CA}_\lambda(N; t, (v_1 \cdot v_2 \cdot \ldots \cdot v_m))$. If for any *t* column projection all combinations of symbols exist **exactly λ times** then it is a ***t***-covering **orthogonal** array of index λ.

   Takeaway: Orthogonal arrays are a special type of covering array.
2. A *t*-covering array is optimal if N is minimal for fixed t, m, and $(v_1 \cdot v_2 \cdot \ldots \cdot v_m)$.
3. When λ=1 (the most common case) we usually drop the subscript.

Covering arrays - Preliminaries

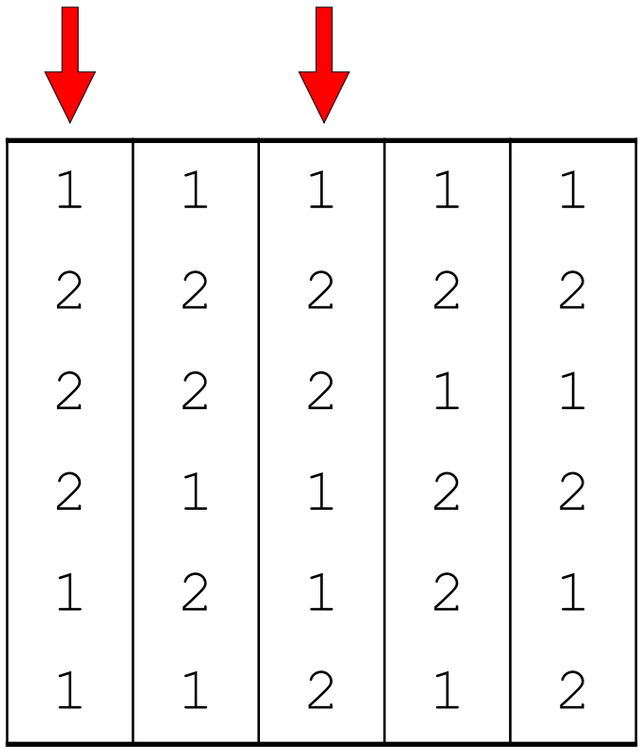
1	1	1	1	1
2	2	2	2	2
2	2	2	1	1
2	1	1	2	2
1	2	1	2	1
1	1	2	1	2

CA(6; 2, 2^5)

1	1	1	1	1
1	1	2	1	1
1	2	1	1	2
1	2	2	1	2
2	1	1	2	1
2	1	2	2	1
2	2	1	2	2
2	2	2	2	2
1	1	1	2	2
2	2	1	1	1
2	1	2	1	2
1	2	2	2	1

CA(12; 3, 2^5)

Covering arrays - Preliminaries



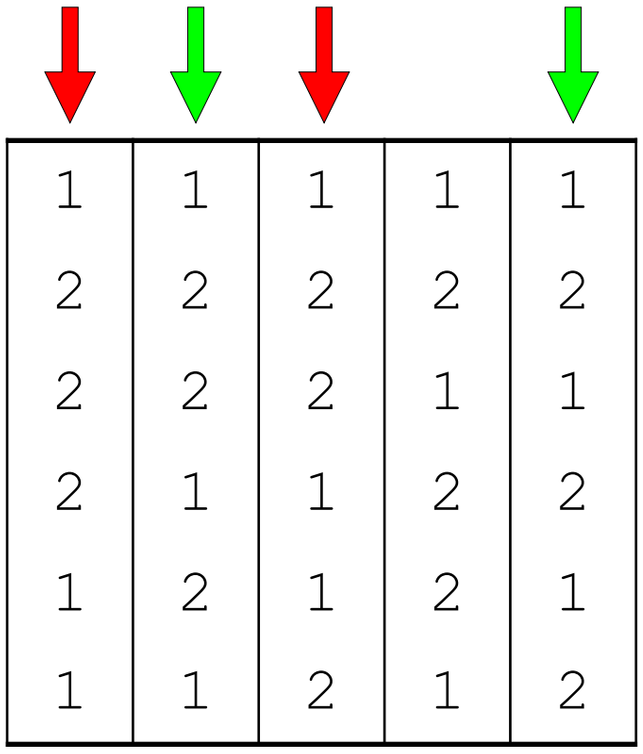
1	1	1	1	1
2	2	2	2	2
2	2	2	1	1
2	1	1	2	2
1	2	1	2	1
1	1	2	1	2

CA(6; 2, 2^5)

CA(12; 3, 2^5)

1	1	1	1	1
1	1	2	1	1
1	2	1	1	2
1	2	2	1	2
2	1	1	2	1
2	1	2	2	1
2	2	1	2	2
2	2	2	2	2
1	1	1	2	2
2	2	1	1	1
2	1	2	1	2
1	2	2	2	1

Covering arrays - Preliminaries



1	1	1	1	1
2	2	2	2	2
2	2	2	1	1
2	1	1	2	2
1	2	1	2	1
1	1	2	1	2

CA(6; 2, 2^5)

CA(12; 3, 2^5)

1	1	1	1	1
1	1	2	1	1
1	2	1	1	2
1	2	2	1	2
2	1	1	2	1
2	1	2	2	1
2	2	1	2	2
2	2	2	2	2
1	1	1	2	2
2	2	1	1	1
2	1	2	1	2
1	2	2	2	1

Covering arrays - Preliminaries

1	1	1	1	1
2	2	2	2	2
2	2	2	1	1
2	1	1	2	2
1	2	1	2	1
1	1	2	1	2

CA(6; 2, 2^5)

1	1	1	1	1
1	1	2	1	1
1	2	1	1	2
1	2	2	1	2
2	1	1	2	1
2	1	2	2	1
2	2	1	2	2
2	2	2	2	2
1	1	1	2	2
2	2	1	1	1
2	1	2	1	2
1	2	2	2	1

CA(12; 3, 2^5)

Covering arrays - Preliminaries

1	1	1	1	1
2	2	2	2	2
2	2	2	1	1
2	1	1	2	2
1	2	1	2	1
1	1	2	1	2

CA(6; 2, 2^5)

1	1	1	1	1
1	1	2	1	1
1	2	1	1	2
1	2	2	1	2
2	1	1	2	1
2	1	2	2	1
2	2	1	2	2
2	2	2	2	2
1	1	1	2	2
2	2	1	1	1
2	1	2	1	2
1	2	2	2	1

CA(12; 3, 2^5)

Covering arrays - Preliminaries

1	1	1	1	1
2	2	2	2	2
2	2	2	1	1
2	1	1	2	2
1	2	1	2	1
1	1	2	1	2

Optimal[1]

CA(6; 2, 2^5)

1	1	1	1	1
1	1	2	1	1
1	2	1	1	2
1	2	2	1	2
2	1	1	2	1
2	1	2	2	1
2	2	1	2	2
2	2	2	2	2
1	1	1	2	2
2	2	1	1	1
2	1	2	1	2
1	2	2	2	1

Not optimal[2]
CAN = 10

CA(12; 3, 2^5)

[1] A. Renyi, 1948
[2] K. Johnson & R. Entringer, 1989

Covering arrays - Preliminaries

1	1	2	1	2	2	2	1	1	1
1	2	1	2	1	1	2	2	2	1
1	3	1	1	2	1	1	1	2	2
2	1	1	2	1	2	1	2	2	2
2	2	2	2	2	1	1	1	1	2
2	3	2	1	2	1	2	1	1	1
3	1	1	1	2	1	1	2	1	1
3	2	1	1	1	2	1	1	1	1
3	3	2	2	1	2	2	2	2	2

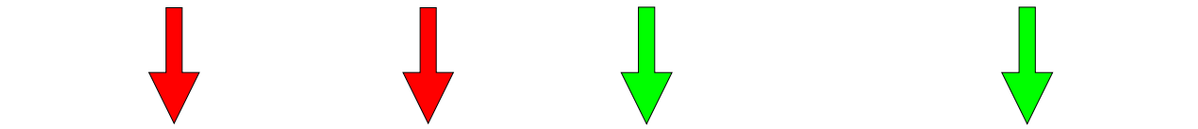
CA$(9; 2, (3^2 \cdot 2^8))$

Covering arrays - Preliminaries

1	1	2	1	2	2	2	1	1	1
1	2	1	2	1	1	2	2	2	1
1	3	1	1	2	1	1	1	2	2
2	1	1	2	1	2	1	2	2	2
2	2	2	2	2	1	1	1	1	2
2	3	2	1	2	1	2	1	1	1
3	1	1	1	2	1	1	2	1	1
3	2	1	1	1	2	1	1	1	1
3	3	2	2	1	2	2	2	2	2

CA$(9; 2, (3^2 \cdot 2^8))$

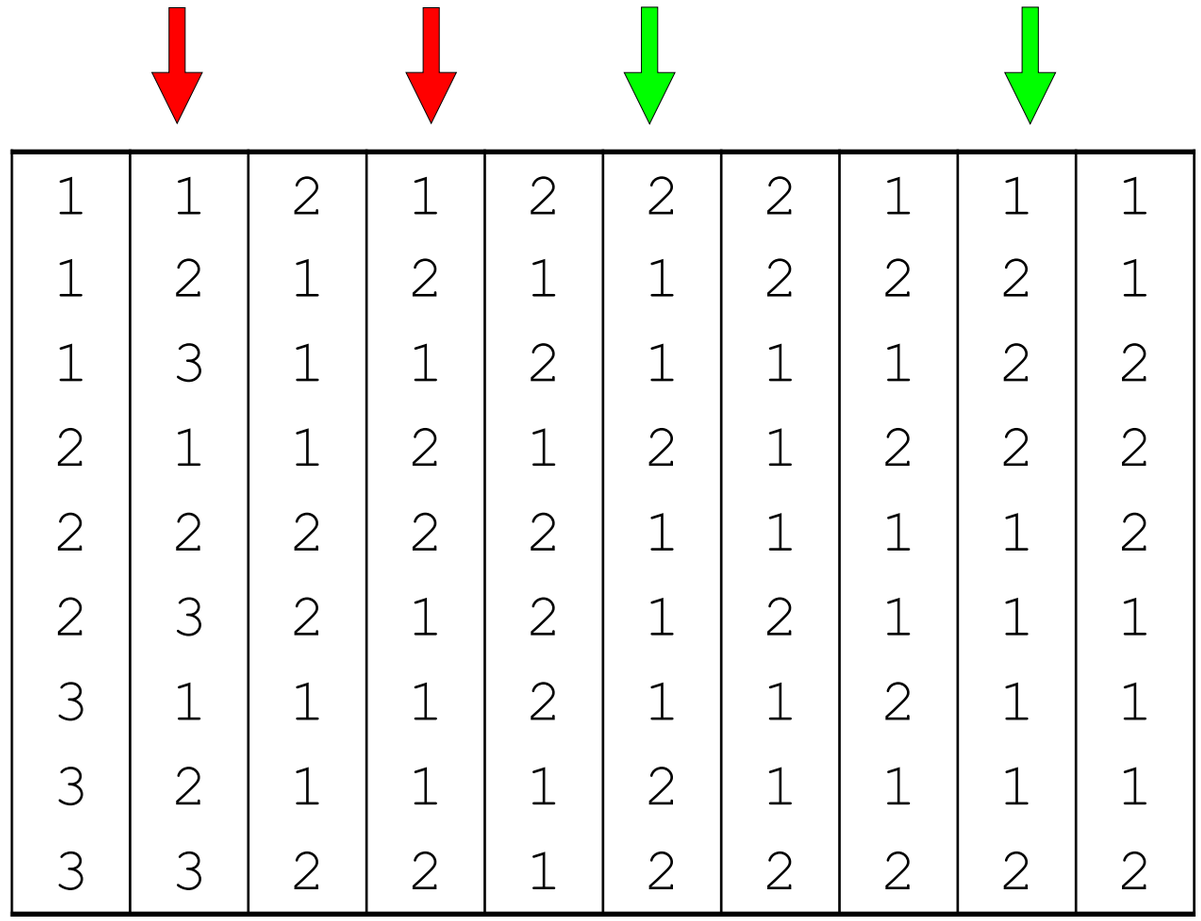
Covering arrays - Preliminaries



1	1	2	1	2	2	2	1	1	1
1	2	1	2	1	1	2	2	2	1
1	3	1	1	2	1	1	1	2	2
2	1	1	2	1	2	1	2	2	2
2	2	2	2	2	1	1	1	1	2
2	3	2	1	2	1	2	1	1	1
3	1	1	1	2	1	1	2	1	1
3	2	1	1	1	2	1	1	1	1
3	3	2	2	1	2	2	2	2	2

CA$(9; 2, (3^2 \cdot 2^8))$

Covering arrays - Preliminaries

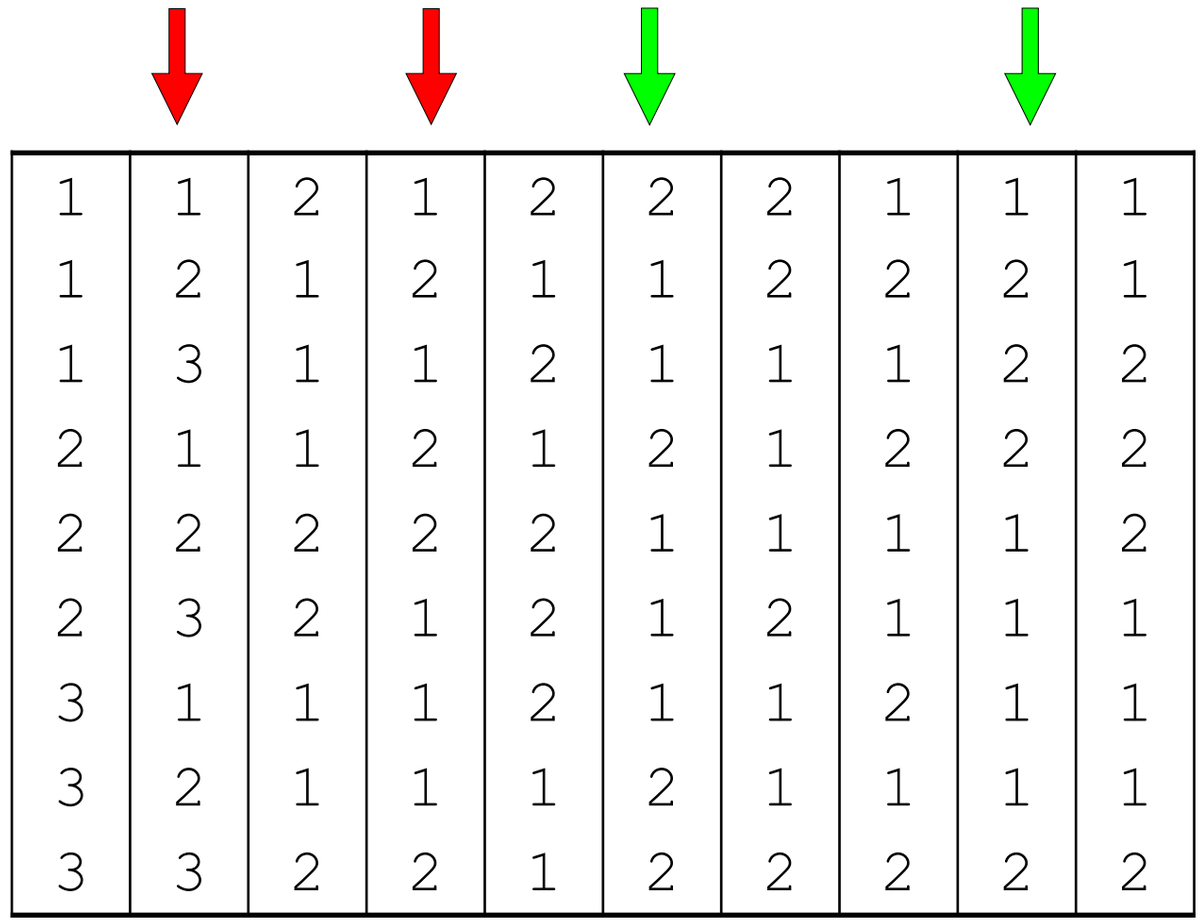


1	1	2	1	2	2	2	1	1	1
1	2	1	2	1	1	2	2	2	1
1	3	1	1	2	1	1	1	2	2
2	1	1	2	1	2	1	2	2	2
2	2	2	2	2	1	1	1	1	2
2	3	2	1	2	1	2	1	1	1
3	1	1	1	2	1	1	2	1	1
3	2	1	1	1	2	1	1	1	1
3	3	2	2	1	2	2	2	2	2

CA(9; 2, $(3^2 \cdot 2^8)$)

Optimal

Covering arrays - Preliminaries



1	1	2	1	2	2	2	1	1	1
1	2	1	2	1	1	2	2	2	1
1	3	1	1	2	1	1	1	2	2
2	1	1	2	1	2	1	2	2	2
2	2	2	2	2	1	1	1	1	2
2	3	2	1	2	1	2	1	1	1
3	1	1	1	2	1	1	2	1	1
3	2	1	1	1	2	1	1	1	1
3	3	2	2	1	2	2	2	2	2

CA(9; 2, $(3^2 \cdot 2^8)$)

Optimal

Note: Using a search algorithm, we know that **CAN**(2,$(3^2 \cdot 2^{20})$) = 9.

Covering arrays - Preliminaries

Definition[1]

Consider the covering array $\mathbf{CA}_\lambda(N; t, (v_1 \cdot v_2 \cdot \ldots \cdot v_m))$. Let n_i be the number of distinct t tuples for the i^{th} projection, p_i the number of possible t tuples *for the* i^{th} projection, r the number of rows, and $M = {}_KC_t$ the number of t column projections that can be obtained from K columns.

Note: $p_i = \prod_{k=1}^{t} v_{f(k)}$ where $f(k)$ is the k^{th} column of the i^{th} projection.

[1]S. Dalal & C. Mallows, "Factor-covering designs for testing software," *Technometrics*, 40(3), 1998, 234-243.

Covering arrays - Preliminaries

Definition[1]

Consider the covering array $\mathbf{CA}_\lambda(N; t, (v_1 \cdot v_2 \cdot \ldots \cdot v_m))$. Let n_i be the number of distinct t tuples for the i^{th} projection, p_i the number of possible t tuples *for the* i^{th} projection, r the number of rows, and $M = {}_KC_t$ the number of t column projections that can be obtained from K columns.

$$t\text{-Coverage} = \frac{\mathbf{1}}{\boldsymbol{M}} \sum_{i=1}^{M} n_i / p_i .$$

Note: $p_i = \prod_{k=1}^{t} v_{f(k)}$ where $f(k)$ is the k^{th} column of the i^{th} projection.

[1]S. Dalal & C. Mallows, "Factor-covering designs for testing software," *Technometrics*, 40(3), 1998, 234-243.

Covering arrays - Preliminaries

Definition[1]

In simple terms, *t*-Coverage is the ratio of the number of distinct *t* tuples, to the number of possible *t* tuples, averaged over all *t* column projections ($1 \leq t \leq K$).

Note: For a comprehensive treatment of covering array metrics see:

Kuhn, D. R., et al. "Combinatorial coverage measurement concepts and applications." *2013 IEEE 6th International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2013.

[1]S. Dalal & C. Mallows, "Factor-covering designs for testing software," *Technometrics*, 40(3), 1998, 234-243.

Covering arrays - Preliminaries

t-Coverage = $\frac{1}{M}\sum_{i=1}^{M} n_i/p_i$

1	1	1	1	1
2	2	2	2	2
2	2	2	1	1
2	1	1	2	2
1	2	1	2	1
1	1	2	1	2

CA(6; 2, 5, 2)

Covering arrays - Preliminaries

t-Coverage = $\frac{1}{M}\sum_{i=1}^{M} n_i/p_i$

1	1	1	1	1
2	2	2	2	2
2	2	2	1	1
2	1	1	2	2
1	2	1	2	1
1	1	2	1	2

CA(6; 2, 5, 2)

t	Coverage
2	100.0
3	70.0
4	37.5

Covering arrays - Preliminaries

t-Coverage = $\frac{1}{M}\sum_{i=1}^{M} n_i/p_i$

1	1	1	1	1
1	1	2	1	1
1	2	1	1	2
1	2	2	1	2
2	1	1	2	1
2	1	2	2	1
2	2	1	2	2
2	2	2	2	2
1	1	1	2	2
2	2	1	1	1
2	1	2	1	2
1	2	2	2	1

CA(12; 3, 5, 2)

Covering arrays - Preliminaries

t-Coverage = $\frac{1}{M}\sum_{i=1}^{M} n_i/p_i$

1	1	1	1	1
1	1	2	1	1
1	2	1	1	2
1	2	2	1	2
2	1	1	2	1
2	1	2	2	1
2	2	1	2	2
2	2	2	2	2
1	1	1	2	2
2	2	1	1	1
2	1	2	1	2
1	2	2	2	1

CA(12; 3, 5, 2)

t	Coverage
3	100.0
4	70.0
5	37.5

Testing Complex Engineered Systems

Question: How do we get to combinatorial testing from covering arrays?

Testing Complex Engineered Systems

Question: How do we get to combinatorial testing from covering arrays?

Answer: It is a mapping exercise!

Testing Complex Engineered Systems

Consider a system to be tested (i.e. SUT), such as a function with *m* arguments/inputs, a GUI with *m* controls/inputs, a configuration with *m* options/inputs (e.g. software system preferences, ML hyper-parameters, compiler options).

Testing Complex Engineered Systems

Consider a system to be tested (i.e. SUT), such as a function with *m* arguments/inputs, a GUI with *m* controls/inputs, a configuration with *m* options/inputs (e.g. software system preferences, ML hyper-parameters, compiler options).

Covering Array		**SUT**
Columns		Inputs

Testing Complex Engineered Systems

Consider a system to be tested (i.e. SUT), such as a function with *m* arguments/inputs, a GUI with *m* controls/inputs, a configuration with *m* options/inputs (e.g. software system preferences, ML hyper-parameters, compiler options).

Covering Array		**SUT**
Columns	⟷	Inputs
Levels per column	⟷	Equivalence classes* per input

Testing Complex Engineered Systems

Consider a system to be tested (i.e. SUT), such as a function with *m* arguments/inputs, a GUI with *m* controls/inputs, a configuration with *m* options/inputs (e.g. software system preferences, ML hyper-parameters, compiler options).

Covering Array		**SUT**
Columns	⟷	Inputs
Levels per column	⟷	Equivalence classes* per input
Strength (i.e. *t*)	⟷	Testing budget

Testing Complex Engineered Systems

Consider a system to be tested (i.e. SUT), such as a function with *m* arguments/inputs, a GUI with *m* controls/inputs, a configuration with *m* options/inputs (e.g. software system preferences, ML hyper-parameters, compiler options).

Covering Array		**SUT**
Columns	⟷	Inputs
Levels per column	⟷	Equivalence classes* per input
Strength (i.e. *t*)	⟷	Testing budget
Rows	⟷	Test cases

Illustrative Example

Air to ground missile system

Combinatorial Testing: Air to ground missile system

Consider a software system controlling the state of an air to ground missile system (Dalal & Mallows[1]).

	Input	Type
1	Altitude	*continuous*
2	Attack angle	*continuous*
3	Bank angle	*continuous*
4	Speed	*continuous*
5	Pitch	*continuous*
6	Roll	*continuous*
7	Yaw	*continuous*
8	Ambient Temperature	*continuous*
9	Pressure	*continuous*
10	Wind Velocity	*continuous*

[1]S. R. Dalal & C. L. Mallows, “Factor-covering designs for testing software,” *Technometrics*, 40(3), 1998, 234-243.

Combinatorial Testing: Air to ground missile system

Consider a software system controlling the state of an air to ground missile system (Dalal & Mallows[1]).

	Input	Type
1	Altitude	*continuous*
2	Attack angle	*continuous*
3	Bank angle	*continuous*
4	Speed	*continuous*
5	Pitch	*continuous*
6	Roll	*continuous*
7	Yaw	*continuous*
8	Ambient Temperature	*continuous*
9	Pressure	*continuous*
10	Wind Velocity	*continuous*

Challenge:

We are interested in deriving test cases to effectively validate "...response during attack maneuvering."

[1]S. R. Dalal & C. L. Mallows, "Factor-covering designs for testing software," *Technometrics*, 40(3), 1998, 234-243.

Combinatorial Testing - Air to ground missile system

Remember that inputs correspond to columns. Let us say that there are two equivalence classes per input (i.e. two levels per column), then the design specification is:

	Column	Levels
1	Altitude	*low, high*
2	Attack angle	*low, high*
3	Bank angle	*low, high*
4	Speed	*low, high*
5	Pitch	*low, high*
6	Roll	*low, high*
7	Yaw	*low, high*
8	Ambient Temperature	*low, high*
9	Pressure	*low, high*
10	Wind Velocity	*low, high*

Combinatorial Testing - Air to ground missile system

Remember that inputs correspond to columns. Let us say that there are two equivalence classes per input (i.e. two levels per column), then the design specification is:

	Column	Levels
1	Altitude	*low, high*
2	Attack angle	*low, high*
3	Bank angle	*low, high*
4	Speed	*low, high*
5	Pitch	*low, high*
6	Roll	*low, high*
7	Yaw	*low, high*
8	Ambient Temperature	*low, high*
9	Pressure	*low, high*
10	Wind Velocity	*low, high*

Note: With two levels per column, the input space is 2^{10} = 1024 points!

Combinatorial Testing - Air to ground missile system

“Naive” approach:

Altitude	Attack Angle	Bank Angle	Speed	Pitch	Roll	Yaw	Ambient Temp	Pressure	Wind Velocity
low	low	low	low	high	low	high	high	high	low
high	high	high	high	low	high	low	low	low	high

Combinatorial Testing - Air to ground missile system

“Naive” approach:

Altitude	Attack Angle	Bank Angle	Speed	Pitch	Roll	Yaw	Ambient Temp	Pressure	Wind Velocity
low high	low high	low high	low high	high low	low high	high low	high low	high low	low high

Combinatorial Testing - Air to ground missile system

“Naive” approach:

Altitude	Attack Angle	Bank Angle	Speed	Pitch	Roll	Yaw	Ambient Temp	Pressure	Wind Velocity
low	low	low	low	high	low	high	high	high	low
high	high	high	high	low	high	low	low	low	high

high low

low high

Missing level combinations!

Combinatorial Testing - Air to ground missile system

"Naive" approach:

Otherwise known as an "All values testing" strategy.

Altitude	Attack Angle	Bank Angle	Speed	Pitch	Roll	Yaw	Ambient Temp	Pressure	Wind Velocity
low	low	low	low	high	low	high	high	high	low
high	high	high	high	low	high	low	low	low	high

high low

low high

Missing level combinations!

Combinatorial Testing - Air to ground missile system

Covering array approach:

Altitude	Attack Angle	Bank Angle	Speed	Pitch	Roll	Yaw	Ambient Temp	Pressure	Wind Velocity
low	low	low	low	low	low	low	low	low	low
low	low	low	low	high	high	high	high	high	high
low	high	high	high	low	low	low	high	high	high
high	low	high	high	low	high	high	low	low	high
high	high	low	high	high	low	high	low	high	low
high	high	high	low	high	high	low	high	low	low

CA(6; 2, 10, 2)

Combinatorial Testing - Air to ground missile system

Covering array approach:

Altitude	Attack Angle	Bank Angle	Speed	Pitch	Roll	Yaw	Ambient Temp	Pressure	Wind Velocity
low	low	low	low	low	low	low	low	low	low
low	low	low	low	high	high	high	high	high	high
low	high	high	high	low	low	low	high	high	high
high	low	high	high	low	high	high	low	low	high
high	high	low	high	high	low	high	low	high	low
high	high	high	low	high	high	low	high	low	low

CA(6; 2, 10, 2)

Combinatorial Testing - Air to ground missile system

Covering array approach:

Altitude	Attack Angle	Bank Angle	Speed	Pitch	Roll	Yaw	Ambient Temp	Pressure	Wind Velocity
low	low	low	low	low	low	low	low	low	low
low	low	low	low	high	high	high	high	high	high
low	high	high	high	low	low	low	high	high	high
high	low	high	high	low	high	high	low	low	high
high	high	low	high	high	low	high	low	high	low
high	high	high	low	high	high	low	high	low	low

CA(6; 2, 10, 2)

Optimal[1]

[1] A. Renyi, 1948

Combinatorial Testing - Air to ground missile system

t-Coverage = $\frac{1}{M}\sum_{i=1}^{M} n_i/p_i$

Air to ground missile system, **CA**(6; 2, 10, 2):

Altitude	Attack Angle	Bank Angle	Speed	Pitch	Roll	Yaw	Ambient Temp	Pressure	Wind Velocity
low	low	low	low	low	low	low	low	low	low
low	low	low	low	high	high	high	high	high	high
low	high	high	high	low	low	low	high	high	high
high	low	high	high	low	high	high	low	low	high
high	high	low	high	high	low	high	low	high	low
high	high	high	low	high	high	low	high	low	low

Combinatorial Testing - Air to ground missile system

t-Coverage = $\frac{1}{M}\sum_{i=1}^{M} n_i/p_i$

Air to ground missile system, **CA**(6; 2, 10, 2):

t	**Coverage**
2	100.0
3	68.8
4	37.1

Altitude	Attack Angle	Bank Angle	Speed	Pitch	Roll	Yaw	Ambient Temp	Pressure	Wind Velocity
low	low	low	low	low	low	low	low	low	low
low	low	low	low	high	high	high	high	high	high
low	high	high	high	low	low	low	high	high	high
high	low	high	high	low	high	high	low	low	high
high	high	low	high	high	low	high	low	high	low
high	high	high	low	high	high	low	high	low	low

Testing Complex Engineered Systems

Combinatorial Testing: Empirical evidence

Question: Is combinatorial testing an effective way to test complex engineered systems?

Testing Complex Engineered Systems

Combinatorial Testing: Empirical evidence

Question: Is combinatorial testing an effective way to test complex engineered systems?

Answer: Yes! Kuhn et al.[1] examined several classes of software systems and discovered the following.

	% faults detected			
	t=2	**t=3**	**t=4**	**t=5**
Medical device software	95	99	100	100
Browser application	70	90	95	96
Server software	75	95	96	99
Network security	62	89	99	100
TCAS - Avionics System	54	74	89	100

[1]D. Kuhn & D. Wallace, & A. Gallo, "Software fault interactions and implications for software testing," IEEE Trans. SE, v30(6) (2004), 418-421.

Testing Complex Engineered Systems

Combinatorial Testing: Empirical evidence

Question: Is combinatorial testing an effective way to test complex engineered systems?

Answer: Yes! Kuhn et al.[1] examined several classes of software systems and discovered the following.

	% faults detected			
	t=2	**t=3**	**t=4**	**t=5**
Medical device software	95	99	100	100
Browser application	70	90	95	96
Server software	75	95	96	99
Network security	62	89	99	100
TCAS - Avionics System	54	74	89	100

Very high availability & reliability requirements!

[1] D. Kuhn & D. Wallace, & A. Gallo, "Software fault interactions and implications for software testing," IEEE Trans. SE, v30(6) (2004), 418-421.

Covering arrays

Note:

In practice, systems and software validation problems come with constraints. Consider a GUI that contains a variety of controls, such as checkboxes, combo boxes, etc., where some setting of a particular control precludes settings of some other control. In discussing testing configurable systems, Myra Cohen[1] states:

> "*Constraints may arise due to any number of reasons, for example, inconsistencies between certain hardware components, limitations due to available memory and software size, or simply marketing decisions.*"

Constraints are referred to as *disallowed combinations* or *forbidden interactions*.

[1]M. Cohen, M. Dwyer, J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," IEEE TSE, 2008, 633-650.

Covering arrays

CA’s are not sufficient, we need an extension!

Constrained covering arrays

Definition[1]

A *constrained covering array* **CCA**(N; t, ($v_1 \cdot v_2 \cdot \ldots \cdot v_k$), ϕ) is an $N \times k$ array such that the *i*-th column contains v_i distinct symbols and ϕ is a set of *p*-tuples ($2 \leq p \leq k$) such that each tuple is a set of two or more column/value pairs identifying a disallowed combination. If for any *t* coordinate projection, all **possible** combinations of symbols exist, then it is a ***t***-covering array and is optimal if N is minimal for fixed t, k, ($v_1 \cdot v_2 \cdot \ldots \cdot v_k$), and ϕ.

Note: Let us say that the symbol set for columns c_1 and c_2 is {1, 2} and the symbol combination (2, 1) is not allowed then $\phi = \{(c_1,2), (c_2,2)\}$.

[1]J. Morgan, "Combinatorial Testing: An approach to systems and software testing based on covering arrays," in *Analytic Methods in Systems and Software Testing,* eds., F. Ruggeri, R. Kennett, & F. Faltin, Wiley, 2018.

Constrained covering arrays

Consider a **CCA**(9; 2, $(3^2 \cdot 2^3)$,ϕ) where ϕ={{(c_1,1),(c_3,1)}}.

It is usually convenient to express ϕ more compactly. Cohen[1], suggests a shorthand exponent notation, p^k, to indicate k *p*-tuples. In this case, ϕ = {2^1}.

c_1	c_2	c_3	c_4	c_5
1	1	2	2	1
1	2	2	1	2
1	3	2	2	2
2	1	1	2	2
2	2	2	2	2
2	3	2	1	1
3	1	2	1	1
3	2	1	2	1
3	3	1	1	2

CCA(9; 2, $(3^2 \cdot 2^3)$, ϕ={2^1})

[1]M. Cohen, M. Dwyer, J. Shi, “Interaction testing of highly-configurable systems in the presence of constraints,” IEEE TSE, 2008, 633-650.

Constrained covering arrays

Consider a **CCA**(9; 2, $(3^2 \cdot 2^3)$,ϕ) where ϕ={{(c_1,1),(c_3,1)}}.

It is usually convenient to express ϕ more compactly. Cohen[1], suggests a shorthand exponent notation, p^k, to indicate k *p*-tuples. In this case, ϕ = $\{2^1\}$.

c_1	c_2	c_3	c_4	c_5
1	1	2	2	1
1	2	2	1	2
1	3	2	2	2
2	1	1	2	2
2	2	2	2	2
2	3	2	1	1
3	1	2	1	1
3	2	1	2	1
3	3	1	1	2

CCA(9; 2, $(3^2 \cdot 2^3)$, ϕ=$\{2^1\}$)

Optimal

Exponent notation

[1]M. Cohen, M. Dwyer, J. Shi, “Interaction testing of highly-configurable systems in the presence of constraints,” IEEE TSE, 2008, 633-650.

Testing Complex Engineered Systems

Combinatorial Testing

If $t << m$ then the cost savings is dramatic when compared to exhaustive testing.

Testing Complex Engineered Systems

Combinatorial Testing

If $t << m$ then the cost savings is dramatic when compared to exhaustive testing.

Consider the **CAN** for strength 2 binary covering arrays:

N	4	5	6	7	8	9	10	11	12	13	14
m	3	4	10	15	35	56	126	210	462	792	1716

CA(N; 2, 2^m), m ≤ 1716

Testing Complex Engineered Systems

Takeaways

1. Given a system with m inputs, a strength t covering array may be used to generate a test suite in which all t-way interactions are covered by at least one test case. If $t << m$ then the cost savings is dramatic when compared to exhaustive testing.
2. Covering arrays are an effective and efficient tool for deriving test suites to validate complex engineered systems such as software systems.

Testing Complex Engineered Systems

Takeaways

1. Given a system with *m* inputs, a strength *t* covering array may be used to generate a test suite in which all *t*-way interactions are covered by at least one test case. If $t << m$ then the cost savings is dramatic when compared to exhaustive testing.
2. Covering arrays are an effective and efficient tool for deriving test suites to validate complex engineered systems such as software systems.

Pseudo-exhaustive testing[1]

[1]D. Kuhn & V. Okum, "Pseudo-exhaustive testing for software," Proc. 30th IEEE/NASA Software Engineering Workshop, (2006).

Testing Complex Engineered Systems

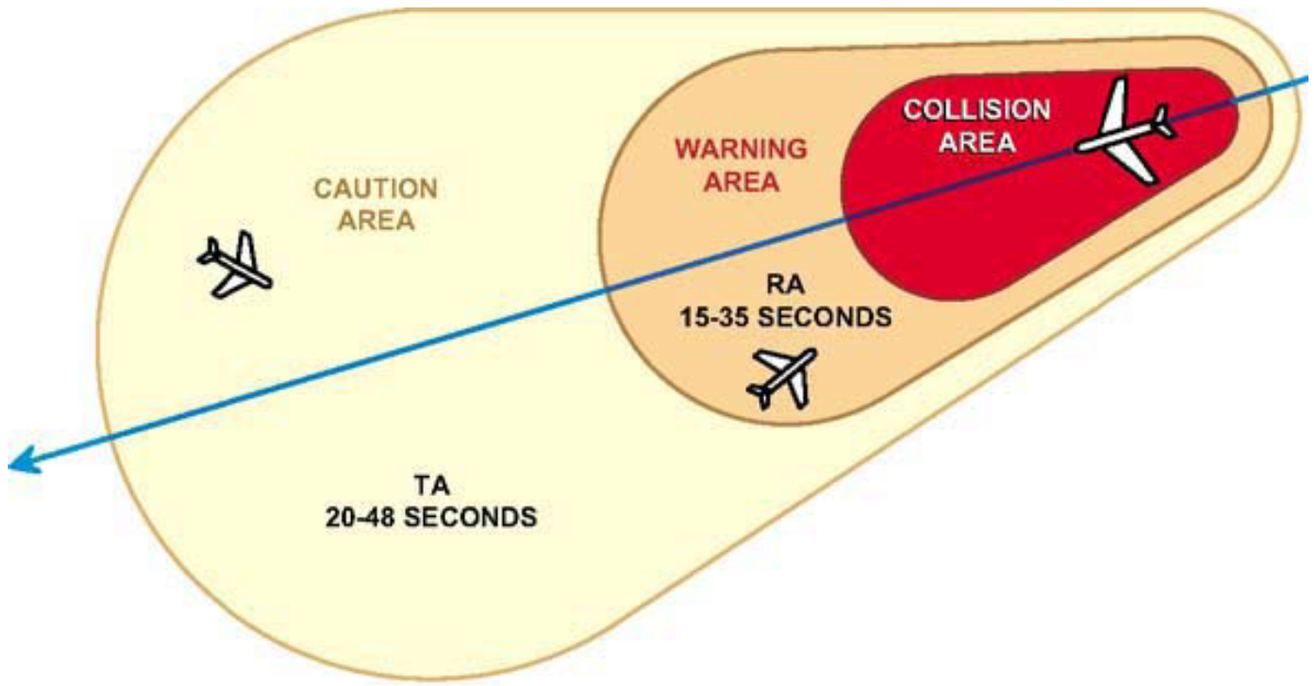
Historical Aside: Combinatorial Testing milestones

1. **Orthogonal latin squares**: In 1985, Mandl proposed orthogonal latin squares as the basis of a test case selection strategy for validating the Ada compiler.
2. **Orthogonal arrays**: In 1987, Tatsumi et al. proposed orthogonal arrays as the basis of a test case selection strategy.
3. **Covering arrays**: In 1996, Cohen et al. proposed covering arrays as the basis of a test case selection strategy.

Case Study #1

Traffic Collision Avoidance System

Traffic Collision Avoidance System (TCAS)



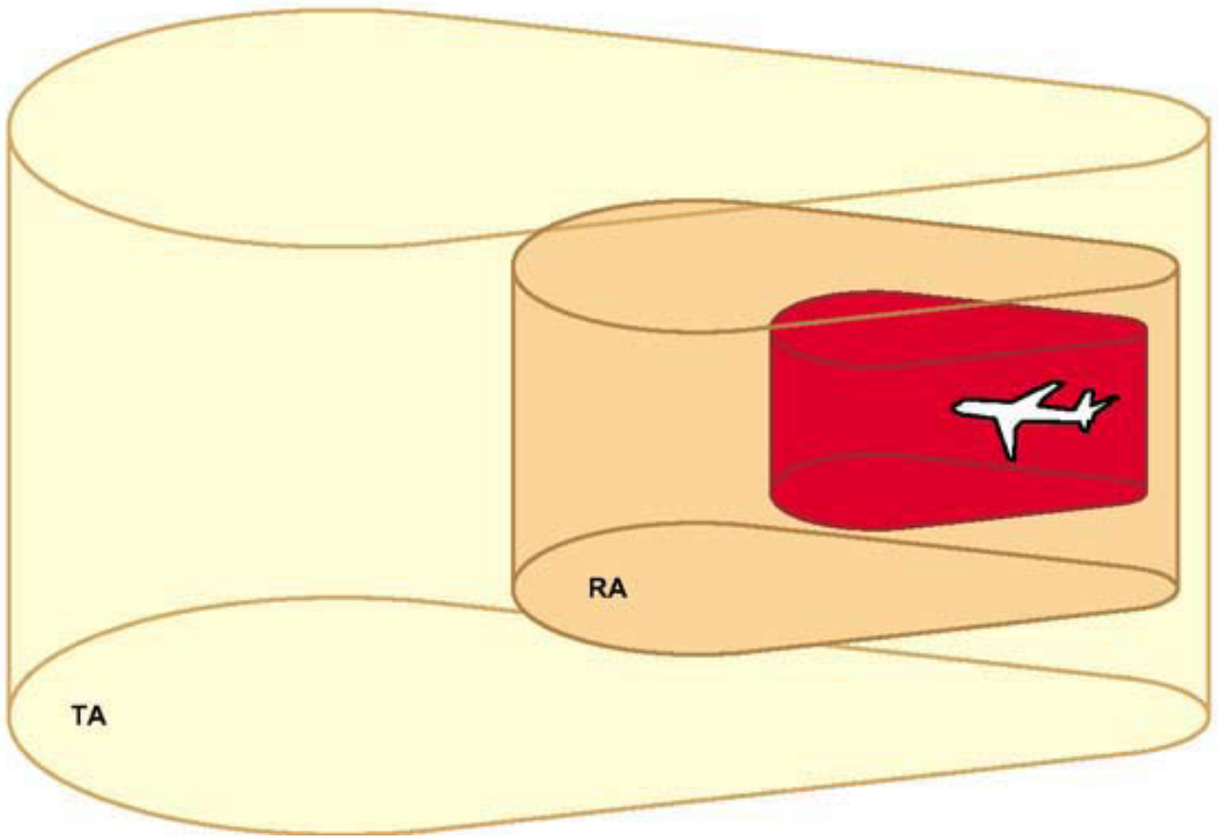


Fig 1. Protection Envelope

FAA (2011) Introduction to TCAS II v7.1, HQ-111358, Wash., DC

TCAS

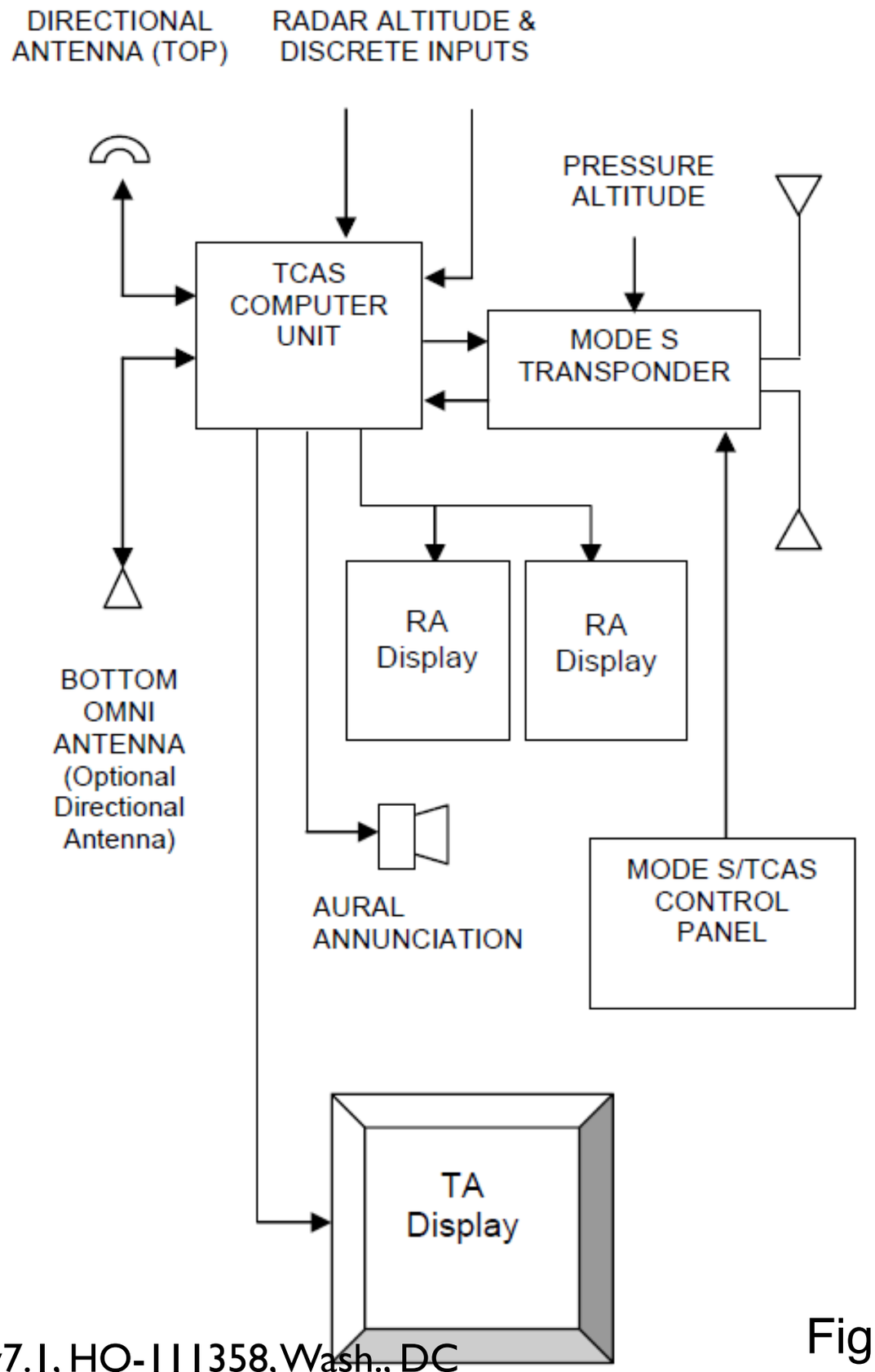
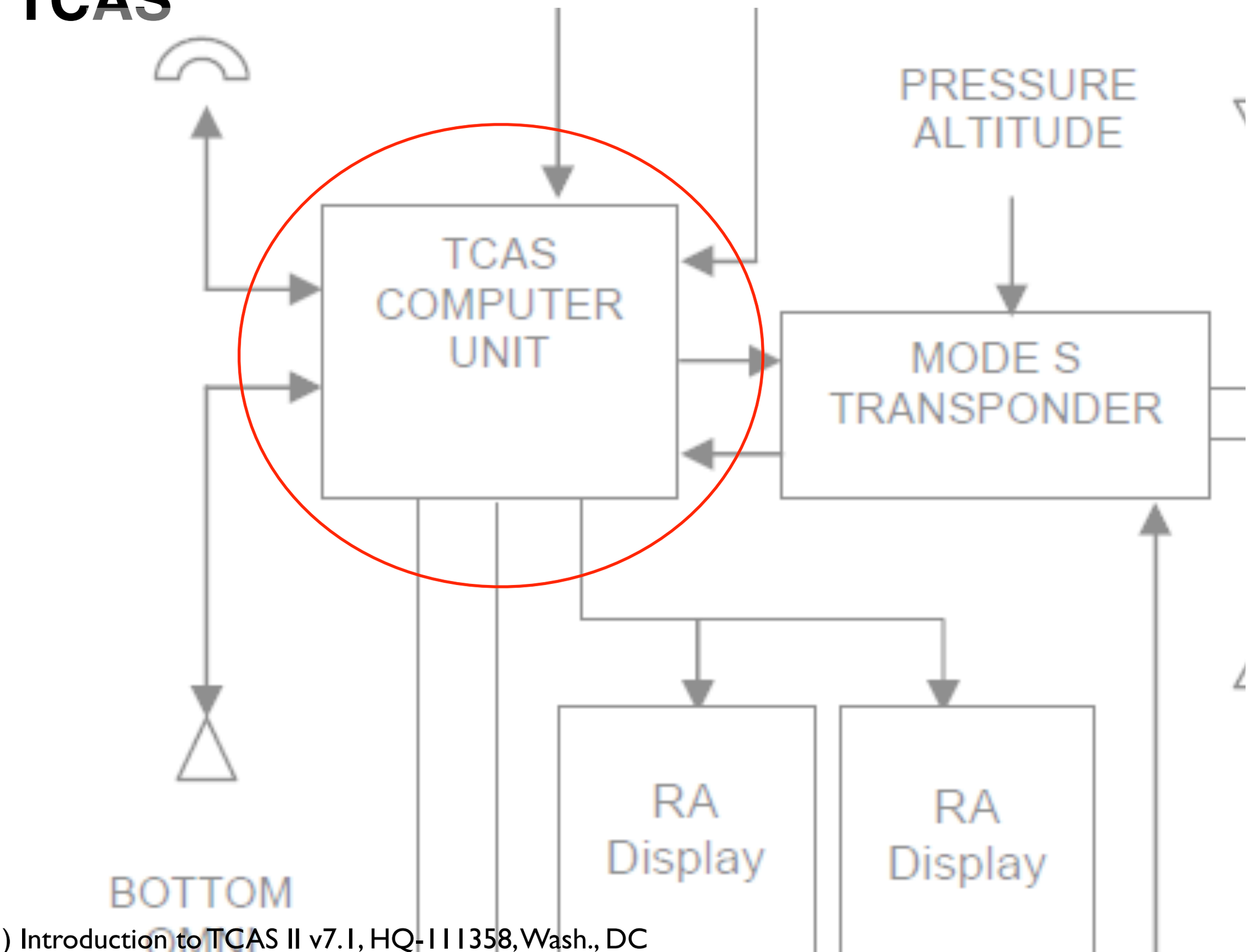


Fig 2. TCAS II Block Diagram

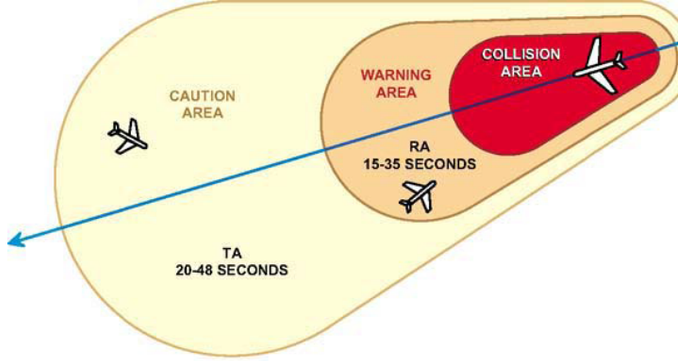
FAA (2011) Introduction to TCAS II v7.1, HQ-111358, Wash., DC

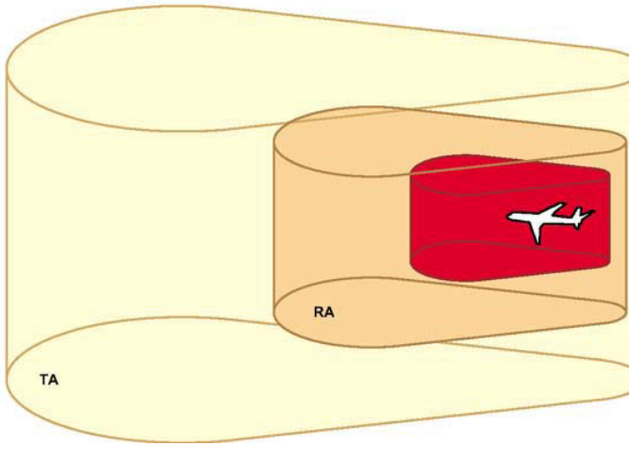
TCAS



FAA (2011) Introduction to TCAS II v7.1, HQ-111358, Wash., DC

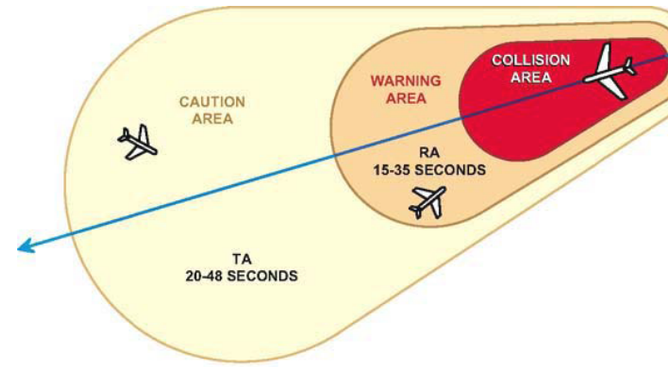
TCAS inputs

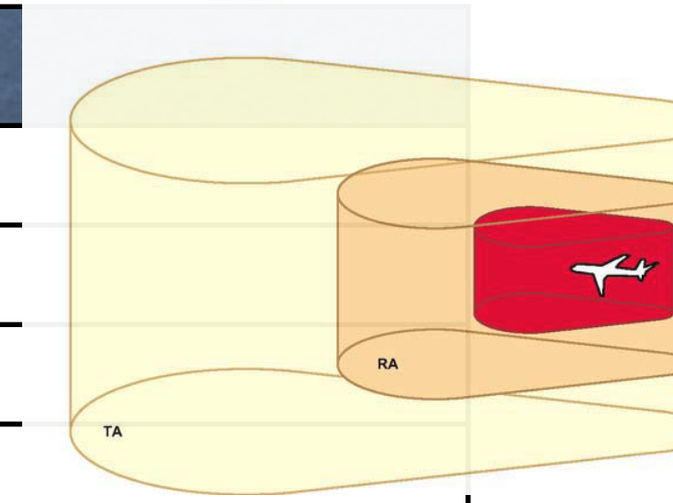




	Input	Type
1	Cur_Vertical_Sep	*continuous*
2	High_Confidence	*boolean*
3	Two_of_Three_Reports_Valid	*boolean*
4	Own_Tracked_Alt	*discrete*
5	Other_Tracked_Alt	*discrete*
6	Own_Tracked_Alt_Rate	*continuous*
7	Alt_Layer_Value	*discrete*
8	Up_Separation	*continuous*
9	Down_Separation	*continuous*
10	Other_RAC	*discrete*
11	Other_Capability	*discrete*
12	Climb_Inhibit	*boolean*

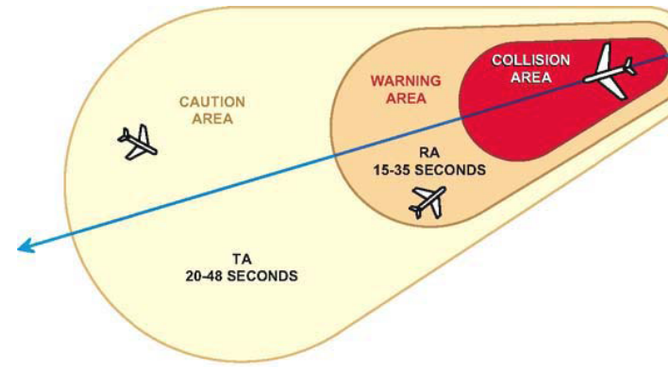
TCAS - design specification



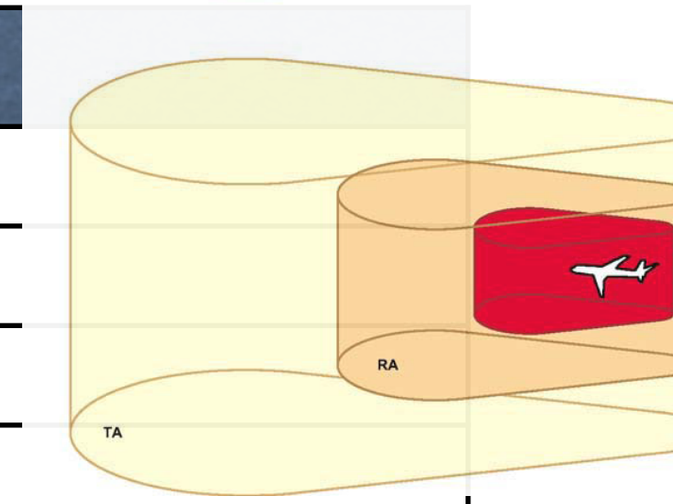


	Input	Levels
1	Cur_Vertical_Sep	*299, 300, 601*
2	High_Confidence	*boolean*
3	Two_of_Three_Reports_Valid	*boolean*
4	Own_Tracked_Alt	*1, 2*
5	Other_Tracked_Alt	*1, 2*
6	Own_Tracked_Alt_Rate	*600, 601*
7	Alt_Layer_Value	*0, 1, 2, 3*
8	Up_Separation	*0, 399, 400, 499, 500, 639, 640, 739, 740, 840*
9	Down_Separation	*0, 399, 400, 499, 500, 639, 640, 739, 740, 840*
10	Other_RAC	*NO_INTENT, DO_NOT_CLIMB, DO_NOT_DESCEND*
11	Other_Capability	*TCAS_TA, OTHER*
12	Climb_Inhibit	*boolean*

TCAS - design specification



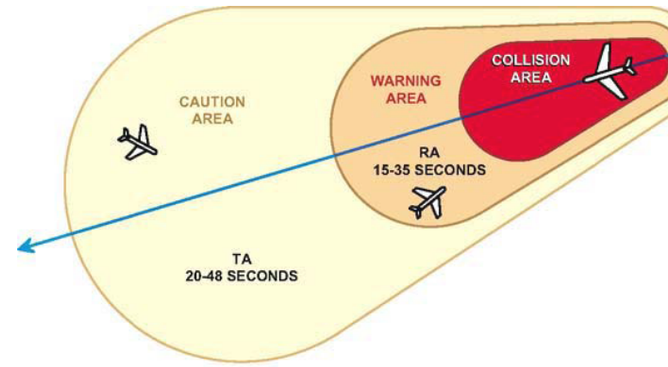
	Input	Levels
1	Cur_Vertical_Sep	*299, 300, 601*
2	High_Confidence	*boolean*
3	Two_of_Three_Reports_Valid	*boolean*
4	Own_Tracked_Alt	*1, 2*
5	Other_Tracked_Alt	*1, 2*
6	Own_Tracked_Alt_Rate	*600, 601*
7	Alt_Layer_Value	*0, 1, 2, 3*
8	Up_Separation	*0, 399, 400, 499, 500, 639, 640, 739, 740, 840*
9	Down_Separation	*0, 399, 400, 499, 500, 639, 640, 739,*
10	Other_RAC	*NO_INTENT, DO_NOT_CLIMB, DO_NOT*
11	Other_Capability	*TCAS_TA, OTHER*
12	Climb_Inhibit	*boolean*



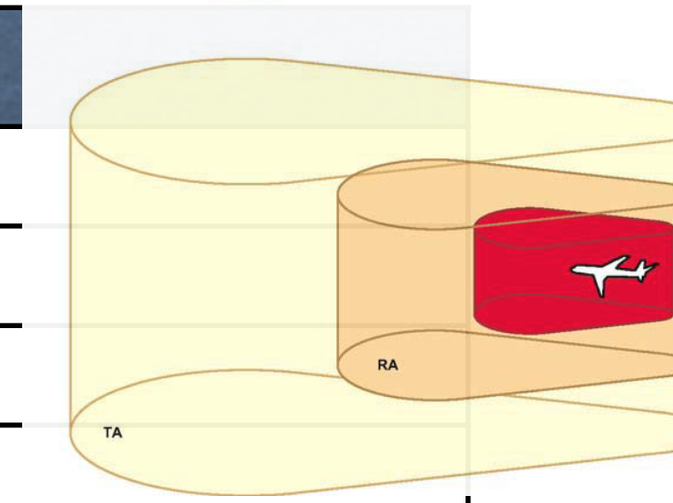
Exhaustive testing is still impossible!

Input space: $3*2^5*4*10^2*3*2^2 = 460{,}800$ points

TCAS - design specification



	Input	Levels
1	Cur_Vertical_Sep	*299, 300, 601*
2	High_Confidence	*boolean*
3	Two_of_Three_Reports_Valid	*boolean*
4	Own_Tracked_Alt	*1, 2*
5	Other_Tracked_Alt	*1, 2*
6	Own_Tracked_Alt_Rate	*600, 601*
7	Alt_Layer_Value	*0, 1, 2, 3*
8	Up_Separation	*0, 399, 400, 499, 500, 639, 640, 739, 740, 840*
9	Down_Separation	*0, 399, 400, 499, 500, 639, 640, 739, 740, 840*
10	Other_RAC	*NO_INTENT, DO_NOT_CLIMB, DO_NOT_DESCEND*
11	Other_Capability	*TCAS_TA, OTHER*
12	Climb_Inhibit	*boolean*



How about using a **CA**(N, t, $(10^2 \cdot 4 \cdot 3^2 \cdot 2^7)$)?

Case Study #2

XGBoost

XGBoost

What is XGBoost?

"An optimized distributed gradient boosting library designed to be highly efficient, flexible and portable.[1]*"*

[1] T. Chen and C. Guestrin. 2016. XGBoost: A Scalable Tree Boosting System.

Combinatorial Testing: XGBoost

	Hyperparameter	Type
1	Max_depth	*continuous*
2	Subsample	*continuous*
3	Colsample_bytree	*continuous*
4	Min_child_weight	*continuous*
5	Alpha	*continuous*
6	Lambda	*continuous*
7	Learning_rate	*continuous*
8	Iterations	*continuous*
9	Tree_method	*categorical (6)*
10	Predictor	*categorical (2)*
11	Grow_policy	*categorical (2)*
12	Booster	*categorical (3)*
13	Process_type	*categorical (2)*
14	Sample_type	*categorical (2)*
15	Feature_selector	*categorical (4)*
16	Colsample_bylevel	*continuous*
17	Colsample_bynode	*continuous*
18	Max_delta_step	*continuous*

	Hyperparameter	Type
19	Gamma	*continuous*
20	Scale_pos_weight	*continuous*
21	Num_parallel_tree	*continuous*
22	Base_score	*continuous*
23	Nthread	*continuous*
24	Seed	*continuous*
25	Sketch_eps	*continuous*
26	Refresh_leaf	*continuous*
27	Max_leaves	*continuous*
28	Max_bin	*continuous*
29	Rate_drop	*continuous*
30	One_drop	*continuous*
31	Skip_drop	*continuous*
32	Top_k	*continuous*
33	Tweedie_variance_power	*continuous*
34	Normalize_type	*categorical (2)*

Combinatorial Testing: XGBoost

	Hyperparameter	Type
1	Max_depth	*continuous*
2	Subsample	*continuous*
3	Colsample_bytree	*continuous*
4	Min_child_weight	*continuous*
5	Alpha	*continuous*
6	Lambda	*continuous*
7	Learning_rate	*continuous*
8	Iterations	*continuous*
9	Tree_method	*categorical (6)*
10	Predictor	*categorical (2)*
11	Grow_policy	*categorical (2)*
12	Booster	*categorical (3)*
13	Process_type	*categorical (2)*
14	Sample_type	*categorical (2)*
15	Feature_selector	*categorical (4)*
16	Colsample_bylevel	*continuous*
17	Colsample_bynode	*continuous*
18	Max_delta_step	*continuous*

	Hyperparameter	Type
19	Gamma	*continuous*
20	Scale_pos_weight	*continuous*
21	Num_parallel_tree	*continuous*
22	Base_score	*continuous*
23	Nthread	*continuous*
24	Seed	*continuous*
25	Sketch_eps	*continuous*
26	Refresh_leaf	*continuous*
27	Max_leaves	*continuous*
28	Max_bin	*continuous*
29	Rate_drop	*continuous*
30	One_drop	*continuous*
31	Skip_drop	*continuous*
32	Top_k	*continuous*
33	Tweedie_variance_power	*continuous*
34	Normalize_type	*categorical (2)*

Challenge: Derive test cases to validate the hyperparameters.

Combinatorial Testing: XGBoost

	Input	# of levels
1	Max_depth	*3*
2	Subsample	*3*
3	Colsample_bytree	*3*
4	Min_child_weight	*3*
5	Alpha	*3*
6	Lambda	*3*
7	Learning_rate	*3*
8	Iterations	*3*
9	Tree_method	*6*
10	Predictor	*2*
11	Grow_policy	*2*
12	Booster	*3*
13	Process_type	*2*
14	Sample_type	*2*
15	Feature_selector	*4*
16	Colsample_bylevel	*3*
17	Colsample_bynode	*3*
18	Max_delta_step	*3*

	Input	# of levels
19	Gamma	*3*
20	Scale_pos_weight	*3*
21	Num_parallel_tree	*3*
22	Base_score	*3*
23	Nthread	*3*
24	Seed	*3*
25	Sketch_eps	*3*
26	Refresh_leaf	*2*
27	Max_leaves	*3*
28	Max_bin	*3*
29	Rate_drop	*3*
30	One_drop	*2*
31	Skip_drop	*3*
32	Top_k	*3*
33	Tweedie_variance_power	*3*
34	Normalize_type	*2*

Combinatorial Testing: XGBoost

	Input	# of levels
1	Max_depth	*3*
2	Subsample	*3*
3	Colsample_bytree	*3*
4	Min_child_weight	*3*
5	Alpha	*3*
6	Lambda	*3*
7	Learning_rate	*3*
8	Iterations	*3*
9	Tree_method	*6*
10	Predictor	*2*
11	Grow_policy	*2*
12	Booster	*3*
13	Process_type	*2*
14	Sample_type	*2*
15	Feature_selector	*4*
16	Colsample_bylevel	*3*
17	Colsample_bynode	*3*
18	Max_delta_step	*3*

	Input	# of levels
19	Gamma	*3*
20	Scale_pos_weight	*3*
21	Num_parallel_tree	*3*
22	Base_score	*3*
23	Nthread	*3*
24	Seed	*3*
25	Sketch_eps	*3*
26	Refresh_leaf	*2*
27	Max_leaves	*3*
28	Max_bin	*3*
29	Rate_drop	*3*
30	One_drop	*2*
31	Skip_drop	*3*
32	Top_k	
33	Tweedie_variance_power	
34	Normalize_type	

Exhaustive testing is impossible!!

Input space: $6*4*3^{25}*2^{7}$ = 2,602,870,608,208,896 points

TCAS, XGBoost, the Profiler & *more*: Combinatorial Testing in Practice

Q&A

sas.com

Thank You

Bibliography

1. B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, 1983.
2. R. Bryce & C. Colbourn, “Prioritized interaction testing for pair-wise coverage with seeding and constraints,” *Information & Software Technology*, 48(10), pp. 960 – 970, 2006.
3. D. Cohen, S. Dalal, M. Fredman, & G. Patton, “The AETG System: An approach to testing based on Combinatorial Design,” *IEEE TSE*, 23(7), 1997, pp. 437-444.
4. M. Cohen, M. Dwyer & J. Shi, “Constructing interaction test suites for highly configurable systems in the presence of constraints: A greedy approach,” *IEEE TSE*, 34(5), 2008, pp. 633-650.
5. C. Colbourn & V. Syrotiuk, “Coverage, location, detection, and measurement,” *IEEE 9th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2016, pp. 19–25.
6. S. Dalal & C. Mallows, “Factor-covering designs for testing software,” *Technometrics*, 40(3), 1998, pp. 234-243.
7. G. Demiroz & C. Yilmaz, “Cost-aware combinatorial interaction testing,” *Proc. of the International Conference on Advances in System Testing and Validation Lifecycles*, 2012, pp. 9–16.
8. I. Dunietz, W. Ehrlich, B. Szablak, C. Mallows, & A. Iannino, “Applying design of experiments to software testing,” *Proceedings of the 19th ICSE*, New York, 1997, pp. 205-215.
9. L. Ghandehari, Y. Lei, D. Kung, R. Kacker, & R. Kuhn,“Fault localization based on failure inducing combinations,” *IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 168–177.
10. A. Hartman & L. Raskin, “Problems and algorithms for covering arrays,” *Discrete Math*, 284(1–3), 2004, pp. 149–156.
11. K. Johnson, & R. Entringer, “Largest induced subgraphs of the n-cube that contain no 4-cycles,” *Journal of Combinatorial Theory, Series B*, 46(3), 1989, pp. 346-355.

Bibliography

12. G. Katona, “Two applications (for search theory and truth functions) of Sperner type theorems,” *Periodica Mathematica Hungarica*, 3(1-2), 1973, pp. 19-26.
13. D. Kleitman & J. Spencer, “Families of k-independent sets,” *Discrete Mathematics*, 6(3), 1973, pp. 255-262.
14. R. Lekivetz, & J. Morgan, “On the testing of statistical software,” *Journal of Statistical Theory and Practice* (2021) (submitted).
15. R. Lekivetz, & J. Morgan, “Fault localization: Analyzing covering arrays given prior information,” *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2018.
16. R. Lekivetz & J. Morgan, “Combinatorial Testing: Using blocking to assign test cases for validating complex software systems,” *Statistical Theory & Related Fields* 5.2 (2021).
17. R. Lekivetz, & J. Morgan, “Covering Arrays: Using Prior Information for Construction, Evaluation and to Facilitate Fault Localization,” *Journal of Statistical Theory and Practice* 14.1 (2020): 7
18. C. King, J. Morgan, & R. Lekivetz. “Design Fractals: A Graphical Method for Evaluating Binary Covering Arrays,” *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2019.
19. J. Morgan, R. Lekivetz, & T. Donnelly. “Covering arrays: Evaluating coverage and diversity in the presence of disallowed combinations,” *2017 IEEE 28th Annual Software Technology Conference (STC)*. IEEE, 2017.
20. J. Morgan, “Combinatorial Testing: An approach to systems and software testing based on covering arrays,” in *Analytic Methods in Systems and Software Testing,* eds., F. Ruggeri, R. Kennett, & F. Faltin, Wiley, pp. 131, 2018.
21. J. Morgan, "Combinatorial Testing,” Wiley StatsRef: Statistics Reference Online (2020): pp. 1-10.
22. G. Myers, *The Art of Software Testing*, Wiley, 1979.