COMS30026 Design Verification

Coverage

Part II: Functional Coverage

Kerstin Eder

(Acknowledgement: Avi Ziv from the IBM Research Labs in Haifa has kindly permitted the re-use of some of his slides.)

University of BRISTOL

Department of
COMPUTER SCIENCE

Outline

- Introduction to coverage
- Part I: Coverage Types
  - Code coverage models
  - (Structural coverage models)
- Part II: Coverage Types (continued)
  - Functional coverage models
- Part III: Coverage Analysis
- Previously: Verification Tools
  - Coverage is part of the Verification Tools.

Functional Coverage

- It is important to cover the **functionality** of the DUV.
  - Most functional requirements can't easily be mapped into lines of code!
- **Functional coverage models** are designed to assure that various aspects of the functionality of the design are verified properly, they link the requirements/specification with the implementation
- Functional coverage models are specific to a given design or family of designs
- Models cover
  - The inputs and the outputs
  - Internal states or microarchitectural features
  - Scenarios
  - Parallel properties
  - Bug Models

Functional Coverage Model Types

1. Discrete set of coverage tasks
   - Set of unrelated or loosely related coverage tasks often derived from the requirements/specification
   - Often used for corner cases
     - Driving data when a FIFO is full
     - Reading from an empty FIFO
   - In many cases, there is a close link between functional coverage tasks and assertions

Functional Coverage Model Types

1. Discrete set of coverage tasks
   - Set of unrelated or loosely related coverage tasks often derived from the requirements/specification
   - Often used for corner cases
     - Driving data when a FIFO is full
     - Reading from an empty FIFO
   - In many cases, there is a close link between functional coverage tasks and assertions
2. Structured coverage models
   - The coverage tasks are defined in a structure that defines relations between the coverage tasks
     - Allow definition of similarity and distance between tasks
     - Most commonly used model types
       - Cross-product
       - Trees
       - Hybrid structures

Cross-Product Coverage Model

[O Lachish, E Marcus, S Ur and A Ziv. Hole Analysis for Functional Coverage Data. In proceedings of the 2002 Design Automation Conference (DAC), June 10-14, 2002, New Orleans, Louisiana, USA.]

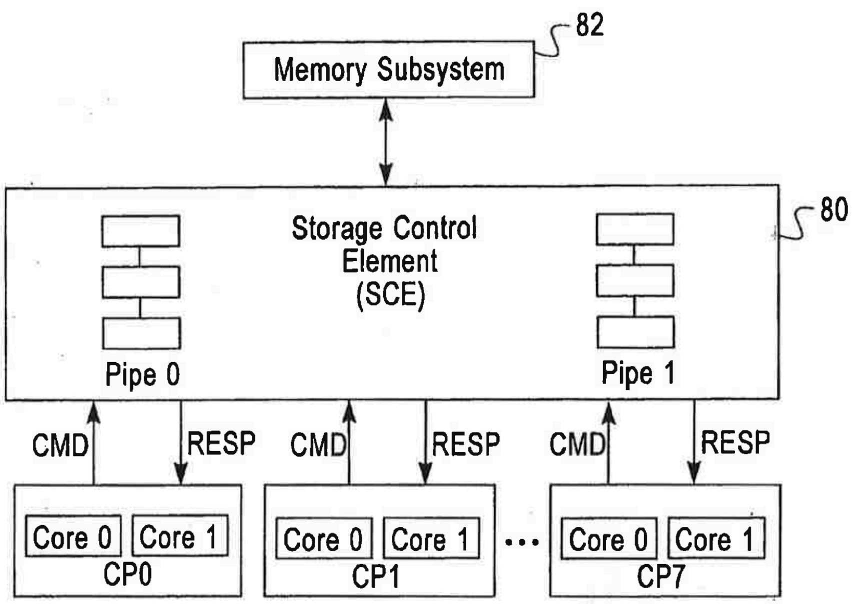
A cross-product coverage model is composed of the following parts:

1. A semantic **description** of the model (story)
2. A list of the **attributes** mentioned in the story
3. A set of all the **possible values** for each attribute (the attribute value **domains**)
4. A list of **restrictions** on the legal combinations in the cross-product of attribute values

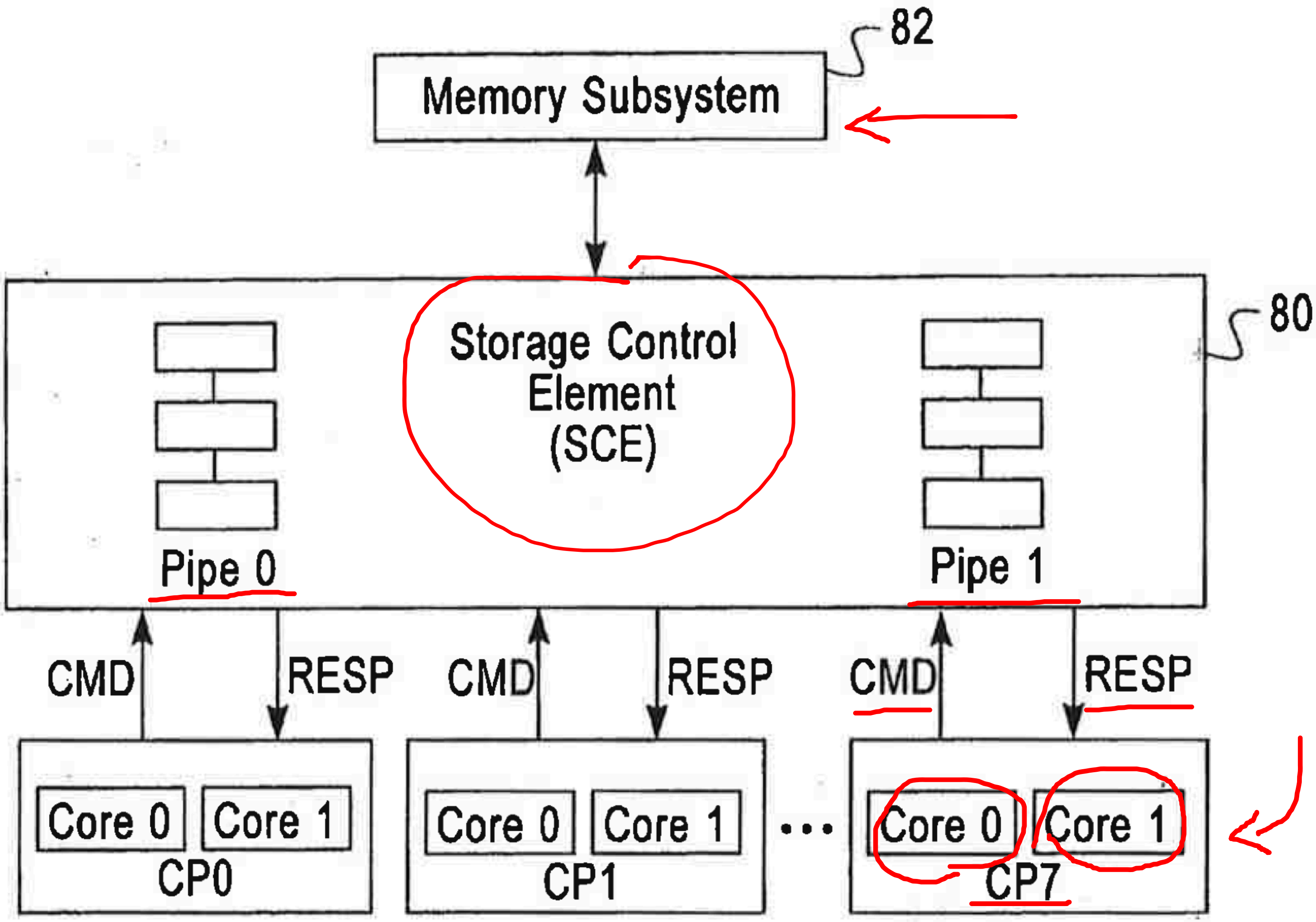
Example: Cross-Product Coverage Model

Design: switch/cache unit

[*G Nativ, S Mittermaier, S Ur and A Ziv. Cost Evaluation of Coverage Directed Test Generation for the IBM Mainframe. In Proceedings of the 2001 International Test Conference, pages 793-802, October 2001.*]

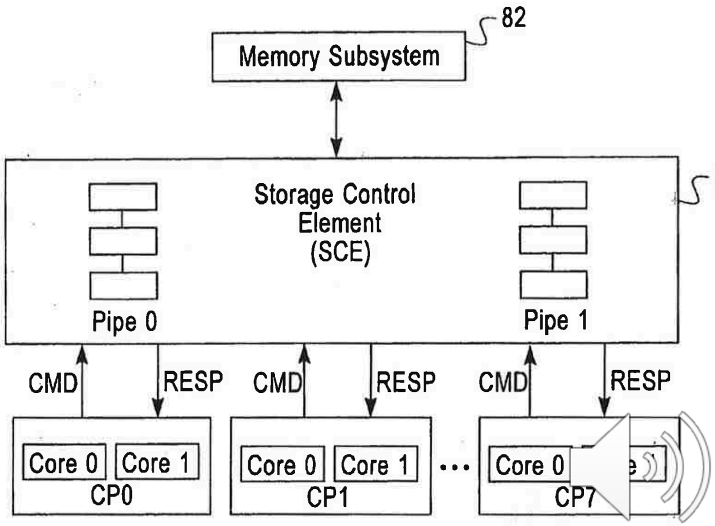


Switch/Cache Unit



Example: Cross-Product Coverage Model

Verification plan: Interactions of core processor unit command-response sequences can create complex and potentially unexpected conditions causing contention within the pipes in the switch/cache unit when many core processors (CPs) are active. **All conditions must be tested to gain confidence in design correctness.**



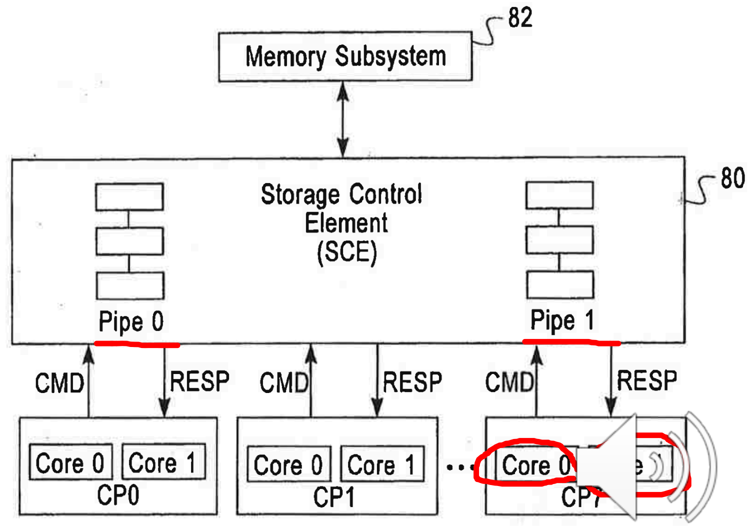
Example: Cross-Product Coverage Model

Verification plan: Interactions of core processor unit command-response sequences can create complex and potentially unexpected conditions causing contention within the pipes in the switch/cache unit when many core processors (CPs) are active. **All conditions must be tested to gain confidence in design correctness.**

the story

Attributes relevant to command-response events:

- Commands - CPs to switch/cache [31]
- Responses - switch/cache to CPs [16]
- Pipes in each switch/cache [2]
- CPs in the system [8]
- (Command generators per CP chip [2])



Example: Cross-Product Coverage Model

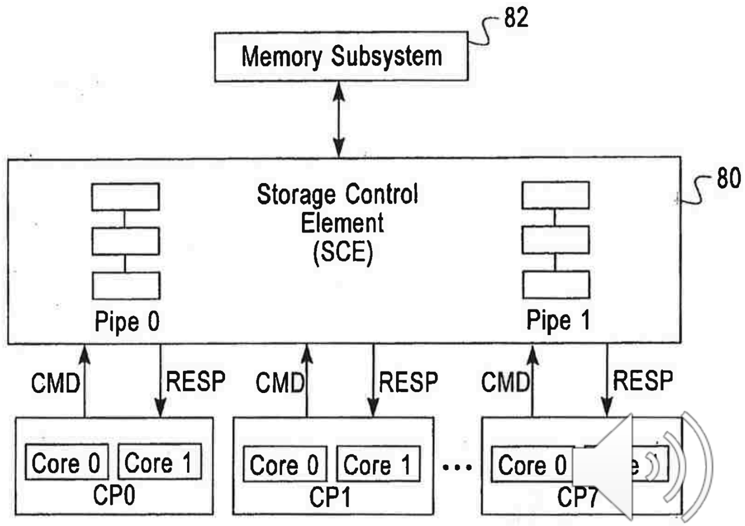
Verification plan: Interactions of core processor unit command-response sequences can create complex and potentially unexpected conditions causing contention within the pipes in the switch/cache unit when many core processors (CPs) are active. **All conditions must be tested to gain confidence in design correctness.**

the story

Attributes relevant to command-response events:

- Commands - CPs to switch/cache [31]
- Responses - switch/cache to CPs [16]
- Pipes in each switch/cache [2]
- CPs in the system [8]
- (Command generators per CP chip [2])

How big is the coverage space, i.e. how many coverage tasks?



Example: Size of Coverage Space

Size of coverage space:

- Coverage space is formed by **cross-product (or, more formally, the Cartesian product) over all attribute value domains.**
- Size of cross-product is product of domain sizes:
  - 31x16x2x8x2 = 15872
- Hence, there are 15872 coverage tasks.

How does such a coverage task look like?

Example: Size of Coverage Space

Size of coverage space:

- Coverage space is formed by **cross-product (or, more formally, the Cartesian product) over all attribute value domains.**
- Size of cross-product is product of domain sizes:
  - 31x16x2x8x2 = 15872
- Hence, there are 15872 coverage tasks.

Example coverage task:

(20,01,1,5,0)

Example: Size of Coverage Space

Size of coverage space:

- Coverage space is formed by **cross-product (or, more formally, the Cartesian product) over all attribute value domains.**
- Size of cross-product is product of domain sizes:
  - 31x16x2x8x2 = 15872
- Hence, there are 15872 coverage tasks.

Example coverage task:

(20,01,1,5,0) = (Command=20, Response=01, Pipe=1, CP=5, CG=0)

Example: Size of Coverage Space

Size of coverage space:

- Coverage space is formed by **cross-product (or, more formally, the Cartesian product) over all attribute value domains.**
- Size of cross-product is product of domain sizes:
  - 31x16x2x8x2 = 15872
- Hence, there are 15872 coverage tasks.

Example coverage task:

(20,01,1,5,0) = (Command=20, Response=01, Pipe=1, CP=5, CG=0)

Are all of these tasks reachable/legal?

Example: Size of Coverage Space

Size of coverage space:

- Coverage space is formed by **cross-product (or, more formally, the Cartesian product) over all attribute value domains.**
- Size of cross-product is product of domain sizes:
  - 31x16x2x8x2 = 15872
- Hence, there are 15872 coverage tasks.

Example coverage task:

(20,01,1,5,0) = (Command=20, Response=01, Pipe=1, CP=5, CG=0)

Are all of these tasks reachable/legal?

- Restrictions on the coverage model are:
  - limited number of possible responses for each command, i.e. not all the 16 responses are valid for each command
  - unimplemented command/response combinations
  - some commands are only executed in pipe 1
- After applying restrictions, there are 1968 legal coverage tasks left

Example: Size of Coverage Space

Size of coverage space:

- Coverage space is formed by **cross-product (or, more formally, the Cartesian product) over all attribute value domains.**
- Size of cross-product is product of domain sizes:
  - 31x16x2x8x2 = 15872
- Hence, there are 15872 coverage tasks.

Example coverage task:

(20,01,1,5,0) = (Command=20, Response=01, Pipe=1, CP=5, CG=0)

Are all of these tasks reachable/legal?

- Restrictions on the coverage model are:
  - limited number of possible responses for each command, i.e. not all the 16 responses are valid for each command
  - unimplemented command/response combinations
  - some commands are only executed in pipe 1
- After applying restrictions, there are 1968 legal coverage tasks left

Example: Size of Coverage Space

Size of coverage space:

- Coverage space is formed by **cross-product (or, more formally, the Cartesian product) over all attribute value domains.**
- Size of cross-product is product of domain sizes:
  - 31x16x2x8x2 = 15872
- Hence, there are 15872 coverage tasks.

Example coverage task:

(20,01,1,5,0) = (Command=20, Response=01, Pipe=1, CP=5, CG=0)

Are all of these tasks reachable/legal?

- Restrictions on the coverage model are:
  - limited number of possible responses for each command, i.e. not all the 16 responses are valid for each command
  - unimplemented command/response combinations
  - some commands are only executed in pipe 1
- After applying restrictions, there are 1968 legal coverage tasks left

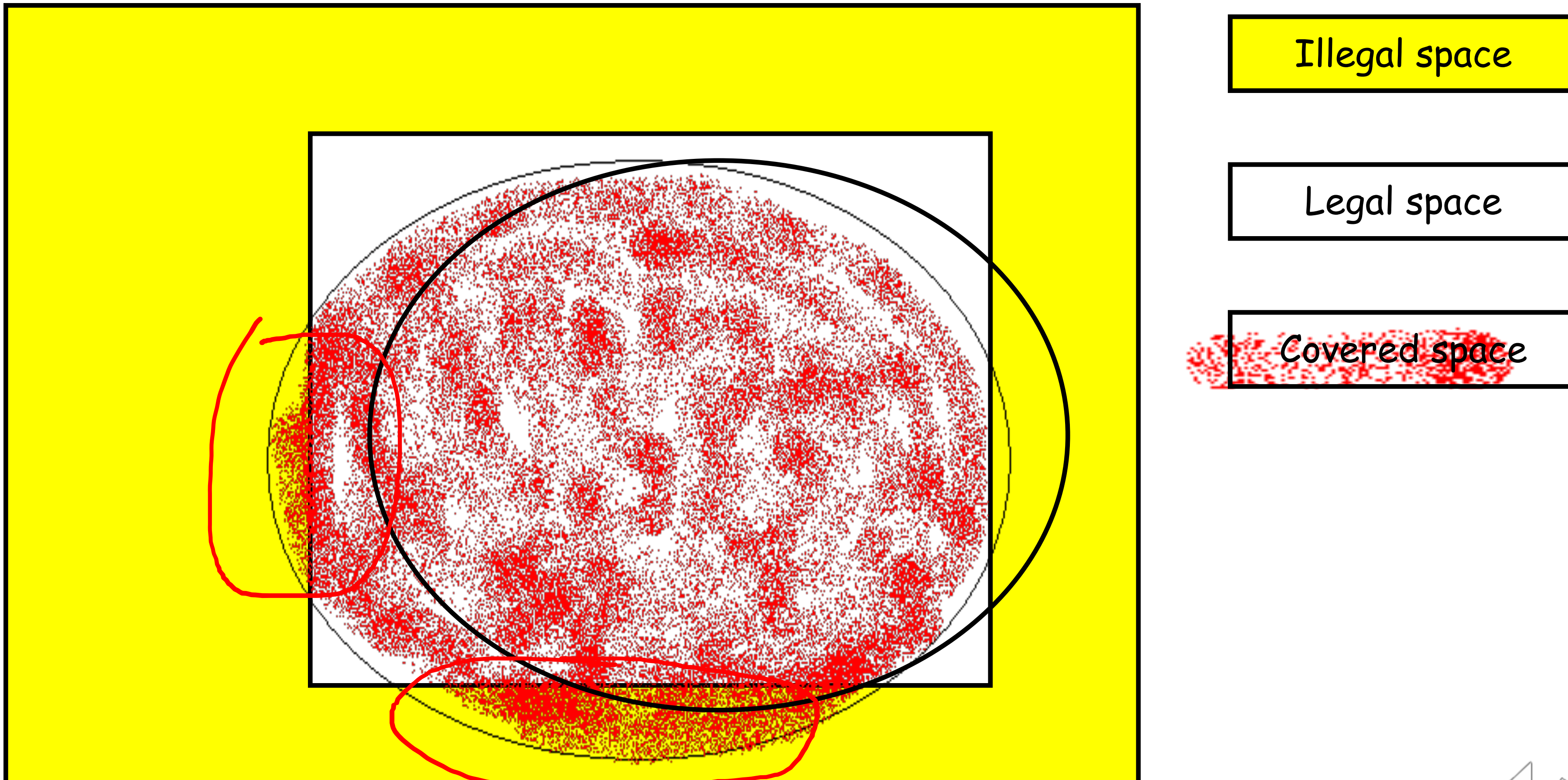
Make sure you identify & apply restrictions before you start!

Defining the Legal and Interesting Spaces

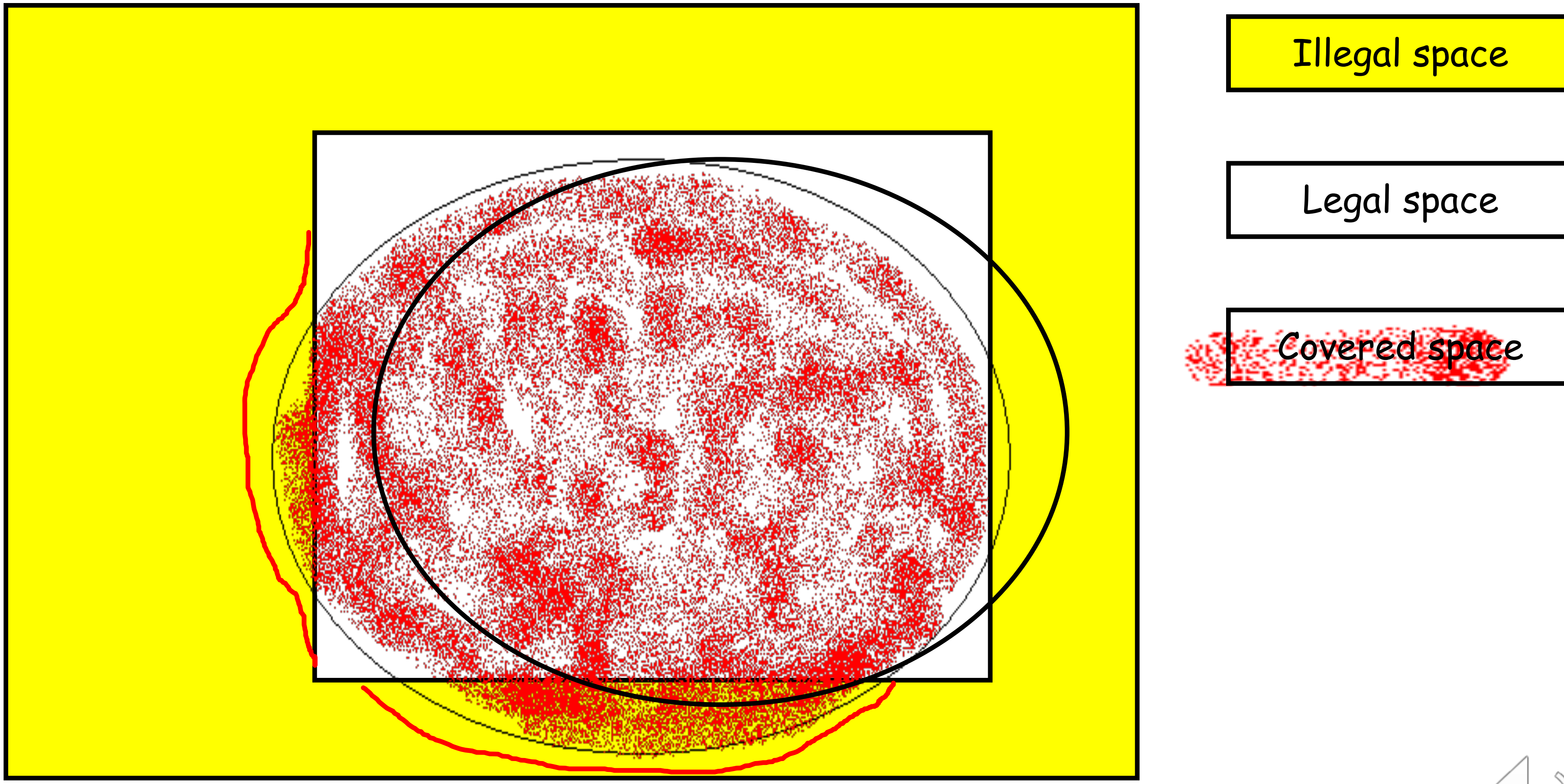
In Practice:

- Boundaries between legal and illegal coverage spaces are often not well understood
- The design and verification team create initial spaces based on their understanding of the design
- Coverage feedback is used to modify the definition of the coverage spaces
- Sub-models are used to economically check and refine the coverage spaces
  - Easy to define as these are sub-crosses!
- Interesting spaces tend to change often due to a shift in focus in the verification process

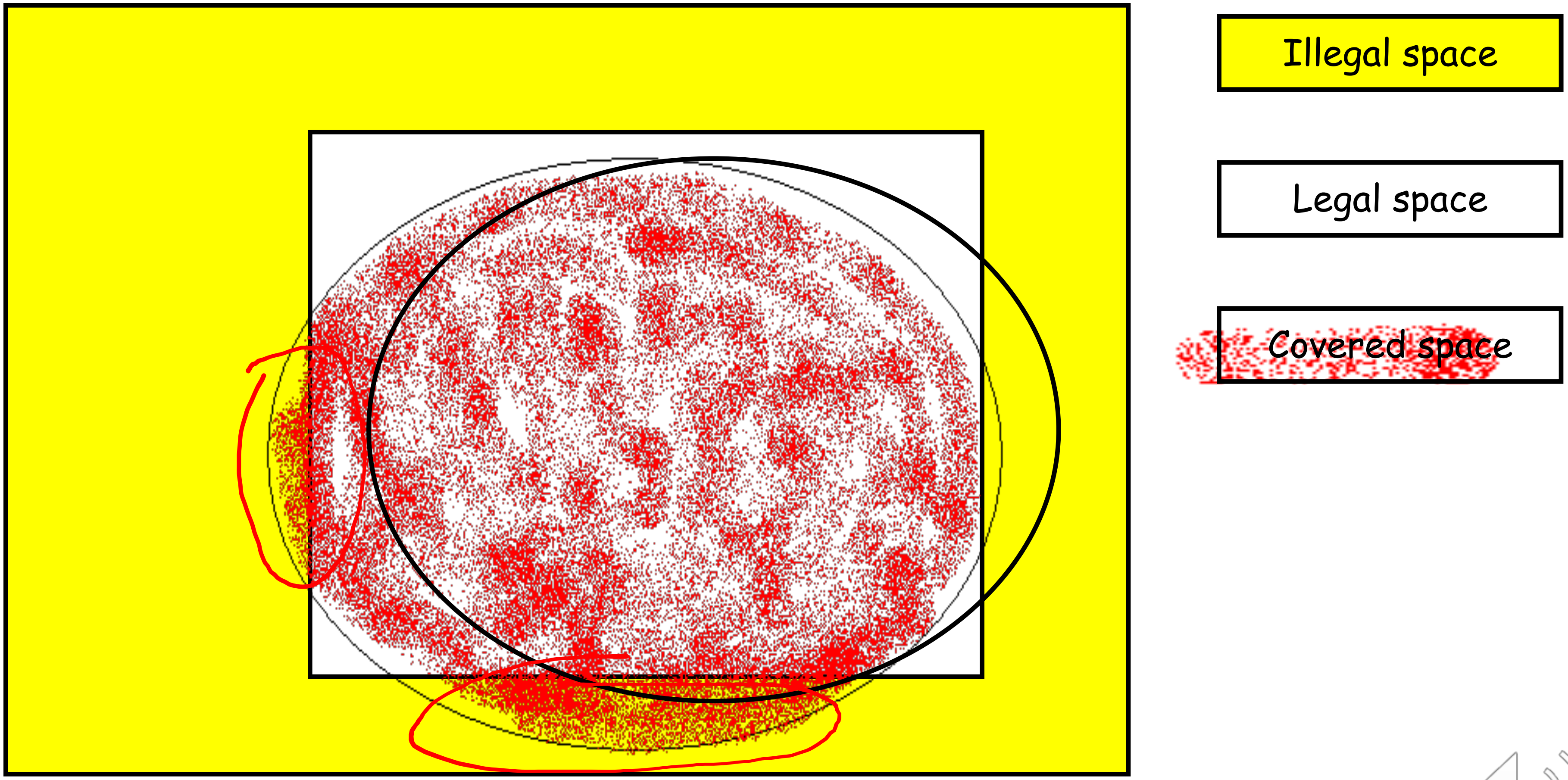
Legal Spaces Are Self-correcting



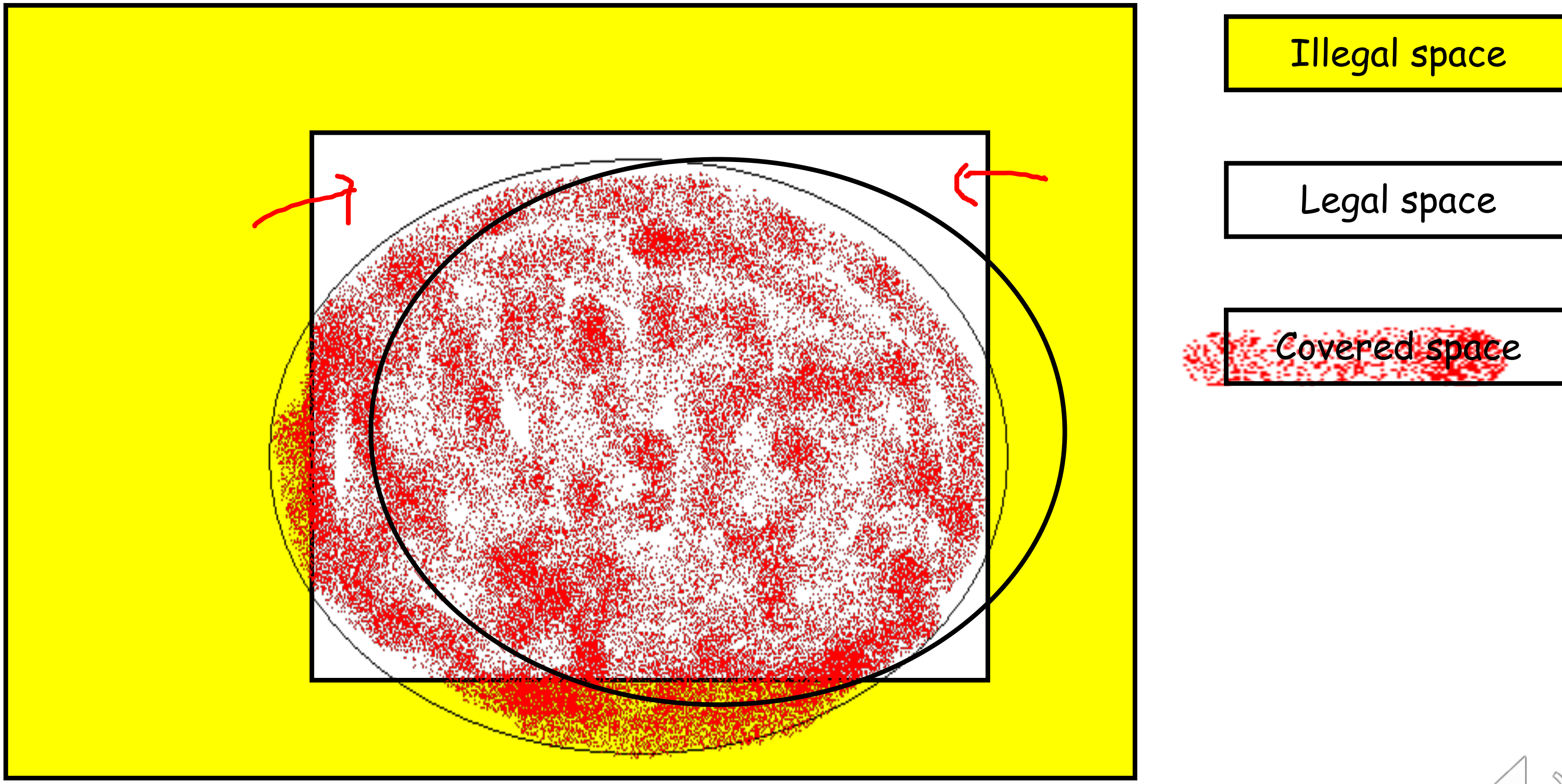
Legal Spaces Are Self-correcting



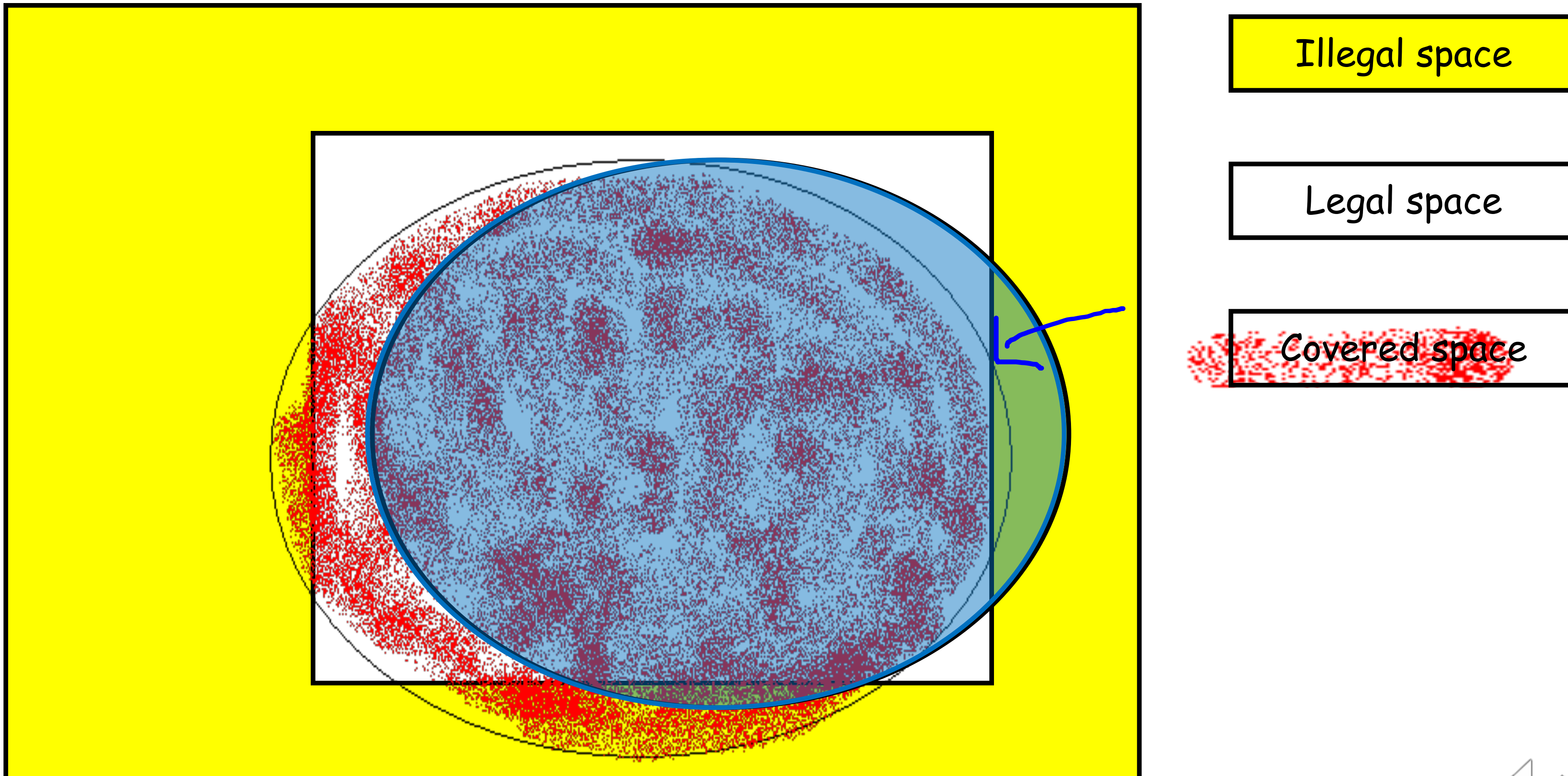
Legal Spaces Are Self-correcting



Legal Spaces Are Self-correcting



Legal Spaces Are Self-correcting



Cross-Product Coverage more formally

- Functional cross-product coverage models can be defined using **multi-dimensional coverage spaces**.
- A **functional coverage space** C_m is defined as the Cartesian product over m signal domains $D_0; \ldots; D_{m-1}$.
  - $C_m = D_0 \, X \ldots X \, D_{m-1}$

Cross-Product Coverage more formally

- Functional cross-product coverage models can be defined using **multi-dimensional coverage spaces**.
- A **functional coverage space** C_m is defined as the Cartesian product over m signal domains $D_0; \ldots; D_{m-1}$.
  - $C_m = D_0 X \ldots X D_{m-1}$
- Let $|D_k| = d_k$ denote the **size of domain** D_k.
- The functional coverage space C_m contains $|C_m| = |D_0| * \ldots * |D_{m-1}| = d$ distinct **coverage points** $p_0; \ldots; p_{d-1}$.

Cross-Product Coverage more formally

- Functional cross-product coverage models can be defined using **multi-dimensional coverage spaces**.
- A **functional coverage space** C_m is defined as the Cartesian product over m signal domains $D_0; \ldots; D_{m-1}$.
  - $C_m = D_0 X \ldots X D_{m-1}$
- Let $|D_k| = d_k$ denote the **size of domain** D_k.
- The functional coverage space C_m contains $|C_m| = |D_0| * \ldots * |D_{m-1}| = d$ distinct **coverage points** $p_0; \ldots; p_{d-1}$.
- A **coverage point** p_i with $i \in \{0; \ldots; d-1\}$ is characterized by an ***m*-tuple of values** $p_i = (v_0; \ldots; v_{m-1})$, where $p_i[k] = v_k$ and each $v_k \in D_k$, for $k \in \{0; \ldots; m-1\}$.

Cross-Product Coverage more formally

- Functional cross-product coverage models can be defined using **multi-dimensional coverage spaces.**
- A **functional coverage space** C_m is defined as the Cartesian product over m signal domains $D_0; \ldots; D_{m-1}$.
  - $C_m = D_0 \, X \ldots X \, D_{m-1}$
- Let $|D_k| = d_k$ denote the **size of domain** D_k.
- The functional coverage space C_m contains $|C_m| = |D_0| * \ldots * |D_{m-1}| = d$ distinct **coverage points** $p_0; \ldots; p_{d-1}$.
- A **coverage point** p_i with $i \in \{0; \ldots; d-1\}$ is characterized by an ***m*-tuple of values** $p_i = (v_0; \ldots; v_{m-1})$, where $p_i[k] = v_k$ and each $v_k \in D_k$, for $k \in \{0; \ldots; m-1\}$.

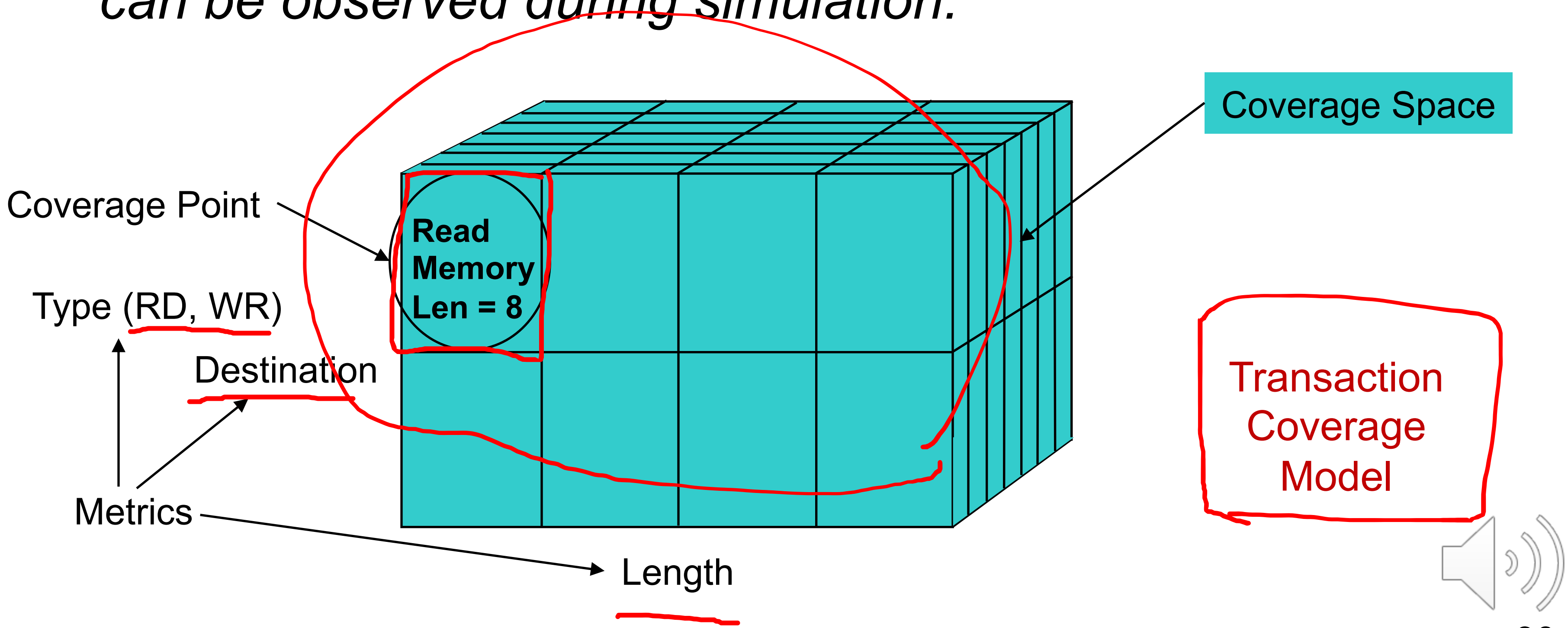
Formalization facilitates automation of coverage analysis e.g. identification of coverage holes.

Coverage Terminology

- **cov·er·age model** *n. 1. A set of legal and interesting coverage points in the coverage space.*
- **cov·er·age point/task** *n. 1. A point within a multi-dimensional coverage space. 2. An event of interest that can be observed during simulation.*

Coverage Terminology

- **cov·er·age model** *n. 1. A set of legal and interesting coverage points in the coverage space.*
- **cov·er·age point/task** *n. 1. A point within a multi-dimensional coverage space. 2. An event of interest that can be observed during simulation.*



Cross-Product Models In *e*

Verification Languages such as e support cross-product coverage models:

- The story is hidden in the event
- The attributes and their values are defined in the coverage items
- The coverage space can be constrained using the illegal and ignore constructs
  - Restrictions can be defined on the coverage items and the cross itself

```
struct instruction {
  opcode: [NOP, ADD, SUB, SHL,
    SHR, AND, OR, XOR] (bits:3);
  !response : uint (bits:2);

  event instruction_complete;

  cover instruction_complete is {
    item opcode;
    item response;
    cross opcode, response
      using
        ignore = (opcode == NOP);
  };
};
```

Cross-Product Models In *e*

Verification Languages such as e support cross-product coverage models:

- The story is hidden in the event
- The attributes and their values are defined in the coverage items
- The coverage space can be constrained using the illegal and ignore constructs
  - Restrictions can be defined on the coverage items and the cross itself

```
struct instruction {
  opcode: [NOP, ADD, SUB, SHL,
    SHR, AND, OR, XOR] (bits:3);
  response : uint (bits:2);

  event stimulus;

  cover stimulus is {
    item opcode;
    item response;
    cross opcode, response
      using
        ignore = (opcode == NOP);
  };
};
```

New: Situation Coverage

	┬	┤	┘	┐	─	╷	╴	┼	┴	├	└	┌	│	╵	╶
Car															
Bike															
HGV															
Ped															

Alexander, Rob; Hawkins, Heather Rebecca; Rae, Andrew John
Situation coverage – a coverage criterion for testing autonomous robots.
Department of Computer Science, University of York, 2015. 21 pages.

PUTTING IT ALL TOGETHER

Summary: Functional Coverage

Determines whether the **functionality** of the DUV has been exercised (and so verified).

- Functional coverage models are **user-defined.**
  - The story is driven by the specification and the verification plan.
  - Defining them is a skill. It needs (lots of) experience!
  - Focus on **control signals.** WHY?

Summary: Functional Coverage

Determines whether the **functionality** of the DUV has been exercised (and so verified).

- Functional coverage models are **user-defined.**
  - The story is driven by the specification and the verification plan.
  - Defining them is a skill. It needs (lots of) experience!
  - Focus on **control signals.** WHY?
- **Strengths:**
  - Highly expressive, can capture cross-correlation, multi-cycle scenarios and sequences over time.
  - Can identify coverage holes by crossing existing items.
  - Results are easy to interpret.
  - Gives an objective measure of progress against verification plan.

Summary: Functional Coverage

Determines whether the **functionality** of the DUV has been exercised (and so verified).

- Functional coverage models are **user-defined.**
  - The story is driven by the specification and the verification plan.
  - Defining them is a skill. It needs (lots of) experience!
  - Focus on **control signals.** WHY?
- **Strengths:**
  - Highly expressive, can capture cross-correlation, multi-cycle scenarios and sequences over time.
  - Can identify coverage holes by crossing existing items.
  - Results are easy to interpret.
  - Gives an objective measure of progress against verification plan.
- **Weaknesses:**
  - Engineering effort is required and a lot of expertise to construct the coverage model.
  - Only as good as the coverage model captures the functionality.

Summary: Code Coverage

Determines whether all the **implementation** has been exercised (and therefore verified).

- Models are implicitly defined by the source code.
  - Code coverage is implementation driven.
  - statement, path, expression, toggle, etc.

Summary: Code Coverage

Determines whether all the **implementation** has been exercised (and therefore verified).

- Models are implicitly defined by the source code.
  - Code coverage is implementation driven.
  - statement, path, expression, toggle, etc.

- **Strengths:**
  - Reveals unexercised parts of design.
  - May reveal gaps in functional verification plan.
  - No manual effort is required to implement the metrics. (Comes for free!)

Summary: Code Coverage

Determines whether all the **implementation** has been exercised (and therefore verified).

- Models are implicitly defined by the source code.
  - Code coverage is implementation driven.
  - statement, path, expression, toggle, etc.

- **Strengths:**
  - Reveals unexercised parts of design.
  - May reveal gaps in functional verification plan.
  - No manual effort is required to implement the metrics. (Comes for free!)
- **Weaknesses:**
  - No cross correlations.
  - Can't see multi-cycle/concurrent scenarios.
  - Manual effort required to interpret results.

Conclusions on Coverage Types

We need both code and functional coverage

Functional Coverage	Code Coverage	Interpretation
Low	Low	There is verification work to do.
Low	High	Multi-cycle scenarios, corner cases, cross-correlations still to be covered.
High	Low	Verification plan and/or functional coverage metrics inadequate. Check for "dead" code.
High	High	High confidence in quality.

- Coverage models complement each other!
- No single coverage model is adequate on its own.