COMS30026 Design Verification

High-level Verification with specman and e

Part 1: Introduction

Kerstin Eder

Trustworthy Systems Laboratory

https://www.bristol.ac.uk/engineering/research/trustworthy-systems-laboratory/

University of BRISTOL

Department of COMPUTER SCIENCE

High-level Verification

- **State-of-the-art Verification Methodology**
  - Focus on **Automation** of the Verification Process
  - Tools: originally from Verisity and now from Cadence (who bought Verisity in April 2005)
    - Specman Elite (SN) and
    - “e” verification language

High-level Verification

- **State-of-the-art Verification Methodology**
  - Focus on **Automation** of the Verification Process
  - Tools: originally from Verisity and now from Cadence (who bought Verisity in April 2005)
    - Specman Elite (SN) and
    - “e” verification language
- **EDA Software Access**
  - Access to specman and the Cadence verification tools should automatically be enabled if you follow the instructions on EDA Software Access as described online at https://uobdv.github.io/Design-Verification/
- *For those taking this unit with coursework, it is beneficial for you to work through at least 75% of the **Specman Fundamentals for Block-Level Environment Developers** online training course.*

[Credits: The material for this lecture is adapted from Verisity/Cadence training material.]

SN Main Enabling Technologies

- **Constraint-driven Test Generation**
  - Create lots of meaningful tests quickly. ☺
  - Control over automatic test generation.
  - Capture constraints from specification and verification plan.

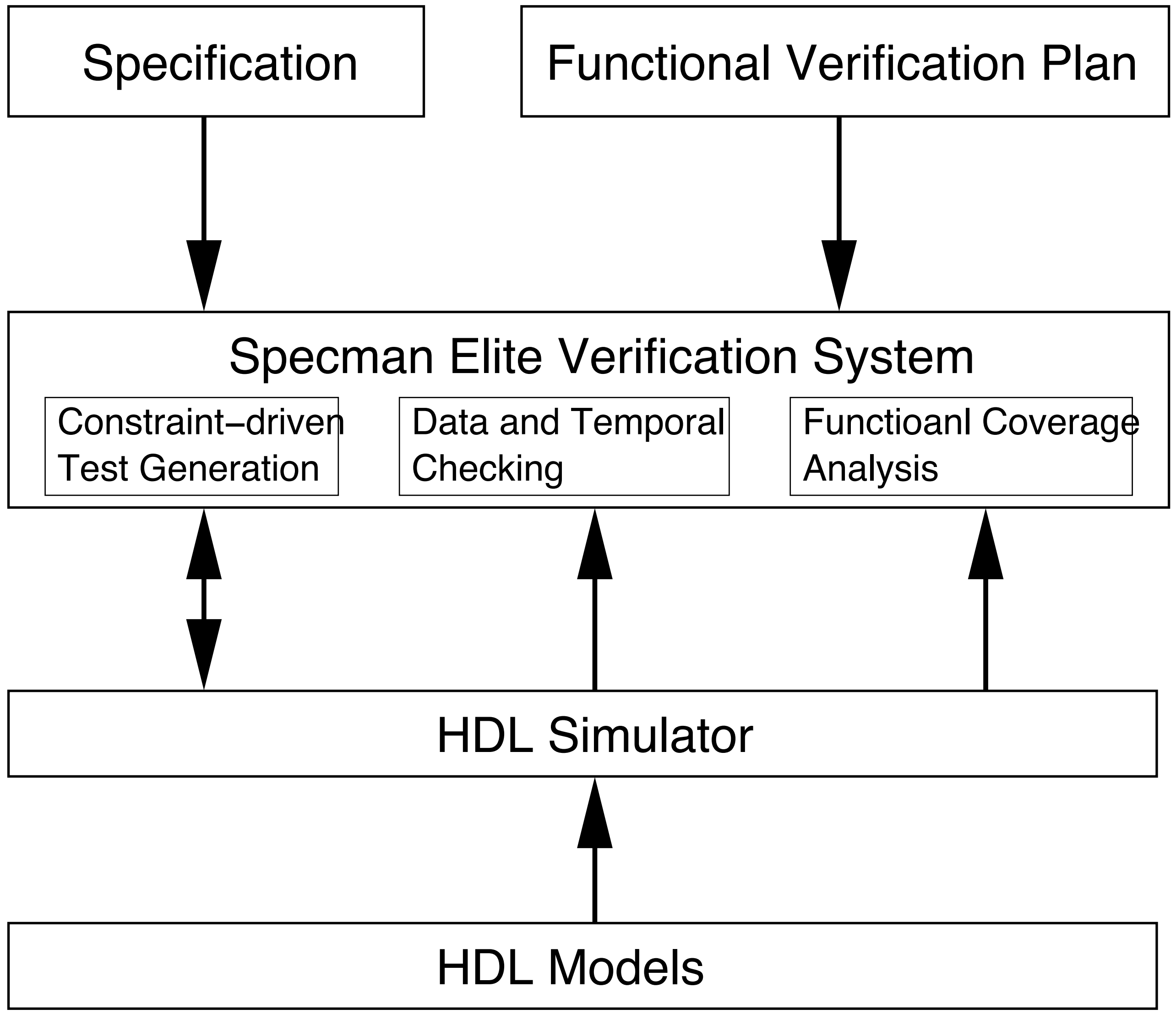
SN Main Enabling Technologies

- **Constraint-driven Test Generation**
  - Create lots of meaningful tests quickly. ☺
  - Control over automatic test generation.
  - Capture constraints from specification and verification plan.
- **Data and Temporal Checking**
  - Self-checking modules ensure data correctness and satisfaction of temporal properties.
  - Checks are always active.
    - Unless turned off by: `set check IGNORE` ;-)

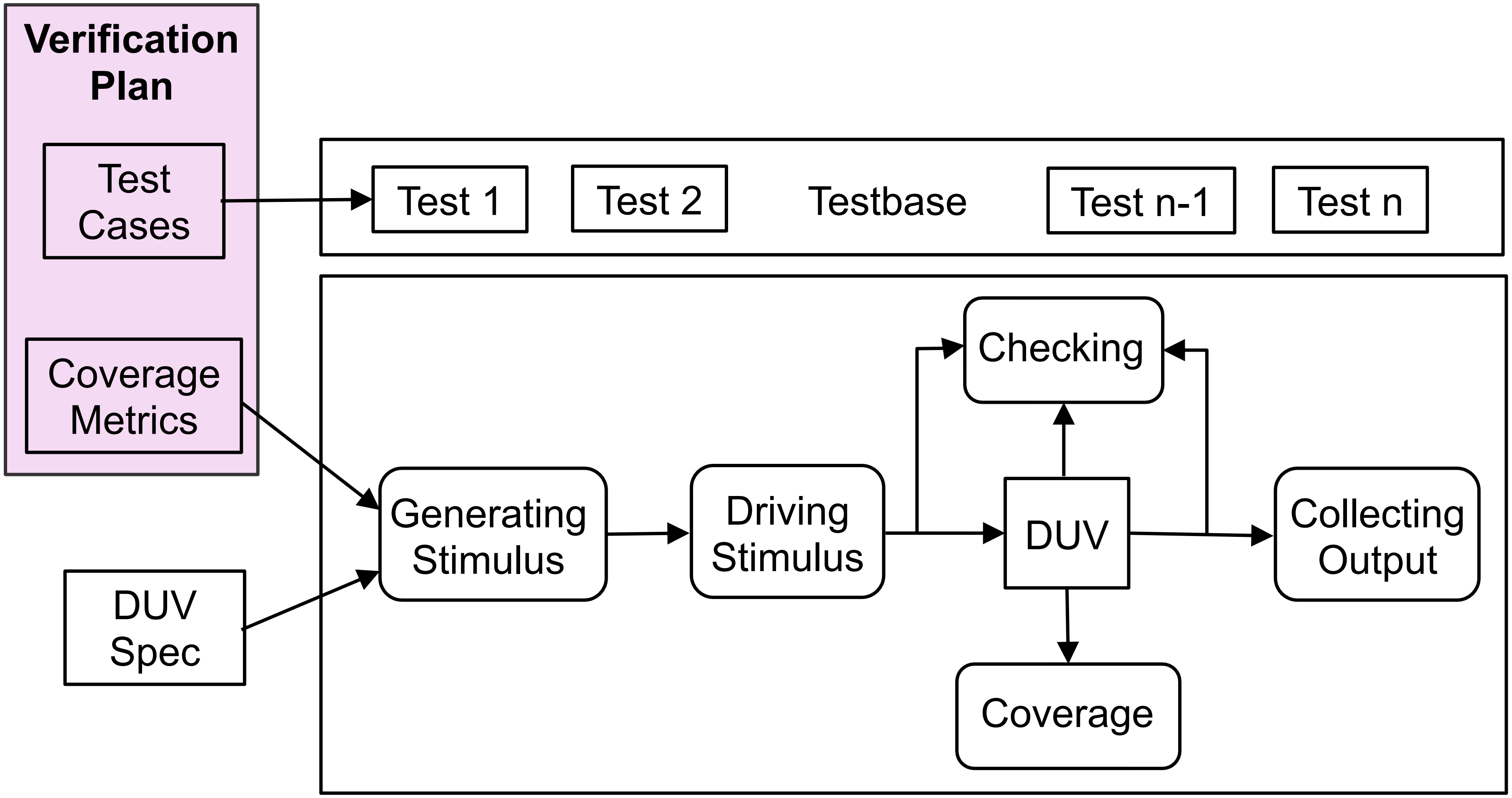
SN Main Enabling Technologies

- **Constraint-driven Test Generation**
  - Create lots of meaningful tests quickly. ☺
  - Control over automatic test generation.
  - Capture constraints from specification and verification plan.
- **Data and Temporal Checking**
  - Self-checking modules ensure data correctness and satisfaction of temporal properties.
  - Checks are always active.
    - Unless turned off by: `set check IGNORE` ;-)
- **Functional Coverage Collection and Analysis**
  - Automatic functional coverage collection.
  - Analyse progress against functional coverage metrics.
- **Promotes Coverage-Driven Verification (CDV)**

SN Verification Environment

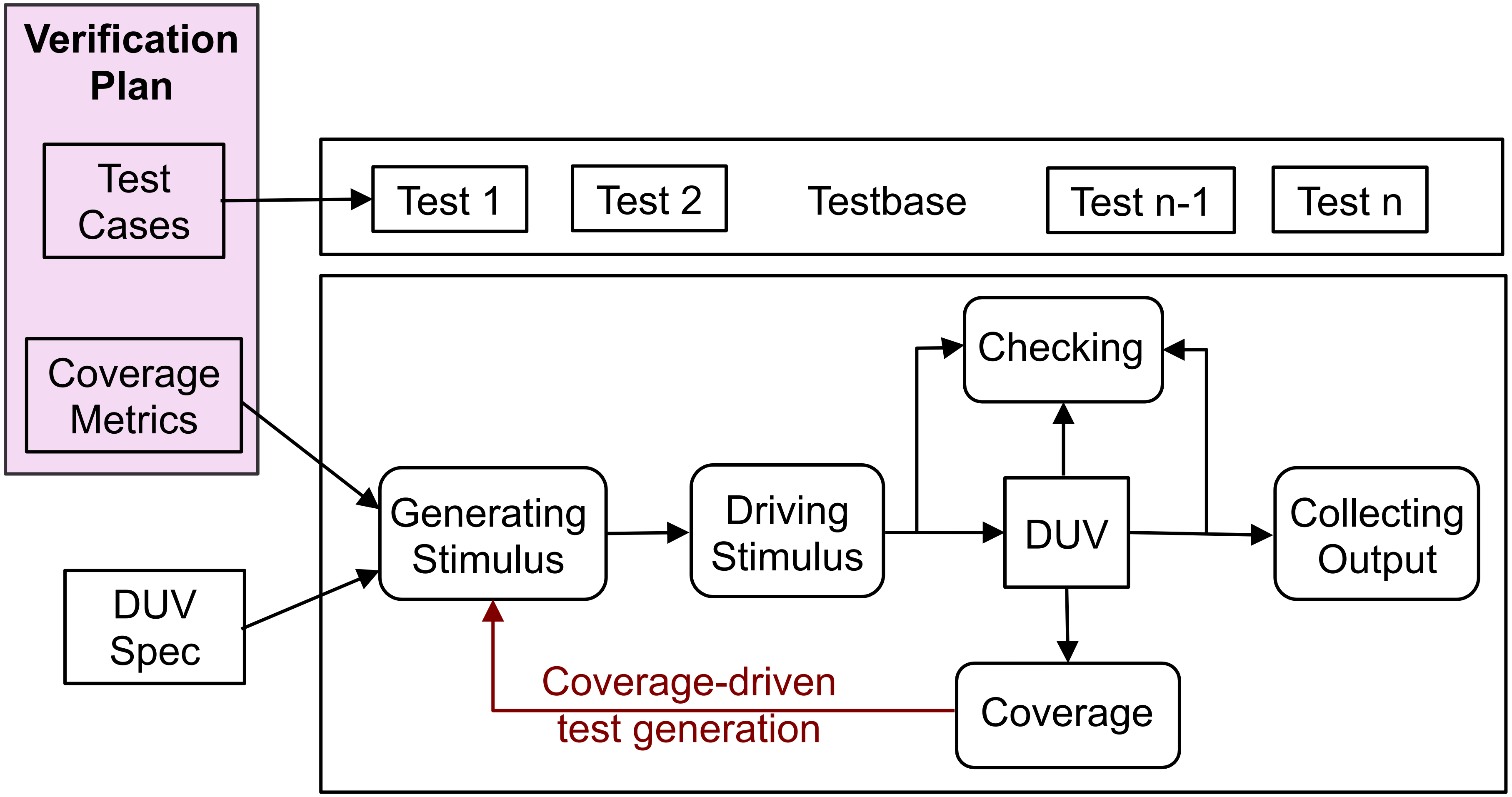

Specification
Functional Verification Plan
Specman Elite Verification System
Constraint–driven Test Generation
Data and Temporal Checking
Functioanl Coverage Analysis
HDL Simulator
HDL Models

Complete SN Verification Process

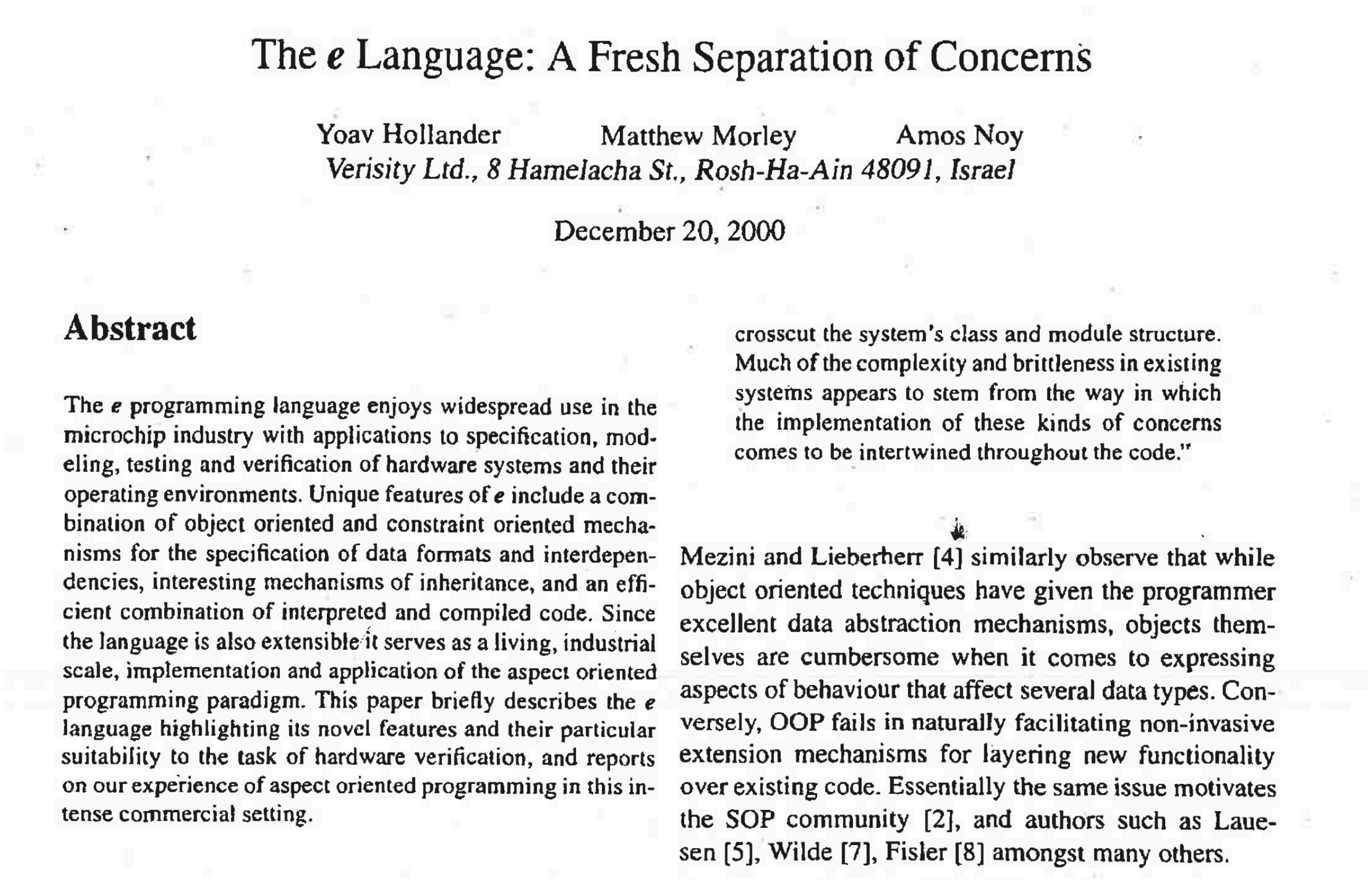


The key is the **Verification Plan!**

Complete SN Verification Process



The key is the **Verification Plan!**



The *e* Language: A Fresh Separation of Concerns

Yoav Hollander Matthew Morley Amos Noy
Verisity Ltd., 8 Hamelacha St., Rosh-Ha-Ain 48091, Israel

December 20, 2000

Abstract

The *e* programming language enjoys widespread use in the microchip industry with applications to specification, modeling, testing and verification of hardware systems and their operating environments. Unique features of *e* include a combination of object oriented and constraint oriented mechanisms for the specification of data formats and interdependencies, interesting mechanisms of inheritance, and an efficient combination of interpreted and compiled code. Since the language is also extensible it serves as a living, industrial scale, implementation and application of the aspect oriented programming paradigm. This paper briefly describes the *e* language highlighting its novel features and their particular suitability to the task of hardware verification, and reports on our experience of aspect oriented programming in this intense commercial setting.

crosscut the system's class and module structure. Much of the complexity and brittleness in existing systems appears to stem from the way in which the implementation of these kinds of concerns comes to be intertwined throughout the code."

Mezini and Lieberherr [4] similarly observe that while object oriented techniques have given the programmer excellent data abstraction mechanisms, objects themselves are cumbersome when it comes to expressing aspects of behaviour that affect several data types. Conversely, OOP fails in naturally facilitating non-invasive extension mechanisms for layering new functionality over existing code. Essentially the same issue motivates the SOP community [2], and authors such as Lauesen [5], Wilde [7], Fisler [8] amongst many others.

Hollander, Y.; Morley, M.; Noy, A., "The e language: a fresh separation of concerns," in *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 38. Proceedings* , vol., no., pp.41-50, 2001
DOI: 10.1109/TOOLS.2001.911754

Basics of the “e” Language

An “e” component is a representation of the “rest of the world” as seen from an interface of the DUV.

- High-level language for writing verification environments:
  - test benches
  - coverage models
  - test generators and checkers
- “e” supports:
  - Modular aspect-oriented design
  - high-level data types
  - pseudo-random constrained-based data generation
  - events
  - high-level checking
  - checking of basic timing properties

Aspect-oriented Programming

- AOP is the “next step up” from object-oriented programming.
  - Testcases have specific purposes:
    - Does the parity check on packets work?
    - Are the timing properties of the transmission protocol satisfied?
  - Both are different concerns: They are orthogonal!
  - Two aspects of same application DUV.

Aspect-oriented Programming

- AOP is the “next step up” from object-oriented programming.
  - Testcases have specific purposes:
    - Does the parity check on packets work?
    - Are the timing properties of the transmission protocol satisfied?
  - Both are different concerns: They are orthogonal!
  - Two aspects of same application DUV.

- AOP provides mechanisms to separate these two concerns into separate aspects of the verification environment.
- Well-defined techniques for adding declarations, inserting or replacing code from the outside of a class, without editing the original class.

File Format

- An “e” code segment is enclosed with a begin-code marker <‘ and an end-code marker ‘>.
  - Both the begin-code marker and the end-code markers must be placed at the beginning of a line (left-most), with no other text on that same line.
- Example “e” code segment:

```
<‘
import cpu_test_env;
‘>
```

- Several e code segments can appear in one file, each segment consists of one or more statements.

Comments

“e” files begin with a comment!

- This comment ends when first begin-code marker <‘ is found.
- Comments in code segments can be marked with — or //.

- Use end-code ‘> and begin-code <‘ markers to write several consecutive lines of comment in the middle of code segments.

Why is this a good idea for a verification language?

Comments

"e" files begin with a comment!

- This comment ends when first begin-code marker `<'` is found.
- Comments in code segments can be marked with `--` or `//`.
- Use end-code `'>` and begin-code `<'` markers to write several consecutive lines of comment in the middle of code segments.

Why is this a good idea for a verification language?

Syntactic Elements

- **Statements** are top-level constructs.
  - Valid within <‘ and ‘> markers.
  - Statements always end with a semicolon “;”!

Syntactic Elements

- **Statements** are top-level constructs.
  - Valid within <‘ and ‘> markers.
  - Statements always end with a semicolon “;”!
- **Struct** members are second-level constructs.
  - Valid only within a `struct` definition.
  - They are associated with dynamic constructs of a testbench e.g. stimulus.
  - (There are also Units which are associated with testbench constructs such as drivers/checkers/scoreboards. They exist for the duration of the simulation.)

Syntactic Elements

- **Statements** are top-level constructs.
  - Valid within <' and '> markers.
  - Statements always end with a semicolon ";"!
- **Struct** members are second-level constructs.
  - Valid only within a `struct` definition.
  - They are associated with dynamic constructs of a testbench e.g. stimulus.
  - (There are also Units which are associated with testbench constructs such as drivers/checkers/scoreboards. They exist for the duration of the simulation.)
- **Actions** are third-level constructs.
  - Valid only when associated with a struct member, such as a method or an event.

Syntactic Elements

- **Statements** are top-level constructs.
  - Valid within <‘ and ‘> markers.
  - Statements always end with a semicolon “;”!
- **Struct** members are second-level constructs.
  - Valid only within a `struct` definition.
  - They are associated with dynamic constructs of a testbench e.g. stimulus.
  - (There are also Units which are associated with testbench constructs such as drivers/checkers/scoreboards. They exist for the duration of the simulation.)
- **Actions** are third-level constructs.
  - Valid only when associated with a struct member, such as a method or an event.
- **Expressions** are lower-level constructs.
  - Can be used only within another “e” construct.

Key Statement Types

`struct`	Defines a new data structure.
`unit`	Defines a new unit.
`type`	Defines an enumerated type or subtype.
`extend`	Extends a previously defined struct/type.
`define`	Extends the language with a definition. `define OFFSET 5;`
`import`	must be first (after defines), otherwise the order of statements is not critical.

... (more, see on line documentation)

Structs vs Units

- **Structs** are the most basic building blocks in “e”.
  - Used to keep data and operations together.
    - packets, instructions, frames
  - Can be created at run-time, i.e. they are dynamic.
  - Data in structs can be generated on-the-fly.

Structs vs Units

- **Structs** are the most basic building blocks in “e”.
  - Used to keep data and operations together.
    - packets, instructions, frames
  - Can be created at run-time, i.e. they are dynamic.
  - Data in structs can be generated on-the-fly.
- **Units** are a special kind of struct.
  - Units are static! Can be generated during test phase only.
  - Allow mapping to HDL path. (Best way to connect to DUV.)

Structs vs Units

- **Structs** are the most basic building blocks in “e”.
  - Used to keep data and operations together.
    - packets, instructions, frames
  - Can be created at run-time, i.e. they are dynamic.
  - Data in structs can be generated on-the-fly.
- **Units** are a special kind of struct.
  - Units are static! Can be generated during test phase only.
  - Allow mapping to HDL path. (Best way to connect to DUV.)
- **Units** are used for generators/checkers/monitors, bus functional models (**BFM**s), self-checking structures, overall testbench.
  - **BFM**s package all bus functional procedures of an interface, i.e. all transactions supported by the interface.
  - The transactions are abstracted from a physical-level interface to a procedural interface.
  - **BFM**s can be used to generate stimulus as well as to check the DUV response.

Structs and Struct Members

- Members are 2nd-level constructs: Valid only within a struct definition.
  - A simple struct for packets to be used in comms protocol:

```
type packet_kind: [atm, eth];
struct packet {
  len: uint;
  keep len < 256;
  kind: packet_kind;
};
```

`keep`: Specifies rules for constraints to influence data generation.

Structs and Struct Members

- Members are 2nd-level constructs: Valid only within a struct definition.
  - A simple struct for packets to be used in comms protocol:

```
type packet_kind: [atm, eth];
struct packet {
  len: uint;
  keep len < 256;
  kind: packet_kind;
};
```

`keep`: Specifies rules for constraints to influence data generation.

  - Another example struct for transactions:

```
struct transaction {
  address: uint;
  data: list of uint;
  transform(multiple:uint) is empty;
};
```

Struct Members

- **Fields:** Define data with an explicit type to be a member of the enclosing `struct`.

Struct Members

- **Fields:** Define data with an explicit type to be a member of the enclosing `struct`.
- **Method:** Define an operational procedure that can manipulate fields of the enclosing `struct` and access run-time values in DUV.

Struct Members

- **Fields:** Define data with an explicit type to be a member of the enclosing **`struct`**.
- **Method:** Define an operational procedure that can manipulate fields of the enclosing **`struct`** and access run-time values in DUV.
- **Subtype declaration:** Defines an instance of a parent **`struct`** in which specific members have particular values or behaviour.
  - Use **`when`** for conditional constraints on possible values of a field.

Struct Members

- **Fields:** Define data with an explicit type to be a member of the enclosing `struct`.
- **Method:** Define an operational procedure that can manipulate fields of the enclosing `struct` and access run-time values in DUV.
- **Subtype declaration:** Defines an instance of a parent `struct` in which specific members have particular values or behaviour.
  - Use `when` for conditional constraints on possible values of a field.
- **Constraint declaration:** Influences distribution of values generated for data entries and the order in which values are generated, e.g. `keep len < 256;`

Struct Members

- **Fields:** Define data with an explicit type to be a member of the enclosing `struct`.
- **Method:** Define an operational procedure that can manipulate fields of the enclosing `struct` and access run-time values in DUV.
- **Subtype declaration:** Defines an instance of a parent `struct` in which specific members have particular values or behaviour.
  - Use `when` for conditional constraints on possible values of a field.
- **Constraint declaration:** Influences distribution of values generated for data entries and the order in which values are generated, e.g. `keep len < 256;`
- **Coverage declaration:** Defines functional verification goals and collects data on how well the testbench is meeting these goals.

  `cover` event-type `is` coverage-item-definition;

Struct Members

- **Fields:** Define data with an explicit type to be a member of the enclosing `struct`.
- **Method:** Define an operational procedure that can manipulate fields of the enclosing `struct` and access run-time values in DUV.
- **Subtype declaration:** Defines an instance of a parent `struct` in which specific members have particular values or behaviour.
  - Use `when` for conditional constraints on possible values of a field.
- **Constraint declaration:** Influences distribution of values generated for data entries and the order in which values are generated, e.g. `keep len < 256;`
- **Coverage declaration:** Defines functional verification goals and collects data on how well the testbench is meeting these goals.

  `cover` event-type `is` coverage-item-definition;
- **Temporal declaration:** Defines "e" events and their associated actions, e.g. `event`

```
type PCICommandType: [ IO_READ=0x2, IO_WRITE=0x3,
                        MEM_READ=0x6, MEM_WRITE=0x7 ];
struct pci_transaction like transaction {
    command: PCICommandType;
    keep soft data.size() in [0..7];
    dual_address: bool;
    when dual_address pci_transaction {
        address2: uint;
    };
    bus_id: uint;
    event initiate;
    on initiate {
      out("An event was initiated on bus ”, bus_id);
    };
    cover initiate is {
        item command;
    };
    transform(multiple: uint) is only {
        address = address * multiple;
    };
};
```

```
type P                    pe: [ IO_READ=0x2, IO_WRITE=0x3,
                               MEM_READ=0           WRITE=0x7 ];
struct pci_transaction like tran
    command: PCICommandType;
    keep soft data.size()                ;
    dual_address: bool;
    when dual_address pci_transaction {
        address2: u
    };
    bus_id: uint;
    event initiate;
    on initiate {
      out("An event was initiated on bus ", bus_id);
    };
    cover initiate is {
        item command;
    };
    transform(multiple: uint) is only {
        address = address * multiple;
    };
};
```

struct

field

field

field

```
type PCICommandType: [ IO_READ=0x2, IO_WRITE=0x3,
                       MEM_READ=0x6, MEM_WRITE=0x7 ];
struct pci_transaction like transaction {
    command: PCICommandType;
    keep soft data.size() in [0..7];
    dual_address: bool;
    when dual_address pci_transaction {
        address2: uint;
    };
    bu
    ev
    or
                                          ous_id);
    };
    co
        item command;
    };
    transform(multiple: uint) is only {
        address = address * multiple;
    };
};
```

constraint declaration

```
struct transaction {
  address: uint;
  data: list of uint;
  transform(multiple:uint) is empty;
};
```

```
type PCICommandType: [ IO_READ=0x2, IO_WRITE=0x3,
                       MEM_READ=0x6, MEM_WRITE=0x7 ];
struct pci_transaction like transaction {
    command: PCICommandType;
    keep soft data.size() in [0..7];
    dual_address: bool;
    when dual_address pci_transaction {
        address2: uint;

                 nt;
                 iate;
    on initiate {
      out("An event was initiated on bus ", bus_id);
    };
    cover initiate is {
        item command;
    };
    transform(multiple: uint) is only {
        address = address * multiple;
    };
};
```

type definition

subtype definition

```
type PCICommandType: [ IO_READ=0x2, IO_WRITE=0x3,
                       MEM_READ=0x6, MEM_WRITE=0x7 ];
struct pci_transaction like transaction {
    command: PCICommandType;
    keep soft data.size() in [0..7];
    dual_address: bool;
    when dual_address pci_transaction {
        address2: uint;
    };
    bus_id: uint;
    event initiate;
    on initiate {
      out("An event was initiated on bus ", bus_id);
    };
    cover initiate is {
        item command;
    };
    transform(multiple: uint) is only {
        address = address * multiple;
    };
};
```

method

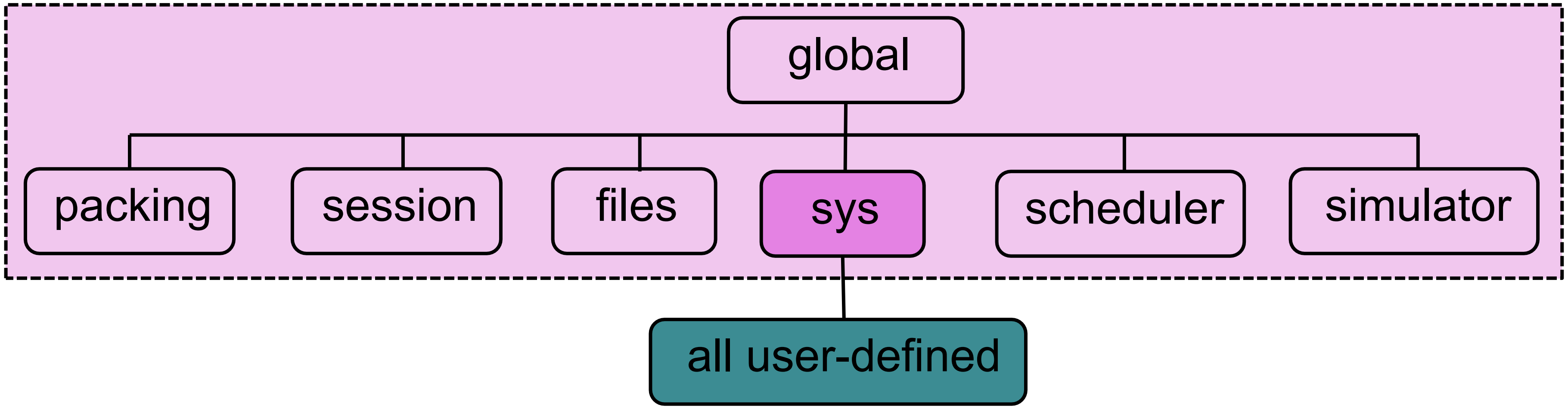
```
type PCICommandType: [ IO_READ=0x2, IO_WRITE=0x3,
                       MEM_READ=0x6, MEM_WRITE=0x7 ];
struct pci_transaction like transaction {
    command: PCICommandType;
    keep soft data.size() in [0..7];
    dual_address: bool;
    when dual_address pci_transaction {
        address2: uint;
    };
    bus_id: uint;
    event initiate;
    on initiate {
      out("An event was initiated on bus ", bus_id);
    };
    cover initiate is {
        item command;
    };
    transform(multiple: uint) is only {
        address = address * multiple;
    };
};
```

event

coverage declaration

Predefined structs

- An “e” environment contains by default a number of predefined structs (and of course some user-defined ones).



- The system struct **`sys`** is the root for user-defined structs.
  - Must instantiate user-defined structs under **`sys`**.
  - Contents of **`sys`** can be viewed via SN GUI.
- Similar to `main` in C.

Instantiation under **`sys`**

Every user-defined struct (including units) must be instantiated as a (sub)field of **`sys`**, e.g.

```
struct packet {
   address : uint (bits : 2);
   payload : uint (bytes : 64);
 };

unit router_bfm {
    packets : list of packet;
};

extend sys {
    router : router_bfm is instance;
};
```

Generation with SN

- Offline (prior to sim, i.e. in *Generate* phase):
  - Use Generate or Test command
    - Test calls Generate command!
  - Recursively generates **everything** under `sys`.
  - BEWARE: Can consume a lot of memory!

- Online (during sim):
  - Allows to dynamically generate values based on DUV state.
  - Use `gen` action.
    - `gen` *gen-item* `[keeping {...}]`

Specifying and Using Constraints

keep *constraint-bool-expr*;

- where *constraint-bool-expr* is a simple or compound Boolean expression.

- State restrictions on the values generated for fields in the struct.
- Describe the required relationships between field values and other struct items.

```
struct packet {
  kind : [tx, rx];
  len : uint;
  keep kind==tx => len==16;
--keep kind!=tx or len==16; exactly the same effect
--when tx packet { keep len==16; }; exactly same effect
};
```

- Hard constraints are applied when the enclosing struct is generated. If constraints can't be met, the generator issues a **constraint contradiction message.**

Generation Order

- Generation order is important:
  - It influences the distribution of values!

```
struct packet {
  kind : [tx, rx];
  len : uint;
  keep len>16 => kind==rx;
};
```

  - If `kind` is generated first, `kind` is `tx` about half the time because there are only two legal values for `kind`.
  - If `len` is generated first, the distribution is different.
  - Consider using: `keep gen (kind) before (len);`

Using Soft Constraints

- Using `keep soft` (e.g. to set default values) and `select`:

```
struct transaction {
  address : uint;
  keep soft address == select {
    10: [0..49];
    60: 50;
    30: [51..99];
  };
};
```

- NOTE: Soft constraints can be overridden by hard constraints!

Using Soft Constraints

- Using keep soft (e.g. to set default values) and select:

```
struct transaction {
  address : uint;
  keep soft address == select {
    10: [0..49];
    60: 50;
    30: [51..99];
  };
};
```

- NOTE: Soft constraints can be overridden by hard constraints!

Does not need to add up to 100!

```
extend instruction {
  keep soft op_code == select {
    40: [ADD, ADDI, SUB, SUBI];
    20: [XOR, XORI];
    10: [JMP, CALL, RET, NOP];
  };
};
```

- In practice, getting the weights/bias right (for **coverage closure**) requires significant engineering skill.

We have now covered

- Basics of the “e” verification language and some features of SN.
  - If you take this unit with coursework, you should be registered for the ***Specman Fundamentals for Block-Level Environment Developers*** *online training course, which introduces you to SN and e in more detail and provides you with exercises.*
- In Part 2 we will explore more advanced features of the e language and also how SN synchronizes with the simulator.