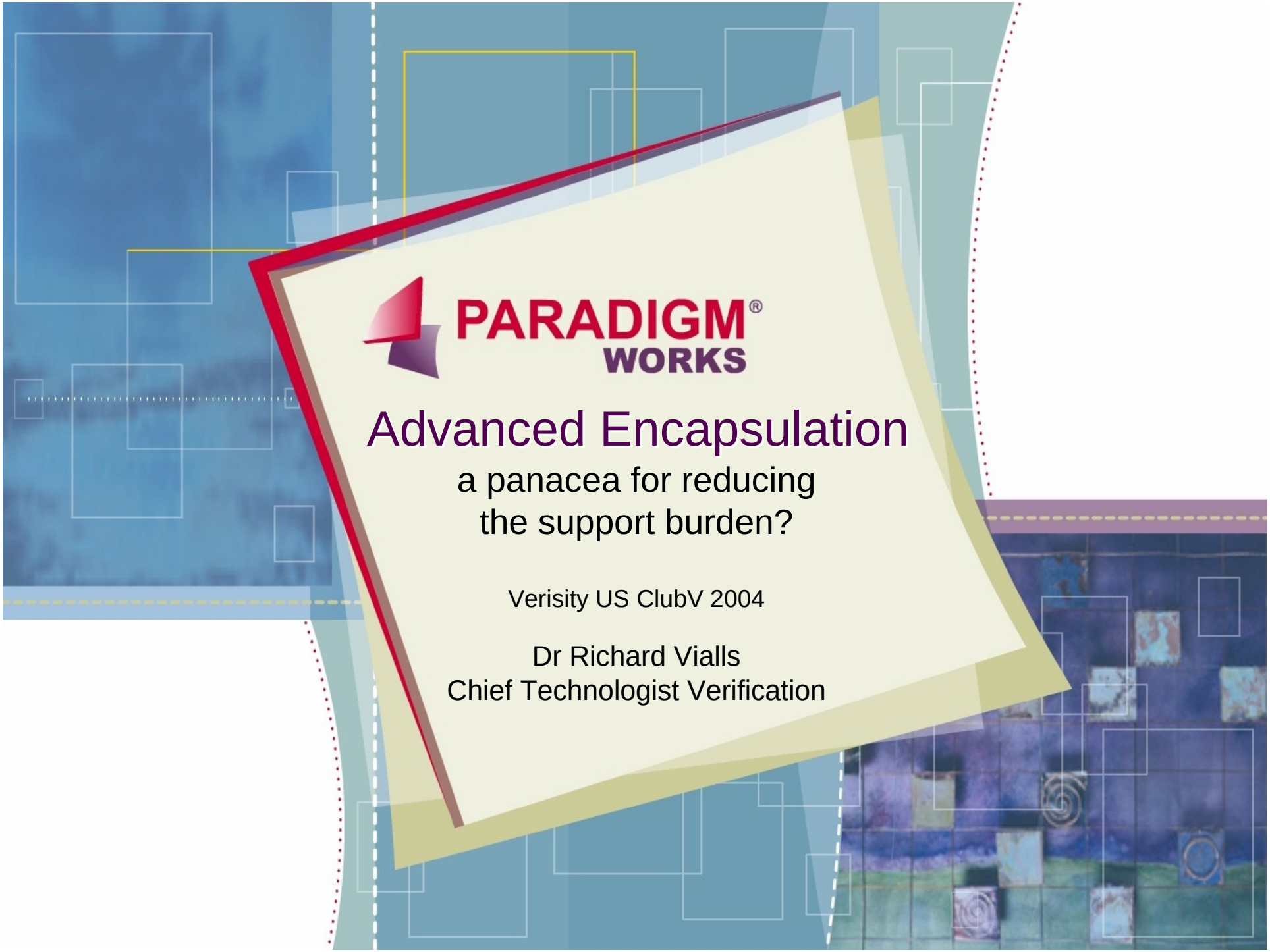

Advanced Encapsulation

a panacea for reducing the support burden?

Verisity US ClubV 2004

Dr Richard Vialls
Chief Technologist Verification

PARADIGM® WORKS

Introduction

- Introduction of ***e***RM has resulted in large number of ***e***VCs being developed:
  - In-house
  - Commercial off-the-shelf
- Verification IP reuse means we have to address the issue of support
- Perhaps biggest current support issue is how to minimise:

USER ERRORS

PARADIGM® WORKS

Introduction

- We will cover:
  - What is a user error?
  - Why are user errors important?
  - Typical user errors
  - How to minimise user errors:
    - Now – encapsulation, other ‘tips and tricks’
    - The future – advanced encapsulation?

PARADIGM® WORKS

What is a ‘user error’?

- ***e***VCs are often complex products
- User API is typically:
  - Large
  - Difficult to document
- User is often:
  - Inexperienced?
  - New to Specman?
  - New to ***e***VCs/***e***RM***?***
- ***Result: User Errors!***

PARADIGM® WORKS

What is a ‘user error’?

- Takes user some time to gain a ‘feel’ for an ***e***VC
- Learning curve typically involves refining understanding of ***e***VC structure
- Even with the best documentation, user may start with misconceptions
  - Initial misconceptions will be gross
  - Later misconceptions will be more subtle

PARADIGM® WORKS

Why are user errors important?

- Most common during early ramp-up
  - Rates typically tail off during ramp-up
  - Assuming good product, more common than ***e***VC failures
- High incidence of user errors during evaluation periods!!!!!
- Ease of use of ***e***VC is largely measured by frequency and severity of user errors
- High impact on technology uptake!
- Need to differentiate user errors from other errors

PARADIGM® WORKS

Typical user errors

- Illegal constraint of field – contradictions!
- Illegal constraint of ‘read-only’ field
- Illegal write to ‘read-only’ field
- Illegal extension of method
- Illegal call of method
- Illegal modification of method parameter in user extension
- etc

PARADIGM® WORKS

Example: illegal constraint of ‘read-only’ field

```
-- eVC Code
unit some_env_u {

  num_squares : uint;

  num_edges : uint;
    keep num_edges == num_squares * 4;

};

-- User Code
extend some_env_u {
  keep num_edges == 8;
};
```

PARADIGM® WORKS

The result…

```
*** Error: Contradiction:
 A contradiction has occurred when generating some_env_u-@0.num_edges :
    Previous constraints reduced its range of possible values,
     then the following constraint contradicted these values:
       keep num_edges == num_squares * 4;       at line 9 in @test
     Reduced: some_env_u-@0.num_edges into []
       Using: some_env_u-@0.num_squares == [1077154777]
 To see details, reload and rerun with "collect gen"
```

PARADIGM® WORKS

Solution 1: documentation

- Need to carefully document API so that user understands correct usage. E.g.:

```
unit some_env_u {

  -- This field controls the number of squares
  -- in the environment. The user should constrain
  -- this field.
  num_squares : uint;

  -- This field indicates the number of edges in the
  -- environment. Its value is automatically constrained
  -- by the eVC and should not be constrained by the user.
  num_edges : uint;
    keep num_edges == num_squares * 4;

};
```

PARADIGM® WORKS

Solution 1: documentation

- Documentation can be auto-extracted to reference manual (using ***e***Doc)
- However…
  - Assumes user reads the manual!!!!!!
- Good documentation is essential, but…
- …good documentation doesn't necessarily prevent user errors
- Hence the term – **RTFM!** Errors:

  *'READ THE ******* MANUAL!'*

PARADIGM® WORKS

The requirement

- Need automated (preferably load-time) solution to inform user of API usage errors
  - Must make clear distinction between API usage errors and other classes of error
  - Must give user sufficient info to debug
  - Should be easy/intuitive to code

PARADIGM® WORKS

Solution 2: encapsulation

- Simple encapsulation added to Specman 4.1
- Provides ability to hide types and struct/unit members
- Three levels of encapsulation: protected, package and private

PARADIGM® WORKS

‘protected’ encapsulation

- Hides struct/unit members from code outside struct/unit

```
unit my_agent_u {
            a : uint;
  protected b : uint;
};

extend sys {
  agent : my_agent_u is instance;
    keep a == 5;  -- this is legal
    keep b == 23; -- this is illegal
};
```

PARADIGM® WORKS

‘protected’ encapsulation

- ‘protected’ is extremely useful tool
- Should be used to hide non-API fields/methods/events/etc. within each struct/unit
- No good for struct members that are used across struct/unit hierarchy
  - Need ‘package’ encapsulation

PARADIGM® WORKS

‘package’ encapsulation

- Introduces concept of encapsulation package
  - a group of files marked as belonging to same package (not the same as ***e***RM package)
- Can be used on types and struct/unit members
- Hides declarations from code outside package

PARADIGM® WORKS

‘package’ encapsulation

```
-- eVC file
package my_evc;

package type my_enum : [A, B];

unit my_agent_u {
          a : uint;
  package b : uint;
};

-- User file
extend my_enum : [C]; -- this is illegal

extend my_agent_u {
  keep a == 5;  -- this is legal
  keep b == 23; -- this is illegal
};
```

PARADIGM® WORKS

‘package’ encapsulation

- Also extremely useful
- Should be used to hide non-API declarations that need to be visible across struct/unit hierarchy

PARADIGM® WORKS

‘private’ encapsulation

- Applies to struct members
- Combines concepts of ‘package’ and ‘protected’
- ‘private’ declarations non visible outside package or outside struct/unit
- ‘private’ typically used extensively
- ‘protected’ typically never used

PARADIGM® WORKS

Other tips and tricks - post_generate() checks

- Usually use hard constraints to limit range of API

```
unit my_agent_u {
  mode : [MODE_A, MODE_B];
  some_control : uint;
};
extend MODE_A my_agent_u {
  keep some_control in [1..3];
};
extend MODE_B my_agent_u {
  keep some_control in [4..6];
};
```

PARADIGM® WORKS

Other tips and tricks - post_generate() checks

- User error leads to contradiction
  - Difficult to debug
    - Especially if result of complex user constraint
- In some cases, can replace with post_generate() check
  - Needs caution – may push problem deeper within ***e***VC

PARADIGM® WORKS

Other tips and tricks - post_generate() checks

```
unit my_agent_u {
  mode : [MODE_A, MODE_B];
  some_control : uint;
};

extend MODE_A my_agent_u {
  post_generate() is also {
    if some_control not in [1..3] {
      error("USER ERROR – in MODE_A, some_control",
            "must be in range 1..3");
    };
  };
};
-- etc.
```

PARADIGM® WORKS

Other tips and tricks - scoreboard hook protection

- Scoreboard hooks make internal monitor data structs visible to the user
  - Potential for user to modify
- Safer to make copy of struct
  - But…comes with a performance penalty

PARADIGM® WORKS

Other tips and tricks - scoreboard hook protection

```
extend my_monitor_u {

  -- This is the scoreboard hook
  packet_done(packet : my_packet_s) is empty;

  private packet_finished(packet : my_packet_s) is {
    ...
    copy_packet = deep_copy(packet);
    packet_done(copy_packet);
  };

};
```

PARADIGM® WORKS

The future?

- What we've seen so far helps…
  - …but it isn't enough
- Simple encapsulation gives us little subtlety
  - A declaration is either visible or invisible
- We want to be able to control ***how*** the user uses the API

PARADIGM® WORKS

Enter ‘Advanced Encapsulation’

- Note… *all that follows is vapourware*!!!
- Basic concept is to define categories of usage for each API construct
- What are the main API constructs?
  - types/structs/units
  - fields
  - events
  - methods
  - method parameters
- What are the possible usage categories for each?

PARADIGM® WORKS

types/structs/units

- types/structs/units
  - declare instance (as field or var)
  - extend
- fields
  - read
  - write (assign)
  - constrain
  - gen/new
- events
  - emit
  - extend
  - use in temporal expression
- methods
  - call
  - start
  - extend
- method parameters
  - read
  - modify

PARADIGM® WORKS

Example syntax

- Exact syntax is less important than concept
- Will use ‘cryptic’ example syntax
  - Define modifier letters for each usage category (similar to unix file permissions)
  - Modifiers are appended to current encapsulation syntax
  - Each modifier specifies an allowable category of action

PARADIGM® WORKS

Example: modifier letters

- types/structs/units
  - **I** declare instance (as field or var)
  - **X** extend
- fields
  - **R** read
  - **W** write (assign)
  - **C** constrain
  - **G** gen/new
- events
  - **E** emit
  - **X** extend
  - **U** use in temporal expression
- methods
  - **C** call
  - **S** start
  - **X** extend
- method parameters
  - **R** read
  - **M** modify

PARADIGM® WORKS

Typical examples

- A field that the user can read but cannot modify in any way (API output):

```
extend my_monitor_u {
  package[R] num_packets_so_far : uint;
};
```

- A field that the user can read and constrain but cannot assign/gen (typical API control):

```
extend my_env_u {
  package[RC] num_agents : uint;
};
```

PARADIGM® WORKS

Typical examples

- An event that the user can use, but cannot emit:

```
extend my_monitor_u {
  package[U] event packet_done is ...;
};
```

- A method with a read-only parameter that the user can extend, but cannot call:

```
unit my_monitor_u {
  package[X] packet_done(package[R] packet : my_packet_s)
    is empty;
};
```

PARADIGM® WORKS

Typical examples

- A field that the user is denied any access to:

```
extend my_monitor_u {
  package some_internal_field : uint;
};
```

  - Note that the current encapsulation solution is a sub-set of the advanced encapsulation proposal.

PARADIGM® WORKS

Advantages

- Developer can specify exactly the intended (and hence legal) usage of API
- User errors result in load-time reporting of exact violation. E.g.:

```
Error: cannot constrain field monitor.num_packets_so_far from
   outside package my_evc
```

  - …guides user to look at API documentation
- Misuse of API results in clean error
  - No possibilities of contradictions or unexpected ‘strange’ behaviour

PARADIGM® WORKS

Advantages

- Advanced encapsulation is backwards compatible with current simple encapsulation solution
- Addition of permissions to current encapsulation syntax is intuitive and scalable
- Could be used to give additional guidance to generator, reducing possibilities for contradictions

PARADIGM® WORKS

Disadvantages

- Verisity haven't implemented this yet!

PARADIGM® WORKS

Final thoughts – two-way encapsulation?

- We've looked at controlling API usage by user.
- What about controlling API usage by eVC developer?
  - Allow definition of API design intent looking both ways.
  - Possible further guidance for generator?

PARADIGM® WORKS

Final thoughts – two-way encapsulation?

- Possible example syntax:

```
extend my_env_u {
  -- num_agents field is a user control. It can be
  -- constrained and read by the user, but can only be
  -- read from within the eVC. If involved in a constraint
  -- within the eVC package, it will be treated as a
  -- non-generatable field.
  package[RC,R] num_agents : uint;
};
```