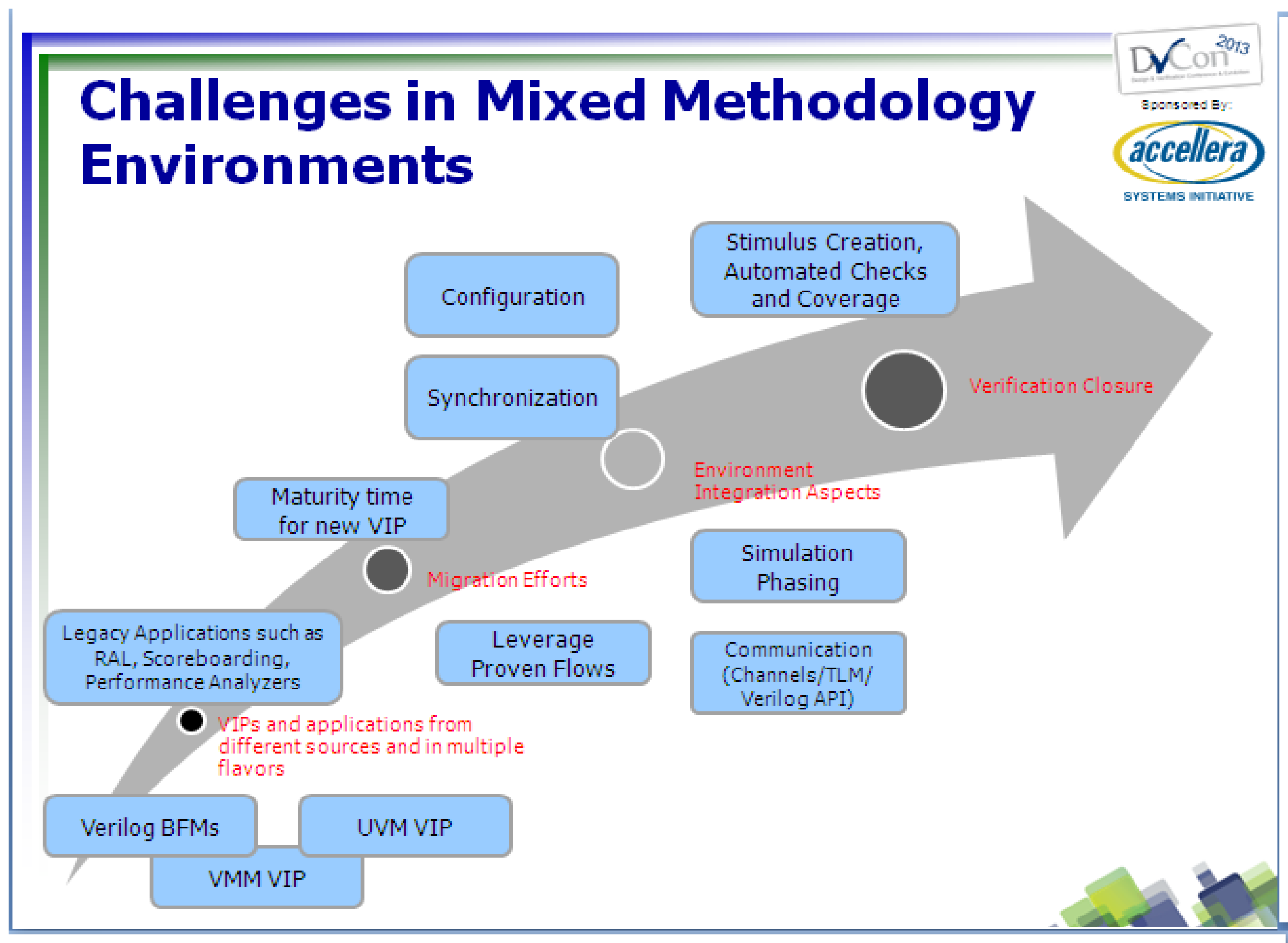
UVM: Conquering Legacy

Santosh Sarma
Wipro Technologies
santosh.sarma@wipro.com

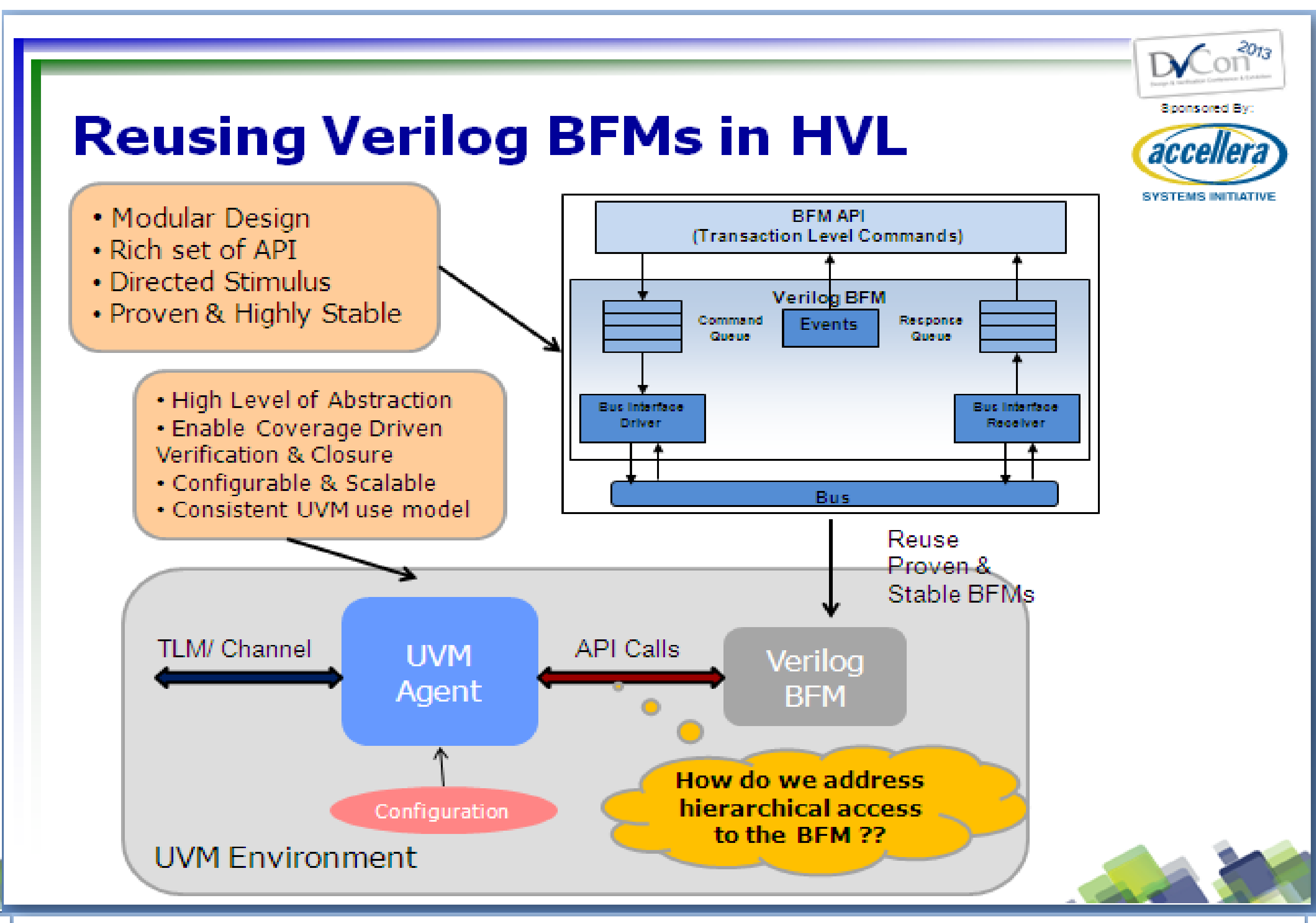
Amit Sharma
Synopsys India Pvt Ltd.
amits@synopsys.com

Adiel Khan
Synopsys, Inc.
adiel@synopsys.com



Challenges in Mixed Methodology Environments

Verilog BFMs
VMM VIP
UVM VIP
VIPs and applications from different sources and in multiple flavors
Legacy Applications such as RAL, Scoreboarding, Performance Analyzers
Migration Efforts
Maturity time for new VIP
Leverage Proven Flows
Configuration
Synchronization
Environment Integration Aspects
Simulation Phasing
Communication (Channels/TLM/ Verilog API)
Stimulus Creation, Automated Checks and Coverage
Verification Closure

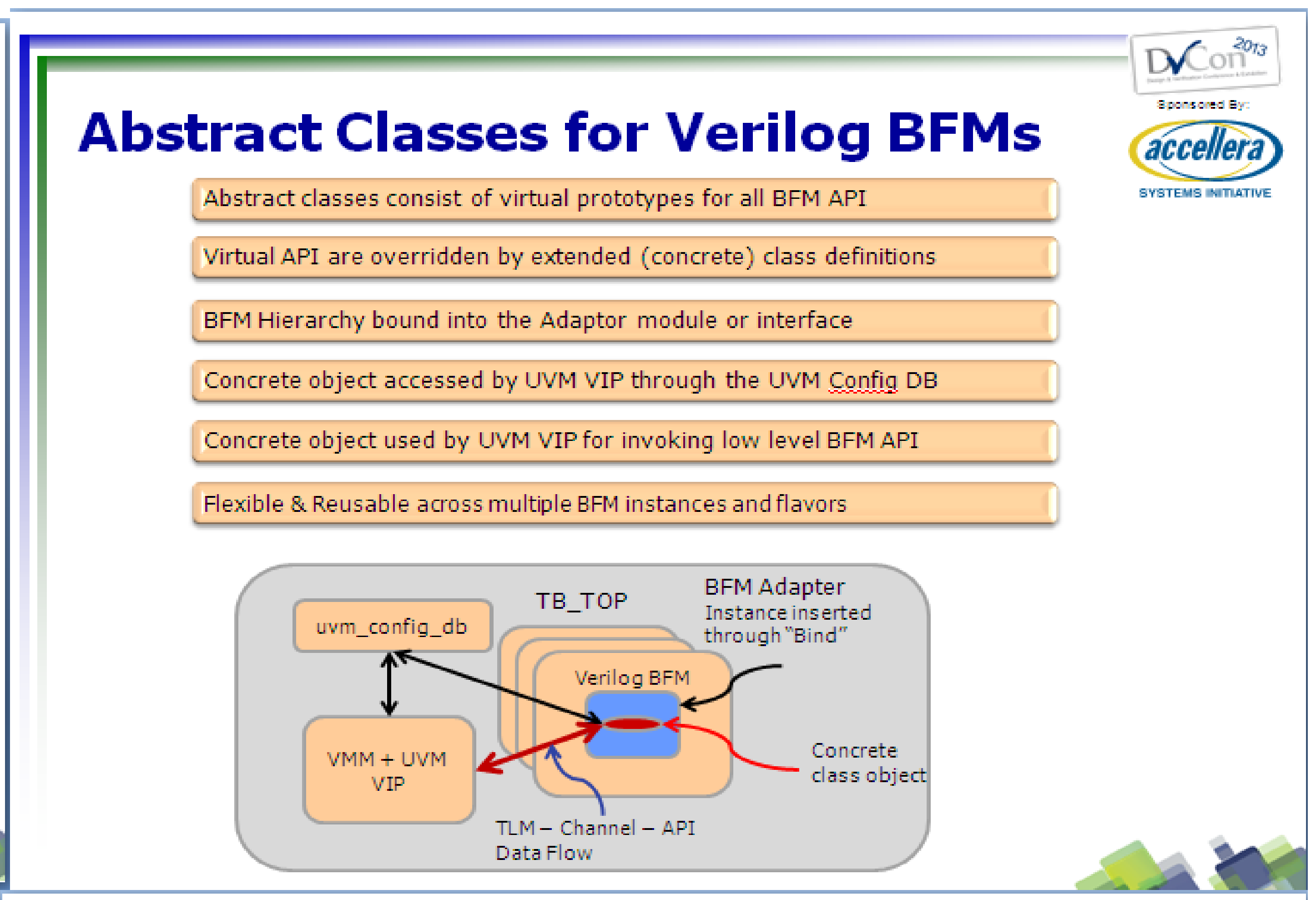


Reusing Verilog BFMs in HVL

- Modular Design
- Rich set of API
- Directed Stimulus
- Proven & Highly Stable

- High Level of Abstraction
- Enable Coverage Driven Verification & Closure
- Configurable & Scalable
- Consistent UVM use model

How do we address hierarchical access to the BFM ??



Abstract Classes for Verilog BFMs

Abstract classes consist of virtual prototypes for all BFM API

Virtual API are overridden by extended (concrete) class definitions

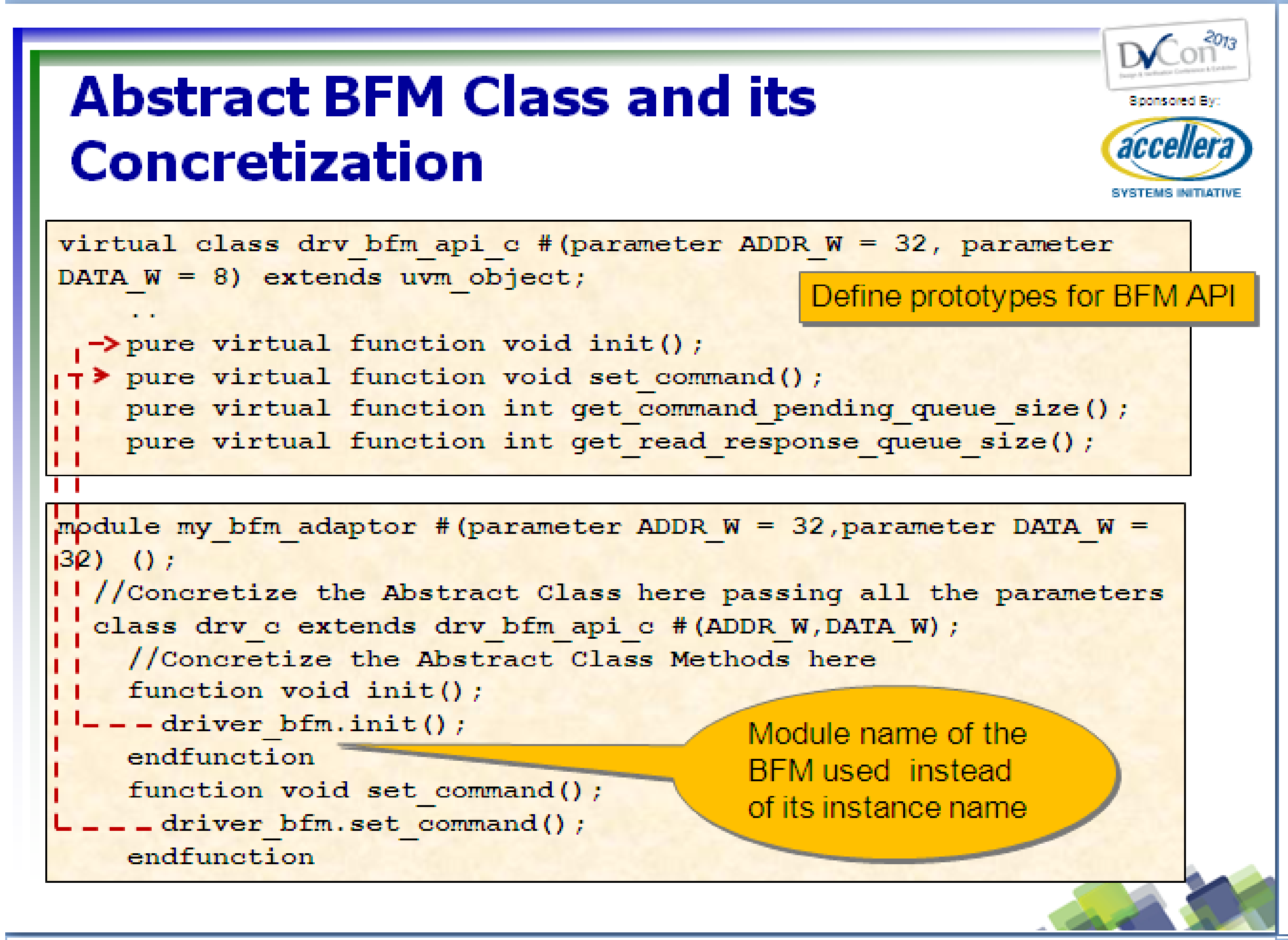
BFM Hierarchy bound into the Adaptor module or interface

Concrete object accessed by UVM VIP through the UVM Config DB

Concrete object used by UVM VIP for invoking low level BFM API

Flexible & Reusable across multiple BFM instances and flavors

uvm_config_db
TB_TOP
BFM Adapter
Instance inserted through "Bind"
Verilog BFM
VMM + UVM VIP
Concrete class object
TLM – Channel – API
Data Flow



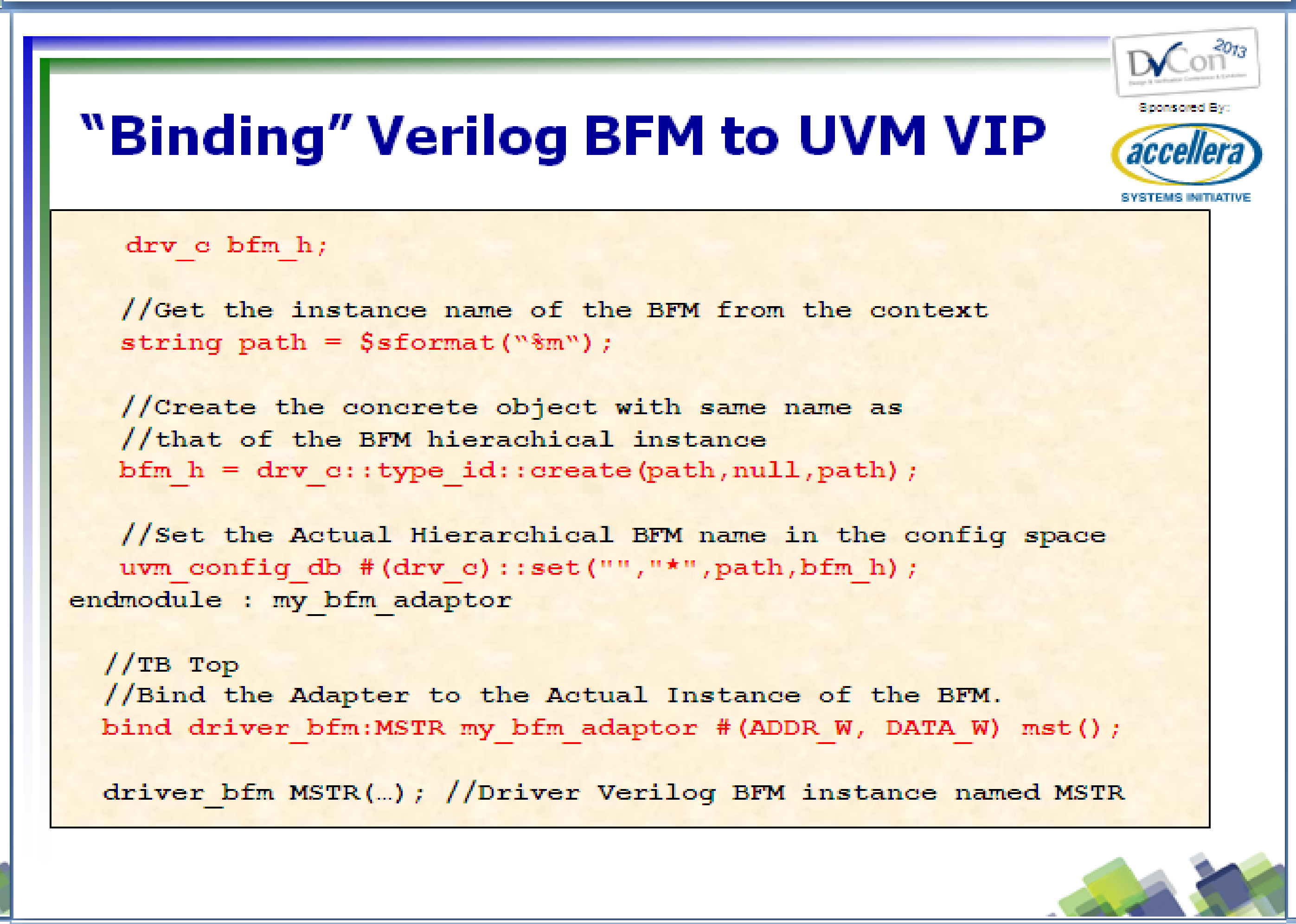
Abstract BFM Class and its Concretization

```
virtual class drv_bfm_api_c #(parameter ADDR_W = 32, parameter
DATA_W = 8) extends uvm_object;
    ..
    pure virtual function void init();
    pure virtual function void set_command();
    pure virtual function int get_command_pending_queue_size();
    pure virtual function int get_read_response_queue_size();
```

Define prototypes for BFM API

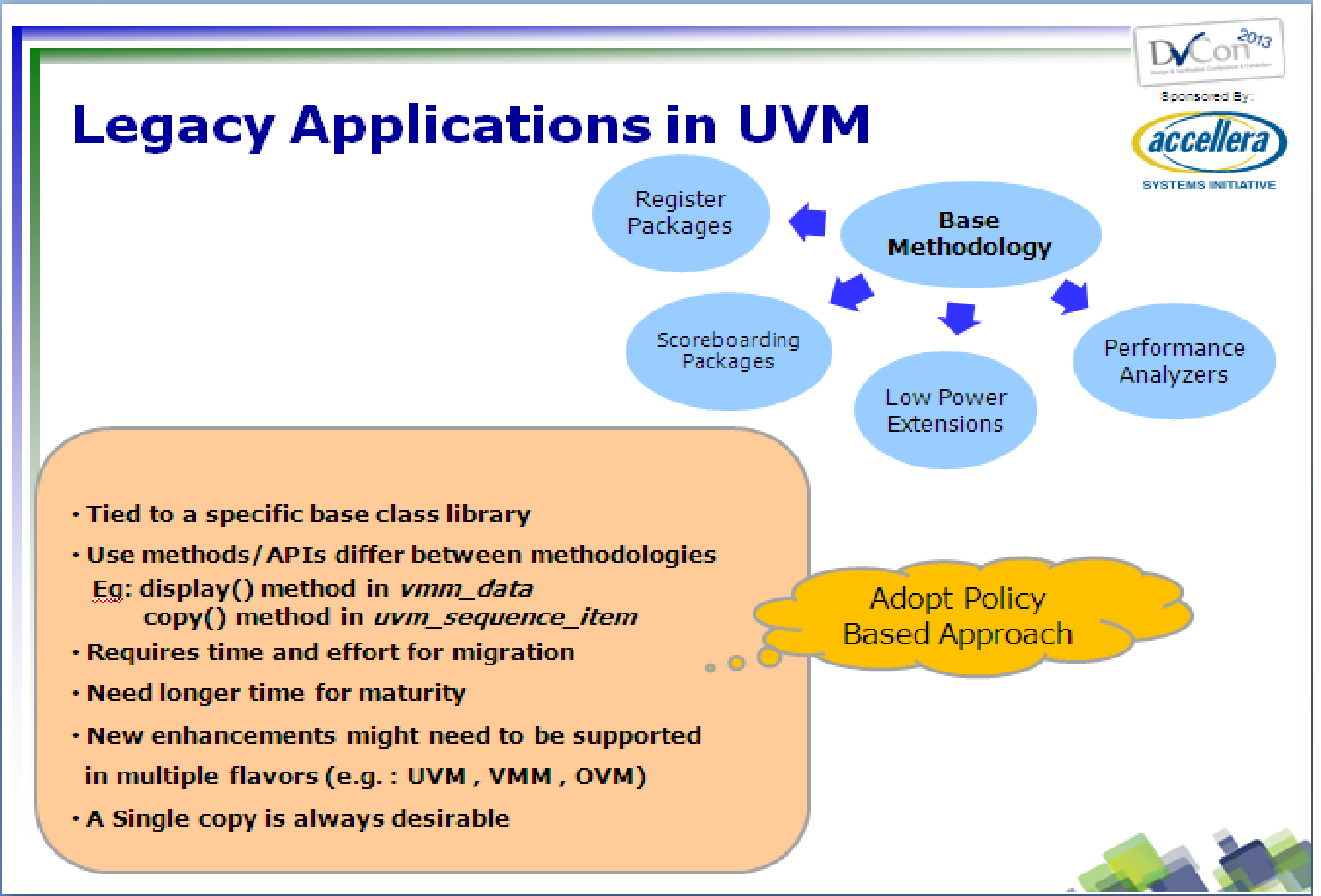
```
module my_bfm_adaptor #(parameter ADDR_W = 32,parameter DATA_W =
32) ();
  //Concretize the Abstract Class here passing all the parameters
  class drv_c extends drv_bfm_api_c #(ADDR_W,DATA_W);
    //Concretize the Abstract Class Methods here
    function void init();
      driver_bfm.init();
    endfunction
    function void set_command();
      driver_bfm.set_command();
    endfunction
```

Module name of the BFM used instead of its instance name



"Binding" Verilog BFM to UVM VIP

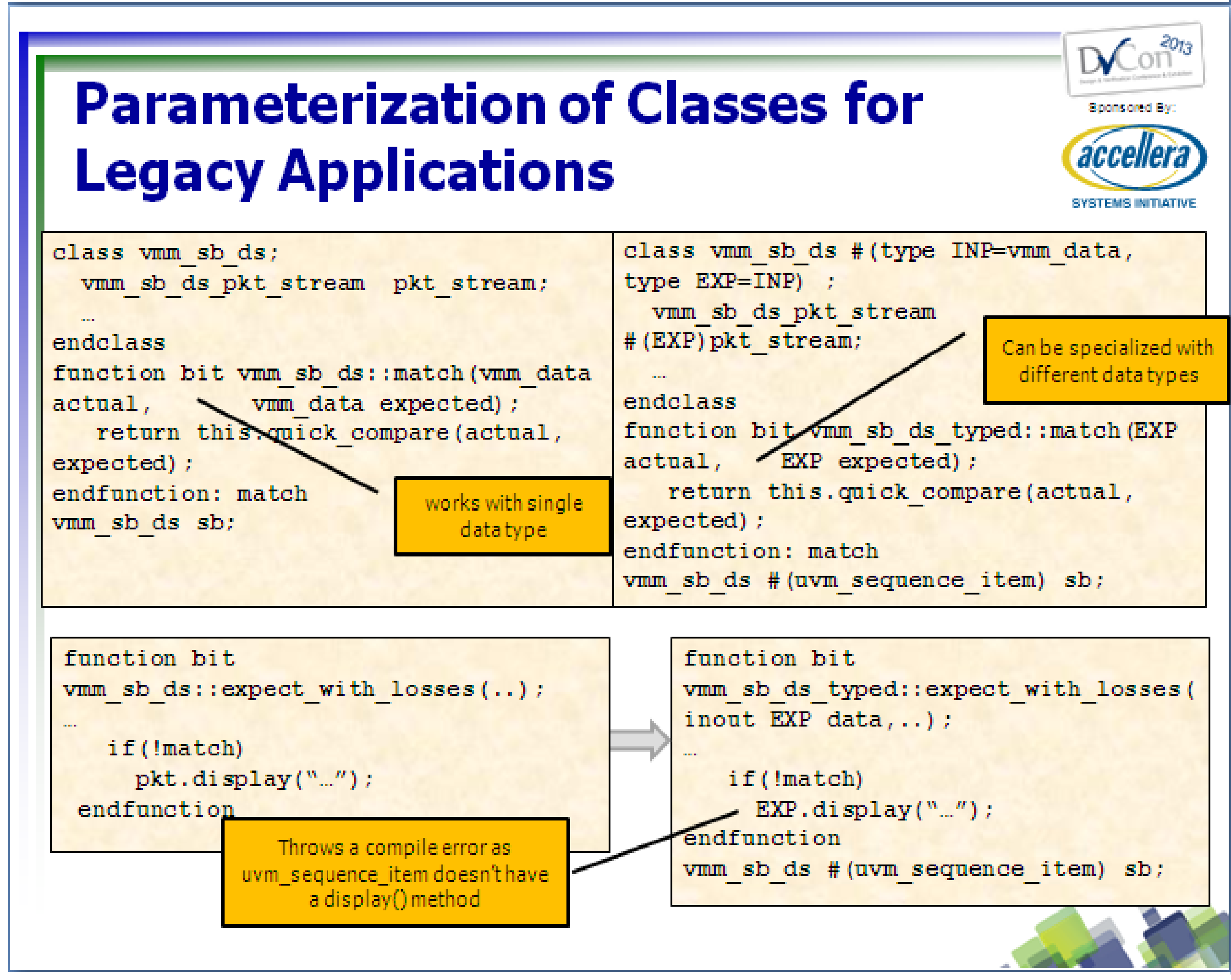
```
    drv_c bfm_h;

    //Get the instance name of the BFM from the context
    string path = $sformat("%m");

    //Create the concrete object with same name as
    //that of the BFM hierachical instance
    bfm_h = drv_c::type_id::create(path,null,path);

    //Set the Actual Hierarchical BFM name in the config space
    uvm_config_db #(drv_c)::set("","*",path,bfm_h);
endmodule : my_bfm_adaptor

   //TB Top
   //Bind the Adapter to the Actual Instance of the BFM.
   bind driver_bfm:MSTR my_bfm_adaptor #(ADDR_W, DATA_W) mst();

   driver_bfm MSTR(…); //Driver Verilog BFM instance named MSTR
```



Legacy Applications in UVM

- Tied to a specific base class library
- Use methods/APIs differ between methodologies
  Eg: display() method in *vmm_data*
  copy() method in *uvm_sequence_item*
- Requires time and effort for migration
- Need longer time for maturity
- New enhancements might need to be supported in multiple flavors (e.g. : UVM , VMM , OVM)
- A Single copy is always desirable

Adopt Policy Based Approach



Parameterization of Classes for Legacy Applications

```
class vmm_sb_ds;
  vmm_sb_ds_pkt_stream  pkt_stream;
  …
endclass
function bit vmm_sb_ds::match(vmm_data
actual,       vmm_data expected);
   return this.quick_compare(actual,
expected);
endfunction: match
vmm_sb_ds sb;
```

works with single data type

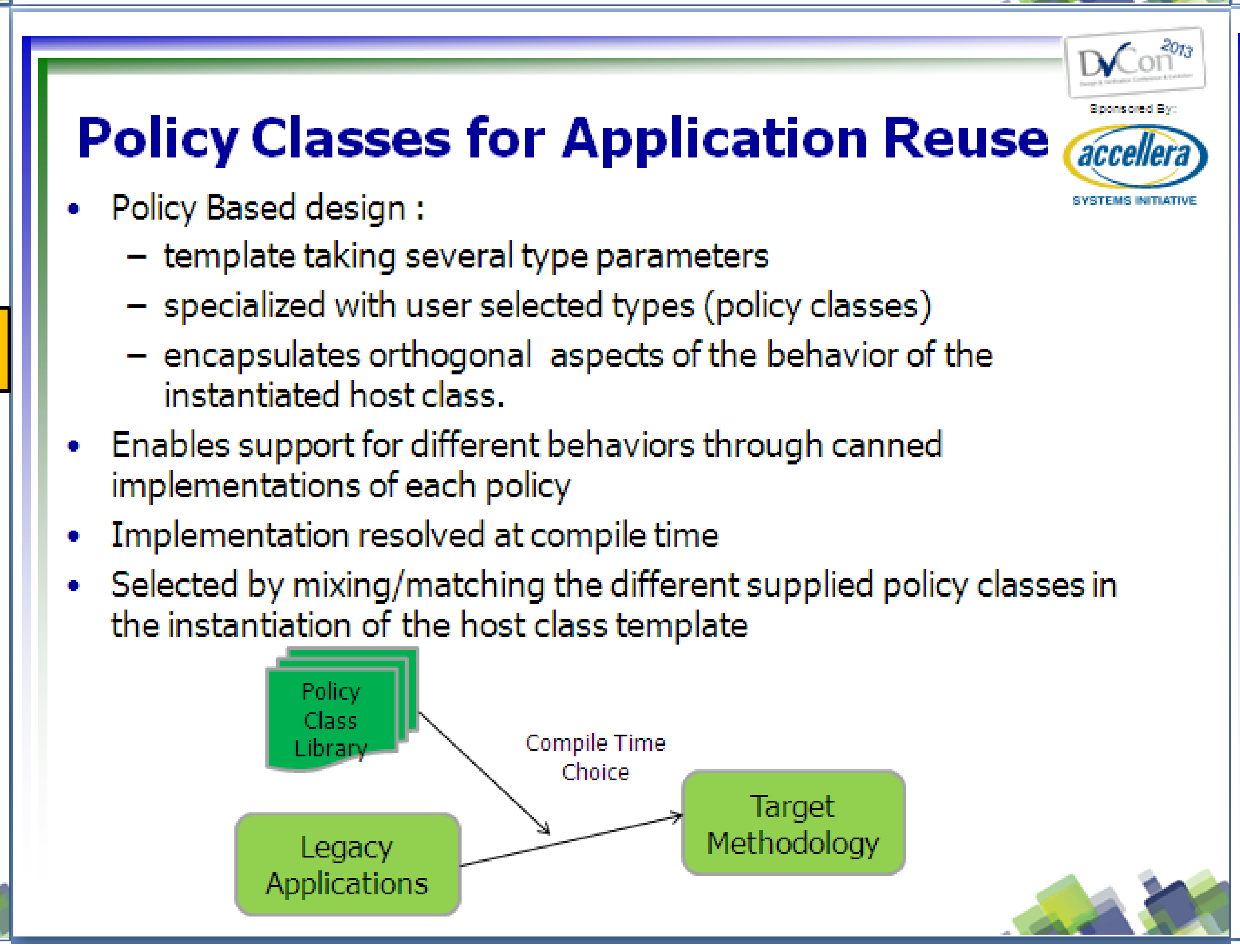
```
class vmm_sb_ds #(type INP=vmm_data,
type EXP=INP) ;
  vmm_sb_ds_pkt_stream
#(EXP)pkt_stream;
  …
endclass
function bit vmm_sb_ds_typed::match(EXP
actual,    EXP expected);
   return this.quick_compare(actual,
expected);
endfunction: match
vmm_sb_ds #(uvm_sequence_item) sb;
```

Can be specialized with different data types

```
function bit
vmm_sb_ds::expect_with_losses(..);
…
   if(!match)
     pkt.display("…");
 endfunction
```

Throws a compile error as uvm_sequence_item doesn't have a display() method

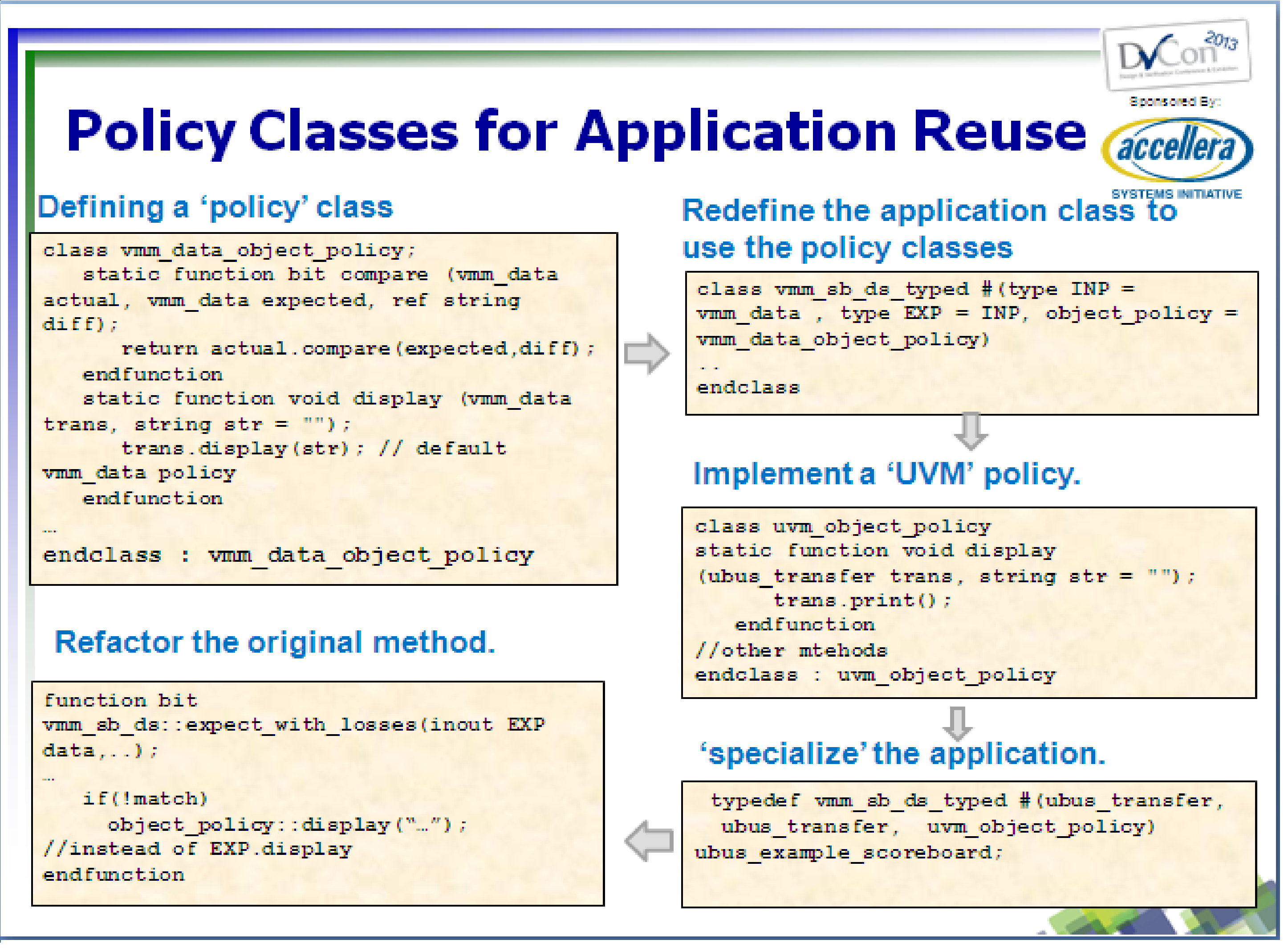
```
function bit
vmm_sb_ds_typed::expect_with_losses(
inout EXP data,..);
…
    if(!match)
      EXP.display("…");
endfunction
vmm_sb_ds #(uvm_sequence_item) sb;
```



Policy Classes for Application Reuse

- Policy Based design :
  - template taking several type parameters
  - specialized with user selected types (policy classes)
  - encapsulates orthogonal aspects of the behavior of the instantiated host class.
- Enables support for different behaviors through canned implementations of each policy
- Implementation resolved at compile time
- Selected by mixing/matching the different supplied policy classes in the instantiation of the host class template

Policy Class Library
Compile Time Choice
Legacy Applications
Target Methodology



Policy Classes for Application Reuse

Defining a 'policy' class

```
class vmm_data_object_policy;
   static function bit compare (vmm_data
actual, vmm_data expected, ref string
diff);
      return actual.compare(expected,diff);
   endfunction
   static function void display (vmm_data
trans, string str = "");
      trans.display(str); // default
vmm_data policy
   endfunction
…
endclass : vmm_data_object_policy
```

Redefine the application class to use the policy classes

```
class vmm_sb_ds_typed #(type INP =
vmm_data , type EXP = INP, object_policy =
vmm_data_object_policy)
..
endclass
```

Implement a 'UVM' policy.

```
class uvm_object_policy
static function void display
(ubus_transfer trans, string str = "");
      trans.print();
   endfunction
//other mtehods
endclass : uvm_object_policy
```

'specialize' the application.

```
 typedef vmm_sb_ds_typed #(ubus_transfer,
  ubus_transfer,  uvm_object_policy)
ubus_example_scoreboard;
```

Refactor the original method.

```
function bit
vmm_sb_ds::expect_with_losses(inout EXP
data,..);
…
   if(!match)
     object_policy::display("…");
//instead of EXP.display
endfunction
```

- **Significant Effort reduction in UVM VIP development leveraging existing Verilog VIP/BFMs with Abstract classes**
- **Abstract BFM classes enable reuse of same base class with different concretized BFM classes**
- **Parameterization and Policy classes enables applications to be interoperable with minimal code refactoring**
- **Policy concept can be used for creating methodology agnostic applications**