2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

A single generated UVM Register Model to handle multiple DUT configurations

Salvatore Marco Rosselli, Giuseppe Falconeri

STMicroelectronics

accellera SYSTEMS INITIATIVE

Modeling Parametric Registers

- Nowadays, parametric designs are a common reality
  - Parameters are flexible
  - They facilitate code reuse
  - The permutation of valid RTL parameters defines the DUT configurations
- Modeling the DUT's registers for dynamic verification is a must
  - Registers represent the way the DUT interacts with the software
  - Verifying registers is one of the most common tasks in RTL verification
- Parameters can modify the structure and behavior of the registers

Modeling Parametric Registers

- Parameters modify
  - The structure of the registers
    - The existence of a field (reserved / non-reserved)
    - The fields' positions (LSB and width)
    - The fields' reset values
  - The behavior of the registers
    - Access rights ("RO", "RW", "WO)
    - Register offset

Arguments of uvm_reg_field::configure()

Arguments of uvm_reg_map::add_reg()

For example, a parameter value can set a register reserved ("RO", always read as zero)

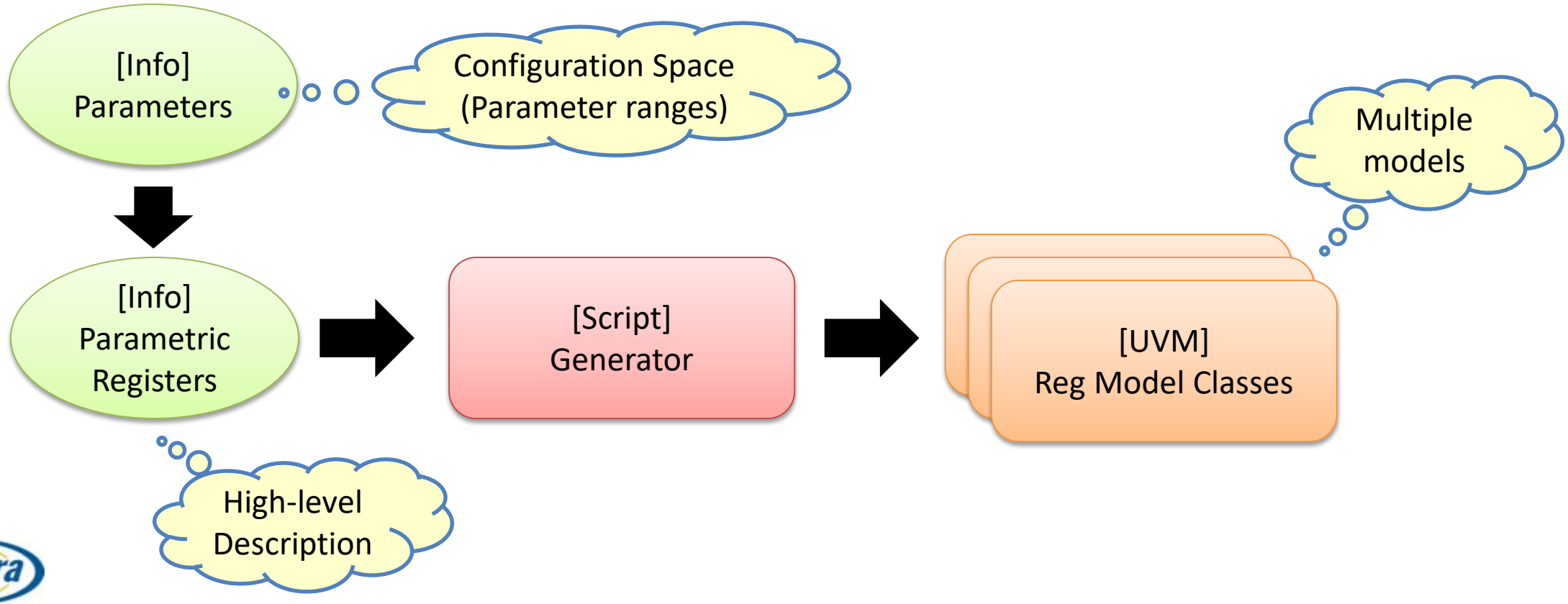
The State Of The Art

- UVM Register Model
  - They are the industry-standard base classes for SystemVerilog

- Techniques and tools exist to describe multiple register configurations
  - Providing high-level descriptions
  - Handling the effects of the parameters on the register model's structure
  - Handling the effects of the parameters on the registers' and fields' behavior
  - Generating UVM-compliant SystemVerilog classes

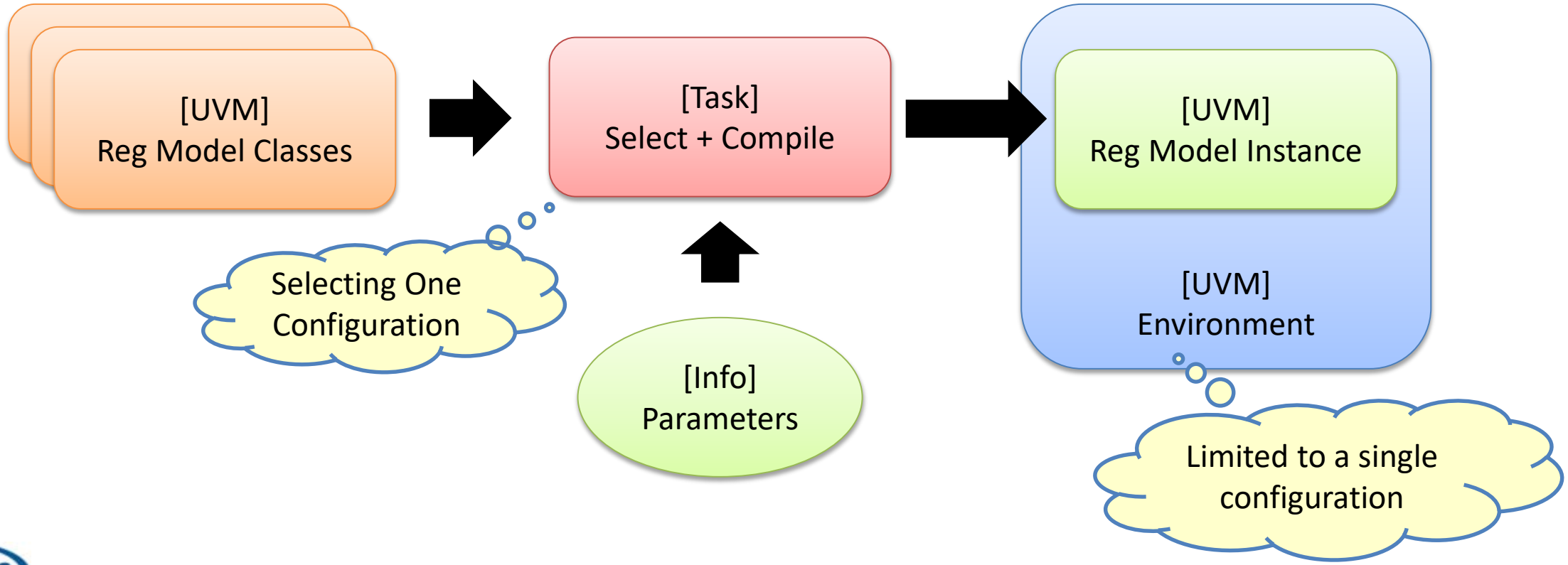
The State Of The Art

- A group of classes is generated for each DUT configuration
  - Each group is a subset of all possible registers, fields and covergroups



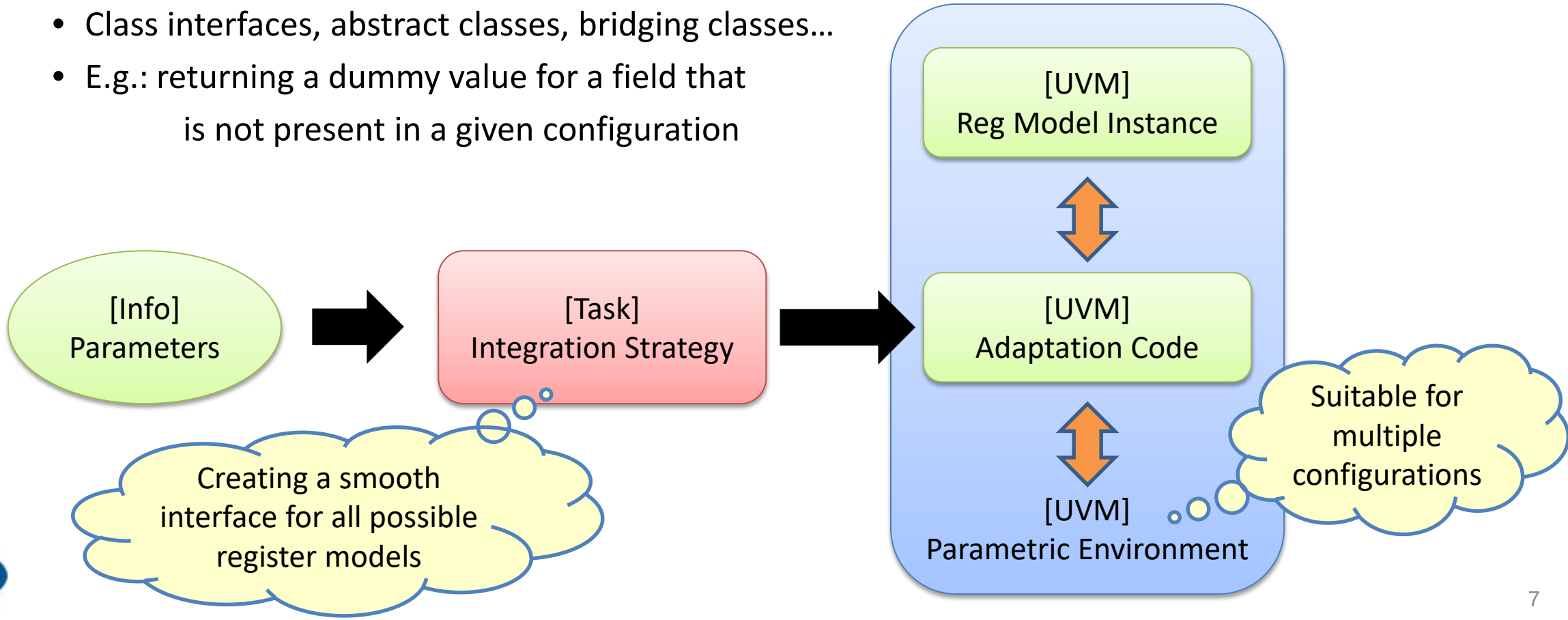
The State Of The Art

- The generated classes are integrated in the verification environment

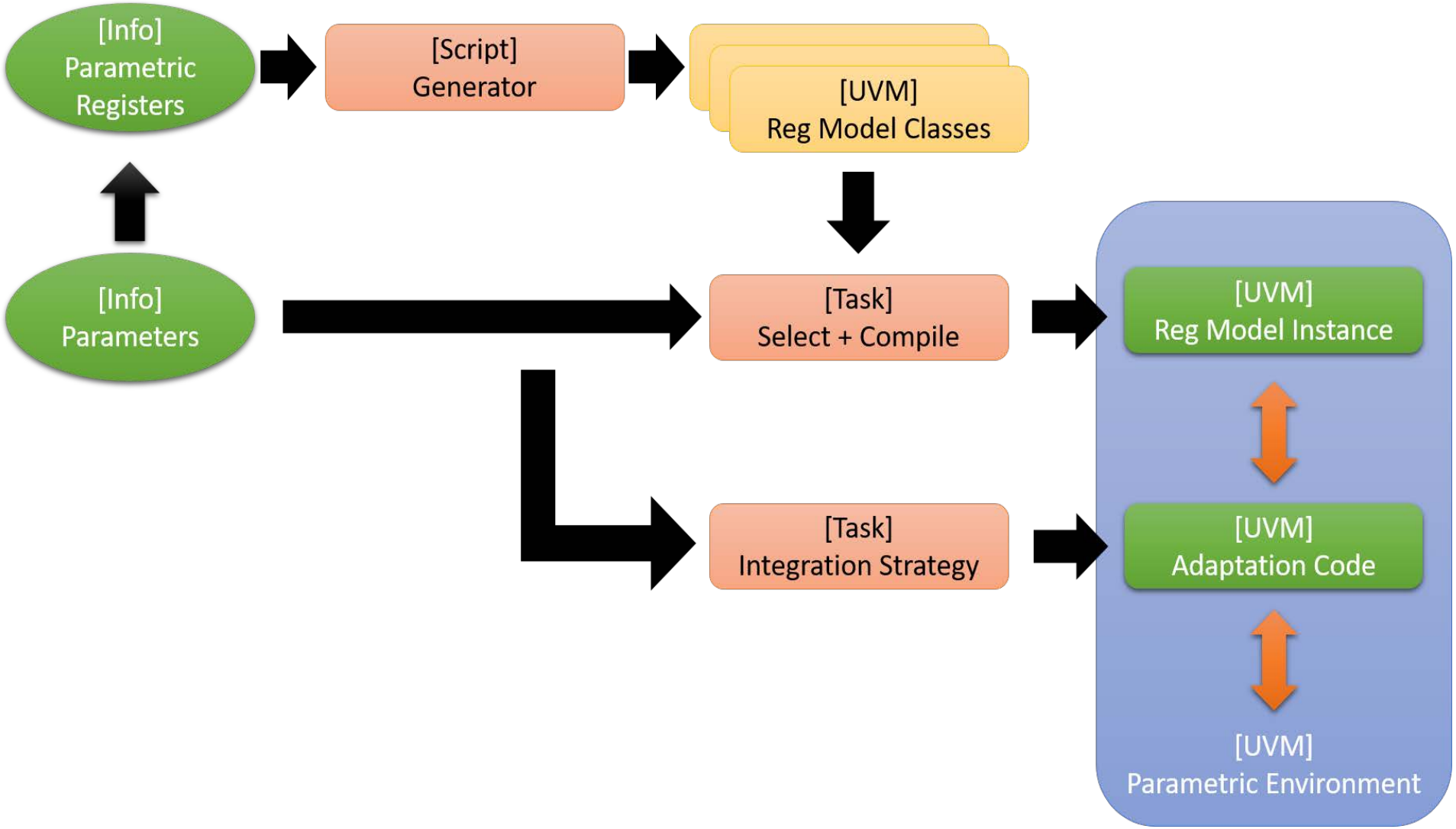


The State Of The Art

- Verification environments are often parametric as well
  - Adaptation code is necessary
    - Class interfaces, abstract classes, bridging classes…
    - E.g.: returning a dummy value for a field that is not present in a given configuration

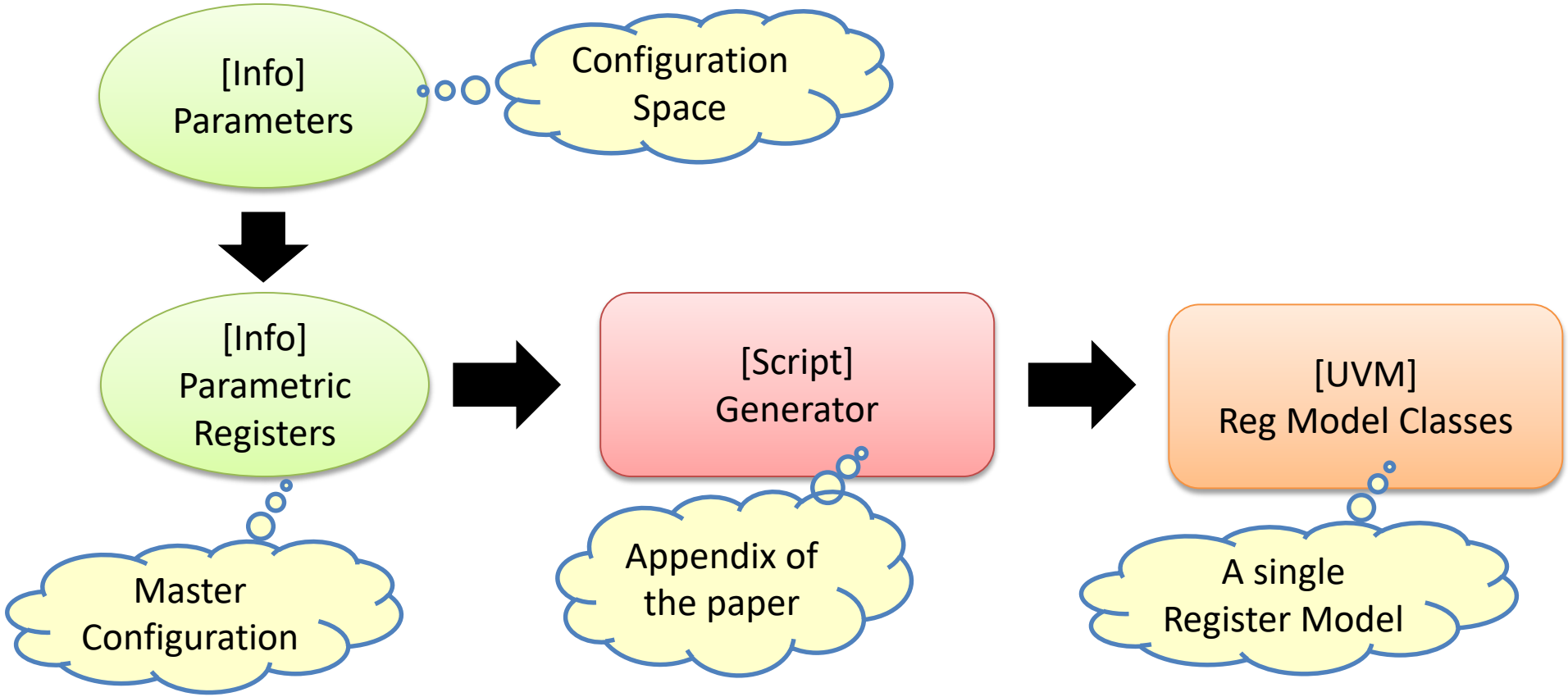


The Traditional Flow

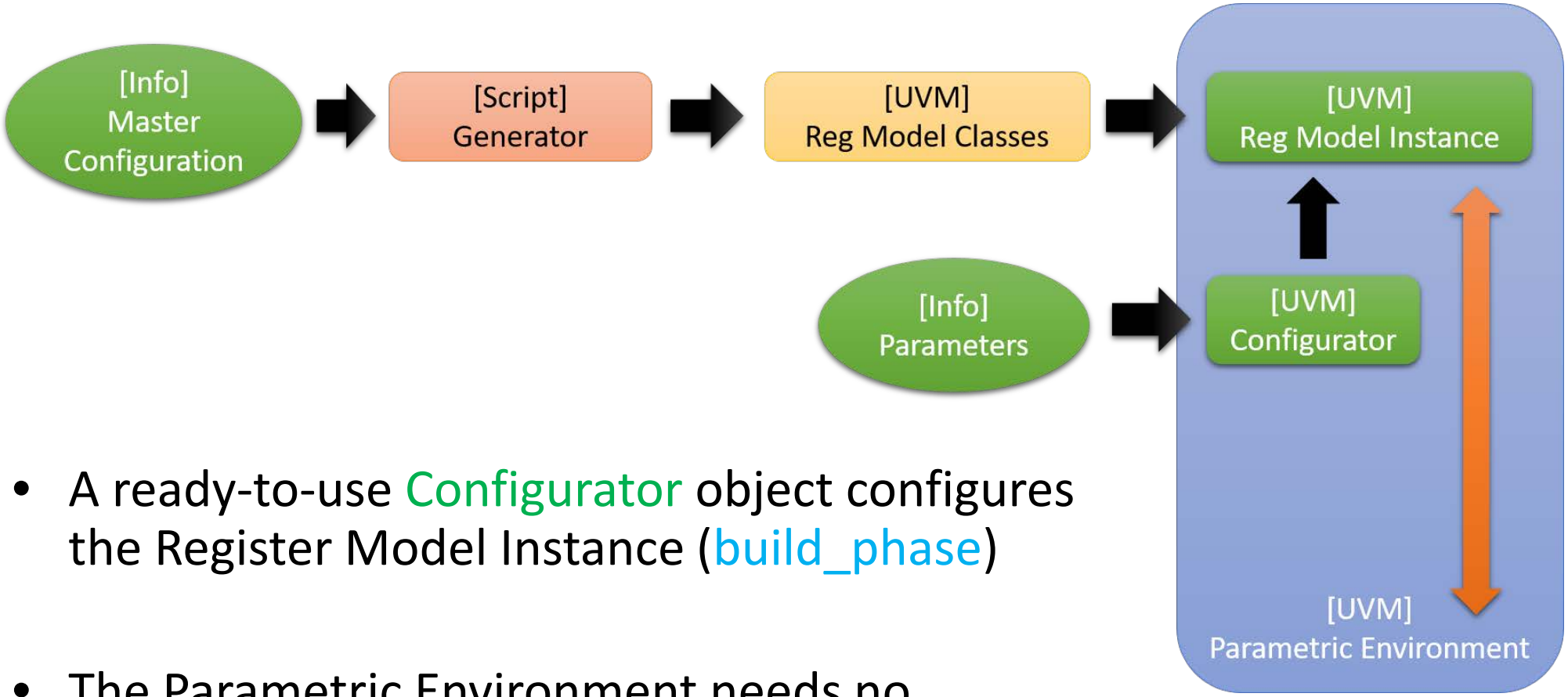


The Proposed Solution

- Generating a single register model for all the configurations
  - This register model contains all the possible fields, registers and covergroups



The Proposed Flow



- A ready-to-use Configurator object configures the Register Model Instance (build_phase)
- The Parametric Environment needs no adaptation code to interface with the registers

Using the Register Model

```
class base_test extends uvm_test;
  virtual function void build_phase(uvm_phase phase);
    Register_Configurator regCfg = new("regCfg");
    regCfg.scope = my_env;

    if(my_param == my_value0) begin: reserve_my_reg1
      string regName = "my_reg1";
      int fieldList [string] = '{"my_field1" : my_field1_reserved_value,
                                 "my_field2" : my_field2_reserved_value};
      regCfg.reserveReg(regName, fieldList);
    end: reserve_my_reg1

    ...  // other actions
  endfunction: build_phase
endclass: base_test
```

Configurator Object

Using the Configurator Object on the registers (before instantiating the Register Model)

Instantiating the Register Model

- Classical and seamless UVM usage of the Register Model

```
class my_env extends uvm_env;
  my_reg_model register_model;

  function void build_phase(uvm_phase phase);
    // creating the register model
    uvm_reg::include_coverage("*", UVM_CVR_ALL);
    register_model = my_reg_model::type_id::create("register_model");
    register_model.set_scope(this);
    register_model.build();
    register_model.lock_model();
    void'(register_model.set_coverage(UVM_CVR_ALL));
  endfunction: build_phase
endclass: my_env
```

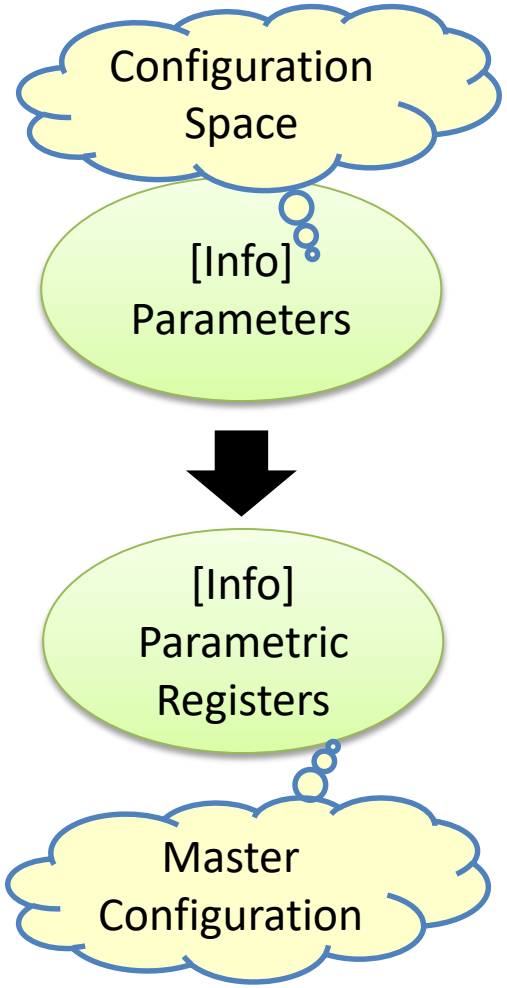
Creating the Register Model

The Register Model is ready and configured

The Master Configuration

- The “master configuration” is a special high-level register description that contains all the possible registers, fields and covergroups of the DUT

- It represents a theoretical configuration that does not necessarily exist in the configuration space
  - Registers, fields and covergroups need only be defined
  - Its content does not need to carry any functional meaning
    - Addresses, positions, reset values (etc) can be dummy values
    - Example: if only one of two registers can exist at a time, both registers are present in the master configuration



The Master Configuration

- The proposed format is YAML
  - Human-readable
  - Easy to write and edit manually
- Many programming languages sport full-fledged parsers and dumpers
  - Simple to generate automatically from other sources of information

```
AREG:
  access: RW
  address: 0
  default: 0
  fields:
    A1FIELD: {end: 15, start: 15}
    A2FIELD: {end: 8, start: 8}
    A3FIELD: {end: 19, start: 19}


BREG:
  access: RW
  address: 0
  default: 123
  fields:
    B1FIELD: {end: 5, start: 4}
    B2FIELD: {end: 23, start: 22}
    B3FIELD: {end: 11, start: 11}
    B4FIELD: {end: 13, start: 13}
```

UVM Implementation - Registers

```
virtual class my_abstract_reg extends uvm_reg;

  uvm_component scope;

  ... // constructor

  virtual function void set_scope(uvm_component scope);
    this.scope = scope;
  endfunction: set_scope

  virtual function void build();
    // empty
  endfunction: build

  // extra user-defined content
endclass: my_abstract_reg
```

Extending from uvm_reg

Reference to a component

Ready to use

Customizable

UVM Implementation - Registers

```
class my_reg1 extends my_abstract_reg;

  ...  // coverage, constructor etc (generated)

  // register fields
  rand uvm_reg_field my_field_1;
  rand uvm_reg_field my_field_2;

  // builder
  extern virtual function void build();
endclass: my_reg1
```

One class for each register

Extending from the abstract class

Generated by the script

- Users should not modify these classes manually: this is generated code
  - In case of a specification change, users can just regenerate the code
  - Only the “master” configuration needs to be modified

UVM Implementation - Registers

Detail of the build() function

Creating the fields

```
virtual function void build();
  super.build();
  my_field_1 = uvm_reg_field::type_id::create("my_field_1");
  my_field_2 = uvm_reg_field::type_id::create("my_field_2");

  begin
    // setting the actual values for my_field_1 at runtime
  end

  begin
    // setting the actual values for my_field_2 at runtime
  end
endfunction: build
```

Next Slide

UVM Implementation - Registers

- To set the actual values for each field at runtime
  - They are looked for in a configuration object
    - in the scope, as "<reg>_<name>_field_cfg"
  - The values from the master configuration are used otherwise

```
// check if the config db contains an object for my_field_1
if(uvm_config_db#(register_field_configuration)::get(this.scope, "",
                  "my_reg1_my_field_1_field_cfg", field_cfg))
   has_cfg = 1;
```

```
// override width
width = ((has_cfg && field_cfg.change_width) ? field_cfg.width : 1);

...  // lsb, access, reset_value, is_rand
this.my_field_1.configure(this, width, lsb, access, 0,
                          reset_value, 1, is_rand, 1);
```

UVM Implementation – Register Block

```
virtual class my_abstract_model extends uvm_reg_block;

  uvm_component scope;

  ... // constructor

  virtual function void set_scope(uvm_component scope);
    this.scope = scope;
  endfunction: set_scope

  virtual function void build();
    // empty
  endfunction: build

  // extra user-defined content (e.g.: search a reg by name)
endclass: my_abstract_model
```

Reference to a component

Extending from uvm_reg_block

Ready to use

Customizable

UVM Implementation – Register Block

```
class my_reg_model extends my_abstract_model;

  ...  // coverage, constructor etc

  // registers
  rand my_reg1 myreg1;
  rand my_reg2 myreg2;

  // builder
  extern virtual function void build();
endclass: my_reg_model
```

Extending from the abstract class

Generated by the script

- Again, users should not modify the generated code

UVM Implementation – Register Block

```
virtual function void build();
  // register instances
  myreg1 = my_reg1::type_id::create("myreg1");
  myreg2 = my_reg2::type_id::create("myreg2");

  // calling build methods
  myreg1.build(); myreg2.build();

  // calling configure methods
  myreg1.configure(...); myreg2.configure(...);

  begin
    // adding myreg1 to the default map
  end

  begin
    // adding myreg2 to the default map
  end
endfunction: build
```

Creating and configuring the registers

Next Slide

UVM Implementation – Register Block

- To set the actual behavior for each register at runtime
  - They are looked for in a configuration object
    - in the scope, as "<reg>_cfg"
  - The values from the master configuration are used otherwise

```
if(uvm_config_db#(register_configuration)::get(scope, "",
                                               "my_reg1_cfg", reg_cfg))
  has_cfg = 1;
```

```
offset = ((has_cfg && reg_cfg.change_offset) ? reg_cfg.offset : 'h0);
access = ((has_cfg && reg_cfg.change_access) ? reg_cfg.access : "RW");
this.default_map.add_reg(myreg1, .offset(offset), .rights(access));
```

UVM Implementation – Configuration Objects

```
virtual class register_abstract_configuration extends uvm_object;
  string access;
  bit change_access;

  function new(string name = "register_abstract_configuration");

  function void setAccess(string access);
    this.change_access = 1;
    this.access = access;
  endfunction: setAccess

  // other methods
endclass: register_abstract_configuration
```

UVM Implementation – Configuration Objects

```
class register_field_configuration extends register_abstract_configuration;
  int unsigned reset_value;
  int unsigned width;
  int unsigned lsb;

  bit change_reset_value;
  bit change_width;
  bit change_lsb;

  function new(string name = "register_field_configuration");

  extern function void setWidth(int unsigned width);
  extern function void setLsb(int unsigned lsb);
  extern function void setResetValue(int unsigned reset_value);
endclass: register_field_configuration
```

UVM Implementation – Configuration Objects

```
class register_configuration extends register_abstract_configuration;
  int unsigned offset;
  bit change_offset;

  function new(string name = "register_configuration");
  extern function void setOffset(int unsigned offset);
endclass: register_configuration
```

UVM Implementation – Reg Configurator

```
class Register_Configurator extends uvm_object;
  uvm_component scope;

  function new(string name = "registerConfigurator");
  extern function void setRegCfg(string regName, register_configuration cfg);
  extern function void setFieldCfg(string regName, string fieldname,
                                   register_field_configuration cfg);

  extern function void reserveField(string regName, string fieldName,
                                    int unsigned resetVal);
  extern function void reserveReg(string regName, int fieldList[string]);

  // other methods
endclass: Register_Configurator
```

It sets the configuration objects in the UVM config db

UVM Implementation – Reg Configurator

```
function void reserveReg(string regName, int fieldList[string]);
  foreach(fieldList[fieldName]) begin
    string cfgName = {fieldName, "_cfg"};
    register_field_configuration cfg = new(cfgName);
    cfg.setResetValue(fieldList[fieldName]);
    cfg.setAccess("RO");
    setFieldCfg(regName, fieldName, cfg);
  end
  setReadOnly(regName);
endfunction: reserveReg
```

An example of implementation of reserveReg

2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Conclusions

- The proposed Configurable Register Model implementation
  - Lets the user regenerate all the configurations entirely after a specification change
  - Provides a seamless interface towards a parametric verification environment
  - Presents a simple format (YAML) to describe the registers' structure ("master" config)
  - Provides a conversion script to generate the UVM classes automatically
  - Decouples the parameterization of the registers from the register classes

- Also, note that the unified coverage model will contain each and every field for all possible configurations
  - This composite coverage model paves the way to interesting scenarios in the context of multi-configuration coverage collection

accellera SYSTEMS INITIATIVE