UVM-SystemC

Language Reference Manual

DRAFT

21 March 2015

Acknowledgements

The creation of this document has been supported by the European Commission as part of the Seventh Framework Programme (FP7) for Research and Technological Development in the project 'VERIFICATION FOR HETEROGENOUS RELIABLE DESIGN AND INTEGRATION' (VERDI). The research leading to this result has received funding from the European Commission under grand agreement No 287562.

More information on the Seventh Framework Programme (FP7) and VERDI project can be found here:

http://cordis.europa.eu/fp7

http://www.verdi-fp7.eu

Special thanks to the Accellera Systems Initiative to grant authorization to use portions of its Universal Verification Methodology Reference Implementation (UVM version 1.1d, March 2013) in this document.

The partners in the VERDI consortium wish to thank Cadence Design Systems Inc. for the initial donation of the UVM-SC Library Reference and documentation (UVM version 1.0, June 2011). This document has been derived from this work, and further enhanced and extended to make it compatible with the UVM 1.1 standard.

Bugs and Suggestions

Please report bugs and suggestions about this document to:

verdi@eas.iis.fraunhofer.de

Table of Contents

1. INTRODUCTION 14
2. UVM-SYSTEMC OVERVIEW 15
   - 2.1 NAMESPACE 15
   - 2.2 HEADER FILES 15
   - 2.3 GLOBAL FUNCTIONS 15
   - 2.4 BASE CLASSES 15
   - 2.5 POLICY CLASSES 15
   - 2.6 REGISTRY AND FACTORY CLASSES 16
   - 2.7 COMPONENT HIERARCHY CLASSES 16
   - 2.8 SEQUENCERS CLASSES 16
   - 2.9 SEQUENCE CLASSES 17
   - 2.10 CONFIGURATION AND RESOURCE CLASSES 17
   - 2.11 PHASING AND SYNCHRONIZATION CLASSES 17
   - 2.12 REPORTING CLASSES 18
   - 2.13 MACROS 18
   - 2.14 EXISTING SYSTEMC FUNCTIONALITY USED IN UVM-SYSTEMC 19
   - 2.15 METHODOLOGY FOR HIERARCHY CONSTRUCTION 20
3. GLOBAL FUNCTIONS 21
   - 3.1 UVM_SET_CONFIG_INT 21
   - 3.2 UVM_SET_CONFIG_STRING 21
   - 3.3 RUN_TEST 21
4. BASE CLASSES 22
   - 4.1 UVM_VOID 22
     - 4.1.1 Class definition 22
   - 4.2 UVM_OBJECT 22
     - 4.2.1 Class definition 22
     - 4.2.2 Construction 24
     - 4.2.3 Seeding 24
     - 4.2.4 Identification 24
     - 4.2.5 Creation 26
     - 4.2.6 Printing 26
     - 4.2.7 Recording 27
     - 4.2.8 Copying 28
     - 4.2.9 Comparing 28
     - 4.2.10 Packing 28
     - 4.2.11 Unpacking 29

4.2.12 Configuration 31
4.2.13 Object macros 31
4.3 UVM_ROOT 31
4.3.1 Class definition 31
4.3.2 Simulation control 32
4.3.3 Topology 33
4.3.4 Data member (variables) 33
4.4 UVM_PORT_BASE 34
4.4.1 Class definition 34
4.4.2 Template parameter IF 35
4.4.3 Constructor 35
4.4.4 Member functions 35
4.5 UVM_COMPONENT_NAME 36
4.5.1 Class definition 36
4.5.2 Constraints on usage 37
4.5.3 Constructor 37
4.5.4 Destructor 37
4.5.5 operator const char* 37
5. POLICY CLASSES 38
5.1 UVM_PACKER 38
5.1.1 Class definition 38
5.1.2 Packing 41
5.1.3 Unpacking 42
5.1.4 operator <<, operator>> 43
5.1.5 Data members (variables) 43
5.2 PRINTER POLICY CLASSES 44
5.2.1 uvm_printer 45
5.2.2 Printing types 47
5.2.3 Printer subtyping 49
5.2.4 Data members 50
5.3 UVM_COMPARER 51
5.3.1 Class definition 51
5.3.2 Member functions 52
5.3.3 Data members (variables) 54
5.4 DEFAULT POLICY OBJECTS 55
6. REGISTRY AND FACTORY CLASSES 57
6.1 UVM_OBJECT_WRAPPER 57
6.1.1 Class definition 57

6.1.2 Member functions 58
6.2 UVM_OBJECT_REGISTRY 58
6.2.1 Class definition 58
6.2.2 Template parameter T 59
6.2.3 Member functions 59
6.3 UVM_COMPONENT_REGISTRY 60
6.3.1 Class definition 60
6.3.2 Template parameter T 61
6.3.3 Member functions 61
6.4 UVM_FACTORY 63
6.4.1 Class definition 63
6.4.2 Registering types 65
6.4.3 Type and instance overrides 65
6.4.4 Creation 66
6.4.5 Debug 68
7. COMPONENT HIERARCHY CLASSES 70
7.1 UVM_COMPONENT 70
7.1.1 Class definition 70
7.1.2 Construction interface 74
7.1.3 Hierarchy interface 74
7.1.4 Phasing interface 76
7.1.5 Process control interface 83
7.1.6 Objection interface 84
7.1.7 Factory interface 85
7.1.8 Hierarchical reporting interface 87
7.1.9 Recording interface 89
7.1.10 Macros 89
7.2 UVM_DRIVER 89
7.2.1 Class definition 90
7.2.2 Template parameters 90
7.2.3 Ports 90
7.2.4 Member functions 90
7.3 UVM_MONITOR 91
7.3.1 Class definition 91
7.3.2 Member functions 91
7.4 UVM_AGENT 92
7.4.1 Class definition 92
7.4.2 Member functions 92

7.5 UVM_ENV 93
7.5.1 Class definition 93
7.5.2 Member functions 93
7.6 UVM_TEST 93
7.6.1 Class definition 93
7.6.2 Member functions 94
7.7 UVM_SCOREBOARD 94
7.7.1 Class definition 94
7.7.2 Member functions 95
7.8 UVM_SUBSCRIBER 95
7.8.1 Class definition 95
7.8.2 Export 96
7.8.3 Member functions 96
8. SEQUENCER CLASSES 97
8.1 UVM_SEQUENCER_BASE 97
8.1.1 Class definition 97
8.1.2 Constructor 98
8.1.3 Member functions 98
8.2 UVM_SEQUENCER_PARAM_BASE 102
8.2.1 Class definition 102
8.2.2 Template parameters 103
8.2.3 Constructor 103
8.2.4 Requests 103
8.3 UVM_SEQUENCER 103
8.3.1 Class definition 103
8.3.2 Template parameters 104
8.3.3 Constructor 104
8.3.4 Exports 105
8.3.5 Sequencer interface 105
8.3.6 Macros 106
8.4 UVM_SRQ_IF_BASE 106
8.4.1 Class definition 106
8.4.2 Template parameters 107
8.4.3 Member functions 107
9. SEQUENCE CLASSES 110
9.1 UVM_TRANSACTION 110
9.1.1 Class definition 110
9.1.2 Constructors 111

9.1.3 Constraints on usage 111
9.1.4 Member functions 111
9.2 UVM_SEQUENCE_ITEM 111
9.2.1 Class definition 111
9.2.2 Constructors 112
9.2.3 Member functions 112
9.3 UVM_SEQUENCE_BASE 114
9.3.1 Class definition 114
9.3.2 Constructor 116
9.3.3 Sequence state 117
9.3.4 Sequence execution 117
9.3.5 Sequence control 119
9.3.6 Sequence item execution 121
9.3.7 Response interface 122
9.3.8 Data members 123
9.4 UVM_SEQUENCE 123
9.4.1 Class definition 124
9.4.2 Template parameters 124
9.4.3 Constructor 124
9.4.4 Member functions 124
10. CONFIGURATION AND RESOURCE CLASSES 126
10.1 UVM_CONFIG_DB 126
10.1.1 Class definition 126
10.1.2 Template parameter T 127
10.1.3 Constraints on usage 127
10.1.4 Member functions 127
10.2 UVM_RESOURCE_DB 128
10.2.1 Class definition 128
10.2.2 Template parameter T 130
10.2.3 Constraints on usage 130
10.2.4 Member functions 130
10.3 UVM_RESOURCE_DB_OPTIONS 132
10.3.1 Class definition 132
10.3.2 Member functions 133
10.4 UVM_RESOURCE_OPTIONS 133
10.4.1 Class definition 133
10.4.2 Member functions 134
10.5 UVM_RESOURCE_BASE 134

10.5.1 Class definition 134
10.5.2 Constructor 135
10.5.3 Resource database interface 136
10.5.4 Read-only interface 136
10.5.5 Notification 136
10.5.6 Scope interface 136
10.5.7 Priority 137
10.5.8 Utility functions 137
10.5.9 Audit trail 137
10.6 UVM_RESOURCE_POOL 138
10.6.1 Class definition 138
10.6.2 get 139
10.6.3 spell_check 140
10.6.4 Set interface 140
10.6.5 Lookup 141
10.6.6 Set priority 142
10.6.7 Debug 143
10.7 UVM_RESOURCE 143
10.7.1 Class definition 143
10.7.2 Template parameter T 144
10.7.3 Type interface 145
10.7.4 Set/Get interface 145
10.7.5 Read/Write interface 146
10.7.6 Priority 146
10.8 UVM_RESOURCE_TYPES 146
10.8.1 Class definition 147
10.8.2 Type definitions (typedefs) 147
10.9 UVM_QUEUE 147
10.9.1 Class definition 147
10.9.2 Template parameter T 148
10.9.3 Member functions 148
10.10 UVM_POOL 149
10.10.1 Class definition 150
10.10.2 Template parameters 150
10.10.3 Member functions 150
11. PHASING AND SYNCHRONIZATION CLASSES 153
11.1 UVM_PHASE 153
11.1.1 Class definition 153

11.1.2 Construction 155
11.1.3 State 155
11.1.4 Callbacks 156
11.1.5 Schedule 157
11.1.6 Synchronization 158
11.1.7 Jumping 159
11.2 UVM_DOMAIN 159
11.2.1 Class definition 160
11.2.2 Constructor 160
11.2.3 Member functions 160
11.3 UVM_BOTTOMUP_PHASE 161
11.3.1 Class definition 161
11.3.2 Constructor 161
11.3.3 Member functions 162
11.4 UVM_TOPDOWN_PHASE 162
11.4.1 Class definition 162
11.4.2 Constructor 163
11.4.3 Member functions 163
11.5 UVM_PROCESS_PHASE° (UVM_TASK_PHASE†) 163
11.5.1 Class definition 163
11.5.2 Member functions 164
11.6 UVM_OBJECTION 164
11.6.1 Class definition 164
11.6.2 Constructor 166
11.6.3 Objection control 166
11.6.4 Callback hooks 168
11.6.5 Objections status 169
11.7 UVM_CALLBACK 170
11.7.1 Class definition 170
11.7.2 Member functions 170
11.8 UVM_CALLBACK_ITER 171
11.8.1 Class definition 171
11.8.2 Template parameter T 171
11.8.3 Template parameter CB 171
11.8.4 Constructor 171
11.8.5 Member functions 172
11.9 UVM_CALLBACKS 172
11.9.1 Class definition 173

11.9.2 Template parameter T 174
11.9.3 Template parameter CB 174
11.9.4 Constructor 174
11.9.5 Add/delete interface 174
11.9.6 Iterator interfaces 175
11.9.7 Debug 176
12. REPORTING CLASSES 177
12.1 UVM_REPORT_OBJECT 177
12.1.1 Class definition 178
12.1.2 Constructors 180
12.1.3 Reporting 180
12.1.4 Verbosity configuration 182
12.1.5 Action configuration 183
12.1.6 File configuration 184
12.1.7 Override configuration 185
12.1.8 Report handler configuration 185
12.2 UVM_REPORT_HANDLER 185
12.2.1 Class definition 186
12.2.2 Constructor 187
12.2.3 Member functions 187
12.3 UVM_REPORT_SERVER 188
12.3.1 Class definition 188
12.3.2 Constructor 189
12.3.3 Member functions 190
12.4 UVM_REPORT_CATCHER 193
12.4.1 Class definition 193
12.4.2 Constructor 195
12.4.3 Current message state 195
12.4.4 Change message state 196
12.4.5 Debug 197
12.4.6 Callback interface 197
12.4.7 Reporting 198
13. MACROS 200
13.1 COMPONENT AND OBJECT REGISTRATION MACROS 200
13.1.1 Macro definitions 200
13.1.2 UVM_OBJECT_UTILS, UVM_OBJECT_PARAM_UTILS 200
13.1.3 UVM_COMPONENT_UTILS, UVM_COMPONENT_PARAM_UTILS 201
13.2 REPORTING MACROS 201

13.2.1 Macro definitions 201
13.2.2 UVM_INFO 201
13.2.3 UVM_WARNING 202
13.2.4 UVM_ERROR 202
13.2.5 UVM_FATAL 202
13.3 SEQUENCE EXECUTION MACROS 202
13.3.1 Macro definitions 202
13.3.2 UVM_DO 203
13.3.3 UVM_DO_PRI 203
13.3.4 UVM_DO_WITH 203
13.3.5 UVM_DO_PRI_WITH 203
13.3.6 UVM_DO_ON 204
13.3.7 UVM_DO_ON_PRI 204
13.3.8 UVM_DO_ON_WITH 204
13.3.9 UVM_DO_ON_PRI_WITH 204
13.3.10 UVM_CREATE 204
13.3.11 UVM_CREATE_ON 205
13.3.12 UVM_DECLARE_P_SEQUENCER 205
13.4 CALLBACK MACROS 205
13.4.1 Macro definitions 205
13.4.2 UVM_REGISTER_CB 205
13.4.3 UVM_DO_CALLBACKS 205
14. TLM INTERFACES 207
14.1 TLM-1 INTERFACES 207
14.2 UVM_BLOCKING_PUT_PORT 208
14.2.1 Class definition 208
14.2.2 Template parameter T 208
14.2.3 Constructor 208
14.2.4 Member functions 208
14.3 UVM_BLOCKING_GET_PORT 209
14.3.1 Class definition 209
14.3.2 Template parameter T 209
14.3.3 Constructor 209
14.3.4 Member functions 210
14.4 UVM_BLOCKING_PEEK_PORT 210
14.4.1 Class definition 210
14.4.2 Template parameter T 210
14.4.3 Constructor 211

14.4.4 Member functions 211
14.5 UVM_BLOCKING_GET_PEEK_PORT 211
14.5.1 Class definition 211
14.5.2 Template parameter T 212
14.5.3 Constructor 212
14.5.4 Member functions 212
14.6 UVM_ANALYSIS_PORT 213
14.6.1 Class definition 213
14.6.2 Template parameter T 213
14.6.3 Constructor 213
14.6.4 Member functions 213
14.7 UVM_ANALYSIS_EXPORT 214
14.7.1 Class definition 214
14.7.2 Template parameter T 214
14.7.3 Constructor 214
14.7.4 Member functions 215
14.8 UVM_TLM_REQ_RSP_CHANNEL 215
14.8.1 Class definition 215
14.8.2 Template parameters 216
14.8.3 Ports and exports 216
14.8.4 Constructor 218
15. GLOBAL DEFINES, TYPEDEFS AND ENUMERATIONS 219
15.1 GLOBAL DEFINES 219
15.1.1 UVM_MAX_STREAMBITS 219
15.1.2 UVM_DEFAULT_TIMEOUT 219
15.2 TYPEDEFS 219
15.2.1 uvm_bitstream_t 219
15.2.2 uvm_integral_t 219
15.2.3 UVM_FILE 219
15.2.4 uvm_report_cb 219
15.2.5 uvm_config_int 219
15.2.6 uvm_config_string 219
15.2.7 uvm_config_object 220
15.2.8 uvm_config_wrapper 220
15.3 ENUMERATION 220
15.3.1 uvm_action 220
15.3.2 uvm_severity 220
15.3.3 uvm_verbosity 220

15.3.4 uvm_active_passive_enum 221
15.3.5 uvm_sequence_state_enum 221
15.3.6 uvm_phase_type 221
16. ANNEX A: UVM-SYSTEMVERILOG FEATURES NOT INCLUDED IN UVM-SYSTEMC 222
16.1 NO FIELD MACROS 222
16.2 NO AUTOMATED CONFIGURATION 222
16.3 NO TRANSACTION RECORDING 222
16.4 NO REGISTER ABSTRACTION LAYER 222
16.5 NO CONSTRAINT RANDOMIZATION AND COVERAGE CLASSES 222
16.6 NO ASSERTIONS 222
17. ANNEX B: DEPRECATED CADENCE UVM-SC FUNCTIONALITY 223
17.1 GLOBAL FUNCTIONS 223
17.1.1 Simulator stop and timeout functions 223
17.1.2 Print configurations 223
17.2 FACTORY 223
17.2.1 Registration macros 223
17.2.2 Global create functions 223
17.2.3 Global type and instance override functions 223
18. ANNEX C: RENAMED FUNCTIONS UVM-SYSTEMC VERSUS UVM-SYSTEMVERILOG 224
19. ANNEX D: TERMINOLOGY 225
19.1 DEFINITIONS 225
19.2 ACRONYMS AND ABBREVIATIONS 227
20. INDEX 229

1. Introduction

UVM-SystemC is a SystemC library extension offering features compatible with the Universal Verification Methodology (UVM). This library is built on top of the SystemC language standard and defines the Application Programming Interface aligned with that of the existing UVM 1.1 standard and associated base class library implementation in SystemVerilog (SV). The UVM-SystemC library does not cover the entire UVM standard, nor the existing UVM implementation in SystemVerilog. However, the UVM-SystemC library offers the essential ingredients to create verification environments which are compliant with the UVM standard.

UVM-SystemC is released as proof-of-concept library that works with any IEEE 1666-2011 compliant SystemC simulation environment. Note that UVM-SystemC uses certain specialized SystemC features introduced since the revision in 2011, such as process control constructs, which are not implemented in all SystemC simulators. The UVM-SystemC functionality can be used together with the Accellera Systems Initiative (formerly OSCI) SystemC proof-of-concept library [1].

UVM-SystemC uses existing SystemC functionality wherever suitable, and introduces new UVM classes on top of the SystemC base classes to facilitate the creation of modular, configurable and reusable verification environments. Certain UVM in SystemVerilog functionality is available as native SystemC language features, and therefore UVM-SystemC uses the existing SystemC classes as foundations for the UVM extensions. Also the transaction-level modeling (TLM) concepts natively exist in SystemC and IEEE Std. 1666-2011, so UVM-SystemC uses the original SystemC TLM definitions and base classes.

Elements which are part of the UVM-SystemC library and language definition and which are *not* part of the UVM 1.1 standard are marked with symbol ϟ. Elements marked with symbol ° are renamed in UVM-SystemC, in contrast to the UVM 1.1 standard, due to their incompatibility due to reserved keywords in C/C++ or an inappropriate name in the context of SystemC base class of member function definitions. The reference to the original UVM 1.1 name is given in brackets and marked with †. Note that these original names are not used in UVM-SystemC.

[1] As process control extensions are only supported in the Accellera Systems Initiative SystemC 2.3.0 release (or later) of the proof-of-concept library, it is required to have this library installed prior to UVM-SystemC installation.

NOTICE

This draft standard is under heavy development. The class definitions and the signatures for the member functions are under discussions and are likely to change. The discussion elements in the documents are labeled red.

2. UVM-SystemC overview

2.1 Namespace

All UVM-SystemC classes and functions shall reside inside the namespace **uvm**.

2.2 Header files

An application shall include the C++ header file **uvm** or **uvm.h** to make use of the UVM-SystemC class library functions. The header file named **uvm** shall only add the name **uvm** to the declarative region in which it is included, whereas the header file named **uvm.h** shall add *all* of the names from the namespace **uvm** to the declarative region in which it is included.

NOTE–It is recommended that an application includes the header file **uvm** rather than the header file **uvm.h**. This means the namespace **uvm** has to be mentioned explicitly when using UVM-SystemC classes and functions. Alternatively, an application may use the C++ *using directive* at the global and local scope to gain access to these classes and functions.

2.3 Global functions

A minimal set of global functions is defined in the global namespace offering generic UVM capabilities and convenience functions for configuration and printing. The global functions are specified in section 3.

2.4 Base classes

These classes define the base UVM class for all other UVM classes, and the base class for data objects:

- **uvm_void**
- **uvm_object**
- **uvm_root**
- **uvm_port_base**
- **uvm_component_name**

The base classes are specified in section 4.

2.5 Policy classes

These classes include policy objects for various operations based on class **uvm_object**:

- **uvm_printer**: performs deep printing of objects derived from **uvm_object**. UVM provides several subtypes to **uvm_printer** that print objects in a specific format: **uvm_table_printer**, **uvm_tree_printer**, and **uvm_line_printer**. Each such printer has many configuration options that govern what and how object members are printed.
- **uvm_comparer**: performs deep comparison of objects derived from **uvm_object**. An application may configure what is compared and how miscompares are reported.
- **uvm_packer**: performs packing (serialize) and unpacking of properties.

The policy classes are specified in section 5.

2.6 Registry and factory classes

The registry and factory classes include the **uvm_factory** and associated classes for object and component registration. The class **uvm_factory** implements a factory pattern. A singleton factory instance is created for a given simulation run. Class types are registered with the factory using the class **uvm_object_wrapper** and its derivatives. The class **uvm_factory** supports type and instance overrides.

The factory classes are:

- **uvm_object_wrapper**
- **uvm_object_registry**
- **uvm_component_registry**
- **uvm_factory**

The registry and factory classes are specified in section 6.

2.7 Component hierarchy classes

These classes define the base class for hierarchical UVM components and the test environment. The class **uvm_component** provides interfaces for:

- Hierarchy—Provides methods for searching and traversing the component hierarchy.
- Configuration—Provides methods for configuring component topology and other parameters before and during component construction.
- Phasing—Defines a phased test flow that all components follow. Methods include the phase callbacks, such as **run_phase** and **report_phase**, overridden by the derived classes. During simulation, these callbacks are executed in precise order.
- Factory—Provides a convenience interface to the **uvm_factory**. The factory is used to create new components and other objects based on type-wide and instance-specific configuration.

All structural component classes **uvm_env**, **uvm_test**, **uvm_agent**, **uvm_driver**, **uvm_monitor**, and **uvm_scoreboard** are derived from the class **uvm_component**.

The UVM component classes are specified in section 7.

2.8 Sequencers classes

The sequencer classes serve as an arbiter for controlling transaction flow from multiple stimulus generators. More specifically, the sequencer controls the flow of transactions of type **uvm_sequence_item** generated by one or more sequences based of type **uvm_sequence**. The sequencer classes are:

- **uvm_sequencer_base**:
- **uvm_sequencer_param_base**
- **uvm_sequencer**
- **uvm_sqr_if_base**

The sequencer classes are specified in section 8.

2.9 Sequence classes

The sequence classes offer the infrastructure to create stimuli descriptions based on transactions, encapsulated as a sequence or sequence item. The following sequence classes are defined:

- **uvm_transaction**
- **uvm_sequence_item**
- **uvm_sequence_base**
- **uvm_sequence**

The sequence classes are specified in section 9.

2.10 Configuration and resource classes

The configuration and resource classes provide access to the configuration and resource database. The configuration database is used to store and retrieve both configuration time and run time properties. The configuration and resource classes are:

- **uvm_config_db**: Configuration database, which acts as interface on top of the resource database.
- **uvm_resource_db**: Resource database.
- **uvm_resource_options**: Provides a namespace for managing options for the resources facility.
- **uvm_resource_base**: Provides a non-parameterized base class for resources.
- **uvm_resource_pool**: Provides the global resource database.
- **uvm_resource**: Defines the parameterized resource.

This configuration and resource classes are specified in section 10.

2.11 Phasing and synchronization classes

The phasing classes define the order of execution of pre-defined callback function and processes, which run either sequentially or concurrently. In addition, dedicated member functions for synchronization are available to coordinate the execution of or status of these processes between all UVM components or objects.

The phasing and synchronization classes are:

- **uvm_phase**: The base class for defining a phase's behavior, state, context.
- **uvm_domain**: Phasing schedule node representing an independent branch of the schedule.
- **uvm_bottomup_phase**: A phase implementation for bottom up function phases.
- **uvm_topdown_phase**: A phase implementation for top-down function phases.
- **uvm_process_phase**° (**uvm_task_phase**†): A phase implementation for phases which are launched as spawned process.
- **uvm_objection**: Mechanism to synchronize phases based on passing execution status information between running processes.

- **uvm_callbacks**: The base class for implementing callbacks, which are typically used to modify or augment component behavior without changing the component base class for user-defined callback classes.
- **uvm_callback_iter**: An class for iterating over callback queues of a specific callback type.
- **uvm_callback:** The base class for user-defined callback classes.

The phasing and synchronization classes are specified in section 11.

2.12 Reporting classes

The reporting classes provide a facility for issuing reports (messages) with consistent formatting and configurable side effects, such as logging to a file or exiting simulation. An application can also filter out reports based on their verbosity, identity, or severity.

The following reporting classes are defined:

- **uvm_report_object**: The base class which provides the interface to the UVM reporting mechanism.
- **uvm_report_handler**: The class which acting as implementation for the member functions defined in the class **uvm_report_object**.
- **uvm_report_server**: The class acting as global server that processes all of the reports generated by the class **uvm_report_handler**.
- **uvm_report_catcher**: The class which captures and counts all reports issued by the class **uvm_report_server**.

The reporting classes are specified in section 12.

2.13 Macros

The UVM-SystemC macros make common code easier to write. It is not imperative to use the macros, but in many cases the macros can save a substantial amount of user-written code. The macros defined in UVM-SystemC are:

- Macros for component and object registration:
    - **UVM_OBJECT_UTILS**
    - **UVM_OBJECT_PARAM_UTILS**
    - **UVM_COMPONENT_UTILS**
    - **UVM_COMPONENT_PARAM_UTILS**
- Sequence execution macros:
    - **UVM_DO**, **UVM_DO_ON** and **UVM_DO_ON_PRI**
    - **UVM_CREATE**, **UVM_CREATE_ON**
    - **UVM_DECLARE_P_SEQUENCER**
- Reporting macros:
    - **UVM_INFO**, **UVM_ERROR**, **UVM_WARNING** and **UVM_FATAL**
- Callback macros:

- **UVM_REGISTER_CB** and **UVM_DO_CALLBACKS**

Detailed information for the macros or the associated member functions are specified in section 13.

2.14 Existing SystemC functionality used in UVM-SystemC

UVM-SystemVerilog defines classes **uvm_*_port**, **uvm_*_export**, **uvm_*_imp** for ports and exports, and the member function **connect** to bind these ports and exports. UVM-SystemC does not define any new base classes for this purpose, but uses the existing SystemC language constructs **sc_core**::**sc_port**, **sc_core**::**sc_export**, and also supports the existing port/export binding mechanisms using the **sc_core**::**sc_port** member functions **bind** or **operator()**.

In UVM-SystemVerilog, the class **uvm_component** inherits from class **uvm_object**. In UVM-SystemC, the same inheritance is followed. This means class **uvm_component** inherits from the existing SystemC class **sc_core**::**sc_module**, and **uvm_report_object**, which is derived from **uvm_object**. Note that the class **uvm_object** is not derived from class **sc_core**::**sc_object**, but from class **uvm_void**.

Because SystemVerilog does not support multiple inheritance, UVM-SystemVerilog is constrained to have only one base class, from which both data elements and hierarchical elements inherit. SystemC supports multiple inheritance. As such, UVM-SystemC uses multiple inheritance where suitable. In addition, the class **sc_module** already offer the hierarchical features that **uvm_component** needs, namely parent and children, and a full instance name. In fact, in SystemC the parent of a component does not need to be explicitly specified as a constructor argument; instead, the language figures out the parent from the top of the current module stack hierarchy.

The class **sc_module** has natural equivalents to some of the UVM pre-run phases, which can used in a UVM-SystemC **uvm_component**. For example:

- The UVM-SystemC callback **before_end_of_elaboration** is mapped onto the UVM callback **build_phase**. Note that UVM-SystemC also provides the callback **build_phase** as an alternative to **before_end_of_elaboration**. It is recommended to use this UVM member function.
- The UVM-SystemC callback **end_of_elaboration** is mapped onto the UVM callback **end_of_elaboration_phase**. UVM-SystemC also provides the callback **end_of_elaboration_phase** with the argument of type **uvm_phase** as an alternative to the callback **end_of_elaboration**, which does give access to the phase information. It is recommended to use this UVM member function.
- The UVM-SystemC callback **start_of_simulation** is mapped onto the UVM callback **start_of_simulation_phase**. UVM-SystemC also provides the callback **start_of_simulation_phase** with the argument of type **uvm_phase** as an alternative to the callback **start_of_simulation**, which does give access to the phase information. It is recommended to use this UVM member function.

UVM-SystemC also defines the callback **run_phase** as a thread process of a **uvm_component**. This works because **sc_module** in SystemC already has the ability to own and spawn thread processes.

UVM-SystemVerilog defines the TLM-1 interfaces like **put** and **get**, as well as some predefined TLM-1 channels like **tlm**::**tlm_fifo**. These already natively exist in the SystemC standard. UVM-SystemC supports the original SystemC TLM-1 definitions. The same holds for the analysis interface in UVM. UVM-SystemC offers a compatibility and convenience layer on top of the TLM interface proper **tlm**::**tlm_analysis_if** and analysis port **tlm**::**tlm_analysis_port,** defining elements such as **uvm_analysis_port** and **uvm_analysis_export**. These TLM classes are specified in section 14.

The SystemC fork-join constructs **SC_FORK** and **SC_JOIN** can be used as a pair to bracket a set of calls to function **sc_core**::**sc_spawn** within a **run_phase**, enabling the creation of concurrent processes.

2.15 Methodology for hierarchy construction

The UVM in SystemVerilog recommends the use of configurations by using the static member function **set** of the **uvm_config_db** in the build phase, followed by hierarchy construction through the factory, in the same phase.

In UVM-SystemVerilog, it is necessary to make the connections (port binding) in the connect phase, which happens after hierarchy construction of components, ports and exports in the build phase. This enables configuration of port/export construction using the configuration database **uvm_config_db**. In that case, if a parent creates a child in the build phase, that child's port/export does not exist at that point, and it has to wait for the next phase to bind child's port/export.

Consistent with UVM in SystemVerilog, UVM-SystemC also recommends configurations using **uvm_config_db** and hierarchy construction through the factory **uvm_factory** in the build phase. This implies that child objects derived from class **uvm_component** should be declared as pointers inside the parent class, and these children should be constructed in the UVM callback **build_phase** through the UVM factory, which does not contradict the SystemC standard, as the SystemC standard allows construction activity in the callback **before_end_of_elaboration**, which is equivalent to the UVM build phase.

In SystemC, the ports/exports are usually becoming members of a **uvm_component** and not pointers. In that case, the ports/exports are automatically created and initialized in the constructor of the parent **uvm_component**. This implies that in UVM-SystemC the ports/export construction is *not* configurable through **uvm_config_db**. Because the bulk of the UVM hierarchy construction occurs in the build phase, the port/export bindings that depend on the entire hierarchy being constructed have to be done in a later phase. Similar as in UVM-SystemVerilog, the connect phase is introduced in UVM-SystemC to perform the port bindings using the **sc_port** member function **bind** or **operator()**. The UVM binding mechanism using the member function **connect** of the ports is made available for compatibility purposes.

3. Global functions

All global functions reside in the UVM namespace. Functions marked with symbol ϟ are not compatible with the UVM 1.1 standard.

3.1 uvm_set_config_intϟ

```
void uvm_set_config_int( const std::string& inst_name,
                         const std::string& field_name,
                         int value );
```

The global function **uvm_set_config_int** shall create and place an integer in a configuration database. The argument *inst_name* shall define the full hierarchical pathname of the object being configured. The argument *field_name* is the specific field that is being searched for. Both arguments *inst_name* and *field_name* may contain wildcards.

NOTE–This global function is made available since there is no command line interface option to pass configuration data.

3.2 uvm_set_config_stringϟ

```
void uvm_set_config_string( const std::string& inst_name,
                        const std::string& field_name,
                        const std::string& value );
```

The global function **uvm_set_config_string** shall create and place a string in a configuration database. The argument *inst_name* shall define the full hierarchical pathname of the object being configured. The argument *field_name* is the specific field that is being searched for. Both arguments *inst_name* and *field_name* may contain wildcards.

NOTE–This global function is made available since there is no command line interface option to pass configuration data.

3.3 run_test

```
void run_test( const std::string& test_name = "" );
```

The function **run_test** is a convenience function to start member function **uvm_root**::**run_test**. (See 4.3.2.1)

4. Base Classes

4.1 uvm_void

The class **uvm_void** shall provide the base class for all UVM classes. It shall be an abstract class with no data members or functions, to allow the creation of a generic container of objects.

An application may derive directly from this class and will inherit none of the UVM functionality, but such classes may be placed in **uvm_void**-typed containers along with other UVM objects.

4.1.1 Class definition

```
namespace uvm {


  class uvm_void
  {
   public:
    uvm_void();
    virtual ~uvm_void();
  }; // class uvm_void


} // namespace uvm
```

4.2 uvm_object

The class **uvm_object** shall provide the base class for all UVM data and hierarchical classes. Its primary role is to define a set of member functions for common operations as create, copy, compare, print, and record. Classes deriving from **uvm_object** shall implement the member functions such as **create** and **get_type_name**.

4.2.1 Class definition

```
namespace uvm {


  class uvm_object : public uvm_void, public uvm_report_object
  {
   public:
    // Group: Construction
    uvm_object();
    explicit uvm_object( const std::string& name );


    // Group: Seeding
    static bool use_uvm_seeding; NOT IMPLEMENTED YET --SHOULD BE TRUE BY DEFAULT
    void reseed(); NOT IMPLEMENTED YET
```

```
// Group: Identification
virtual void set_name( const std::string& name );
virtual const std::string get_name() const;
virtual const std::string get_full_name() const;
virtual int get_inst_id() const;
static int get_inst_count();
static const uvm_object_wrapper* get_type();
virtual const uvm_object_wrapper* get_object_type() const;
virtual const std::string get_type_name() const;

// Group: Creation
virtual uvm_object* create( const std::string& name = "" );
virtual uvm_object* clone();

// Group: Printing
void print( uvm_printer* printer = NULL ) const;
std::string sprint( uvm_printer* printer = NULL ) const;
virtual void do_print( const uvm_printer& printer ) const;
virtual std::string convert2string() const;

// Group: Recording
void record( uvm_recorder* recorder = NULL );
virtual void do_record( const uvm_recorder& recorder );

// Group: Copying
void copy( const uvm_object& rhs );
virtual void do_copy( const uvm_object& rhs );

// Group: Comparing
bool compare( const uvm_object& rhs,
              const uvm_comparer* comparer = NULL ) const;

virtual bool do_compare( const uvm_object& rhs,
                         const uvm_comparer* comparer = NULL ) const;

// Group: Packing
int pack( std::vector<bool>& bitstream, uvm_packer* packer = NULL );
```

```
    int pack_bytes( std::vector<unsigned char>& bytestream, uvm_packer* packer = NULL );
    int pack_ints( std::vector<unsigned int>& intstream, uvm_packer* packer = NULL );
    virtual void do_pack( uvm_packer& packer ) const;

    // Group: Unpacking
    int unpack( const std::vector<bool>& v, uvm_packer* packer = NULL );
    int unpack_bytes( const std::vector<unsigned char>& v, uvm_packer* packer = NULL );
    int unpack_ints( const std::vector<unsigned int>& v, uvm_packer* packer = NULL );
    virtual void do_unpack( uvm_packer& packer );

    // Group: Configuration NOT IMPLEMENTED

  }; // class uvm_object

} // namespace uvm
```

4.2.2 Construction

4.2.2.1 Constructors

```
uvm_object();
explicit uvm_object( const std::string& name );
```

The constructor shall create a new **uvm_object** with the given instance *name* passed as argument. If no argument is given, the default constructor shall call function **sc_core**::**sc_gen_unique_name**("object") to generate a unique string name as instance name of this object.

4.2.3 Seeding

NOTE–Randomization not implemented yet.

4.2.4 Identification

4.2.4.1 set_name

```
virtual void set_name( const std::string& name );
```

The member function **set_name** shall set the instance name of this object passed as argument, overwriting any previously given name. It shall be an error if the name is already in use for another object.

4.2.4.2 get_name

```
virtual const std::string get_name() const;
```

The member function **get_name** shall return the name of the object, as provided by the argument *name* via the constructor or member function **set_name**.

4.2.4.3 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the full hierarchical name of this object. The default implementation is the same as **get_name**, as objects of type **uvm_object** do not inherently possess hierarchy.

Objects possessing hierarchy, such as objects of type **uvm_component**, override the default implementation. Other objects might be associated with component hierarchy, but are not themselves components. For example, sequence classes of type **uvm_sequence** are typically associated with a sequencer class of type **uvm_sequencer**. In this case, it is useful to override **get_full_name** to return the sequencer's full name concatenated with the sequence's name. This provides the sequence a full context, which is useful when debugging.

4.2.4.4 get_inst_id

```
virtual int get_inst_id() const;
```

The member function **get_inst_id** shall return the object's unique, numeric instance identifier.

4.2.4.5 get_inst_count

```
static int get_inst_count();
```

The member function **get_inst_count** shall return the current value of the instance counter, which represents the total number of objects of type **uvm_object** that have been allocated in simulation. The instance counter is used to form a unique numeric instance identifier.

4.2.4.6 get_type

```
static const uvm_object_wrapper* get_type();
```

The member function **get_type** shall return the type-proxy (wrapper) for this object. The **uvm_factory**'s type-based override and creation member functions take arguments of **uvm_object_wrapper**. The default implementation of this member function shall produce an error and return NULL.

To enable use of this member function, a user's subtype must implement a version that returns the subtype's wrapper.

4.2.4.7 get_object_type

```
virtual const uvm_object_wrapper* get_object_type() const;
```

The member function **get_object_type** shall the return the type-proxy (wrapper) for this object. The **uvm_factory**'s type-based override and creation member functions take arguments of **uvm_object_wrapper**. The default implementation of this member function does a factory lookup of the proxy using the return value from **get_type_name**. If the type returned by **get_type_name** is not registered with the factory, then the member function shall return NULL.

This member function behaves the same as the static member function **get_type**, but uses an already allocated object to determine the type-proxy to access (instead of using the static object).

4.2.4.8 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the object, which is typically the type identifier enclosed in quotes. It is used for various debugging functions in the library, and it is used by the factory for creating objects.

4.2.5 Creation

4.2.5.1 create

```
virtual uvm_object* create( const std::string& name = "" );
```

The member function **create** shall allocate a new object of the same type as this object and returns it via a base handle **uvm_object**. Every class deriving from **uvm_object**, directly or indirectly, shall implement the member function **create**.

4.2.5.2 clone

```
virtual uvm_object* clone();
```

The member function **clone** shall create and return an exact copy of this object.

NOTE–As the member function **clone** is virtual, derived classes may override this implementation if desired.

4.2.6 Printing

4.2.6.1 print

```
void print( uvm_printer* printer = NULL ) const;
```

The member function **print** shall deep-print this object's properties in a format and manner governed by the given argument *printer*. If the argument *printer* is not provided, the global **uvm_default_printer** shall be used (see 5.4.1.4)

The member function **print** is not virtual and shall not be overloaded. To include custom information in the **print** and **sprint** operations, derived classes shall override the member function **do_print** and can use the provided printer policy class to format the output.

4.2.6.2 sprint

```
std::string sprint( uvm_printer* printer = NULL ) const;
```

The member function **sprint** shall return the object's properties as a string and manner governed by the given argument *printer*. If the argument *printer* is not provided, the global **uvm_default_printer** shall be used (see 5.4.1.4)

The member function **sprint** is not virtual and must not be overloaded. To include additional fields in the **print** and **sprint** operation, derived classes must override the member function **do_print** and use the provided printer policy class to format the output. The printer policy will manage all string concatenations and provide the string to **sprint** to return to the caller.

4.2.6.3 do_print

```
virtual void do_print( const uvm_printer& printer ) const;
```

The member function **do_print** shall provide a context called by the member functions **print** and **sprint** that allows an application to customize what gets printed. The argument *printer* is the policy object that governs the format and content of the output. To ensure correct **print** and **sprint** operation, and to ensure a consistent output format, the printer shall be used by all **do_print** implementations.

4.2.6.4 convert2string

```
virtual std::string convert2string() const;
```

The member function **do_print** shall provide a context which may be called directly by the application, to provide object information in the form of a string. Unlike the member function **sprint**, there is no requirement to use a **uvm_printer** policy object. As such, the format and content of the output is fully customizable, which may be suitable for applications not requiring the consistent formatting offered by the **print**/**sprint**/**do_print** API.

4.2.7 Recording

4.2.7.1 record

```
void record( uvm_recorder* recorder = NULL );
```

The member function **record** shall deep-records this object's properties according to an optional recorder policy. The member function is not virtual and shall not be overloaded. To include additional fields in the record operation, derived classes should override the member function **do_record**.

The optional argument *recorder* specifies the recording policy, which governs how recording takes place. If a recorder policy is not provided explicitly, then the global **uvm_default_recorder** policy is used (see 5.4.1.7).

NOTE–The recording mechanism is vendor-specific. By providing access via a common interface, the **uvm_recorder** policy provides vendor-independent access to a simulator's recording capabilities.

4.2.7.2 do_record

```
virtual void do_record( const uvm_recorder& recorder );
```

The member function **do_record** shall provide a context called by the member function **record**. A derived class should overload this member function to include its fields in a record operation.

The argument *recorder* is policy object for recording this object. A **do_record** implementation should call the appropriate recorder member function for each of its fields.

NOTE–Vendor-specific recording implementations are encapsulated in the recorder policy, thereby insulating user-code from vendor-specific behavior.

4.2.8 Copying

4.2.8.1 copy

```
void copy( const uvm_object& rhs );
```

The member function **copy** shall make a copy of the specified object passed as argument.

NOTE–The member function is not virtual and should not be overloaded in derived classes. To copy the fields of a derived class, that class should overload the member function **do_copy**.

4.2.8.2 do_copy

```
virtual void do_copy( const uvm_object& rhs );
```

The member function **do_copy** shall provide a context called by the member function **copy**. A derived class should overload this member function to include its fields in a copy operation.

4.2.9 Comparing

4.2.9.1 compare

```
bool compare( const uvm_object& rhs,
              const uvm_comparer* comparer = NULL ) const;
```

The member function **compare** shall compare members of this data object with those of the object provided in the rhs (right-hand side) argument. It shall return true on a match; otherwise it shall return false.

The optional argument *comparer* specifies the comparison policy. It allows an application to control some aspects of the comparison operation. It also stores the results of the comparison, such as field-by-field miscompare information and the total number of miscompares. If a compare policy is not provided or set to NULL, then the global **uvm_default_comparer** policy is used (see 5.4.1.6).

NOTE–The member function is not virtual and should not be overloaded in derived classes. To compare the fields of a derived class, that class should overload the member function **do_compare**.

4.2.9.2 do_compare

```
virtual bool do_compare( const uvm_object& rhs,
                         const uvm_comparer* comparer = NULL ) const;
```

The member function **do_compare** shall provide a context called by the member function **compare**. A derived class should overload this member function to include its fields in a compare operation. The member function shall return true if the comparison succeeds; otherwise it shall return false.

4.2.10 Packing

4.2.10.1 pack

```
int pack( std::vector<bool>& bitstream, uvm_packer* packer = NULL );
```

The member function **pack** shall concatenate the object properties into a vector of bits. The member function shall return the total number of bits packed into the given vector.

The optional argument *packer* specifies the packing policy, which governs the packing operation. If a packer policy is not provided or set to NULL, the global **uvm_default_packer** policy shall be used (see 5.4.1.5).

NOTE–The member function is not virtual and should not be overloaded in derived classes. To include additional fields in the pack operation, derived classes should overload the member function **do_pack**.

4.2.10.2 pack_bytes

```
int pack_bytes( std::vector<char>& bytestream, uvm_packer* packer = NULL );
```

The member function **pack_bytes** shall concatenate the object properties into a vector of bytes. The member function shall return the total number of bytes packed into the given vector.

The optional argument *packer* specifies the packing policy, which governs the packing operation. If a packer policy is not provided or set to NULL, the global **uvm_default_packer** policy shall be used (see 5.4.1.5).

NOTE–The member function is not virtual and should not be overloaded in derived classes. To include additional fields in the pack operation, derived classes should overload the member function **do_pack**.

4.2.10.3 pack_ints

```
int pack_ints( std::vector<int>& intstream, uvm_packer* packer = NULL );
```

The member function **pack_ints** shall concatenate the object properties into a vector of integers. The member function shall return the total number of integers packed into the given vector.

The optional argument *packer* specifies the packing policy, which governs the packing operation. If a packer policy is not provided or set to NULL, the global **uvm_default_packer** policy shall be used (see 5.4.1.5).

NOTE–The member function is not virtual and should not be overloaded in derived classes. To include additional fields in the pack operation, derived classes should overload the member function **do_pack**.

4.2.10.4 do_pack

```
virtual void do_pack( uvm_packer& packer ) const;
```

The member function **do_pack** shall provide a context called by the member functions **pack**, **pack_bytes** and **pack_ints**. A derived class should overload this member function to include its fields in a packing operation.

The argument *packer* is the policy object for packing and should be used to pack objects.

4.2.11 Unpacking

4.2.11.1 unpack

```
int unpack( const std::vector<bool>& bitstream, uvm_packer* packer = NULL );
```

The member function **unpack** shall extract the values from a vector of bits. The member function shall return the total number of bits unpacked from the given vector.

The optional argument *packer* specifies the packing policy, which governs both the pack and unpack operation. If a packer policy is not provided or set to NULL, the global **uvm_default_packer** policy shall be used (see 5.4.1.5).

NOTE 1–The member function is not virtual and should not be overloaded in derived classes. To include additional fields in the unpack operation, derived classes should overload the member function **do_unpack**.

NOTE 2–The application of the member function for unpacking shall exactly correspond to the member function for packing. This is assured if (a) the same packer policy is used to pack and unpack, and (b) the order of unpacking is the same as the order of packing used to create the input vector.

4.2.11.2 unpack_bytes

```
int unpack_bytes( const std::vector<char>& bytestream, uvm_packer* packer = NULL );
```

The member function **unpack_bytes** shall extract the values from a vector of bytes. The member function shall return the total number of bytes unpacked from the given vector.

The optional argument *packer* specifies the packing policy, which governs the pack and unpack operation. If a packer policy is not provided or set to NULL, the global **uvm_default_packer** policy shall be used (see 5.4.1.5).

NOTE 1–The member function is not virtual and should not be overloaded in derived classes. To include additional fields in the unpack operation, derived classes should overload the member function **do_unpack**.

NOTE 2–The application of the member function for unpacking shall exactly correspond to the member function for packing. This is assured if (a) the same packer policy is used to pack and unpack, and (b) the order of unpacking is the same as the order of packing used to create the input vector.

4.2.11.3 unpack_ints

```
int unpack_ints( const std::vector<int>& intstream, uvm_packer* packer = NULL );
```

The member function **unpack_ints** shall extract the values from a vector of integers. The member function shall return the total number of integers unpacked from the given vector.

The optional argument *packer* specifies the packing policy, which governs the pack and unpack operation. If a packer policy is not provided or set to NULL, the global **uvm_default_packer** policy shall be used (see 5.4.1.5).

NOTE 1–The member function is not virtual and should not be overloaded in derived classes. To include additional fields in the unpack operation, derived classes should overload the member function **do_unpack**.

NOTE 2–The application of the member function for unpacking shall exactly correspond to the member function for packing. This is assured if (a) the same packer policy is used to pack and unpack, and (b) the order of unpacking is the same as the order of packing used to create the input vector.

4.2.11.4 do_unpack

```
virtual void do_unpack( uvm_packer& packer ) const;
```

The member function **do_unpack** shall provide a context called by the member functions **unpack**, **unpack_bytes** and **unpack_ints**. A derived class should overload this member function to include its fields in a unpacking operation. The member function shall return true if the unpacking succeeds; otherwise it shall return false.

The argument *packer* is the policy object for unpacking and should be used to unpack objects.

NOTE–The application of the member function for unpacking shall exactly correspond to the member function for packing. This is assured if (a) the same packer policy is used to pack and unpack, and (b) the order of unpacking is the same as the order of packing used to create the input vector.

4.2.12 Configuration

NOT IMPLEMENTED

4.2.13 Object macros

UVM-SystemC provides the following macros for a **uvm_object**:

- utility macro **UVM_OBJECT_UTILS**(*classname*) is to be used inside the class definition that expands to:
  - The declaration of the member function **get_type_name,** which returns the type of a class as string
  - The declaration of the member function **get_type**, which returns a factory proxy object for the type
  - The declaration of the proxy class **uvm_object_registry**<*classname*> used by the factory.

Template classes shall use the macro **UVM_OBJECT_PARAM_UTILS**, to guarantee correct registration of one or more parameters passed to the class template. Note that template classes are not evaluated at compile-time, and thus not registered with the factory. Due to this, name-based lookup with the factory for template classes is not possible. Instead, an application shall use the member function **get_type** for factory overrides.

4.3 uvm_root

The class **uvm_root** serves as the implicit top-level and phase controller for all UVM components. An application shall not directly instantiate **uvm_root**. A UVM implementation shall create a single instance of **uvm_root** that an application can access via the global variable **uvm_top**.

4.3.1 Class definition

```
namespace uvm {


  class uvm_root : public uvm_component
  {
   public:
    static uvm_root* get();


    // Simulation control
    virtual void run_test( const std::string& test_name = "" );
    virtual void die();
    void set_timeout( const sc_core::sc_time& timeout, bool overridable = true );


    // Topology
    uvm_component* find( const std::string& comp_match );
```

```
    void find_all( const std::string& comp_match,
                   std::vector<uvm_component*>& comps,
                   uvm_component* comp = NULL );
    void print_topology( uvm_printer* printer = NULL );

    // variables
    bool enable_print_topology;
    bool finish_on_completion;
    const uvm_root* uvm_top;

  }; // class uvm_root
} // namespace uvm
```

4.3.2 Simulation control

4.3.2.1 run_test

```
virtual void run_test( const std::string& test_name = "" );
```

The member function **run_test** shall register the test component and the UVM phasing mechanism. If the optional argument *test_name* is provided, then the specified component is created just prior to phasing, if and only if this component is derived from class **uvm_test**. Otherwise it shall be an error. The test component may contain new verification components or the entire testbench, in which case the test and testbench can be chosen from.

The phasing mechanism is used during test execution, where all components are called following a defined set of registered phases. The member function **run_test** will register both the common phases as well as the UVM run-time phases. (See section 11).

NOTE 1–Selection of the test via the command line interface is not yet available.

NOTE 2–The test execution is started using the SystemC function **sc_core**::**sc_start**. It is recommended not to specify the simulation stop time, as the end-of-test is automatically managed by the phasing mechanism.

4.3.2.2 die

```
virtual void die();
```

The member function **die** shall be called by the report server if a report reaches the maximum quit count or has a **UVM_EXIT** action associated with it, e.g., as with fatal errors. The member function shall call the member function **uvm_component**::**pre_abort** on the entire UVM component hierarchy in a bottom-up fashion. It then shall call **uvm_report_server**::**report_summarize** and terminates the simulation.

4.3.2.3 set_timeout

```
void set_timeout( const sc_core::sc_time& timeout, bool overridable = true );
```

The member function **set_timeout** shall define the timeout for the run phases. If not called, the default timeout shall be set to **UVM_DEFAULT_TIMEOUT** (see 15.1.2).

4.3.3 Topology

4.3.3.1 find

```
uvm_component* find( const std::string& comp_match );
```

The member function **find** shall return a component handle matching the given string *comp_match*. The string may contain the wildcards '*' and '?'. Strings beginning with character '.' are absolute path names.

4.3.3.2 find_all

```
void find_all( const std::string& comp_match,
               std::vector<uvm_component*>& comps,
               uvm_component* comp = NULL );
```

The member function **find_all** shall return a vector of component handles matching the given string *comp_match*. The string may contain the wildcards '*' and '?'. Strings beginning with character '.' are absolute path names. If the optional component argument *comp* is provided, then the search begins from that component down; otherwise it searches all components.

4.3.3.3 print_topology

```
void print_topology( uvm_printer* printer = NULL );
```

The member function **print_topology** shall print the verification environment's component topology. The argument *printer* shall be an object of class **uvm_printer** that controls the format of the topology printout; a NULL printer prints with the default output.

4.3.4 Data member (variables)

4.3.4.1 enable_print_topology

```
bool enable_print_topology;
```

If the data member **enable_print_topology** is set to true, then the entire testbench topology is printed just after completion of the end_of_elaboration phase. By default, this data member is set to false (no topology printed).

4.3.4.2 finish_on_completion

```
bool finish_on_completion;
```

If the data member **finish_on_completion** is set to true, then **sc_core**::**sc_stop** is called after all phases are executed. By default, this data member is set to true.

4.3.4.3 uvm_top

```
const uvm_root* uvm_top;
```

The data member **uvm_top** is a handle to the top-level (root) component that governs phase execution and provides the component search interface. By default, this handle is provided by the **uvm_root** singleton.

The **uvm_top** instance of **uvm_root** plays several key roles in the UVM:

- *Implicit top-level*: The **uvm_top** serves as an implicit top-level component. Any UVM component which is not instantiated in another UVM component (e.g. when instantiated in an **sc_core**::**sc_module** or in **sc_main**) becomes a child of **uvm_top**. Thus, all UVM components in simulation are descendants of **uvm_top**.
- *Phase control*: **uvm_top** manages the phasing for all components.
- *Search*: An application may use **uvm_top** to search for components based on their hierarchical name. See member functions **find** and **find_all**.
- *Report configuration*: An application may use **uvm_top** to globally configure report verbosity, log files, and actions. For example, **uvm_top**.**set_report_verbosity_level_hier**(**UVM_FULL**) would set full verbosity for all components in simulation.
- *Global reporter*: Because **uvm_top** is globally accessible, the UVM reporting mechanism is accessible from anywhere outside **uvm_component**, such as in modules and sequences. See **uvm_report_error**, **uvm_report_warning**, and other global methods.

The **uvm_top** instance checks during the **end_of_elaboration_phase** if any errors have been generated so far. If errors are found a **UVM_FATAL** error is being generated as result so that the simulation will not continue to the **start_of_simulation_phase**. CHECK

4.4 uvm_port_base

The class **uvm_port_base** shall provide methods to bind ports to interfaces or to other ports or exports, and to forward interface method calls to the channel to which the port is bound, according to the same mechanism as defined in SystemC. Therefore this class shall be derived from the class **sc_core**::**sc_port**.

4.4.1 Class definition

```
namespace uvm {


  template <class IF>
  class uvm_port_base : public sc_core::sc_port<IF>
  {
   public:
    uvm_port_base();
    explicit uvm_port_base( const std::string& name );


    virtual const std::string get_name() const;
```

```
    virtual const std::string get_full_name() const;
    virtual uvm_component* get_parent() const;
    virtual const std::string get_type_name() const;

    virtual void connect( IF& );
    virtual void connect( uvm_port_base<IF>& );

  // class uvm_port_base

} // namespace uvm
```

4.4.2 Template parameter IF

The template parameter IF shall specify the name of the interface type used for the port. The port can only be bound to a channel which is derived from the same type, or to another port or export which is derived from this type.

4.4.3 Constructor

```
uvm_port_base();
explicit uvm_port_base( const std::string& name );
```

The constructor shall create and initialize an instance of the class with the name *name*, if passed as an argument.

4.4.4 Member functions

4.4.4.1 get_name

```
virtual const std::string get_name() const;
```

The member function **get_name** shall return the leaf name of this port.

4.4.4.2 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the full hierarchical name of this port.

4.4.4.3 get_parent

```
virtual uvm_component* get_parent() const;
```

The member function **get_parent** shall return the handle to this port's parent, or NULL if it has no parent.

4.4.4.4 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name to this port. Derived port classes shall implement this member function to return the concrete type.

4.4.4.5 connect

```
virtual void connect( IF& );
virtual void connect( uvm_port_base<IF>& );
```

The member function **connect** shall bind this port to the interface given as argument.

NOTE–The member function **connect** implements the same functionality as the SystemC member function **bind**.

4.5 uvm_component_name

The class **uvm_component_name** is shall provide the mechanism for building the hierarchical names of component instances and component hierarchy during elaboration.

An implementation shall maintain the UVM component hierarchy, that is, it shall build a list of hierarchical component names, where each component instance is named as if it were a child of another component (its parent). The mechanism to implement such component hierarchy is implementation-defined.

NOTE 1–The hierarchical name of an instance in the component hierarchy is returned from member function **get_full_name** of class **uvm_component**, which is the base class of all component instances.

NOTE 2–An object of type **uvm_object** may have a hierarchical name and may have a parent in the component hierarchy, but such object is not part of the component hierarchy.

4.5.1 Class definition

```
namespace uvm {


  class uvm_component_name
  {
   public:
    uvm_component_name( const char* name );
    uvm_component_name( const uvm_component_name& name );
    ~uvm_component_name();
    operator const char*() const;


   private:
    // Disabled
    uvm_component_name();
    uvm_component_name& operator= ( const uvm_component_name& name );
  }; // class uvm_component_name


} // namespace uvm
```

4.5.2 Constraints on usage

The class **uvm_component_name** shall only be used as argument in a constructor of a class derived from class **uvm_component**. Such constructor shall only contain this argument of type **uvm_component_name**.

4.5.3 Constructor

```
uvm_component_name( const char* name );
```

The constructor **uvm_component_name(** const char* *name*) shall store the name in the component hierarchy. The constructor argument *name* shall be used as the string name for that component being instantiated within the component hierarchy.

NOTE–An application shall define for each class derived directly or indirectly from class **uvm_component** a constructor with a single argument of type **uvm_component_name,** where the constructor **uvm_component_name(** const char*) is called as an implicit conversion.

```
uvm_component_name( const uvm_componet_name& name );
```

The constructor **uvm_component_name(** const **uvm_component_name**& *name*) shall copy the constructor argument but shall not modify the component hierarchy.

NOTE–When an application derives a class directly or indirectly from class **uvm_component**, the derived class constructor calls the base class constructor with an argument of class **uvm_component_name** and thus this copy constructor is called.

4.5.4 Destructor

```
~uvm_component_name();
```

The destructor shall remove the object from the component hierarchy if, and only if, the object being destroyed was constructed by using the constructor signature **uvm_component_name(** const char* *name*).

4.5.5 operator const char*

```
operator const char*() const;
```

This conversion function shall return the string name (not the hierarchical name) associated with the **uvm_component_name**.

5. Policy classes

The UVM policy classes provide specific tasks for printing, comparing, recording, packing, and unpacking of objects derived from class **uvm_object**. They are implemented separately from class **uvm_object** so that an application can plug in different ways to print, compare, etc. without modifying the object class being operated on. The user can simply apply a different printer or compare "policy" to change how an object is printed or compared.

Each policy class includes several user-configurable parameters that control the operation. An application may also customize operations by deriving new policy subtypes from these base types. For example, the UVM provides four different printer policy classes derived from the policy base class **uvm_printer**, each of which print objects in a different format.

The following policy classes are defined:

- **uvm_packer**
- **uvm_printer**
- **uvm_recorder**
- **uvm_comparer**

5.1 uvm_packer

The class **uvm_packer** provides a policy object for packing and unpacking objects of type **uvm_object**. The policies determine how packing and unpacking should be done. Packing an object causes the object to be placed into a packed array of type byte or int. Unpacking an object causes the object to be filled from the pack array. The logic values X and Z are lost on packing. The maximum size of the packed array is limited to 4096.

5.1.1 Class definition

```
namespace uvm {


  class uvm_packer
  {
   public:
    uvm_packer();


    // Group: Packing
    virtual void pack_field( const uvm_bitstream_t& value, int size );
    virtual void pack_field_int( const uvm_integral_t& value, int size );
    virtual void pack_string( const std::string& value );
    virtual void pack_time( const sc_core::sc_time& value );
    virtual void pack_real( double value );
    virtual void pack_real( float value );
    virtual void pack_object( const uvm_object& value );
    virtual uvm_packer& operator<< ( bool value );
```

```
virtual uvm_packer& operator<< ( double& value );
virtual uvm_packer& operator<< ( float& value );
virtual uvm_packer& operator<< ( char value );
virtual uvm_packer& operator<< ( unsigned char value );
virtual uvm_packer& operator<< ( short value );
virtual uvm_packer& operator<< ( unsigned short value );
virtual uvm_packer& operator<< ( int value );
virtual uvm_packer& operator<< ( unsigned int value );
virtual uvm_packer& operator<< ( long value );
virtual uvm_packer& operator<< ( unsigned long value );
virtual uvm_packer& operator<< ( long long value );
virtual uvm_packer& operator<< ( unsigned long long value );
virtual uvm_packer& operator<< ( const std::string& value );
virtual uvm_packer& operator<< ( const char* value );
virtual uvm_packer& operator<< ( const uvm_object& value );
virtual uvm_packer& operator<< ( const sc_dt::sc_logic& value );
virtual uvm_packer& operator<< ( const sc_dt::sc_bv_base& value );
virtual uvm_packer& operator<< ( const sc_dt::sc_lv_base& value );
virtual uvm_packer& operator<< ( const sc_dt::sc_int_base& value );
virtual uvm_packer& operator<< ( const sc_dt::sc_uint_base& value );
virtual uvm_packer& operator<< ( const sc_dt::sc_signed& value );
virtual uvm_packer& operator<< ( const sc_dt::sc_unsigned& value );

template <class T>
uvm_packer& operator<< ( const std::vector<T>& value );

// Group: Unpacking
virtual bool is_null();
virtual uvm_integral_t unpack_field_int( int size );
virtual uvm_bitstream_t unpack_field( int size );
virtual std::string unpack_string( int num_chars = -1 );
virtual sc_core::sc_time unpack_time();
virtual double unpack_real();
virtual float unpack_real();
virtual void unpack_object( uvm_object& value );
virtual unsigned int get_packed_size() const;

virtual uvm_packer& operator>> ( bool& value );
```

```
    virtual uvm_packer& operator>> ( double& value );
    virtual uvm_packer& operator>> ( float& value );
    virtual uvm_packer& operator>> ( char& value );
    virtual uvm_packer& operator>> ( unsigned char& value );
    virtual uvm_packer& operator>> ( short& value );
    virtual uvm_packer& operator>> ( unsigned short& value );
    virtual uvm_packer& operator>> ( int& value );
    virtual uvm_packer& operator>> ( unsigned int& value );
    virtual uvm_packer& operator>> ( long& value );
    virtual uvm_packer& operator>> ( unsigned long& value );
    virtual uvm_packer& operator>> ( long long& value );
    virtual uvm_packer& operator>> ( unsigned long long& value );
    virtual uvm_packer& operator>> ( std::string& value );
    virtual uvm_packer& operator>> ( uvm_object& value );
    virtual uvm_packer& operator>> ( sc_dt::sc_logic& value );
    virtual uvm_packer& operator>> ( sc_dt::sc_bv_base& value );
    virtual uvm_packer& operator>> ( sc_dt::sc_lv_base& value );
    virtual uvm_packer& operator>> ( sc_dt::sc_int_base& value );
    virtual uvm_packer& operator>> ( sc_dt::sc_uint_base& value );
    virtual uvm_packer& operator>> ( sc_dt::sc_signed& value );
    virtual uvm_packer& operator>> ( sc_dt::sc_unsigned& value );

    template <class T>
    virtual uvm_packer& operator>> ( std::vector<T>& value );

    // Data members (variables)
    bool physical;
    bool abstract;
    bool use_metadata;
    bool big_endian;

  }; // class uvm_packer

} // namespace uvm
```

5.1.2 Packing

5.1.2.1 pack_field

```
virtual void pack_field( const uvm_bitstream_t& value, int size );
```

The member function **pack_field** shall pack an integral value (less than or equal to 4096 bits) into the packed array. The argument *size* is the number of bits of value to pack.

5.1.2.2 pack_field_int

```
virtual void pack_field_int( const uvm_integral_t& value, int size );
```

The member function **pack_field_int** shall pack the integral value (less than or equal to 64 bits) into the packed array. The argument *size* is the number of bits of value to pack.

NOTE–This member function is the optimized version of **pack_field** is useful for sizes up to 64 bits.

5.1.2.3 pack_string

```
virtual void pack_string( const std::string& value );
```

The member function **pack_string** shall pack a string value into the packed array. When the variable **metadata** is set, the packed string is terminated by a NULL character to mark the end of the string.

5.1.2.4 pack_time

```
virtual void pack_time( const sc_core::sc_time& value );
```

The member function **pack_time** shall pack a time value as 64 bits into the packed array.

5.1.2.5 pack_real

```
virtual void pack_real( double value );
virtual void pack_real( float value );
```

The member function **pack_real** shall pack a real value as binary vector into the packed array. When the argument is a double precision floating point value of type double, a 64 bit binary vector shall be used. When the argument is a single precision floating point value of type float, a 32 bit binary vector shall be used. The convertion of the floating point representation to binary vector shall be according to the IEEE Standard for Floating-Point Arithmetic (IEEE Std. 754-1985).

5.1.2.6 pack_object

```
virtual void pack_object( const uvm_object& value );
```

The member function **pack_object** shall pack an object value into the packed array. A 4-bit header is inserted ahead of the string to indicate the number of bits that was packed. If a NULL object was packed, then this header will be 0.

5.1.3 Unpacking

5.1.3.1 is_null

```
virtual bool is_null();
```

The member function **is_null** shall be used during unpack operations to peek at the next 4-bit chunk of the pack data and determine if it is 0. If the next four bits are all 0, then the return value is a true; otherwise it returns false.

NOTE–This member function is useful when unpacking objects, to decide whether a new object needs to be allocated or not.

5.1.3.2 unpack_field_int

```
virtual uvm_integral_t unpack_field_int( int size );
```

The member function **unpack_field_int** shall unpack bits from the packed array and returns the bit-stream that was unpacked. The argument *size* is the number of bits to unpack; the maximum is 64 bits.

NOTE–This member function is a more efficient variant than **unpack_field** when unpacking into smaller vectors.

5.1.3.3 unpack_field

```
virtual uvm_bitstream_t unpack_field( int size );
```

The member function **unpack_field** shall unpack bits from the packed array and returns the bit-stream that was unpacked. The argument *size* is the number of bits to unpack; the maximum is 4096 bits.

5.1.3.4 unpack_string

```
virtual std::string unpack_string( int num_chars = -1 );
```

The member function **unpack_string** shall unpack a string. The argument *num_chars* specifies the number of bytes that are unpacked into a string. If *num_chars* is -1, then unpacking stops on at the first NULL character that is encountered.

5.1.3.5 unpack_time

```
virtual sc_core::sc_time unpack_time();
```

The member function **unpack_time** shall unpack the next 64 bits of the packed array and places them into a time variable.

5.1.3.6 unpack_real

```
virtual double unpack_real();
virtual float unpack_real();
```

The member function **unpack_real** shall unpack the next 64 bits of the packed array and places them into a real variable. The 64 bits of packed data shall be converted to double precision floating point notation according to the IEEE Standard for Floating-Point Arithmetic (IEEE Std. 754-1985).

5.1.3.7 unpack_object

```
virtual void unpack_object( uvm_object& value );
```

The member function **unpack_object** shall unpack an object and stores the result into *value*. Argument *value* must be an allocated object that has enough space for the data being unpacked. The first four bits of packed data are used to determine if a null object was packed into the array. The member function **is_null** can be used to peek at the next four bits in the pack array before calling this member function.

5.1.3.8 get_packed_size

```
virtual unsigned int get_packed_size() const;
```

The member function **get_packed_size** returns the number of bits that were packed.

5.1.4 operator <<, operator>>

The class **uvm_packer** defines **operator**<< for packing, and **operator** >> for unpacking basic C++ types, SystemC types, the type **uvm_object**, and std::vector types. The supported data types are:

- Basic C++ types: bool, double, float, char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long.
- SystemC types: **sc_dt**::**sc_logic**, **sc_dt**::**sc_bv**, **sc_dt**::**sc_lv**, **sc_dt**::**sc_int**, **sc_dt**::**sc_uint**, **sc_dt**::**sc_signed**, and **sc_dt**::**sc_unsigned**.
- String of type std::string and const char*
    - When packing, an additional NULL byte is packed after the string is packed. CHECK
- Any type that derives from class **uvm_object**
- Vector types: std::vector<T>, where T is one of the supported data types listed above, and has an **operator**<< defined for it:
    - When packing, additional 32 bits are packed indicating the size of the vector, prior to packing individual elements.

An application may use **operator**<< or **operator**>> for the implementation of the member function **do_pack** and **do_unpack** as part of an application-specific object definition derived from class **uvm_object**.

5.1.5 Data members (variables)

5.1.5.1 physical

```
bool physical;
```

The data member **physical** shall provides a filtering mechanism for fields. The abstract and physical settings allow an object to distinguish between two different classes of fields. An application may, in the member functions **uvm_object**::**do_pack** and **uvm_object**::**do_unpack**, test the setting of this field, to use it as a filter. By default, the data member **physical** is set to true in the constructor of **uvm_packer**.

5.1.5.2 abstract

```
bool abstract;
```

The data member **abstract** shall provides a filtering mechanism for fields. The abstract and physical settings allow an object to distinguish between two different classes of fields. An application may, in the member functions **uvm_object**::**do_pack** and **uvm_object**::**do_unpack**, test the setting of this field, to use it as a filter. By default, the data member **abstract** is set to false in the constructor of **uvm_packer**.

5.1.5.3 use_metadata

```
bool use_metadata;
```

The data member **use_metadata** shall indicate whether to encode metadata when packing dynamic data, or to decode metadata when unpacking. Implementations of **uvm_object**::**do_pack** and **uvm_object**::**do_unpack** should regard this bit when performing their respective operation. When set to true, metadata should be encoded as follows:

- For strings, pack an additional NULL byte after the string is packed.
- For objects, pack 4 bits prior to packing the object itself. Use 0b0000 to indicate the object being packed is null, otherwise pack 0b0001 (the remaining 3 bits are reserved).
- For queues, dynamic arrays, and associative arrays, pack 32 bits indicating the size of the array prior to to packing individual elements.

5.1.5.4 big_endian

```
bool big_endian;
```

The data member **big_endian** shall determine the order that integral data is packed (using the member functions **pack_field**, **pack_field_int**, **pack_time**, or **pack_real**) and how the data is unpacked from the pack array (using the member functions **unpack_field**, **unpack_field_int**, **unpack_time**, or **unpack_real**). By default, the data member is set to true in the constructor of **uvm_packer**. When the data member is set, data is associated msb to lsb; otherwise, it is associated lsb to msb.

5.2 Printer policy classes

The class **uvm_printer** provides an interface for printing objects of type **uvm_object** in various formats.

Classes derived from class **uvm_printer** implement pre-defined printing formats or policies:

- The class **uvm_table_printer** prints the object in a tabular form.
- The class **uvm_tree_printer** prints the object in a tree form.
- The class **uvm_line_printer** prints the information on a single line, but uses the same object separators as the tree printer.

These printer classes have ‘knobs’ that an application may use to control what and how information is printed. These knobs are contained in a separate knob class **uvm_printer_knobs**

5.2.1 uvm_printer

The class **uvm_printer** shall provide the basic printer functionality, which shall be overloaded by derived classes to support various pre-defined printing formats.

5.2.1.1 Class definition

```
namespace uvm {


  class uvm_printer
  {
   public:
    uvm_printer();
    virtual ~uvm_printer();


    // Group: Printing types


    virtual void print_field( const std::string& name,
                              const uvm_bitstream_t& value,
                              int size = -1,
                              uvm_radix_enum radix = UVM_NORADIX,
                              const char* scope_separator = ".",
                              const std::string& type_name = "" ) const;


    virtual void print_field_int( const std::string& name,
                                  const uvm_integral_t& value,
                                  int size = -1,
                                  uvm_radix_enum radix = UVM_NORADIX,
                                  const char* scope_separator = ".",
                                  const std::string& type_name = "" ) const;


    virtual void print_real( const std::string& name,
                             double value,
                             const char* scope_separator = "." ) const;


    virtual void print_real( const std::string& name,
                             float value,
                             const char* scope_separator = "." ) const;


    virtual void print_object( const std::string& name,
```

```
                              uvm_object* value,
                              const char* scope_separator = "." ) const;

  virtual void print_object_header( const std::string& name, not in UVM-SV LRM??
                                    uvm_object* value,
                                    const char* scope_separator = "." ) const;

  virtual void print_string( const std::string& name,
                             const std::string& value,
                             const char* scope_separator = "." ) const;

  virtual void print_time( const std::string& name,
                           const sc_core::sc_time& value,
                           const char* scope_separator = "." ) const;

  virtual void print_generic( const std::string& name,
                              const std::string& type_name,
                              int size,
                              const std::string& value,
                              const char* scope_separator = "." ) const;

  // Group: Printer subtyping
  virtual std::string emit();
  virtual std::string format_row( const uvm_printer_row_info& row );
  virtual std::string format_header();
  virtual std::string format_footer();

  std::string adjust_name( const std::string& id,
                           const char* scope_separator = "." ) const;

  virtual void print_array_header( const std::string& name,
                                   int size,
                                   const std::string& arraytype = "array",
                                   const char* scope_separator = "." ) const;

  void print_array_range( int min, int max ) const;
  void print_array_footer( int size = 0 ) const;
```

```
    // Data members
    uvm_printer_knobs knobs;


  }; // class uvm_printer


} // namespace uvm
```

5.2.2 Printing types

5.2.2.1 print_field

```
virtual void print_field( const std::string& name,
                          const uvm_bitstream_t& value,
                          int size = -1, // default value not in UVM standard
                          uvm_radix_enum radix = UVM_NORADIX,
                          const char* scope_separator = ".",
                          const std::string& type_name = "" );
```

The member function **print_field** shall print a field of type **uvm_bitstream_t**. The argument *name* defines the name of the field. The argument *value* contains the value of the field. The argument *size* defines the number of bits of the field. The argument radix defined radix to use for printing. The printer knob for radix is used if no radix is specified. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of a field. Typical values for the separator are a “.” (dot) or “[” (open bracket).

5.2.2.2 print_field_int

```
virtual void print_field_int( const std::string& name,
                              const uvm_integral_t& value,
                              int size = -1, // default value not in UVM standard
                              uvm_radix_enum radix = UVM_NORADIX,
                              const char* scope_separator = ".",
                              const std::string& type_name = "" );
```

The member function **print_field_int** shall print an integer field. The argument *name* defines the name of the field. The argument *value* contains the value of the field. The argument *size* defines the number of bits of the field. The argument radix defined radix to use for printing. The printer knob for radix is used if no radix is specified. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of a field. Typical values for the separator are a “.” (dot) or “[” (open bracket).

5.2.2.3 print_real

```
virtual void print_real( const std::string& name,
                         double value,
```

```
                          const char* scope_separator = "." );
```

The member function **print_real** shall print a real (double) field. The argument *name* defines the name of the field. The argument *value* contains the value of the field. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of a field.

5.2.2.4 print_double

```
virtual void print_double( const std::string& name,
                           double value,
                           const char* scope_separator = "." );
```

The member function **print_double** shall print a real (double) field. The argument *name* defines the name of the field. The argument *value* contains the value of the field. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of a field.

NOTE–This member function has been introduced to be more compatible with C++/SystemC coding styles and types. The member function has similar functionality as **print_real**.

5.2.2.5 print_object

```
virtual void print_object( const std::string& name,
                           const uvm_object& value,
                           const char* scope_separator = "." ) const;
```

The member function **print_object** shall print an object. The argument *name* defines the name of the object. The argument *value* contains the reference to the object. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of the object.

Whether the object is recursed depends on a variety of knobs, such as the depth knob; if the current depth is at or below the depth setting, then the object is not recursed. By default, the children of objects of type **uvm_component** are printed. To disable automatically printing of these objects, an application can set the member function **uvm_component::print_enabled** to false for the specific children to be excluded from printing.

5.2.2.6 print_object_header

```
virtual void print_object_header( const std::string& name, not in UVM-SV LRM??
                                  const uvm_object& value,
                                  const char* scope_separator = "." ) const;
```

The member function **print_object_header** shall print an object header. The argument *name* defines the name of the object. The argument *value* contains the reference to the object. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of a field.

5.2.2.7 print_string

```
virtual void print_string( const std::string& name,
                           const std::string& value,
```

```
                              const char* scope_separator = "." );
```

The member function **print_string** shall print a string field. The argument *name* defines the name of the field. The argument *value* contains the value of the field. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of a field.

5.2.2.8 print_time

```
virtual void print_time( const std::string& name,
                         const sc_core::sc_time& value,
                         const char* scope_separator = "." );
```

The member function **print_time** shall print the time. The argument *name* defines the name of the field. The argument *value* contains the value of the field. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of a field.

5.2.2.9 print_generic

```
virtual void print_generic( const std::string& name,
                            const std::string& type_name,
                            int size,
                            const std::string& value,
                            const char* scope_separator = "." );
```

The member function **print_generic** shall print a field using the arguments *name*, *type_name*, *size*, and *value*. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of a field.

5.2.3 Printer subtyping

5.2.3.1 emit

```
virtual std::string emit();
```

The member **emit** shall return a string representing the contents of an object in a format defined by an extension of this object.

5.2.3.2 format_row

```
virtual std::string format_row( const uvm_printer_row_info& row );
```

The member **format_row** shall offer a hook for producing custom output of a single field (row).

5.2.3.3 format_header

```
virtual std::string format_header();
```

The member **format_header** shall offer a hook to override the base header with a custom header.

5.2.3.4 format_footer

```
virtual std::string format_footer();
```

The member **format_footer** shall offer a hook to override the base footer with a custom footer.

5.2.3.5 adjust_name

```
std::string adjust_name( const std::string& id,
                         const char* scope_separator = "." ) const;
```

The member function **adjust_name** shall print a field's name, or *id*, which is the full instance name. The intent of the separator is to mark where the leaf name starts if the printer is configured to print only the leaf name of the identifier.

5.2.3.6 print_array_header

```
virtual void print_array_header( const std::string& name,
                                 int size,
                                 const std::string& arraytype = "array",
                                 const char* scope_separator = "." ) const;
```

The member function **print_array_header** shall print the header of an array. This member function shall be called before each individual element is printed. The member function **print_array_footer** shall be called to mark the completion of array printing.

5.2.3.7 print_array_range

```
void print_array_range( int min, int max ) const;
```

The member function **print_array_range** shall print a range using ellipses for values. This method is used when honoring the array knobs for partial printing of large arrays, **uvm_printer_knobs**::**begin_elements** and **uvm_printer_knobs**::**end_elements**. This member function should be called after **uvm_printer_knobs**::**begin_elements** have been printed and before **uvm_printer_knobs**::**end_elements** have been printed.

5.2.3.8 print_array_footer

```
void print_array_footer( int size = 0 ) const;
```

The member function **print_array_footer** shall print the footer of an array. This member function marks the end of an array print. Generally, there is no output associated with the array footer, but this method lets the printer know that the array printing is complete.

5.2.4 Data members

5.2.4.1 knobs

```
uvm_printer_knobs knobs;
```

The data member **knobs** shall provide access to the variety of knobs associated with a specific printer instance.

5.3 uvm_comparer

The class **uvm_comparer** shall provide a policy object for doing comparisons. The policies determine how miscompares are treated and counted. Results of a comparison are stored in the comparer object. The member functions **uvm_object::compare** and **uvm_object::do_compare** are passed a **uvm_comparer** policy object.

5.3.1 Class definition

```
namespace uvm {


  class uvm_comparer
  {
   public:
    // Group: member functions
    virtual bool compare_field( const std::string& name,
                                const uvm_bitstream_t& lhs,
                                const uvm_bitstream_t& rhs,
                                int size,
                                uvm_radix_enum radix = UVM_NORADIX ) const;

    virtual bool compare_field_int( const std::string& name,
                                    const uvm_integral_t& lhs,
                                    const uvm_integral_t& rhs,
                                    int size,
                                    uvm_radix_enum radix = UVM_NORADIX ) const;

    virtual bool compare_field_real( const std::string& name,
                                     double lhs,
                                     double rhs ) const;

    virtual bool compare_field_real( const std::string& name,
                                     float lhs,
                                     float rhs ) const;

    virtual bool compare_object( const std::string& name,
                                 const uvm_object& lhs,
                                 const uvm_object& rhs ) const;
```

```
    virtual bool compare_string( const std::string& name,
                                 const std::string& lhs,
                                 const std::string& rhs ) const;

    void print_msg( const std::string& msg ) const;

    // Group: data members (variables)
    uvm_recursion_policy_enum policy;
    unsigned int show_max;
    unsigned int verbosity;
    uvm_severity sev;
    mutable std::string miscompares; // ok to make mutable?
    bool physical;
    bool abstract;
    bool check_type;
    mutable unsigned int result; // ok to make mutable?
  }; // class uvm_comparer

} // namespace uvm
```

5.3.2 Member functions

5.3.2.1 compare_field

```
virtual bool compare_field( const std::string& name,
                            const uvm_bitstream_t& lhs,
                            const uvm_bitstream_t& rhs,
                            int size,
                            uvm_radix_enum radix = UVM_NORADIX ) const;
```

The member function **compare_field** shall compare two integral values. The argument *name* is used for purposes of storing and printing a miscompare. The left-hand-side *lhs* and right-hand-side *rhs* objects are the two objects used for comparison. The size variable indicates the number of bits to compare; size must be less than or equal to 64. The argument *radix* is used for reporting purposes, the default radix is hex. CHECK

5.3.2.2 compare_field_int

```
virtual bool compare_field_int( const std::string& name,
                                const uvm_integral_t& lhs,
                                const uvm_integral_t& rhs,
                                int size,
```

```
                                    uvm_radix_enum radix = UVM_NORADIX ) const;
```

The member function **compare_field_int** shall compare two integral values. This member function is same as **compare_field** except that the arguments are small integers, less than or equal to 64 bits. It is automatically called by **compare_field** if the operand size is less than or equal to 64.

The argument *name* is used for purposes of storing and printing a miscompare. The left-hand-side *lhs* and right-hand-side *rhs* objects are the two objects used for comparison. The size variable indicates the number of bits to compare; size must be less than or equal to 64. The argument *radix* is used for reporting purposes, the default radix is hex.

5.3.2.3 compare_field_real

```
virtual bool compare_field_real( const std::string& name,
                                 double lhs,
                                 double rhs ) const;


virtual bool compare_field_real( const std::string& name,
                                 float lhs,
                                 float rhs ) const;
```

The member function **compare_field_real** shall compare two real numbers, represented by type double or float. The left-hand-side *lhs* and right-hand-side *rhs* arguments are used for comparison.

5.3.2.4 compare_object

```
virtual bool compare_object( const std::string& name,
                             const uvm_object& lhs,
                             const uvm_object& rhs ) const;
```

The member function **compare_object** shall compare two class objects using the data member *policy* to determine whether the comparison should be deep, shallow, or reference. The argument *name* is used for purposes of storing and printing a miscompare. The *lhs* and *rhs* objects are the two objects used for comparison. The data member *check_type* determines whether or not to verify the object types match (the return from *lhs*.**get_type_name**() matches *rhs*.**get_type_name**()).

5.3.2.5 compare_string

```
virtual bool compare_string( const std::string& name,
                             const std::string& lhs,
                             const std::string& rhs ) const;
```

The member function **compare_string** shall compare two two string variables. The argument *name* is used for purposes of storing and printing a miscompare. The *lhs* and *rhs* objects are the two objects used for comparison.

5.3.2.6 print_msg

```
void print_msg( const std::string& msg ) const;
```

The member function **print_msg** shall cause the error count to be incremented and the message passed as argument to be appended to the miscompares string (a newline is used to separate messages). If the message count is less than the data member *show_max* setting, then the message is printed to standard-out using the current verbosity (see 5.3.3.3) and severity (see 5.3.3.4) settings.

5.3.3 Data members (variables)

5.3.3.1 policy

```
uvm_recursion_policy_enum policy;
```

The data member **policy** determines whether comparison is **UVM_DEEP**, **UVM_REFERENCE**, or **UVM_SHALLOW**. The default policy shall be set to **UVM_DEFAULT_POLICY**.

5.3.3.2 show_max

```
unsigned int show_max;
```

The data member **show_max** sets the maximum number of messages to send to the printer for miscompares of an object. The default number of messages shall be set to one.

5.3.3.3 verbosity

```
unsigned int verbosity;
```

The data member **verbosity** shall set the verbosity for printed messages. The verbosity setting is used by the messaging mechanism to determine whether messages should be suppressed or shown. The default verbosity shall be set to **UVM_LOW**.

5.3.3.4 sev

```
uvm_severity sev;
```

The data member **sev** shall set the severity for printed messages. The severity setting is used by the messaging mechanism for printing and filtering messages. The default severity shall be set to **UVM_INFO**.

5.3.3.5 miscompares

```
mutable std::string miscompares;
```

The data member **miscompares** shall contain the miscompare string. This string is reset to an empty string when a comparison is started. The string holds the last set of miscompares that occurred during a comparison. The default miscompares string shall be empty.

5.3.3.6 physical

```
bool physical;
```

The data member **physical** shall provide a filtering mechanism for fields. The abstract and physical settings allow an object to distinguish between two different classes of fields. An application shall use the member function **uvm_object**::**do_compare** to test the setting of this field if it wants to use the physical trait as a filter. By default, the data member **physical** shall be set to true.

5.3.3.7 abstract

```
bool abstract;
```

The data member **abstract** shall provide a filtering mechanism for fields. The abstract and physical settings allow an object to distinguish between two different classes of fields. An application shall use the member function **uvm_object**::**do_compare** to test the setting of this field if it wants to use the abstract trait as a filter. By default, the data member **abstract** shall be set to true.

5.3.3.8 check_type

```
bool check_type;
```

The data member **check_type** shall determine whether the type, given by **uvm_object**::**get_type_name**, is used to verify that the types of two objects are the same. This data member is used by the member function **compare_object**. In some cases it is useful to set this data member to false, when the two operands are related by inheritance but are different types.

5.3.3.9 result

```
mutable unsigned int result;
```

The data member **result** shall store the number of miscompares for a given compare operation. An application can use the result to determine the number of miscompares that were found.

5.4 Default policy objects

This section lists the default policy objects.

5.4.1.1 uvm_default_table_printer

```
extern uvm_table_printer* uvm_default_table_printer;
```

The global object **uvm_default_table_printer** shall define a handle to an object of type **uvm_table_printer**, which can be used with **uvm_object**::**do_print** to get tabular style printing.

5.4.1.2 uvm_default_tree_printer

```
extern uvm_tree_printer* uvm_default_tree_printer;
```

The global object **uvm_default_tree_printer** shall define a handle to an object of type **uvm_tree_printer**, which can be used with **uvm_object**::**do_print** to get a multi-line tree style printing.

5.4.1.3 uvm_default_line_printer

```
extern uvm_line_printer* uvm_default_line_printer;
```

The global object **uvm_default_line_printer** shall define a handle to an object of type **uvm_line_printer**, which can be used with **uvm_object**::**do_print** to get a single-line style printing.

5.4.1.4 uvm_default_printer

```
extern uvm_printer* uvm_default_printer;
```

The global object **uvm_default_printer** shall define the default printer policy, which shall be set to **uvm_default_table_printer.** An application can redefine the default printer, by setting it to any legal **uvm_printer** derived type, including the global line, tree, and table printers in the previous sections.

5.4.1.5 uvm_default_packer

```
extern uvm_printer* uvm_default_printer;
```

The global object **uvm_default_packer** shall define the default packer policy. It shall be used when calls to **uvm_object**::**pack** and **uvm_object**::**unpack** do not specify a packer policy.

5.4.1.6 uvm_default_comparer

```
extern uvm_comparer* uvm_default_comparer;
```

The global object **uvm_default_comparer** shall define the default comparer policy. It shall be used when calls to **uvm_object**::**compare** do not specify a comparer policy.

5.4.1.7 uvm_default_recorder

```
extern uvm_recorder* uvm_default_recorder;
```

The global object **uvm_default_recorder** shall define the default recorder policy. It shall be used when calls to **uvm_object**::**record** do not specify a recorder policy.

6. Registry and factory classes

The registry and factory classes offer the interface to register and use UVM objects and components via the factory.

The following classes are defined:

- **uvm_object_wrapper**
- **uvm_object_registry**
- **uvm_component_registry**
- **uvm_factory**

The class **uvm_object_wrapper** forms the base class for the registry classes **uvm_object_registry** and **uvm_component_registry**, which act as lightweight proxies for UVM objects and components, respectively.

UVM object and component types are registered with the factory via typedef or macro invocation. When the application requests a new object or component from the factory, the factory will determine what type of object to create based on its configuration, and will ask that type's proxy to create an instance of the type, which is returned to the application.

6.1 uvm_object_wrapper

The class **uvm_object_wrapper** shall provide an abstract interface for creating object and component proxies. Instances of these lightweight proxies, representing every object or component derived from **uvm_object** or **uvm_component** respectively in the test environment, are registered with the **uvm_factory**. When the factory is called upon to create an object or component, it shall find and delegate the request to the appropriate proxy.

6.1.1 Class definition

```
namespace uvm {


  class uvm_object_wrapper
  {
   public:
    virtual uvm_object* create_object( const std::string& name = "" );
    virtual uvm_component* create_component( const std::string& name,
                                             uvm_component* parent );
    virtual const std::string get_type_name() const = 0;
  };


} // namespace uvm
```

6.1.2 Member functions

6.1.2.1 create_object

```
virtual uvm_object* create_object( const std::string& name = "" );
```

The member function **create_object** shall create a new object with the optional name passed as argument. An object proxy (e.g., **uvm_object_registry**<T>) implements this member function to create an object of a specific type, T (see 6.2).

6.1.2.2 create_component

```
virtual uvm_object* create_object( const std::string& name = "" );
```

The member function **create_component** shall create a new component, by passing to its constructor the given name and parent. The component proxy (e.g. **uvm_component_registry**<T>) implements this member function to create a component of a specific type, T (see 6.2.2).

6.1.2.3 get_type_name

```
virtual const std::string get_type_name() const = 0;
```

The implementation of the pure virtual member function **get_type_name** shall return the type name of the object created by **create_component** or **create_object**. The factory uses this name when matching against the requested type in name-based lookups.

6.2 uvm_object_registry

The class **uvm_object_registry** shall provide a lightweight proxy for a **uvm_object** of type T. The proxy enables efficient registration with the **uvm_factory**. Without it, registration would require an instance of the object itself.

The macros **UVM_OBJECT_UTILS** or **UVM_OBJECT_PARAM_UTILS** shall create the appropriate class **uvm_object_registry** necessary to register that particular object wth the factory.

6.2.1 Class definition

```
namespace uvm {


  template <typename T = uvm_object>
  class uvm_object_registry<T> : public uvm_object_wrapper
  {
   public:
    virtual uvm_object* create_object( const std::string& name = "" );
    virtual const std::string get_type_name() const;
    static uvm_object_registry<T>* get();


    static T* create( const std::string& name = "",
```

```
                      uvm_component* parent = NULL,
                      const std::string& contxt = "" );

    static void set_type_override( uvm_object_wrapper* override_type,
                                   bool replace = true );

    static void set_inst_override( uvm_object_wrapper* override_type,
                                   const std::string& inst_path,
                                   uvm_component* parent = NULL );
  }; // class uvm_object_registry

} // namespace uvm
```

6.2.2 Template parameter T

The template parameter T specifies the object type of the objects being registered. The object type must be a derivative of class **uvm_object**.

6.2.3 Member functions

6.2.3.1 create_object

```
virtual uvm_object* create_object( const std::string& name = "" );
```

The member function **create_object** shall create an object of type T and returns it as a handle to a **uvm_object**. This is an overload of the member function in **uvm_object_wrapper**. It is called by the factory after determining the type of object to create. An application shall not call this member function directly. Instead, an application shall call the static member function **create**.

6.2.3.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the object. This member function overloads the member function in **uvm_object_wrapper**.

6.2.3.3 get

```
static uvm_object_registry<T>* get();
```

The member function **get** shall return the singleton instance of this type. Type-based factory operation depends on there being a single proxy instance for each registered type.

6.2.3.4 create

```
static T* create( const std::string& name = "",
```

```
                    uvm_component* parent = NULL,
                    const std::string& contxt = "" );
```

The member function **create** shall return a new instance of the object type, T, represented by this proxy, subject to any factory overrides based on the context provided by the parent's full name. The new instance shall have the given leaf name *name*, if provided as argument. The argument *contxt*, if supplied, supersedes the parent's context.

6.2.3.5 set_type_override

```
static void set_type_override( uvm_object_wrapper* override_type,
                               bool replace = true );
```

The member function **set_type_override** shall configure the factory to create an object of the type represented by *override_type* whenever a request is made to create an object of the type represented by this proxy, provided no instance override applies. The original type, T, is typically a super class of the override type.

When replace is true, a previous override on *original_type* is replaced, otherwise a previous override, if any, remains intact.

6.2.3.6 set_inst_override

```
static void set_inst_override( uvm_object_wrapper* override_type,
                               const std::string& inst_path,
                               uvm_component* parent = NULL );
```

The member function **set_inst_override** shall configure the factory to create an object of the type represented by argument *override_type* whenever a request is made to create an object of the type represented by this proxy, with matching instance paths. The original type, T, is typically a super class of the override type.

If argument *parent* is not specified, argument *inst_path* is interpreted as an absolute instance path, which enables instance overrides to be set from outside component classes. If argument *parent* is specified, argument *inst_path* is interpreted as being relative to the parent's hierarchical instance path. The argument *inst_path* may contain wildcards for matching against multiple contexts.

6.3 uvm_component_registry

The class **uvm_component_registry** shall provide a lightweight proxy for a **uvm_component** of type T. The proxy enables efficient registration with the **uvm_factory**. Without it, registration would require an instance of the component itself.

The macros **UVM_COMPONENT_UTILS** and **UVM_COMPONENT_PARAM_UTILS** shall create the appropriate class **uvm_component_registry** necessary to register that particular component with the factory.

6.3.1 Class definition

```
namespace uvm {


  template <typename T = uvm_component>
```

```
  class uvm_component_registry : public uvm_object_wrapper
  {
   public:
    virtual uvm_component* create_component( const std::string& name,
                                             uvm_component* parent );

    virtual const std::string get_type_name() const;
    static uvm_component_registry<T>* get();

    static T* create( const std::string& name = "",
                      uvm_component* parent = NULL,
                      const std::string& contxt = "" );

    static void set_type_override( uvm_object_wrapper* override_type,
                                   bool replace = true );

    static void set_inst_override( uvm_object_wrapper* override_type,
                                   const std::string& inst_path,
                                   uvm_component* parent = NULL );
  }; // class uvm_component_registry

} // namespace uvm
```

6.3.2 Template parameter T

The template parameter T specifies the object type of the components being registered. The object type must be a derivative of class **uvm_component**.

6.3.3 Member functions

6.3.3.1 create_component

```
virtual uvm_component* create_component( const std::string& name,
                                         uvm_component* parent );
```

The member function **create_component** shall create an object of type T having the provided *name* and *parent*, and returns it as a handle to a **uvm_component**. This is an overload of the member function in **uvm_object_wrapper**. It is called by the factory after determining the type of component to create. An application shall not call this member function directly. Instead, an application shall call the static member function **create**.

6.3.3.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the component. This member function overloads the member function in **uvm_object_wrapper**.

6.3.3.3 get

```
static uvm_component_registry<T>* get();
```

The member function **get** shall return the singleton instance of this type. Type-based factory operation depends on there being a single proxy instance for each registered type.

6.3.3.4 create

```
static T* create( const std::string& name = "",
                  uvm_component* parent = NULL,
                  const std::string& contxt = "" );
```

The member function **create** shall return a new instance of the component type, T, represented by this proxy, subject to any factory overrides based on the context provided by the parent's full name. The new instance shall have the given leaf name *name*, if provided as argument. The argument *contxt*, if supplied, supersedes the parent's context.

6.3.3.5 set_type_override

```
static void set_type_override( uvm_object_wrapper* override_type,
                               bool replace = true );
```

The member function **set_type_override** shall configure the factory to create a component of the type represented by argument *override_type* whenever a request is made to create a component of the type represented by this proxy, provided no instance override applies. The override type shall be derived from the original type, T.

When replace is true, a previous override on *original_type* is replaced, otherwise a previous override, if any, remains intact.

6.3.3.6 set_inst_override

```
static void set_inst_override( uvm_object_wrapper* override_type,
                               const std::string& inst_path,
                               uvm_component* parent = NULL );
```

The member function **set_inst_override** shall configure the factory to create a component of the type represented by argument *override_type* whenever a request is made to create a component of the type represented by this proxy, with matching instance paths. The override type shall be derived from the original type, T.

If argument *parent* is not specified, argument *inst_path* is interpreted as an absolute instance path, which enables instance overrides to be set from outside component classes. If argument *parent* is specified, argument *inst_path* is interpreted as being relative to the parent's hierarchical instance path. The argument *inst_path* may contain wildcards for matching against multiple contexts.

6.4 uvm_factory

The class **uvm_factory** implements a factory pattern. A singleton factory instance is created for a given simulation run. Object and component types are registered with the factory using proxies to the actual objects and components being created. The classes **uvm_object_registry**<T> and **uvm_component_registry**<T> are used to proxy objects of type **uvm_object** and **uvm_component** respectively. These registry classes both use the **uvm_object_wrapper** as abstract base class.

6.4.1 Class definition

```
namespace uvm {


  class uvm_factory {
   public:
    uvm_factory();


    // Group: Registering types

    void do_register° ( uvm_object_wrapper* obj ); // is 'register' in UVM standard


    // Group: Type & instance overrides
    void set_inst_override_by_type( uvm_object_wrapper* original_type,
                                    uvm_object_wrapper* override_type,
                                    const std::string& full_inst_path );

    void set_inst_override_by_name( const std::string& original_type_name,
                                    const std::string& override_type_name,
                                    const std::string& full_inst_path );

    void set_type_override_by_type( uvm_object_wrapper* original_type,
                                    uvm_object_wrapper* override_type,
                                    bool replace = true );

    void set_type_override_by_name( const std::string& original_type_name,
                                    const std::string& override_type_name,
                                    bool replace = true );

    // Group: Creation
    uvm_object* create_object_by_type( uvm_object_wrapper* requested_type,
                                       const std::string& parent_inst_path = "",
                                       const std::string& name = "" );
```

```
    uvm_object* create_object_by_name( const std::string& requested_type_name,
                                       const std::string& parent_inst_path = "",
                                       const std::string& name = "" );

    uvm_component* create_component_by_type( uvm_object_wrapper* requested_type,
                                             const std::string& parent_inst_path = "",
                                             const std::string& name = "",
                                             uvm_component* parent = NULL );

    uvm_component* create_component_by_name( const std::string& requested_type_name,
                                             const std::string& parent_inst_path = "",
                                             const std::string& name = "",
                                             uvm_component* parent = NULL );

    // Group: Debug
    void debug_create_by_type( uvm_object_wrapper* requested_type,
                               const std::string& parent_inst_path = "",
                               const std::string& name = "" );

    void debug_create_by_name( const std::string& requested_type_name,
                               const std::string& parent_inst_path = "",
                               const std::string& name = "" );

    uvm_object_wrapper* find_override_by_type( uvm_object_wrapper* requested_type,
                                               const std::string& full_inst_path );

    uvm_object_wrapper* find_override_by_name( const std::string& requested_type_name,
                                               const std::string& full_inst_path );

    void print( int all_types = 1 );

  }; // class uvm_factory

} // namespace uvm
```

6.4.2 Registering types

6.4.2.1 do_register° (register†)

```
Void do_register( uvm_object_wrapper* obj );
```

The member function **do_register**° shall be used to register an object or a component with the factory. Usually, an application will invoke the macros **UVM_OBJECT_UTILS**, **UVM_OBJECT_PARAM_UTILS**, **UVM_COMPONENT_UTILS**, or **UVM_COMPONENT_PARAM_UTILS** to register a particular object or component respectively with the factory.

NOTE–The UVM standard defines the member function **register**† for factory registration. As 'register' is a reserved keyword in C++, this member function has been renamed to **do_register**° in UVM-SystemC.

6.4.3 Type and instance overrides

6.4.3.1 set_inst_override_by_type

```
void set_inst_override_by_type( uvm_object_wrapper* original_type,
                                uvm_object_wrapper* override_type,
                                const std::string& full_inst_path );
```

The member function **set_inst_override_by_type** shall configure the factory to create an object of the override's type whenever a request is made to create an object of the original type using a context that matches *full_inst_path*. The override type shall be derived from the original type, T.

Both the *original_type* and *override_type* are handles to the types' proxy objects. Preregistration is not required.

The argument *full_inst_path* is matched against the concatenation of parent instance path and name (*parent_inst_path.name*) provided in future create requests. The argument *full_inst_path* may include wildcards ('*' and '?') such that a single instance override can be applied in multiple contexts. An argument *full_inst_path* of '*' is effectively a type override, as it will match all contexts.

When the factory processes instance overrides, the instance queue shall be processed in order of the override call. Thus, more specific overrides should be set in place first, followed by more general overrides. This way, the general override will not override the specific override.

6.4.3.2 set_inst_override_by_name

```
void set_inst_override_by_name( const std::string& original_type_name,
                                const std::string& override_type_name,
                                const std::string& full_inst_path );
```

The member function **set_inst_override_by_name** shall configure the factory to create an object of the override's type whenever a request is made to create an object of the original type using a context that matches *full_inst_path*. The original type is typically a super class of the override type.

The *original_type_name* typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Future calls to any of the member functions **create_object_by_type**, **create_object_by_name**,

create_component_by_type or **create_component_by_name** with the same string and matching instance path will produce the type represented by *override_type_name*, which must be preregistered with the factory.

The argument *full_inst_path* is matched against the concatenation of parent instance path and name (*parent_inst_path.name*) provided in future create requests. The argument *full_inst_path* may include wildcards ('*' and '?') such that a single instance override can be applied in multiple contexts. An argument *full_inst_path* of '*' is effectively a type override, as it will match all contexts.

When the factory processes instance overrides, the instance queue shall be processed in order of the override call. Thus, more specific overrides should be set in place first, followed by more general overrides. This way, the general override will not override the specific override.

6.4.3.3 set_type_override_by_type

```
void set_type_override_by_type( uvm_object_wrapper* original_type,
                                uvm_object_wrapper* override_type,
                                bool replace = true );
```

The member function **set_inst_override_by_type** shall configure the factory to create an object of the override's type whenever a request is made to create an object of the original type, provided no instance override applies. The override type shall be derived from the original type, T.

Both the *original_type* and *override_type* are handles to the types' proxy objects. Preregistration is not required.

When replace is true, a previous override on *original_type* is replaced, otherwise a previous override, if any, remains intact.

6.4.3.4 set_type_override_by_name

```
void set_type_override_by_name( const std::string& original_type_name,
                                const std::string& override_type_name,
                                bool replace = true );
```

The member function **set_inst_override_by_name** shall configure the factory to create an object of the override's type whenever a request is made to create an object of the original type, provided no instance override applies. The original type is typically a super class of the override type.

The *original_type_name* typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Future calls to any of the member functions **create_object_by_type**, **create_object_by_name**, **create_component_by_type** or **create_component_by_name** with the same string and matching instance path will produce the type represented by *override_type_name*, which must be preregistered with the factory.

When replace is true, a previous override on *original_type_name* is replaced, otherwise a previous override, if any, remains intact.

6.4.4 Creation

6.4.4.1 create_object_by_type

```
uvm_object* create_object_by_type( uvm_object_wrapper* requested_type,
```

```
                                    const std::string& parent_inst_path = "",
                                    const std::string& name = "" );
```

The member function **create_object_by_type** shall create and return an object of the requested type, which is specified by argument *requested_type*. A requested object shall be derived from the base class **uvm_object**.

The argument *parent_inst_path* is an optional hierarchical anchor for the object being created. If this argument is provided, then the concatenation, *parent_inst_path.name*, forms the instance path (context) that is used to search for an instance override. Newly created object shall have the given *name*, if provided.

6.4.4.2 create_object_by_name

```
uvm_object* create_object_by_name( const std::string& requested_type_name,
                                    const std::string& parent_inst_path = "",
                                    const std::string& name = "" );
```

The member function **create_object_by_name** shall create and return an object of the requested type, which is specified by argument *requested_type_name*. The requested type must have been registered with the factory with that name prior to the request. If the factory does not recognize the *requested_type_name*, an error is produced and the member function shall return NULL. A requested object shall be derived from the base class **uvm_object**.

The argument *parent_inst_path* is an optional hierarchical anchor for the object being created. If this argument is provided, then the concatenation, *parent_inst_path.name*, forms the instance path (context) that is used to search for an instance override. If no instance override is found, the factory then searches for a type override. Newly created object shall have the given *name*, if provided.

NOTE–The convenience function **create_object** is available in the class **uvm_component** for the creation of an object (See Section 7.1.1). Alternatively, an application can create an object by using the static member function **create** via the **uvm_object_registry**, which is made available via the macro **UVM_OBJECT_UTILS** or **UVM_OBJECT_PARAM_UTILS**.

6.4.4.3 create_component_by_type

```
uvm_component* create_component_by_type( uvm_object_wrapper* requested_type,
                                         const std::string& parent_inst_path = "",
                                         const std::string& name = "",
                                         uvm_component* parent = NULL );
```

The member function **create_component_by_type** shall create and return a component of the requested type, which is specified by argument *requested_type*. A requested component shall be derived from the base class **uvm_component**.

The argument *parent_inst_path* is an optional hierarchical anchor for the component being created. If this argument is provided, then the concatenation, *parent_inst_path.name*, forms the instance path (context) that is used to search for an instance override. Newly created components shall have the given *name* and *parent*.

6.4.4.4 create_component_by_name

```
uvm_component* create_component_by_name( const std::string& requested_type_name,
                                         const std::string& parent_inst_path = "",
```

```
                                        const std::string& name = "",
                                        uvm_component* parent = NULL );
```

The member function **create_component_by_name** shall create and return a component of the requested type, which is specified by argument *requested_type_name*. The requested type must have been registered with the factory with that name prior to the request. If the factory does not recognize the *requested_type_name*, an error is produced and the member function shall return NULL. A requested component shall be derived from the base class **uvm_component**.

The argument *parent_inst_path* is an optional hierarchical anchor for the component being created. If this argument is provided, then the concatenation, *parent_inst_path.name*, forms the instance path (context) that is used to search for an instance override. If no instance override is found, the factory then searches for a type override. Newly created components shall have the given *name* and *parent*.

NOTE–The convenience function **create_component** is available in the class **uvm_component** for the creation of a component (see section 7.1.1). Alternatively, an application can create an object by using the static member function **create** via the **uvm_component_registry** which is made available via the macro **UVM_COMPONENT_UTILS** or **UVM_COMPONENT_PARAM_UTILS**.

6.4.5 Debug

6.4.5.1 debug_create_by_type

```
void debug_create_by_type( uvm_object_wrapper* requested_type,
                           const std::string& parent_inst_path = "",
                           const std::string& name = "" );
```

The member function **debug_create_by_type** shall perform the same search algorithm as the member function **create_object_by_type**, but it shall not create a new object. Instead, it provides detailed information about what type of object it would return, listing each override that was applied to arrive at the result. Interpretation of the arguments are exactly as with the member function **create_object_by_type**.

6.4.5.2 debug_create_by_name

```
void debug_create_by_name( const std::string& requested_type_name,
                           const std::string& parent_inst_path = "",
                           const std::string& name = "" );
```

The member function **debug_create_by_name** shall perform the same search algorithm as the member function **create_object_by_name**, but it shall not create a new object. Instead, it provides detailed information about what type of object it would return, listing each override that was applied to arrive at the result. Interpretation of the arguments are exactly as with the member function **create_object_by_name**.

6.4.5.3 find_override_by_type

```
uvm_object_wrapper* find_override_by_type( uvm_object_wrapper* requested_type,
                                           const std::string& full_inst_path );
```

The member function **find_override_by_type** shall return the proxy to the object that would be created given the arguments. The argument *full_inst_path* is typically derived from the parent's instance path and the leaf name of the object to be created.

6.4.5.4 find_override_by_name

```
uvm_object_wrapper* find_override_by_name( const std::string& requested_type_name,
                                           const std::string& full_inst_path );
```

The member function **find_override_by_name** shall return the proxy to the object that would be created given the arguments. The argument *full_inst_path* is typically derived from the parent's instance path and the leaf name of the object to be created.

6.4.5.5 print

```
void print( int all_types = 1 );
```

The member function **print** shall print the state of the **uvm_factory**, including registered types, instance overrides, and type overrides.

When argument *all_types* is set to zero, only type and instance overrides are displayed. When *all_types* is set to 1 (default), all registered user-defined types are printed as well, provided they have names associated with them. When *all_types* is set to 2, the UVM types (prefixed with **uvm_**) are included in the list of registered types.

7. Component hierarchy classes

The UVM components form the foundation of the UVM. They are used to assemble the actual verification environment in a hierarchical and modular fashion, offering a basic set of building blocks such as sequencers, drivers, monitors, scoreboards, and other components. The UVM class library provides a set of predefined component types, all derived directly or indirectly from class **uvm_component**. The following classes are defined:

- **uvm_component**
- **uvm_agent**
- **uvm_driver**
- **uvm_monitor**
- **uvm_env**
- **uvm_scoreboard**
- **uvm_subscriber**
- **uvm_test**
- **uvm_sequencer** (see section 8)

7.1 uvm_component

The class **uvm_component** is the root base class for all structural elements. It provides interfaces for:

- Hierarchy
- Phasing: Pre-run phases, run phase, and post-run phases
- Factory: convenience interface to **uvm_factory**
- Process control: to suspend and resume processes
- Objection: to handle raised and dropped objections
- Reporting: hierarchical reporting of messages
- Recording: transaction recording

7.1.1 Class definition

```
namespace uvm {


  class uvm_component : public uvm_report_object
  {
   public:
    // Group: Construction
    explicit uvm_component( uvm_component_name name );
```

```
// Group: Hierarchy Interface
virtual uvm_component* get_parent() const;
virtual const std::string get_full_name() const;
void get_children( std::vector<uvm_component*>& children ) const;
uvm_component* get_child( const std::string& name ) const;
int get_next_child( std::string& name ) const;
int get_first_child( std::string& name ) const;
int get_num_children() const;
bool has_child( const std::string& name ) const;
uvm_component* lookup( const std::string& name ) const;
unsigned int get_depth() const;

// Group: Phasing Interface
virtual void build_phase( uvm_phase& phase );
virtual void connect_phase( uvm_phase& phase );
virtual void end_of_elaboration_phase( uvm_phase& phase );
virtual void start_of_simulation_phase( uvm_phase& phase );
virtual void run_phase( uvm_phase& phase );
virtual void pre_reset_phase( uvm_phase& phase );
virtual void reset_phase( uvm_phase& phase );
virtual void post_reset_phase( uvm_phase& phase );
virtual void pre_configure_phase( uvm_phase& phase );
virtual void configure_phase( uvm_phase& phase );
virtual void post_configure_phase( uvm_phase& phase );
virtual void pre_main_phase( uvm_phase& phase );
virtual void main_phase( uvm_phase& phase );
virtual void post_main_phase( uvm_phase& phase );
virtual void pre_shutdown_phase( uvm_phase& phase );
virtual void shutdown_phase( uvm_phase& phase );
virtual void post_shutdown_phase( uvm_phase& phase );
virtual void extract_phase( uvm_phase& phase );
virtual void check_phase( uvm_phase& phase );
virtual void report_phase( uvm_phase& phase );
virtual void final_phase( uvm_phase& phase );
virtual void phase_started( uvm_phase& phase );
virtual void phase_ready_to_end( uvm_phase& phase );
virtual void phase_ended( uvm_phase& phase );
void set_domain( uvm_domain* domain, int hier = 1 );
```

```
uvm_domain* get_domain() const;
void define_domain( uvm_domain* domain );
void set_phase_imp( uvm_phase* phase, uvm_phase* imp, int hier = 1 );
virtual void resolve_bindings();

// Group: Process control interface
virtual bool suspend();
virtual bool resume();

// Group: Objection Interface
virtual void raised( uvm_objection* objection,
                     uvm_object* source_obj,
                     const std::string& description,
                     int count );

virtual void dropped( uvm_objection* objection,
                      uvm_object* source_obj,
                      const std::string& description,
                      int count );

virtual void all_dropped( uvm_objection* objection,
                          uvm_object* source_obj,
                          const std::string& description,
                          int count );

// Group: Factory Interface
uvm_component* create_component( const std::string& requested_type_name,
                                 const std::string& name );

uvm_object* create_object( const std::string& requested_type_name,
                           const std::string& name );

static void set_type_override_by_type( uvm_object_wrapper* original_type,
                                       uvm_object_wrapper* override_type,
                                       bool replace = true );

void set_inst_override_by_type( const std::string& relative_inst_path,
                                uvm_object_wrapper* original_type,
```

```
                                uvm_object_wrapper* override_type );

  static void set_type_override( const std::string& original_type_name,
                                 const std::string& override_type_name,
                                 bool replace = true );

  void set_inst_override( const std::string& relative_inst_path,
                          const std::string& original_type_name,
                          const std::string& override_type_name );

  void print_override_info( const std::string& requested_type_name = "",
                            const std::string& name = "" );

  // Group: Hierarchical reporting interface
  void set_report_id_verbosity_hier( const std::string& id,
                                     int verbosity );

  void set_report_severity_id_verbosity_hier( uvm_severity severity,
                                              const std::string& id,
                                              int verbosity );

  void set_report_severity_action_hier( uvm_severity severity,
                                        uvm_action action );

  void set_report_id_action_hier( const std::string& id,
                                  uvm_action action );

  void set_report_severity_id_action_hier( uvm_severity severity,
                                           const std::string& id,
                                           uvm_action action );

  void set_report_default_file_hier( UVM_FILE file );

  void set_report_severity_file_hier( uvm_severity severity,
                                      UVM_FILE file );

  void set_report_id_file_hier( const std::string& id,
                                UVM_FILE file );
```

```
    void set_report_severity_id_file_hier( uvm_severity severity,
                                           const std::string& id,
                                           UVM_FILE file );

    void set_report_verbosity_level_hier( int verbosity );

    virtual void pre_abort();

    // Group: Recording interface
    NOT IMPLEMENTED

    // Data members
    static bool print_config_matches;

  }; // class uvm_component

} // namespace uvm
```

7.1.2 Construction interface

When creating a new UVM component, an application must always provide a local leaf name. The parent is traced from the current **uvm_component** at top of the hierarchy stack. The **uvm_component** hierarchy stack is built during module construction, in the pre-run phases **build_phase** and **connect_phase**. If the parent component is not derived from **uvm_component**, the leaf object becomes part of the object **uvm_root**. The full hierarchical name must be unique; if it is not unique, a warning message is generated, and a number is appended at the end of the hierarchical name to make it unique.

Compatible with SystemC, it is illegal to create a component after the **before_end_of_elaboration** phase or UVM pre-run phases **build_phase** and **connect_phase**. The constructor for **uvm_component** spawns off the member function **run_phase** of that component.

7.1.2.1 Constructor

```
explicit uvm_component( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.1.3 Hierarchy interface

The following member functions provide user access to information about the component hierarchy, for example, topology.

7.1.3.1 get_parent

```
virtual uvm_component* get_parent() const;
```

The member function **get_parent** shall return a pointer to the component's parent, or NULL if it has no parent.

7.1.3.2 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the full hierarchical name of the component. It shall concatenate the hierarchical name of the parent, if any, with the leaf name of the component, as returned by member function **uvm_object**::**get_name** (see 4.2.4.2).

7.1.3.3 get_children

```
void get_children( std::vector<uvm_component*>& children ) const;
```

The member function **get_children** shall return a vector of type std::vector containing a pointer to every instance of the component's children of class **uvm_component**.

7.1.3.4 get_child

```
uvm_component* get_child( const std::string& name ) const;
```

The member function **get_child** shall return a pointer to the component's child which matches the argument string *name*.

7.1.3.5 get_first_child

```
int get_first_child( std::string& name ) const;
```

The member function **get_first_child** shall pass the name of the first child of a component to the argument *name*. The member function returns true of the first child has been found; otherwise it shall return false.

7.1.3.6 get_next_child

```
int get_next_child( std::string& name ) const;
```

The member function **get_next_child** shall pass the name of the next child of a component, followed after a call to member function **get_first_child**, to the argument *name*. The member function returns true of the next child has been found; otherwise it shall return false.

7.1.3.7 get_num_children

```
int get_num_children() const;
```

The member function **get_num_children** shall return the number of the component's children.

7.1.3.8 has_child

```
bool has_child( const std::string& name ) const;
```

The member function **has_child** shall return true if this component has a child with the given name; otherwise it shall return false;

7.1.3.9 lookup

```
uvm_component* lookup( const std::string& name ) const;
```

The member function **lookup** shall return a pointer to a component with the passed hierarchical name *name* relative to the component. If the argument *name* is preceded with a '.' (dot), then the search shall begin relative to the top level (absolute lookup). The member function shall return NULL if no component has been found. The argument *name* shall not contain wildcards.

7.1.3.10 get_depth

```
unsigned int get_depth() const;
```

The member function **get_depth** shall return the component's depth from the root level. **uvm_top** has a depth of 0. The test and any other top level components have a depth of 1, and so on.

7.1.4 Phasing interface

UVM components execute their behavior in strictly ordered, pre-defined phases. Each phase is defined by its own member function, which derived components can override to incorporate component-specific behavior. During simulation, the phases are executed one by one, where one phase must complete before the next phase begins.

The phases can be grouped in three main categories:

- Pre-run phases
- Run-time phases
- Post-run phases

7.1.4.1 Pre-run phases

The pre-run phases are responsible for the construction, connection and elaboration of the structural composition. In the pre-run phases, there is neither notion nor progress of time. It consists of the following phases:

- **build_phase**: The component constructs its children in this phase. It may use the static member function **uvm_config_db**::**get** to obtain any configuration for itself, the member function **uvm_config_db**::**set** to define any configuration for its own children, and the factory interface for actually creating the children and other objects it might need. An application shall declare child objects derived from **uvm_component** as pointers, instead of member fields of a component, such that they can be created via the factory in this phase.
- **connect_phase**: After creating the children in the **build_phase**, the component makes connections (binding of (TLM) ports and exports) from child-to-child or from child-to-self (that is, to promote a child or export up the hierarchy for external access).

- **end_of_elaboration_phase**: At this point, the entire testbench environment has been built and connected. No new components and connections shall be created from this point forward. Components do final checks for proper connectivity.
- **start_of_simulation_phase**: The simulation is about to begin, and this phase is used to perform any pre-run activity such as displaying banners, printing final testbench topology and configuration information.

As UVM components are derived from class **sc_module**, the inherited callbacks **before_end_of_elaboration**, **end_of_elaboration**, and **start_of_simulation** are available. It is recommended *not* to use these member functions for the construction of testbenches, but to use the UVM pre-run phases. Main reason is to support maximum reusability and flexibility for building, configuration and connecting various verification components using the same construction mechanism.

7.1.4.2 Run-time phases

The run-time phases are used to perform the actual verification. These phases are exclusively designed only for objects derived from class **uvm_component**. Run-time phases consume time.

A component's primary function is implemented in the member function **run_phase**. The component should not declare 'run_phase' as a thread process. The UVM-SystemC library spawns **run_phase** as a thread process. Other processes may be spawned from the run phase, if desired. When a component returns from executing its member function **run_phase**, it does not signify completion of its run phase. Any processes that it may have spawned still continue to run.

The run phase executes along with the other processes in the SystemC language: no special status is provided to the **run_phase** processes; for example, there is no guarantee that the **run_phase** processes is the first on the runable queue at time 0s, and hence there is no guarantee that the **run_phase** processes execute ahead of the other SystemC processes.

Concurrently to the execution of the **run_phase**, UVM defines a pre-defined schedule which consists of four groups of phases which are executed sequentially.

- Reset phases: Phases to apply reset signals for the DUT. Consists of three phases called **pre_reset_phase**, **reset_phase**, and **post_reset_phase**.
- Configure phases: Phases which can be used for the configuration of the DUT. Consists of three phases called **pre_configure_phase**, **configure_phase**, and **post_configure_phase**.
- Main phases: Phases which are used to apply the primary test stimulus to DUT. Consists of three phases called **pre_main_phase**, **main_phase**, and **post_main_phase**.
- Shutdown phase: Phases to wait for all data to be drained out of the DUT and to disable DUT. Consists of three phases called **pre_shutdown_phase**, **shutdown_phase**, and **post_shutdown_phase**.

7.1.4.3 Post-run phases

The post-run phases are:

- **extract_phase**: This phase occurs after the run phase is over. This phase is specific to objects derived from class **uvm_component** and does not apply to objects derived from class **sc_module**. It is used to extract simulation results from coverage collectors and scoreboards, collect status/error counts, statistics, and other information from components in bottom-up order. Being a separate phase, the extract phase ensures all

relevant data from potentially independent sources (that is, other components) are collected before being checked in the next phase.

- **check_phase**: This phase is specific to objects derived from class **uvm_component** and does not apply to objects derived from class **sc_module**. Having extracted vital simulation results in the previous phase, the check phase is used to validate such data and determine the overall simulation outcome. It executes bottom-up.
- **report_phase**: Finally, the report phase is used to output results to files and/or the screen. This phase is also be specific to objects derived from class **uvm_component** and does not apply to objects derived from class **sc_module**.
- **final_phase**: This phase is called as soon as all tests have been executed and completed. This phase is used to close created or used files before the simulation exits.

7.1.4.4 build_phase

```
virtual void build_phase( uvm_phase& phase );
```

The member function **build_phase** shall provide a context to implement functionality as part of the build phase. The application shall not call this member function directly.

7.1.4.5 connect_phase

```
virtual void connect_phase( uvm_phase& phase );
```

The member function **connect_phase** shall provide a context to implement functionality as part of the connect phase. The application shall not call this member function directly.

7.1.4.6 end_of_elaboration_phase

```
virtual void end_of_elaboration_phase( uvm_phase& phase );
```

The member function **end_of_elaboration_phase** shall provide a context to implement functionality as part of the end of elaboration phase. The application shall not call this member function directly.

7.1.4.7 start_of_simulation_phase

```
virtual void start_of_simulation_phase( uvm_phase& phase );
```

The member function **start_of_simulation_phase** shall provide a context to implement functionality as part of the start of simulation phase. The application shall not call this member function directly.

7.1.4.8 run_phase

```
virtual void run_phase( uvm_phase& phase );
```

The member function **run_phase** shall provide a context to implement functionality as part of the run phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the

member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.9 pre_reset_phase

```
virtual void pre_reset_phase( uvm_phase& phase );
```

The member function **pre_reset_phase** shall provide a context to implement functionality as part of the pre-reset phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.10 reset_phase

```
virtual void reset_phase( uvm_phase& phase );
```

The member function **reset_phase** shall provide a context to implement functionality as part of the reset phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.11 post_reset_phase

```
virtual void post_reset_phase( uvm_phase& phase );
```

The member function **post_reset_phase** shall provide a context to implement functionality as part of the post-reset phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.12 pre_configuration_phase

```
virtual void pre_configuration_phase( uvm_phase& phase );
```

The member function **pre_configuration_phase** shall provide a context to implement functionality as part of the pre-configuration phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.13 configuration_phase

```
virtual void configuration_phase( uvm_phase& phase );
```

The member function **configuration_phase** shall provide a context to implement functionality as part of the configuration phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.14 post_configuration_phase

```
virtual void post_configuration_phase( uvm_phase& phase );
```

The member function **post_configuration_phase** shall provide a context to implement functionality as part of the post-configuration phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.15 pre_main_phase

```
virtual void pre_main_phase( uvm_phase& phase );
```

The member function **pre_main_phase** shall provide a context to implement functionality as part of the pre-main phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.16 main_phase

```
virtual void main_phase( uvm_phase& phase );
```

The member function **main_phase** shall provide a context to implement functionality as part of the main phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.17 post_main_phase

```
virtual void post_main_phase( uvm_phase& phase );
```

The member function **post_main_phase** shall provide a context to implement functionality as part of the post-main phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist.

Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.18 pre_shutdown_phase

```
virtual void pre_shutdown_phase( uvm_phase& phase );
```

The member function **pre_shutdown_phase** shall provide a context to implement functionality as part of the pre-shutdown phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.19 shutdown_phase

```
virtual void shutdown_phase( uvm_phase& phase );
```

The member function **shutdown_phase** shall provide a context to implement functionality as part of the shutdown phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.20 post_shutdown_phase

```
virtual void post_shutdown_phase( uvm_phase& phase );
```

The member function **post_shutdown_phase** shall provide a context to implement functionality as part of the post-shutdown phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.21 extract_phase

```
virtual void extract_phase( uvm_phase& phase );
```

The member function **extract_phase** shall provide a context to implement functionality as part of the extract phase. The application shall not call this member function directly.

7.1.4.22 check_phase

```
virtual void check_phase( uvm_phase& phase );
```

The member function **check_phase** shall provide a context to implement functionality as part of the check phase. The application shall not call this member function directly.

7.1.4.23 report_phase

```
virtual void report_phase( uvm_phase& phase );
```

The member function **report_phase** shall provide a context to implement functionality as part of the report phase. The application shall not call this member function directly.

7.1.4.24 final_phase

```
virtual void final_phase( uvm_phase& phase );
```

The member function **final_phase** shall provide a context to implement functionality as part of the final phase. The application shall not call this member function directly.

7.1.4.25 phase_started

```
virtual void phase_started( uvm_phase& phase );
```

The member function **phase_started** shall provide a context to implement functionality as part of the start of each phase. The argument *phase* specifies the phase being started. Any threads spawned in this callback are not affected when the phase ends.

7.1.4.26 phase_ready_to_end

```
virtual void phase_ready_to_end( uvm_phase& phase );
```

The member function **phase_ready_to_end** shall provide a context to implement functionality as part of the ending of each phase. The argument *phase* specifies the phase being ended. The member function shall be invoked when all objections to ending the given phase have been dropped, thus indicating that phase is ready to end. All this component's threads spawned for the given phase will be killed upon return from this member function. Components needing to consume delta cycles or advance time to perform a clean exit from the phase may raise the phase's objection.

7.1.4.27 phase_ended

```
virtual void phase_ended( uvm_phase& phase );
```

The member function **phase_ended** shall provide a context to implement functionality at the end of each phase. The argument *phase* specifies the phase that has ended. Any threads spawned in this callback are not affected when the phase ends.

7.1.4.28 set_domain

```
void set_domain( uvm_domain* domain, int hier = 1 );
```

The member function **set_domain** shall set the phase domain to this component and, if *hier* is set, recursively to all its children.

7.1.4.29 get_domain

```
uvm_domain* get_domain() const;
```

The member function **get_domain** shall return a pointer to the phase domain set on this component.

7.1.4.30 define_domain

```
void define_domain( uvm_domain* domain ); make protected?
```

The member function **define_domain** shall build a custom phase schedules into the provided domain passed as pointer.

7.1.4.31 set_phase_imp

```
void set_phase_imp( uvm_phase* phase, uvm_phase* imp, int hier = 1 );
```

The member function **set_phase_imp** shall provide a context for an application-specific phase implementation, which shall be created as a singleton object extending the default one and implementing required behavior for the member functions **execute** and **traverse**.

The optional argument *hier* specifies whether to apply the custom functor to the whole tree or just this component.

7.1.4.32 resolve_bindings

```
virtual void resolve_bindings();
```

The member function **resolve_bindings** shall check whether each port's min and max connection requirements are met. It shall process all port, export, and imp connections. It shall be called just before the **end_of_elaboration_phase**. The application shall not call this member function directly.

7.1.5 Process control interface

The class **uvm_component** has the following member functions to support process control constructs on the run process handle:

- **suspend**
- **resume**

The default implementation of these member functions is to invoke the corresponding process control construct on the component's run process handle, if the run process is active (that is, not already terminated), for those simulators that support process control constructs. Each of these member functions return true if the simulator supports process control constructs. For those simulators that do not support process control constructs, these member functions do nothing and return false. CHECK

NOTE–Process control extensions are only supported when using the Accellera Systems Initiative SystemC 2.3.0 release of the proof-of-concept library.

7.1.5.1 suspend

```
virtual bool suspend();
```

The member function **suspend** shall suspend operation of this component. It shall return true if suspending succeeds; otherwise it shall return false.

NOTE–This member function shall be implemented by the application to suspend the component according to the protocol and functionality it implements. A suspended component can be subsequently resumed by calling the member function **resume**.

7.1.5.2 resume

```
virtual bool resume();
```

The member function **resume** shall resume operation of this component. It shall return true if resuming succeeds; otherwise it shall return false.

NOTE–This member function shall be implemented by the application to resume a component that was previously suspended using member function **suspend**. Some components may start in the suspended state and may need to be explicitly resumed.

7.1.6 Objection interface

These member functions provide object level access into the **uvm_objection** mechanism.

7.1.6.1 raised

```
virtual void raised( uvm_objection* objection,
                     uvm_object* source_obj,
                     const std::string& description,
                     int count );
```

The member function **raised** shall be called when this or a descendant of this component instance raises the specified objection. The argument *source_obj* is the object that originally raised the objection. The argument *description* is optionally provided by the *source_obj* to give a reason for raising the objection. The argument *count* indicates the number of objections raised by the *source_obj*.

7.1.6.2 dropped

```
virtual void dropped( uvm_objection* objection,
                      uvm_object* source_obj,
                      const std::string& description,
                      int count );
```

The member function **dropped** shall be called when this or a descendant of this component instance drops the specified objection. The argument *source_obj* is the object that originally dropped the objection. The argument *description* is optionally provided by the *source_obj* to give a reason for dropping the objection. The argument *count* indicates the number of objections dropped by the *source_obj*.

7.1.6.3 all_dropped

```
virtual void all_dropped( uvm_objection* objection,
                          uvm_object* source_obj,
```

```
                              const std::string& description,
                              int count );
```

The member function **all_dropped** shall be called when all objections have been dropped by this component and all its descendants. The argument *source_obj* is the object that dropped the last objection. The argument *description* is optionally provided by the *source_obj* to give a reason for raising the objection. The argument *count* indicates the number of objections dropped by the *source_obj*.

7.1.7 Factory interface

The factory interface provides components with convenient access to the UVM's central **uvm_factory** object. The member functions **create_***, and **set_*** call the corresponding member functions in **uvm_factory**, passing whatever arguments it can to reduce the number of arguments required of the user.

7.1.7.1 create_component

```
uvm_component* create_component( const std::string& requested_type_name,
                                 const std::string& name );
```

The member function **create_component** shall provide a convenience layer to the member function **uvm_factory**::**create_component_by_name**, which calls upon the factory to create a new child component whose type corresponds to the preregistered type name, *requested_type_name*, and instance name, *name* (see 6.4.4.4).

7.1.7.2 create_object

```
uvm_object* create_object( const std::string& requested_type_name,
                           const std::string& name );
```

The member function **create_object** shall provide a convenience layer to the member function **uvm_factory**::**create_object_by_name**, which calls upon the factory to create a new object whose type corresponds to the preregistered type name, *requested_type_name*, and instance name, *name* (see 6.4.4.2).

7.1.7.3 set_type_override_by_type

```
static void set_type_override_by_type( uvm_object_wrapper* original_type,
                                       uvm_object_wrapper* override_type,
                                       bool replace = true );
```

The member function **set_type_override_by_type** shall provide a convenience layer to the member function **uvm_factory**::**set_type_override_by_type**, which registers a factory override for components and objects created at this level of hierarchy or below (see 6.4.3.3).

The argument *original_type* represents the type that is being overridden. In subsequent calls to **uvm_factory**::**create_object_by_type** or **uvm_factory**::**create_component_by_type**, if the argument *requested_type* matches the *original_type* and the instance paths match, the factory will produce the *override_type*.

7.1.7.4 set_inst_override_by_type

```
void set_inst_override_by_type( const std::string& relative_inst_path,
                                uvm_object_wrapper* original_type,
                                uvm_object_wrapper* override_type );
```

The member function **set_inst_override_by_type** shall provide a convenience layer to the member function **uvm_factory**::**set_inst_override_by_type**, which registers a factory override for components and objects created at this level of hierarchy or below (see 6.4.3.1).

The argument *relative_inst_path* is relative to this component and may include wildcards. The argument *original_type* represents the type that is being overridden. In subsequent calls to **uvm_factory**::**create_object_by_type** or **uvm_factory**::**create_component_by_type**, if the *requested_type* matches the *original_type* and the instance paths match, the factory will produce the *override_type*.

7.1.7.5 set_type_override

```
static void set_type_override( const std::string& original_type_name,
                               const std::string& override_type_name,
                               bool replace = true );
```

The member function **set_type_override** shall provide a convenience layer to the member function **uvm_factory**::**set_type_override_by_name**, which configures the factory to create an object of type *override_type_name* whenever the factory is asked to produce a type represented by *original_type_name* (see 6.4.3.4).

The argument *original_type_name* typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Subsequent calls to **create_component** or **create_object** with the same string and matching instance path will produce the type represented by *override_type_name*. The argument *override_type_name* must refer to a preregistered type in the factory.

7.1.7.6 set_inst_override

```
void set_inst_override( const std::string& relative_inst_path,
                        const std::string& original_type_name,
                        const std::string& override_type_name );
```

The member function **set_inst_override** shall provide a convenience layer to the member function **uvm_factory**::**set_inst_override_by_name**, which registers a factory override for components created at this level of hierarchy or below (see 6.4.3.2).

The argument *relative_inst_path* is relative to this component and may include wildcards. The argument *original_type_name* typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Subsequent calls to **create_component** or **create_object** with the same string and matching instance path will produce the type represented by *override_type_name*. The *override_type_name* must refer to a preregistered type in the factory.

7.1.7.7 print_override_info

```
void print_override_info( const std::string& requested_type_name = "",
                          const std::string& name = "" );
```

The member function **print_override_info** shall provide the same lookup process as **create_object** and **create_component**, but instead of creating an object, it prints information about what type of object would be created given the provided arguments.

7.1.8 Hierarchical reporting interface

This interface provides versions of the member function **set_report_*** in the base class **uvm_report_object** that are applied recursively to this component and all its children. When a report is issued and its associated action **UVM_LOG** is set, the report will be sent to its associated file descriptor.

7.1.8.1 set_report_id_verbosity_hier

```
void set_report_id_verbosity_hier( const std::string& id,
                                   int verbosity );
```

The member function **set_report_id_verbosity_hier** shall recursively associate the specified verbosity with reports of the given *id*. A verbosity associated with a particular severity-id pair, using member function **set_report_severity_id_verbosity_hier**, shall take precedence over a verbosity associated by this member function.

7.1.8.2 set_report_severity_id_verbosity_hier

```
void set_report_severity_id_verbosity_hier( uvm_severity severity,
                                            const std::string& id,
                                            int verbosity );
```

The member function **set_report_severity_id_verbosity_hier** shall recursively associate the specified verbosity with reports of the given *severity* with *id* pair. An verbosity associated with a particular severity-id pair takes precedence over an verbosity associated with id, which takes precedence over a verbosity associated with a severity.

7.1.8.3 set_report_severity_action_hier

```
void set_report_severity_action_hier( uvm_severity severity,
                                      uvm_action action );
```

The member function **set_report_severity_action_hier** shall recursively associate the specified action with reports of the given *severity*. An action associated with a particular severity-id pair shall take precedence over an action associated with id, which shall take precedence over an action associated with a severity as defined in this member function.

7.1.8.4 set_report_id_action_hier

```
void set_report_id_action_hier( const std::string& id,
                                uvm_action action );
```

The member function **set_report_id_action_hier** shall recursively associate the specified action with reports of the given *id*. An action associated with a particular severity-id pair shall take precedence over an action associated with id as defined in this member function.

7.1.8.5 set_report_severity_id_action_hier

```
void set_report_severity_id_action_hier( uvm_severity severity,
                                         const std::string& id,
                                         uvm_action action );
```

The member function **set_report_severity_id_action_hier** shall recursively associate the specified action with reports of the given *severity* with *id* pair. An action associated with a particular severity-id pair shall take precedence over an action associated with id, which shall take precedence over an action associated with a severity.

7.1.8.6 set_report_default_file_hier

```
void set_report_default_file_hier( UVM_FILE file );
```

The member function **set_report_default_file_hier** shall recursively associate the report to the default *file* descriptor. A file associated with a particular severity-id pair shall take precedence over a file associated with id, which shall take precedence over a file associated with a severity, which shall take precedence over the default file descriptor as defined in this member function.

7.1.8.7 set_report_severity_file_hier

```
void set_report_severity_file_hier( uvm_severity severity,
                                    UVM_FILE file );
```

The member function **set_report_severity_file_hier** shall recursively associate the specified *file* descriptor with reports of the given *severity*. A file associated with a particular severity-id pair shall take precedence over a file associated with id, which shall take precedence over a file associated with a severity as defined in this member function.

7.1.8.8 set_report_id_file_hier

```
void set_report_id_file_hier( const std::string& id,
                              UVM_FILE file );
```

The member function **set_report_id_file_hier** shall recursively associate the specified *file* descriptor with reports of the given *id*. A file associated with a particular severity-id pair shall take precedence over a file associated with id as defined in this member function.

7.1.8.9 set_report_severity_id_file_hier

```
void set_report_severity_id_file_hier( uvm_severity severity,
                                       const std::string& id,
                                       UVM_FILE file );
```

The member function **set_report_severity_id_file_hier** shall recursively associate the specified *file* descriptor with reports of the given *severity* and *id* pair. A file associated with a particular severity-id pair shall take precedence over a file associated with id, which shall take precedence over a file associated with a severity, which shall take precedence over the default file descriptor.

7.1.8.10 set_report_verbosity_level_hier

```
void set_report_verbosity_level_hier( int verbosity );
```

The member function **set_report_verbosity_level_hier** shall recursively set the maximum verbosity level for reports for this component and all those below it. Any report from this component sub-tree whose verbosity exceeds this maximum will be ignored.

7.1.8.11 pre_abort

```
virtual void pre_abort();
```

The member function **pre_abort** shall be executed when the message system is executing a **UVM_EXIT** action. The exit action causes an immediate termination of the simulation, but the **pre_abort** callback hook gives components an opportunity to provide additional information to the application before the termination happens. For example, a test may want to execute the report function of a particular component even when an error condition has happened to force a premature termination. The member function **pre_abort** shall be called for all UVM components in the hierarchy in a bottom-up fashion.

7.1.9 Recording interface

TODO

7.1.10 Macros

UVM-SystemC defines the following macros for class **uvm_component**:

- Utility macro **UVM_COMPONENT_UTILS**(*classname*) to be used inside the Class definition, that expands to:
    - The declaration of the member function **get_type_name**, which returns the type of the class as string
    - The declaration of the member function **get_type**, which returns a factory proxy object for the type
    - The class **uvm_component_registry**<*classname*> used by the factory.

Template classes shall use the macro **UVM_COMPONENT_PARAM_UTILS**, to guarantee correct registration of one or more parameters passed to the class template. Note that template classes are not evaluated at compile-time, and thus not registered with the factory. Due to this, name-based lookup with the factory for template classes is not possible. Instead, an application shall use the member function **get_type** for factory overrides.

7.2 uvm_driver

The class **uvm_driver** is the base class for drivers that initiate requests for new transactions. The ports are typically connected to the exports of an appropriate sequencer component of class **uvm_sequencer**.

7.2.1 Class definition

```
namespace uvm {


  template <typename REQ = uvm_sequence_item, typename RSP = REQ>
  class uvm_driver : public uvm_component
  {
   public:
    uvm_port_base< uvm_sqr_if_base<REQ, RSP> > seq_item_port;


    explicit uvm_driver( uvm_component_name name );


    virtual const std::string get_type_name() const;


  }; // class uvm_driver


} // namespace uvm
```

7.2.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively.These object types must be a derivative of class **uvm_sequence_item**.

7.2.3 Ports

7.2.3.1 seq_item_port

```
uvm_port_base< uvm_sqr_if_base<REQ, RSP> > seq_item_port;
```

A port **seq_item_port** of type **uvm_port_base** using interface **uvm_sqr_if_base**<REQ, RSP> shall be defined to connect (bind) the driver to the corresponding export in the sequencer.

NOTE–In line with the UVM-SystemVerilog syntax, the member function **connect** can be used to establish the binding between the driver and the sequencer. The UVM-SystemC implementation also supports the SystemC syntax using the member function **bind** or using **operator()** to perform the binding.

7.2.4 Member functions

7.2.4.1 Constructor

```
explicit uvm_driver( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.2.4.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the object derived from this class as an object of type std::string.

7.3 uvm_monitor

The class **uvm_monitor** is the base class for monitors. Deriving from **uvm_monitor** allows an application to distinguish monitors from generic component types inheriting from uvm_component. Such monitors will automatically inherit features that may be added to **uvm_monitor** in the future.

7.3.1 Class definition

```
namespace uvm {


  class uvm_monitor : public uvm_component
  {
   public:
    explicit uvm_monitor( uvm_component_name name );


    virtual const std::string get_type_name() const;


  }; // class uvm_monitor


} // namespace uvm
```

7.3.2 Member functions

7.3.2.1 Constructor

```
explicit uvm_monitor( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.3.2.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the object derived from this class as an object of type std::string.

7.4 uvm_agent

The class **uvm_agent** is the base class for the creation of agents. Deriving from **uvm_agent** will allow an application to distinguish agents from other component types also using its inheritance. Such agents will automatically inherit features that may be added to **uvm_agent** in the future.

While an agent's build function, inherited from **uvm_component**, can be implemented to define any agent topology, an agent typically contains three subcomponents: a driver, sequencer, and monitor. If the agent is active, subtypes should contain all three subcomponents. If the agent is passive, subtypes should contain only the monitor.

7.4.1 Class definition

```
namespace uvm {


  class uvm_agent : public uvm_component
  {
   public:
    explicit uvm_agent( uvm_component_name name );
    virtual const std::string get_type_name() const;
    uvm_active_passive_enum get_is_active() const;
  }; // class uvm_agent


} // namespace uvm
```

7.4.2 Member functions

7.4.2.1 Constructor

```
explicit uvm_agent( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.4.2.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the object derived from this class as an object of type std::string.

7.4.2.3 get_is_active

```
uvm_active_passive_enum get_is_active();
```

The member function **get_is_active** shall return **UVM_ACTIVE** if the agent is acting as an active agent and **UVM_PASSIVE** if it is acting as a passive agent (see 15.3.4). An application may override this behavior if a more complex algorithm is needed to determine the active/passive nature of the agent.

7.5 uvm_env

The class **uvm_env** is the base class for the creation of a self-containing verification environment, such as a verification component which contains multiple agents.

7.5.1 Class definition

```
namespace uvm {


  class uvm_env : public uvm_component
  {
   public:
    explicit uvm_env( uvm_component_name name );


    virtual const std::string get_type_name() const;


  }; // class uvm_env


} // namespace uvm
```

7.5.2 Member functions

7.5.2.1 Constructor

```
explicit uvm_env( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.5.2.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the object derived from this class as an object of type std::string.

7.6 uvm_test

The class **uvm_test** is the base class for the test environment.

7.6.1 Class definition

```
namespace uvm {


  class uvm_test : public uvm_component
  {
```

```
   public:
    explicit uvm_test( uvm_component_name name );

    virtual const std::string get_type_name() const;

  }; // class uvm_test

} // namespace uvm
```

7.6.2 Member functions

7.6.2.1 Constructor

```
explicit uvm_test( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.6.2.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the object derived from this class as an object of type std::string.

7.7 uvm_scoreboard

The class **uvm_scoreboard** is the base class for the creation of a scoreboard. Deriving from **uvm_scoreboard** will allow an application to distinguish scoreboards from other component types inheriting directly from **uvm_component**. Such scoreboards will automatically inherit and benefit from features that may be added to **uvm_scoreboard** in the future.

7.7.1 Class definition

```
namespace uvm {

  class uvm_scoreboard : public uvm_component
  {
   public:
    explicit uvm_scoreboard( uvm_component_name name );

    virtual const std::string get_type_name() const;

  }; // class uvm_scoreboard
```

```
} // namespace uvm
```

7.7.2 Member functions

7.7.2.1 Constructor

```
explicit uvm_scoreboard( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.7.2.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the component derived from this class as an object of type std::string.

7.8 uvm_subscriber

The class **uvm_subscriber** is the base class for the creation of a subscriber. It provides an analysis export for receiving transactions from a connected analysis export. Making such a connection “subscribes” this component to any transactions emitted by the connected analysis port.

Subtypes of this class must define the member function **write** to process the incoming transactions. This class is particularly useful when designing a coverage collector that attaches to a monitor.

7.8.1 Class definition

```
namespace uvm {

  class uvm_subscriber : public uvm_component
  {
   public:
    uvm_analysis_export<T> analysis_export;

    explicit uvm_scoreboard( uvm_component_name name );

    virtual const std::string get_type_name() const;

  }; // class uvm_subscriber

} // namespace uvm
```

7.8.2 Export

7.8.2.1 analysis_export

```
uvm_analysis_export<T> analysis_export;
```

The export **analysis_export** shall provide access to the member function **write** method, which derived subscribers shall implement.

7.8.3 Member functions

7.8.3.1 Constructor

```
explicit uvm_subscriber( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.8.3.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the component derived from this class as an object of type std::string.

8. Sequencer classes

The sequencer classes offer the interface between the stimuli generators (by means of sequences) and the structural composition of the test infrastructure using verification components. The sequencer is integral part of a verification component, which can be enabled in case the verification component is marked as ‘active’ (driving) element.

The sequencer processes the transactions, defined as objects derived from class **uvm_sequence_item** or class **uvm_sequence** and passes these transactions to the driver (object derived from class **uvm_driver**).

The following sequencer classes are defined:

- **uvm_sequencer_base**
- **uvm_sequencer_param_base**
- **uvm_sequencer**
- **uvm_sqr_if_base**

NOTE–The UVM-SystemC sequencer classes only partially implement the standardized UVM sequencer capabilities. However, these definitions are sufficient to build a functional sequencer.

8.1 uvm_sequencer_base

The class **uvm_sequencer_base** is the root base class for all sequencer classes.

8.1.1 Class definition

```
namespace uvm {


  class uvm_sequencer_base : public uvm_component
  {
   public:
    explicit uvm_sequencer_base( uvm_component_name name );


    bool is_child ( uvm_sequence_base* parent, const uvm_sequence_base* child ) const;


    virtual int user_priority_arbitration( vector< uvm_sequence_request* > avail_sequences );
    virtual void execute_item( uvm_sequence_item* item ); NOT IMPLEMENTED YET
    virtual void start_phase_sequence( uvm_phase& phase );


    virtual void wait_for_grant( uvm_sequence_base* sequence_ptr,
                                 int item_priority = -1,
                                 bool lock_request = false);


    virtual void wait_for_item_done( uvm_sequence_base* sequence_ptr,
                                     int transaction_id = -1 );
```

```
    bool is_blocked( const uvm_sequence_base* sequence_ptr ) const;
    bool has_lock( uvm_sequence_base* sequence_ptr );
    virtual void lock( uvm_sequence_base* sequence_ptr );
    virtual void grab( uvm_sequence_base* sequence_ptr );
    virtual void unlock( uvm_sequence_base* sequence_ptr );
    virtual void ungrab( uvm_sequence_base* sequence_ptr );
    virtual void stop_sequences();
    virtual bool is_grabbed() const;
    virtual uvm_sequence_base* current_grabber() const;
    virtual bool has_do_available();
    void set_arbitration( SEQ_ARB_TYPE mode );
    SEQ_ARB_TYPE get_arbitration() const;
    virtual void wait_for_sequences();

    virtual void send_request( uvm_sequence_base* sequence_ptr,
                               uvm_sequence_item* seq_item,
                               bool rerandomize = false);

  }; // class uvm_sequencer_base

} // namespace uvm
```

8.1.2 Constructor

```
explicit uvm_sequencer_base( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

8.1.3 Member functions

8.1.3.1 is_child

```
bool is_child ( uvm_sequence_base* parent, const uvm_sequence_base* child ) const;
```

The member function **is_child** shall return true if the child sequence is a child of the parent sequence and false otherwise.

8.1.3.2 user_priority_arbitration

```
virtual int user_priority_arbitration( vector< uvm_sequence_request* > avail_sequences );
```

The member function **user_priority_arbitration** shall be called by an application when the sequencer arbitration mode is set to **SEQ_ARB_USER** (via the member function **set_arbitration**) each time that it needs to arbitrate among sequences. Derived sequencers may override this member function to perform a custom arbitration policy. The override shall return one of the entries from the *avail_sequences* queue, which are indexes into an internal queue of type vector< **uvm_sequence_request*** >. The default implementation shall behave similar as **SEQ_ARB_FIFO**, which returns the first entry of *avail_sequences*.

8.1.3.3 start_phase_sequence

```
virtual void start_phase_sequence( uvm_phase phase );
```

The member function **start_phase_sequence** shall start the default sequence for the phase given as argument. The default sequence is configured via resources using either a sequence instance or sequence type (object wrapper). If both are used, the sequence instance takes precedence. When attempting to override a previous default sequence setting, an application shall override both the instance and type (wrapper) resources, else the override may not take effect.

8.1.3.4 wait_for_grant

```
virtual void wait_for_grant( uvm_sequence_base* sequence_ptr,
                             int item_priority = -1,
                             bool lock_request = false);
```

The member function **wait_for_grant** shall issue a request for the specified sequence. If *item_priority* is not specified, then the current sequence priority shall be used by the arbiter. If a *lock_request* is made, then the sequencer shall issue a lock immediately before granting the sequence. The lock may be granted without the sequence being granted if the member function **is_relevant** of the sequence instance is not asserted.

When this member function returns, the sequencer has granted the sequence, and the sequence must call **send_request** without inserting any simulation delay other than delta cycles. The driver is currently waiting for the next item to be sent via the **send_request** call.

8.1.3.5 wait_for_item_done

```
virtual void wait_for_item_done( uvm_sequence_base* sequence_ptr,
                                 int transaction_id = -1 );
```

The member function **wait_for_item_done** shall block the sequence until the driver calls **item_done** or **put** on a transaction issued by the specified sequence. If no *transaction_id* parameter is specified, then the call will return the next time that the driver calls **item_done** or **put**. If a specific *transaction_id* is specified, then the call will only return when the driver indicates that it has completed that specific item.

8.1.3.6 is_blocked

```
bool is_blocked( const uvm_sequence_base* sequence_ptr ) const;
```

The member function **is_blocked** shall return true if the sequence referred to by *sequence_ptr* is currently locked out of the sequencer. It shall return false if the sequence is currently allowed to issue operations.

Even when a sequence is not blocked, it is possible for another sequence to issue a lock before this sequence is able to issue a request or lock.

8.1.3.7 has_lock

```
bool has_lock( uvm_sequence_base* sequence_ptr );
```

The member function **has_lock** shall return true if the sequence referred to in the parameter currently has a lock on the sequencer; otherwise it shall return false. Even if this sequence has a lock, a child sequence may also have a lock, in which case the sequence is still blocked from issuing operations on the sequencer.

8.1.3.8 lock

```
virtual void lock( uvm_sequence_base* sequence_ptr );
```

The member function **lock** shall request a lock for the sequence specified by the specified argument *sequence_ptr*. A lock request will be arbitrated the same as any other request. A lock is granted after all earlier requests are completed and no other locks or grabs are blocking this sequence. The lock call shall return when the lock has been granted.

8.1.3.9 grab

```
virtual void grab( uvm_sequence_base* sequence_ptr );
```

The member function **grab** shall request a grab for the sequence specified by the specified argument *sequence_ptr*. A grab request is put in front of the arbitration queue. It will be arbitrated before any other requests. A grab is granted when no other grabs or locks are blocking this sequence. The grab call shall return when the grab has been granted.

8.1.3.10 unlock

```
virtual void unlock( uvm_sequence_base* sequence_ptr );
```

The member function **unlock** shall remove any locks and grabs obtained by the specified argument *sequence_ptr*.

8.1.3.11 ungrab

```
virtual void ungrab( uvm_sequence_base* sequence_ptr );
```

The member function **ungrab** shall remove any locks and grabs obtained by the specified argument *sequence_ptr*.

8.1.3.12 stop_sequences

```
virtual void stop_sequences();
```

The member function **stop_sequences** shall inform the the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks and responses that are currently queued. This

essentially resets the sequencer to an idle state.

8.1.3.13 is_grabbed

```
virtual bool is_grabbed() const;
```

The member function **is_grabbed** shall return true if any sequence currently has a lock or grab on this sequencer; otherwise it shall return false.

8.1.3.14 current_grabber

```
virtual uvm_sequence_base* current_grabber() const;
```

The member function **current_grabber** shall return a pointer to the sequence that currently has a lock or grab on the sequence. If multiple hierarchical sequences have a lock, it returns the child that is currently allowed to perform operations on the sequencer. Should return reference?

8.1.3.15 has_do_available

```
virtual bool has_do_available();
```

The member function **has_do_available** shall return true if any sequence running on this sequencer is ready to supply a transaction, otherwise it shall return false.

8.1.3.16 set_arbitration

```
void set_arbitration( SEQ_ARB_TYPE mode );
```

The member function **set_arbitration** shall set the arbitration mode for the sequencer. The argument *mode* shall be of type **SEQ_ARB_TYPE** and set to

- **SEQ_ARB_FIFO**: Requests are granted in FIFO order (default).
- **SEQ_ARB_WEIGHTED**: Requests are granted randomly by weight.
- **SEQ_ARB_RANDOM**: Requests are granted randomly.
- **SEQ_ARB_STRICT_FIFO**: Requests at highest priority granted in FIFO order.
- **SEQ_ARB_STRICT_RANDOM**: Requests at highest priority granted in randomly.
- **SEQ_ARB_USER**: Arbitration is delegated to the user-defined member function; **user_priority_arbitration**. That member function will specify the next sequence to grant.

The default arbitration mechanism shall be set to **SEQ_ARB_FIFO**.

8.1.3.17 get_arbitration

```
SEQ_ARB_TYPE get_arbitration() const;
```

The member function **get_arbitration** shall return the current arbitration mode set for the sequencer (see 8.1.3.16).

8.1.3.18 wait_for_sequences

```
virtual void wait_for_sequences();
```

The member function **wait_for_sequences** shall wait for a sequence to have a new item available.

8.1.3.19 send_request

```
virtual void send_request( uvm_sequence_base* sequence_ptr,
                           uvm_sequence_item* seq_item,
                           bool rerandomize = false);
```

Derived classes shall implement the member function **send_request** to send a request item to the sequencer, which shall forward it to the driver. (See 8.2.2).

This function shall only be called after a **wait_for_grant** call.

NOTE–Rerandomize capabilities are not yet implemented for UVM-SystemC.

8.2 uvm_sequencer_param_base

The class **uvm_sequencer_param_base** extends the base class **uvm_sequencer_base** for specific request (REQ) and response (RSP) types, which are specified as template arguments.

8.2.1 Class definition

```
namespace uvm {

  template <typename REQ = uvm_sequence_item, typename RSP = REQ>
  class uvm_sequencer_param_base : public uvm_sequencer_base
  {
   public:
    tlm::tlm_fifo<REQ> m_req_fifo;

    explicit uvm_sequencer_param_base( uvm_component_name name );

    // Group: Requests
    void send_request( uvm_sequence_base* sequence_ptr,
                       uvm_sequence_item* seq_item,
                       bool rerandomize = false );

    REQ get_current_item();

  }; // class uvm_sequencer_param_base

} // namespace uvm
```

8.2.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. These object types must be a derivative of class **uvm_sequence_item**.

8.2.3 Constructor

```
explicit uvm_sequencer_param_base( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

8.2.4 Requests

8.2.4.1 send_request

```
virtual void send_request( uvm_sequence_base* sequence_ptr,
                           uvm_sequence_item* seq_item,
                           bool rerandomize = false );
```

The member function **send_request** sends a request item pointed to by *seq_item* to the sequencer pointed to by *sequence_ptr*. The sequencer shall forward it to the driver. This member function shall only be called after a call to member function **wait_for_grant**.

NOTE–Rerandomize capabilities are not yet implemented.

8.2.4.2 get_current_item

```
REQ get_current_item();
```

The member function **get_current_item** shall return the requested item of type REQ, which is currently being executed by the sequencer. If the sequencer is not currently executing an item, this member function shall return NULL.

The sequencer is executing an item from the time that **get_next_item** or **peek** is called by the driver until the time that member function **get** or **item_done** is called by the driver. In case a driver calls member function **get**, the current item cannot be shown, since the item is completed at the same time as it is requested.

8.3 uvm_sequencer

The class **uvm_sequencer** defines the interface for the TLM communication of sequences or sequence-items by providing access via an export object of class **sc_export**.

8.3.1 Class definition

```
namespace uvm {


  template <typename REQ = uvm_sequence_item, typename RSP = REQ>
  class uvm_sequencer : public uvm_sequencer_param_base<REQ,RSP>,
```

```
                        public uvm_sqr_if_base<REQ, RSP>
  {
   public:
    explicit uvm_sequencer( uvm_component_name name );
    virtual ~uvm_sequencer();

    // Group: Exports
    sc_core::sc_export< uvm_sqr_if_base<REQ, RSP> > seq_item_export; or uvm_seq_item_pull_imp?

    // Group: Sequencer interface
    virtual REQ get_next_item( tlm::tlm_tag<REQ>* req = NULL );
    virtual bool try_next_item( REQ& req );
    virtual void item_done( const RSP& item, bool use_item = true );
    virtual void item_done();
    virtual REQ get( tlm::tlm_tag<REQ>* req = NULL );
    virtual void get( REQ& req );
    virtual REQ peek( tlm::tlm_tag<REQ>* req = NULL ); make argument const?
    virtual void put( const RSP& rsp );
    virtual void stop_sequences(); NEEDED?
    virtual void wait_for_sequences(); NEEDED?
    virtual bool has_do_available(); NEEDED?

  }; // class uvm_sequencer

} // namespace uvm
```

8.3.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. These object types must be a derivative of class **uvm_sequence_item**.

8.3.3 Constructor

```
explicit uvm_sequencer( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

8.3.4 Exports

8.3.4.1 seq_item_export

```
sc_core::sc_export< uvm_sqr_if_base<REQ, RSP> > seq_item_export;
```

The export **seq_item_export** shall provide access to the sequencer's implementation via the sequencer interface, **uvm_sqr_if_base** <REQ, RSP> (see 8.4).

8.3.5 Sequencer interface

8.3.5.1 get_next_item

```
virtual REQ get_next_item( tlm::tlm_tag<REQ>* req = NULL );
```

The member function **get_next_item** shall retrieve the next available item from a sequence (see also 8.4.3.1).

8.3.5.2 try_next_item

```
virtual bool try_next_item( REQ& req );
```

The member function **try_next_item** shall retrieve the next available item from a sequence if one is available (see also 8.4.3.2).

8.3.5.3 item_done

```
virtual void item_done( const RSP& item, bool use_item = true );
virtual void item_done();
```

The member function **item_done** shall indicate that the request is completed (see also 8.4.3.3).

8.3.5.4 get

```
virtual REQ get( tlm::tlm_tag<REQ>* req = NULL );
virtual void get( REQ& req );
```

The member function **get** shall retrieve the next available item from a sequence (see also 8.4.3.4).

8.3.5.5 peek

```
virtual REQ peek( tlm::tlm_tag<REQ>* req = NULL ); make argument const?
```

The member function **peek** shall return the current request item if one is in the FIFO (see also 8.4.3.5).

8.3.5.6 put

```
virtual void put( const RSP& rsp );
```

The member function **put** shall send a response back to the sequence that issued the request (see also 8.4.3.6).

8.3.5.7 stop_sequences

```
virtual void stop_sequences(); NEEDED?
```

The member function **stop_sequences** shall tell the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks and responses that are currently queued. This essentially resets the sequencer to an idle state.

8.3.5.8 wait_for_sequences

```
virtual void wait_for_sequences(); NEEDED?
```

The member function **wait_for_sequences** shall wait for a sequence to have a new item available (see also 8.1.3.18).

8.3.5.9 has_do_available

```
virtual bool has_do_available(); NEEDED?
```

The member function **has_do_available** shall return true if any sequence running on this sequencer is ready to supply a transaction, otherwise it shall return false.

8.3.6 Macros

8.3.6.1 UVM_DECLARE_P_SEQUENCER

```
UVM_DECLARE_P_SEQUENCER( SEQUENCER )
```

The macro **UVM_DECLARE_P_SEQUENCER** shall declare a variable **p_sequencer** whose type is specified by the argument *SEQUENCER*.

8.4 uvm_srq_if_base

The class **uvm_srq_if_base** shall define an interface for sequence drivers to communicate with sequencers. The driver requires the interface via a port, and the sequencer implements it and provides it via an export.

8.4.1 Class definition

```
namespace uvm {


  template <typename REQ, typename RSP = REQ>
  class uvm_sqr_if_base : public virtual sc_core::sc_interface
  {
   public:
    virtual REQ get_next_item( tlm::tlm_tag<REQ> *req = NULL ) = 0;
    virtual bool try_next_item( REQ& req ) = 0;
    virtual void item_done( const RSP& item, bool use_item = true ) = 0;
```

```
    virtual void item_done() = 0;
    virtual void put( const RSP& rsp ) = 0;
    virtual REQ get( tlm::tlm_tag<REQ> *req = NULL ) = 0;
    virtual REQ peek( tlm::tlm_tag<REQ> *req = NULL ) = 0; // make const method

}; // class uvm_sqr_if_base

} // namespace uvm
```

8.4.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. These object types must be a derivative of class **uvm_sequence_item**.

8.4.3 Member functions

8.4.3.1 get_next_item

```
virtual REQ get_next_item( tlm::tlm_tag<REQ> *req = NULL ) = 0;
```

The member function **get_next_item** shall retrieve the next available item from a sequence. The call will block until an item is available. The following steps occur on this call:

1. Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.
2. The chosen sequence will return from member function **wait_for_grant** (see 9.3.6.4).
3. The chosen sequence's member function **uvm_sequence_base**::**pre_do** is called (see 9.3.4.4).
4. The chosen sequence item is randomized. NOT IMPLEMENTED
5. The chosen sequence's member function **uvm_sequence_base**::**post_do** is called (see 9.3.4.7).
6. Return with a reference to the item.

Once member function **get_next_item** is called, the member function **item_done** must be called to indicate the completion of the request to the sequencer. This will remove the request item from the sequencer FIFO.

8.4.3.2 try_next_item

```
virtual bool try_next_item( REQ& req ) = 0;
```

The member function **try_next_item** shall retrieve the next available item from a sequence if one is available. If available, it shall return true. Otherwise, the member function shall return false. The following steps occur on this call:

1. Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, the member function returns false.

2. The chosen sequence will return from member function **uvm_sequence_base**::**wait_for_grant** (see 9.3.6.4).
3. The chosen sequence's member function **uvm_sequence_base**::**pre_do** is called (see 9.3.4.4).
4. The chosen sequence item is randomized. NOT IMPLEMENTED
5. The chosen sequence **uvm_sequence_base**::**post_do** is called (see 9.3.4.7).
6. Return with a reference to the item.

Once the member function **try_next_item** is called, the member function **item_done** must be called to indicate the completion of the request to the sequencer. This will remove the request item from the sequencer FIFO.

8.4.3.3 item_done

```
virtual void item_done( const RSP& item, bool use_item = true ) = 0;
virtual void item_done() = 0;
```

The member function **item_done** shall indicate that the request is completed to the sequencer. Any **uvm_sequence_base**::**wait_for_item_done** calls made by a sequence for this item will return.

The current item is removed from the sequencer FIFO.

If a response item is provided, then it will be sent back to the requesting sequence. The response item must have its sequence ID and transaction ID set correctly, using the member function **uvm_sequence_item**::**set_id_info**.

Before the member function **item_done** is called, any calls to the member function **peek** will retrieve the current item that was obtained by member function **get_next_item**. After the member function **item_done** is called, member function **peek** will cause the sequencer to arbitrate for a new item.

8.4.3.4 get

```
virtual REQ get( tlm::tlm_tag<REQ> *req = NULL ) = 0;
```

The member function get shall retrieve the next available item from a sequence. The call blocks until an item is available. The following steps occur on this call:

1. Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.
2. The chosen sequence will return from member function **uvm_sequence_base**::**wait_for_grant** (see 9.3.6.4).
3. The chosen sequence's member function **uvm_sequence_base**::**pre_do** is called (see 9.3.4.4).
4. The chosen sequence item is randomized. NOT IMPLEMENTED YET
5. The chosen sequence's member function **uvm_sequence_base**::**post_do** is called (see 9.3.4.7).
6. Indicate **item_done** to the sequencer
7. Return with a reference to the item.

When the member function **get** is called, the member function **item_done** may not be called. A new item can be obtained by calling the member function **get** again, or a response may be sent using either member function **put**, or **uvm_driver**::**rsp_port**.**write**(). rsp_port not implemented yet

8.4.3.5 peek

```
virtual void put( const RSP& rsp ) = 0;
```

The member function **peek** shall return the current request item if one is in the sequencer FIFO. If no item is in the FIFO, then the call will block until the sequencer has a new request. The following steps will occur if the sequencer FIFO is empty:

1. Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.
2. The chosen sequence will return from member function **uvm_sequence_base**::**wait_for_grant** (see 9.3.6.4).
3. The chosen sequence's member function **uvm_sequence_base**::**pre_do** is called (see 9.3.4.4).
4. The chosen sequence item is randomized. NOT IMPLEMENTED YET
5. The chosen sequence's member function **uvm_sequence_base**::**post_do** is called (see 9.3.4.7).

Once a request item has been retrieved and is in the sequencer FIFO, subsequent calls to member function **peek** will return the same item. The item will stay in the FIFO until either the member function **get** or **item_done** is called.

8.4.3.6 put

```
virtual void put( const RSP& rsp ) = 0;
```

The member function **put** shall send a response back to the sequence that issued the request. Before the response is put, it must have its sequence ID and transaction ID set to match the request. This can be done using the member function **uvm_sequence_item**::**set_id_info**.

This member function will not block. The response will be put into the sequence response queue or it will be sent to the sequence response handler.

9. Sequence classes

The sequence classes offer the infrastructure to create stimuli descriptions based on transactions, encapsulated as a sequence or sequence item. As the sequences and sequence items only describe stimuli, they are independent and thus not part of the structural hierarchy of a UVM agent (in which sequencer, driver and monitor resides). Instead, they are included at a higher functional layer defined within the UVM environment (e.g. encapsulated within a verification component derived from class **uvm_env**) or as part of a UVM test environment (component derived from class **uvm_test**).

The following sequence classes are defined:

- **uvm_transaction**
- **uvm_sequence_item**
- **uvm_sequence_base**
- **uvm_sequence**

When sequences are executed parallel, the sequencer will arbitrate among the parallel sequences. By default, requests are granted in a first-in-first-out (FIFO) order. Other arbitration modes are not yet support in UVM-SystemC.

NOTE–The UVM-SystemC sequence classes only partially implement the standardized UVM sequence capabilities. However, these definitions are sufficient to create UVM compatible sequences.

9.1 uvm_transaction

The class **uvm_transaction** is the root base class for all UVM transactions. As such, the class **uvm_sequence_item** will be derived from this class. The main purpose of this class is to provide timestamp properties, notification events, and transaction recording.

9.1.1 Class definition

```
namespace uvm {


  class uvm_transaction : public uvm_object
  {
   public:
    uvm_transaction();
    explicit uvm_transaction( const std::string& name );


    void set_transaction_id( int id );
    int get_transaction_id() const;


  }; // class uvm_transaction


} // namespace uvm
```

9.1.2 Constructors

```
uvm_transaction();

explicit uvm_transaction( const std::string& name );
```

The constructor shall create and initialize an instance of the class, which is derived from class **uvm_object**, with the name *name* passed as an argument. Add statement on default name

9.1.3 Constraints on usage

An application shall not create transactions based on this base class. Instead, it shall use the class **uvm_sequence_item** or class **uvm_sequence**.

9.1.4 Member functions

9.1.4.1 set_transaction_id

```
void set_transaction_id( int id );
```

The member function **set_transaction_id** shall set the transaction's numeric identifier (ID), passed as argument *id*. If the transaction ID is not set via this member function, the transaction ID defaults to -1.

When using sequences to generate stimulus, the transaction ID is used along with the sequence ID to route responses in sequencers and to correlate responses to requests.

9.1.4.2 get_transaction_id

```
int get_transaction_id() const;
```

The member function **get_transaction_id** shall return the transaction's numeric identifier (ID), which is -1 if not set explicitly by **set_transaction_id**.

When using an object derived from class **uvm_sequence** <REQ, RSP> to generate stimulus, the transaction ID is used along with the sequence ID to route responses in sequencers and to correlate responses to requests.

9.2 uvm_sequence_item

The class **uvm_sequence_item** is the base class for application-defined sequence items and also serves as the base class for class **uvm_sequence**. The class **uvm_sequence_item** provides basic functionality for transactional objects, both sequence items and sequences, to operate in the sequence mechanism.

9.2.1 Class definition

```
namespace uvm {


  class uvm_sequence_item : public uvm_transaction
  {
   public:
```

```
    uvm_sequence_item();
    explicit uvm_sequence_item( const std::string& name );
    virtual ~uvm_sequence_item();

    void set_use_sequence_info( bool value );
    bool get_use_sequence_info() const;
    void set_id_info( uvm_sequence_item& item );
    virtual void set_sequencer( uvm_sequencer_base* sequencer );
    uvm_sequencer_base* get_sequencer() const;
    void set_parent_sequence( uvm_sequence_base* parent );
    uvm_sequence_base* get_parent_sequence() const;
    void set_depth( int value );
    int get_depth() const;
    virtual bool is_item() const;
    const std::string get_root_sequence_name() const;
    const uvm_sequence_base* get_root_sequence() const;
    const std::string get_sequence_path() const;

  }; // class uvm_sequence_item

} // namespace uvm
```

9.2.2 Constructors

```
uvm_sequence_item();
explicit uvm_sequence_item( const std::string& name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

9.2.3 Member functions

9.2.3.1 set_use_sequence_info

```
void set_use_sequence_info( bool value );
```

The member function **set_use_sequence_info** shall enable or disable printing, copying, or recording of sequence information (sequencer, parent_sequence, sequence_id, etc.). When the argument of this member function is set to false, then the usage of sequence information shall be disabled. When the argument of this member function is set to true, the printing and copying of sequence information shall be enabled.

9.2.3.2 get_use_sequence_info

```
bool get_use_sequence_info() const;
```

The member function **get_use_sequence_info** shall return true if the usage of sequence information, such as printing and copying of sequence information, has been enabled. The member function shall return false if the usage of sequence information has been disabled.

9.2.3.3 set_id_info

```
void set_id_info( uvm_sequence_item& item );
```

The member function **set_id_info** shall copy the sequence ID and transaction ID from the referenced item into the calling item. This routine should always be used by drivers to initialize responses for future compatibility.

9.2.3.4 set_sequencer

```
virtual void set_sequencer( uvm_sequencer_base* sequencer );
```

The member function **set_sequencer** shall set the default sequencer, passed as argument, to be used for the sequence or sequence item for which this member function is called. It shall take effect immediately, so it should not be called while the sequence is actively communicating with the sequencer.

9.2.3.5 get_sequencer

```
uvm_sequencer_base* get_sequencer() const;
```

The member function **get_sequencer** shall return a pointer to the default sequencer used by the sequence or sequence item for which this member function is called.

9.2.3.6 set_parent_sequence

```
void set_parent_sequence( uvm_sequence_base* parent );
```

The member function **set_parent_sequence** shall set the parent sequence, passed as an argument, of the sequence or sequence item.

9.2.3.7 get_parent_sequence

```
uvm_sequence_base* get_parent_sequence() const;
```

The member function **set_parent_sequence** shall return a pointer to the parent sequence of any sequence for which this member function was called. If this is a parent sequence, the member function shall return NULL.

9.2.3.8 set_depth

```
void set_depth( int value );
```

The member function **set_depth** shall set the depth of a particular sequence. If this member function is not called, the depth of any sequence shall be calculated automatically. When called, the member function shall override the automatically calculated depth, even if it is incorrect.

9.2.3.9 get_depth

```
int get_depth() const;
```

The member function **get_depth** shall return the depth of sequence from its parent. A parent sequence will have a depth of 1, its child will have a depth of 2, and its grandchild will have a depth of 3.

9.2.3.10 is_item

```
virtual bool is_item() const;
```

The member function **is_item** shall return true when the object for which the member function is called is derived from **uvm_sequence_item**. It shall return false if the object is derived from class **uvm_sequence**.

9.2.3.11 get_root_sequence_name

```
const std::string get_root_sequence_name() const;
```

The member function **get_root_sequence_name** shall provide the name of the root sequence (the top-most parent sequence).

9.2.3.12 get_root_sequence

```
const uvm_sequence_base* get_root_sequence() const;
```

The member function **get_root_sequence** shall provide a reference to the root sequence (the top-most parent sequence).

9.2.3.13 get_sequence_path

```
const std::string get_sequence_path() const;
```

The member function **get_sequence_path** shall provide a string of names of each sequence in the full hierarchical path. The dot character '.' is used as the separator between each sequence.

9.3 uvm_sequence_base

The class **uvm_sequence_base** defines the primary interface member functions to create, control and execute the sequences.

9.3.1 Class definition

```
namespace uvm {


  class uvm_sequence_base : public uvm_sequence_item
  {
   public:
    explicit uvm_sequence_base( const std::string& name );
```

```
  virtual ~uvm_sequence_base();

  // Group: Sequence state
  uvm_sequence_state_enum get_sequence_state() const;
  void wait_for_sequence_state( uvm_sequence_state_enum state );

  // Group: Sequence execution
  virtual void start( uvm_sequencer_base* sqr,
                      uvm_sequence_base* parent_sequence = NULL,
                      int this_priority = -1,
                      bool call_pre_post = true );

  virtual void pre_start();
  virtual void pre_body();
  virtual void pre_do( bool is_item );
  virtual void mid_do( uvm_sequence_item* this_item );
  virtual void body();
  virtual void post_do( uvm_sequence_item* this_item );
  virtual void post_body();
  virtual void post_start();

  // Group: Sequence control
  void set_priority( int value );
  int get_priority() const;
  virtual bool is_relevant() const;
  virtual void wait_for_relevant();NOT IMPLEMENTED YET
  void lock( uvm_sequencer_base* sequencer = NULL );
  void grab( uvm_sequencer_base* sequencer = NULL );
  void unlock( uvm_sequencer_base* sequencer = NULL );
  void ungrab( uvm_sequencer_base* sequencer = NULL );
  bool is_blocked() const;
  bool has_lock();
  void kill();
  virtual void do_kill();

  // Group: Sequence item execution
  uvm_sequence_item* create_item( uvm_object_wrapper* type_var,
                                  uvm_sequencer_base* l_sequencer,
```

```
                                    const std::string& name );

    virtual void start_item( uvm_sequence_item* item,
                             int set_priority = -1,
                             uvm_sequencer_base* sequencer = NULL );

    virtual void finish_item( uvm_sequence_item* item,
                              int set_priority = -1 );

    virtual void wait_for_grant( int item_priority = -1,
                                 bool lock_request = false );

    virtual void send_request( uvm_sequence_item* request,
                               bool rerandomize = false );

    virtual void wait_for_item_done( int transaction_id = -1 );

    // Group: Response interface
    void use_response_handler( bool enable );
    bool get_use_response_handler() const;
    virtual void response_handler( const uvm_sequence_item* response );
    void set_response_queue_error_report_disabled( bool value ); NOT IMPLEMENTED YET
    bool get_response_queue_error_report_disabled() const; NOT IMPLEMENTED YET
    void set_response_queue_depth( int value ); NOT IMPLEMENTED YET
    int get_response_queue_depth() const; NOT IMPLEMENTED YET
    virtual void clear_response_queue();

    // Data members
    uvm_phase* starting_phase;

  }; // class uvm_sequence_base

} // namespace uvm
```

9.3.2 Constructor

```
explicit uvm_sequence_base( const std::string& name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

9.3.3 Sequence state

9.3.3.1 get_sequence_state

```
uvm_sequence_state_enum get_sequence_state() const;
```

The member function **get_sequence_state** shall return the sequence state as an enumerated value of type **uvm_sequence_state_enum** (see 15.3.5). This member function can be used to wait on the sequence reaching or changing from one or more states.

9.3.4 Sequence execution

9.3.4.1 start

```
virtual void start( uvm_sequencer_base* sequencer,
                    uvm_sequence_base* parent_sequence = NULL,
                    int this_priority = -1,
                    bool call_pre_post = true );
```

The member function **start** shall execute the sequence. The argument *sequencer* specifies the sequencer on which to run this sequence. The sequencer must be compatible with the sequence, that is, the sequencer shall recognize the communicated request and response types.

If *parent_sequence* is not passed as argument or set to NULL, then the sequence is treated as a root sequence, otherwise it is a child of a parent sequence. In the latter case, the parent sequence's member functions **pre_do**, **mid_do**, and **post_do** shall be called during the execution of this sequence.

If *this_priority* is not passed as argument or set to -1, the priority of a sequence is set to priority of its parent sequence. If it is a root (parent) sequence, its default priority is 100. A different priority greater than zero may be specified using this argument. Higher numbers indicate higher priority.

If argument *call_pre_post* is not passed or set to true, then the member functions **pre_body** and **post_body** will be called before and after calling the member function **body** of the sequence.

9.3.4.2 pre_start

```
virtual void pre_start();
```

The member function **pre_start** shall be provided as a callback for the application that is called before the optional execution of member function **pre_body**. The application shall not call this member function.

9.3.4.3 pre_body

```
virtual void pre_body();
```

The member function **pre_body** shall be provided as a callback for the application that is called before the execution of member function **body**, but only when the sequence is started by using member function **start**. If **start** is called with argument *call_pre_post* set to false, the member function **pre_body** shall not be called. The application shall not call this member function.

9.3.4.4 pre_do

```
virtual void pre_do( bool is_item );
```

The member function **pre_do** shall be provided as a callback for the application that is called on the parent sequence, if the sequence has issued a **wait_for_grant** call and after the sequencer has selected this sequence, and before the item is randomized. The application shall not call this member function.

9.3.4.5 mid_do

```
virtual void mid_do( uvm_sequence_item* this_item );
```

The member function **mid_do** shall be provided as a callback for the application that is called after the sequence item has been randomized, and just before the item is sent to the driver. The application shall not call this member function.

9.3.4.6 body

```
virtual void body();
```

The member function **body** shall be provided as a callback for the application that is called before the optional execution of member function **post_body**. The application shall not call this member function.

NOTE–In an application, the implementation of the sequence resides in this member function.

9.3.4.7 post_do

```
virtual void post_do( uvm_sequence_item* this_item );
```

The member function **post_do** shall be provided as a callback for the application that is called after the driver has indicated that it has completed the sequence item, calling either the member function **item_done** or **put**. The application shall not call this member function.

9.3.4.8 post_body

```
virtual void post_body();
```

The member function **post_body** shall be provided as a callback for the application that is called before the execution of member function **post_start**, but only when the sequence is started by using member function **start**. If **start** is called with argument *call_pre_post* set to false, the member function **post_body** shall not be called. The application shall not call this member function.

9.3.4.9 post_start

```
virtual void post_start();
```

The member function **post_start** shall be provided as a callback for the application that is called after the optional execution of member function **post_body**. The application shall not call this member function.

9.3.5 Sequence control

9.3.5.1 set_priority

```
void set_priority( int value );
```

The member function **set_priority** shall set the priority of a sequence. The default priority value for a sequence is 100. Higher values result in higher priorities. When the priority of a sequence is changed, the new priority will be used by the sequencer the next time that it arbitrates between sequences.

9.3.5.2 get_priority

```
int get_priority() const;
```

The member function **get_priority** shall return the current priority of the sequence.

9.3.5.3 is_relevant

```
virtual bool is_relevant() const;
```

The member function **is_relevant** shall mark a sequence as being relevant or not. By default, the member function **is_relevant** shall return true, indicating that the sequence is always relevant.

An application may choose to overload this member function to indicate to the sequencer that the sequence is not currently relevant after a request has been made. Any sequence that implements the member function **is_relevant** shall also implement **wait_for_relevant**, to enable a sequencer to wait for a sequence to become relevant.

When the sequencer arbitrates, it shall call the member function **is_relevant** on each requesting, unblocked sequence to see if it is relevant. If this member function returns false, then the sequence will not be chosen.

If all requesting sequences are not relevant, then the sequencer shall call **wait_for_relevant** on all sequences and re-arbitrate upon its return.

9.3.5.4 lock

```
void lock( uvm_sequencer_base* sequencer = NULL );
```

The member function **lock** shall request a lock on the specified sequencer. If sequencer is NULL, the lock will be requested on the current default sequencer. A lock request will be arbitrated the same as any other request. A lock is granted after all earlier requests are completed and no other locks or grabs are blocking this sequence. The lock call shall return when the lock has been granted.

9.3.5.5 grab

```
void grab( uvm_sequencer_base* sequencer = NULL );
```

The member function **grab** shall request a lock on the specified sequencer. If sequencer is NULL, the grab will be requested on the current default sequencer. A grab request is put in front of the arbitration queue. It will be arbitrated before any other requests. A grab is granted when no other grabs or locks are blocking this sequence. The grab call shall return when the grab has been granted.

9.3.5.6 unlock

```
void unlock( uvm_sequencer_base* sequencer = NULL );
```

The member function **unlock** shall remove any locks or grabs obtained by this sequence on the specified sequencer. If the sequencer is NULL, then the unlock will be done on the current default sequencer.

9.3.5.7 ungrab

```
void ungrab( uvm_sequencer_base* sequencer = NULL );
```

The member function **ungrab** shall remove any locks or grabs obtained by this sequence on the specified sequencer. If the sequencer is NULL, then the ungrab will be done on the current default sequencer.

9.3.5.8 is_blocked

```
bool is_blocked() const;
```

The member function **is_blocked** shall return a Boolean type indicating whether this sequence is currently prevented from running due to another lock or grab. A true is returned if the sequence is currently blocked. A false is returned if no lock or grab prevents this sequence from executing. Even if a sequence is not blocked, it is possible for another sequence to issue a lock or grab before this sequence can issue a request.

9.3.5.9 has_lock

```
bool has_lock();
```

The member function **has_lock** shall return true if this sequence has a lock; otherwise it shall return false. Even if this sequence has a lock, a child sequence may also have a lock, in which case the sequence is still blocked from issuing operations on the sequencer.

9.3.5.10 kill

```
    void kill();
```

The member function **kill** shall shall kill the sequence, and cause all current locks and requests in the sequence's default sequencer to be removed. The sequence state shall be changed to STOPPED and the callback functions **post_body** and **post_start** are not being executed.

9.3.5.11 do_kill

```
virtual void do_kill();
```

The member function **do_kill** shall provide a callback for an application that is called whenever a sequence is terminated by using either **kill** or **stop_sequences**.

9.3.6 Sequence item execution

9.3.6.1 create_item

```
uvm_sequence_item* create_item( uvm_object_wrapper* type_var,
                                uvm_sequencer_base* l_sequencer,
                                const std::string& name );
```

The member function **create_item** shall create and initialize a sequence item of class **uvm_sequence_item** or sequence of class **uvm_sequence** using the factory. The type of the created object, being a sequence item or sequence, is defined by the first argument *type_var*, which shall be of type **uvm_sequence_item** or **uvm_sequence** only. The sequence item or sequence will be initialized to communicate with the specified sequencer *l_sequencer* passed as second argument. The *name* of the created item shall be passed as third argument.

9.3.6.2 start_item

```
virtual void start_item( uvm_sequence_item* item,
                         int set_priority = -1,
                         uvm_sequencer_base* sequencer = NULL );
```

The member function **start_item** shall initiate execution of a sequence item specified as argument *item*. If the item has not already been initialized using member function **create_item**, then it will be initialized here by using the sequencer specified by argument *sequencer*. If argument *sequencer* is not specified or set to NULL, the default sequencer will be used (see also 9.2.3.4). The argument *set_priority* can be used to specify the priority for the execution. If argument *set_priority* is not specified or set to -1, the default priority shall be 100. Randomization, or other member functions, may be done between **start_item** and **finish_item** to ensure late generation. CHECK

9.3.6.3 finish_item

```
    virtual void finish_item( uvm_sequence_item* item,
                              int set_priority = -1 );
```

The member function **finish_item** shall finalize execution of execution of a sequence item specified as argument *item*. The member function shall be called after **start_item** with no delays or delta-cycles. The argument *set_priority* can be used to specify the priority for the execution. If argument *set_priority* is not specified or set to -1, the default priority shall be 100. Randomization, or other member functions, may be called between **start_item** and **finish_item**.

9.3.6.4 wait_for_grant

```
virtual void wait_for_grant( int item_priority = -1,
                             bool lock_request = false );
```

The member function **wait_for_grant** shall issue a request to the current sequencer. If argument *item_priority* is not specified or set to -1, then the current sequence priority will be used by the arbiter. If the argument *lock_request* is set to true, then the sequencer will issue a lock immediately before granting the sequence.

NOTE–The lock may be granted without the sequence being granted if member function **is_relevant** is not asserted.

9.3.6.5 send_request

```
virtual void send_request( uvm_sequence_item* request,
                           bool rerandomize = false );
```

The member function **send_request** shall send the request item, passed as an argument, to the sequencer, which shall forward it to the driver. If argument *rerandomize* is set to true, the item will be randomized before being sent to the driver. RANDOMIZATION NOT IMPLEMENTED YET

NOTE–In an application, the member function **send_request** shall only be called after a call to **wait_for_grant**.

9.3.6.6 wait_for_item_done

```
virtual void wait_for_item_done( int transaction_id = -1 );
```

The member function **wait_for_item_done** shall block until the driver calls **item_done** or **put**. If no *transaction_id* argument is specified, then the call will return the next time that the driver calls **item_done** or **put**. If a specific *transaction_id* is specified, then the call will return when the driver indicates completion of that specific item.

NOTE–If a specific *transaction_id* has been specified, and the driver has already issued an **item_done** or put for that transaction, then the call will hang, having missed the earlier notification.

9.3.7 Response interface

9.3.7.1 use_response_handler

```
void use_response_handler( bool enable );
```

The member function **use_response_handler** shall send responses to the response handler when argument *enable* is set to true. By default, responses from the driver are retrieved in the sequence by calling member function **get_response**.

9.3.7.2 get_use_response_handler

```
bool get_use_response_handler() const;
```

The member function **get_use_response_handler** shall return the state set by **use_response_handler**. If this member function returns false, the response handler is disabled.

9.3.7.3 response_handler

```
virtual void response_handler( const uvm_sequence_item* response );
```

The member function **response_handler** shall be provided to enable the sequencer, in case returns true, to call this member function for each response that arrives for this sequence.

9.3.7.4 set_response_queue_error_report_disabled

```
void set_response_queue_error_report_disabled( bool value ); NOT IMPLEMENTED YET
```

The member function **set_response_queue_error_report_disabled** shall enable error reporting of overflows of the reponse queue. The response queue will overflow if more responses are sent to this sequence from the driver than calls to member function **get_response** are made. If argument *value* is set to false, error reporting is disabled. If argument *value* is set to true, error reporting is enabled. By default, if the response queue overflows, an error is reported.

9.3.7.5 get_response_queue_error_report_disabled

```
bool get_response_queue_error_report_disabled() const; NOT IMPLEMENTED YET
```

The member function **get_response_queue_error_report_disabled** shall return the reporting status of an overflow of the response queue. It returns false when error reports are generated and returns true if no such error reports are generated.

9.3.7.6 set_response_queue_depth

```
void set_response_queue_depth( int value ); NOT IMPLEMENTED YET
```

The member function **set_response_queue_depth** shall set the depth of the reponse queue. The default maximum depth of the response queue is 8. An argument *value* of -1 defines an unbound response queue.

9.3.7.7 get_response_queue_depth

```
int get_response_queue_depth() const; NOT IMPLEMENTED YET
```

The member function **get_response_queue_depth** shall return the current depth for the response queue. An unbound response queue returns the value -1.

9.3.7.8 clear_response_queue

```
virtual void clear_response_queue();
```

The member function **clear_response_queue** shall empty the response queue for the sequence.

9.3.8 Data members

9.3.8.1 starting_phase

```
uvm_phase* starting_phase;
```

The data member *starting_phase* shall specify the phase in which this sequence was started. The *starting_phase* shall be set when the sequence is started as the default sequence (see 8.1.3.3).

9.4 uvm_sequence

The class **uvm_sequence** extends the base class **uvm_sequence_base** for specific request (REQ) and response (RSP) types, which are specified as template arguments.

9.4.1 Class definition

```
namespace uvm {


  template <typename REQ = uvm_sequence_item, typename RSP = REQ>
  class uvm_sequence : public uvm_sequence_base
  {
   public:
    explicit uvm_sequence( const std::string& name );
    virtual ~uvm_sequence();

    void send_request( uvm_sequence_item* request,
                       bool rerandomize = false );

    REQ get_current_item();

    virtual void get_response( RSP*& response,
                               int transaction_id = -1 );

  }; // class uvm_sequence

} // namespace uvm
```

9.4.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. These object types must be a derivative of class **uvm_sequence_item**.

9.4.3 Constructor

```
explicit uvm_sequence( const std::string& name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

9.4.4 Member functions

9.4.4.1 send_request

```
void send_request( uvm_sequence_item* request,
                   bool rerandomize = false );
```

The member function **send_request** shall send the request item, passed as an argument, to the sequencer, which shall forward it to the driver. If argument *rerandomize* is set to true, the item will be randomized before being sent to the driver. RANDOMIZATION NOT IMPLEMENTED YET

NOTE–In an application, the member function **send_request** shall only be called after a call to **wait_for_grant**.

9.4.4.2 get_current_item

```
REQ get_current_item();
```

The member function **get_current_item** shall return the request item currently being executed by the sequencer. If the sequencer is not currently executing an item, this method will return NULL. The sequencer is executing an item from the time that **get_next_item** or **peek** is called until the time that get or **item_done** is called.

NOTE–A driver that only calls **get** will never show a current item, since the item is completed at the same time as it is requested.

9.4.4.3 get_response

```
virtual void get_response( RSP*& response,
                           int transaction_id = -1 );
```

The member function **get_response** shall retrieve a response via the response queue. If no response is available in the response queue, the member function will block until a response is received.

If no *transaction_id* is passed as an argument, this member function will return the next response sent to this sequence. If a *transaction_id* is specified, the member function will block until a response with that transaction ID is received in the response queue.

10. Configuration and resource classes

The configuration and resource classes provide access to a centralized database where type specific information can be stored and retrieved. A configuration or resource item may be associated with a specific hierarchical scope of an object derived from class **uvm_component** or it may be visible to all components regardless of their hierarchical position.

The following configuration and resource classes are defined:

- **uvm_config_db**
- **uvm_resource_db**
- **uvm_resource_db_options**
- **uvm_resource_options**
- **uvm_resource_base**
- **uvm_resource_pool**
- **uvm_resource**
- **uvm_resource_types**
- **uvm_queue**
- **uvm_pool**

10.1 uvm_config_db

The class **uvm_config_db** provides a typed interface for object-centric configuration. It is consistent with the configuration mechanism as defined for the class **uvm_component**. Information can be read from or written to the database at any time during simulation.

10.1.1 Class definition

```
namespace uvm {


  template <class T>
  class uvm_config_db
  {
   public:
    uvm_config_db();
    static void set( uvm_component* cntxt,
                     const std::string& inst_name,
                     const std::string& field_name,
                     const T& value );


    static bool get( uvm_component* cntxt,
```

```
                    const std::string& inst_name,
                    const std::string& field_name,
                    T& value );

    static bool exists( uvm_component* cntxt,
                        const std::string& inst_name,
                        const std::string& field_name,
                        bool spell_chk = false );

    static void wait_modified( uvm_component* cntxt,
                               const std::string& inst_name,
                               const std::string& field_name );

  }; // class uvm_config_db

} // namespace uvm
```

10.1.2 Template parameter T

The template parameter T specifies the object type of the objects being stored in or retrieved from the configuration database.

10.1.3 Constraints on usage

To remain compatible with UVM-SystemVerilog, all of the member functions in class **uvm_config_db** are static, so they must be called using the **operator::**.

10.1.4 Member functions

10.1.4.1 set

```
static void set( uvm_component* cntxt,
                 const std::string& instname,
                 const std::string& fieldname,
                 const T& value );
```

The member function **set** shall create a new or update an existing configuration setting using target field *field_name* in instance with name *inst_name* from the context *cntxt* in which it is defined. If argument *cntxt* is set to NULL, then *inst_name* defines the complete scope for the configuration setting; otherwise, the full name of the component referenced to by *cntxt* shall be added to the instance name. An application may define *inst_name* and *field_name* to be glob-style or regular expression style expressions.

10.1.4.2 get

```
static bool get( uvm_component* cntxt,
                 const std::string& instname,
                 const std::string& fieldname,
                 T& value );
```

The member function **get** shall retrieve a configuration setting via arguments *inst_name* and *field_name*, using a component pointer *cntxt* as the starting search point. The argument *inst_name* shall be an explicit instance name relative to *cntxt* and may be an empty string if the *cntxt* is the instance that the configuration object applies to. The argument *field_name* is the specific field in the scope that is being searched for.

The member function returns true if the value is being found; otherwise, false is returned.

10.1.4.3 exists

```
static bool exists( uvm_component* cntxt,
                    const std::string& inst_name,
                    const std::string& field_name,
                    bool spell_chk = false );
```

The member function **exists** shall check if a value for *field_name* is available in *inst_name*, using component *cntxt* as the starting search point. *inst_name* is an explicit instance name relative to cntxt and may be an empty string if the cntxt is the instance that the configuration object applies to. *field_name* is the specific field in the scope that is being searched for. The argument *spell_chk* can be set to true to turn spell checking on if it is expected that the field should exist in the database. The function returns true if a config parameter exists and false if it does not exist.

10.1.4.4 wait_modified

```
static void wait_modified( uvm_component* cntxt,
                           const std::string& inst_name,
                           const std::string& field_name );
```

The member function **wait_modified** shall wait for a configuration setting to be set for *field_name* in *cntxt* and *inst_name*. The member function blocks until a new configuration setting is applied that effects the specified field.

10.2 uvm_resource_db

The class **uvm_resource_db** provides a convenience interface for the resources facility. In many cases basic operations such as creating and setting a resource or getting a resource could take multiple lines of code using the interfaces in class **uvm_resource_base** or class **uvm_resource**. The convenience layer in class **uvm_resource_db** reduces many of those operations to a single line of code.

10.2.1 Class definition

```
namespace uvm {
```

```
template < typename T = uvm_object* >
class uvm_resource_db
{
 public:
  static uvm_resource<T>* get_by_type( const std::string& scope );

  static uvm_resource<T>* get_by_name( const std::string& scope,
                                       const std::string& name,
                                       bool rpterr = true );

  static uvm_resource<T>* set_default( const std::string& scope,
                                       const std::string& name );

  static void set( const std::string& scope,
                   const std::string& name,
                   const T& val,
                   uvm_object* accessor = NULL );

  static void set_anonymous( const std::string& scope,
                             const T& val,
                             uvm_object* accessor = NULL );

  static bool read_by_name( const std::string& scope,
                            const std::string& name,
                            T val,
                            uvm_object* accessor = NULL );

  static bool read_by_type( const std::string& scope,
                            T val,
                            uvm_object* accessor = NULL );

  static bool write_by_name( const std::string& scope,
                             const std::string& name,
                             const T& val,
                             uvm_object* accessor = NULL );

  static bool write_by_type( const std::string& scope,
                             const T& val,
```

```
                                uvm_object* accessor = NULL );

    static void dump();

   private:
    // disabled
    uvm_resource_db();

  }; // class uvm_config_db

} // namespace uvm
```

10.2.2 Template parameter T

The template parameter T specifies the object type of the objects being stored in or retrieved from the resource database.

10.2.3 Constraints on usage

To remain compatible with UVM-SystemVerilog, all of the member functions in class **uvm_resource_db** are static, so they must be called using the **operator::**. An application shall not instantiate this class, but shall call the static member functions directly.

10.2.4 Member functions

10.2.4.1 get_by_type

```
static uvm_resource<T>* get_by_type( const std::string& scope );
```

The member function **get_by_type** shall return the resource by type. The type is specified in the database class parameter so the only argument to this member function is the scope.

10.2.4.2 get_by_name

```
static uvm_resource<T>* get_by_name( const std::string& scope,
                                     const std::string& name,
                                     bool rpterr = true );
```

The member function **get_by_name** shall return the resource by name. The first argument is the current scope and the second argument is the name of the resource to be retrieved. If the argument *rpterr* is set to true, a warning shall be generated if no matching resource is found.

10.2.4.3 set_default

```
static uvm_resource<T>* set_default( const std::string& scope,
```

```
                                        const std::string& name );
```

The member function **set_default** shall create a new resource with a default value and add it to the resource database using arguments *name* and *scope* as the lookup parameters. UNCLEAR WHAT THE DEFAULT VALUE IS

10.2.4.4 set

```
static void set( const std::string& scope,
                 const std::string& name,
                 const T& val,
                 uvm_object* accessor = NULL );
```

The member function **set** shall create a new resource, write a value *val* to it, and add it to the resource database using arguments *name* and *scope* as the lookup parameters. The argument *accessor* is used for auditing

10.2.4.5 set_anonymous

```
static void set_anonymous( const std::string& scope,
                           const T& val,
                           uvm_object* accessor = NULL );
```

The member function **set_anonymous** shall create a new resource, write a value *val* to it, and add it to the resource database. As the resource has no argument *name*, it will not be entered into the name map. But is does have an argument *scope* for lookup purposes. The argument *accessor* is used for auditing.

10.2.4.6 read_by_name

```
static bool read_by_name( const std::string& scope,
                          const std::string& name,
                          T val,
                          uvm_object* accessor = NULL );
```

The member function **read_by_name** shall locate a resource by arguments *name* and *scope* and returns the value through argument *val*. The member function shall return true if the read was successful; otherwise it shall return false. The argument *accessor* is used for auditing.

10.2.4.7 read_by_type

```
static bool read_by_type( const std::string& scope,
                          T val,
                          uvm_object* accessor = NULL );
```

The member function **read_by_type** shall read a value by type. The value is returned through the argument *val*. The argument *scope* is used for the lookup. The member function shall return true if the read was successful; otherwise it shall return false. The argument *accessor* is used for auditing.

10.2.4.8 write_by_name

```
static bool write_by_name( const std::string& scope,
                           const std::string& name,
                           const T& val,
                           uvm_object* accessor = NULL );
```

The member function **write_by_name** shall write the argument *val* into the resources database. First, look up the resource by using arguments *name* and *scope*. If it is not located then add a new resource to the database and then write its value. WHAT DOES IT RETURN?

10.2.4.9 write_by_type

```
static bool write_by_type( const std::string& scope,
                           const T& val,
                           uvm_object* accessor = NULL );
```

The member function **write_by_type** shall write the argument *val* into the resources database. First, look up the resource by type. If it is not located then add a new resource to the database and then write its value.

Because the scope is matched to a resource which may be a regular expression, and consequently may target other scopes beyond the scope argument. Care must be taken with this function. If a **get_by_name** match is found for name and scope then val will be written to that matching resource and thus may impact other scopes which also match the resource. WHAT DOES IT RETURN?

10.3 uvm_resource_db_options

The class **uvm_resource_db_options** shall provide a namespace for managing options for the resources database facility. The class shall define static member functions for manipulating and retrieving the value of the data members. The static data members represent options and settings that control the behavior of the resources database facility.

10.3.1 Class definition

```
namespace uvm {


  class uvm_resource_db_options
  {
   public:
    static void turn_on_tracing();
    static void turn_off_tracing();
    static bool is_tracing();


   private:
    // Disabled
```

```
    uvm_resource_db_options();

  }; // class uvm_resource_db_options

} // namespace uvm
```

10.3.2 Member functions

10.3.2.1 turn_on_tracing

```
static void turn_on_tracing();
```

The member function **turn_on_tracing** shall enable tracing for the resource database. This causes all reads and writes to the database to display information about the accesses.

10.3.2.2 turn_off_tracing

```
static void turn_off_tracing();
```

The member function **turn_off_tracing** shall disable tracing for the resource database.

10.3.2.3 is_tracing

```
static bool is_tracing();
```

The member function **is_tracing** shall return true if the tracing facility is enabled; otherwise it shall return false.

10.4 uvm_resource_options

The class **uvm_resource_options** shall provide a namespace for managing options for the resources facility. The class shall only provide static member functions for manipulating and retrieving the value of its data members.

10.4.1 Class definition

```
namespace uvm {

  class uvm_resource_options
  {
   public:
    static void turn_on_auditing();
    static void turn_off_auditing();
    static bool is_auditing();

   private:
    // Disabled
```

```
    uvm_resource_options();

  }; // class uvm_resource_options

} // namespace uvm
```

10.4.2 Member functions

10.4.2.1 turn_on_auditing

```
static void turn_on_auditing();
```

The member function **turn_on_auditing** shall enable auditing for the resource database. This causes all reads and writes to the database to store information about the accesses. Auditing is enabled by default.

10.4.2.2 turn_off_auditing

```
static void turn_off_auditing();
```

The member function **turn_off_auditing** shall disable auditing for the resource database. If auditing is disabled, it is not possible to get extra information about resource database accesses.

10.4.2.3 is_auditing

```
static bool is_auditing();
```

The member function **is_auditing** shall return true if auditing is enabled; otherwise it shall return false.

10.5 uvm_resource_base

The class **uvm_resource_base** shall provide a non-parameterized base class for resources. It supports interfaces for scope matching and virtual member functions for printing the resource and accessors list.

10.5.1 Class definition

```
namespace uvm {

  class uvm_resource_base : public uvm_object
  {
   public:
    uvm_resource_base( const std::string& name = "",
                       const std::string& scope = "*" );
    ~uvm_resource_base();

    // Group: Resource database interface
```

```
    virtual uvm_resource_base* get_type_handle() const = 0;

    // Group: Read-only interface
    void set_read_only();
    bool is_read_only() const;

    // Group: Notification
    void wait_modified();

    // Group: Scope interface
    void set_scope( const std::string* scope );
    std::string get_scope() const;
    bool match_scope( const std::string& scope );

    // Group: Priority
    virtual void set_priority( uvm_resource_types::priority_e pri ) = 0;

    // Group: Utility functions
    void do_print( const uvm_printer& printer ) const;

    // Group: Audit trail
    void record_read_access( uvm_object* accessor = NULL );
    void record_write_access( uvm_object* accessor = NULL );
    virtual void print_accessors() const;
    void init_access_record( uvm_resource_types::access_t access_record );

    // Data members: Precedence
    unsigned int precedence;
    static int unsigned default_precedence;

  }; // class uvm_resource_base

} // namespace uvm
```

10.5.2 Constructor

```
uvm_resource_base( const std::string& name = "",
                   const std::string& scope = "*" );
```

The constructor takes two arguments, the name of the resource *name* and a regular expression *scope* which represents the set of scopes over which this resource is visible.

10.5.3 Resource database interface

10.5.3.1 get_type_handle

```
virtual uvm_resource_base* get_type_handle() const = 0;
```

The member function **get_type_handle** shall return the type handle of the resource container.

10.5.4 Read-only interface

10.5.4.1 set_read_only

```
void set_read_only();
```

The member function **set_read_only** shall define the resource as a read-only resource. An attempt to call **uvm_resource**<T>::**write** on the resource will cause an error.

10.5.4.2 is_read_only

```
bool is_read_only() const;
```

The member function **is_read_only** shall return true if this resource has been set to read-only; otherwise it shall return false.

10.5.5 Notification

10.5.5.1 wait_modified

```
void wait_modified();
```

The member function **wait_modified** shall block execution until the resource has been modified, that is, it waits till a **uvm_resource**<T>::**write** operation has been performed.

10.5.6 Scope interface

10.5.6.1 set_scope

```
void set_scope( const std::string& scope );
```

The member function **set_scope** shall set the value of the regular expression that identifies the set of scopes over which this resource is visible. If the supplied argument is a glob it will be converted to a regular expression before it is stored.

10.5.6.2 get_scope

```
std::string get_scope() const;
```

The member function **get_scope** shall retrieve the regular expression string that identifies the set of scopes over which this resource is visible.

10.5.6.3 match_scope

```
bool match_scope( const std::string& scope );
```

The member function **match_scope** shall return true if this resource is visible in a scope. The scope is specified as argument and may use regular expressions.

10.5.7 Priority

10.5.7.1 set_priority

```
virtual void set_priority( uvm_resource_types::priority_e pri ) = 0;
```

The member function **set_priority** shall change the search priority of the resource based on the value of the priority enumeration given as argument.

10.5.8 Utility functions

10.5.8.1 do_print

```
void do_print( const uvm_printer& printer ) const;
```

The member function **do_print** shall be called by member function **print**. It allows an application to implement application-specific printing routines.

10.5.9 Audit trail

10.5.9.1 record_read_access

```
void record_read_access( uvm_object* accessor = NULL );
```

The member function **record_read_access** shall record the read access for this resource.

10.5.9.2 record_write_access

```
void record_write_access( uvm_object* accessor = NULL );
```

The member function **record_write_access** shall record the write access for this resource.

10.5.9.3 print_accessors

```
virtual void print_accessors() const;
```

The member function **print_accessors** shall print the access records for this resource.

10.5.9.4 init_access_record

```
void init_access_record( uvm_resource_types::access_t access_record );
```

The member function **init_access_record** shall initialize a new access record.

10.6 uvm_resource_pool

The class **uvm_resource_pool** shall provide the centralized resource pool to store each resource both by primary name and by type handle.

10.6.1 Class definition

```
namespace uvm {


  class uvm_resource_pool
  {
   public:
    static uvm_resource_pool* get();
    bool spell_check( const std::string& s );

    // Group: Set interface
    void set( uvm_resource_base* rsrc, int override = 0 );
    void set_override( uvm_resource_base* rsrc );
    void set_name_override( uvm_resource_base* rsrc );
    void set_type_override( uvm_resource_base* rsrc );

    // Group: Lookup
    uvm_resource_types::rsrc_q_t* lookup_name( const std::string& scope,
                                               const std::string& name,
                                               uvm_resource_base* type_handle,
                                               bool rpterr = true );

    uvm_resource_base* get_highest_precedence( uvm_resource_types::rsrc_q_t* q );
MAKE ARGUMENT A REFERENCE?
    static void sort_by_precedence( uvm_resource_types::rsrc_q_t* q );
MAKE ARGUMENT A REFERENCE?
    uvm_resource_base* get_by_name( const std::string& scope,
                                    const std::string& name,
                                    uvm_resource_base* type_handle,
                                    bool rpterr = true );
```

```
    uvm_resource_types::rsrc_q_t* lookup_type( const std::string& scope,
                                               uvm_resource_base* type_handle );

    uvm_resource_base* get_by_type( const std::string& scope,
                                    uvm_resource_base* type_handle );

    uvm_resource_types::rsrc_q_t* lookup_regex_names( const std::string& scope,
                                                      const std::string& name,
                                                      uvm_resource_base* type_handle = NULL );

    uvm_resource_types::rsrc_q_t* lookup_regex( const std::string& re,
                                                const std::string& scope );

    uvm_resource_types::rsrc_q_t* lookup_scope( const std::string& scope );

    // Group: Set Priority
    void set_priority_type( uvm_resource_base* rsrc,
                            uvm_resource_types::priority_e pri );

    void set_priority_name( uvm_resource_base* rsrc,
                            uvm_resource_types::priority_e pri );

    void set_priority( uvm_resource_base* rsrc,
                       uvm_resource_types::priority_e pri );

    // Group: Debug
    uvm_resource_types::rsrc_q_t* find_unused_resources();
    void print_resources( uvm_resource_types::rsrc_q_t rq, bool audit = false );
    void dump( bool audit = false );

  }; // class uvm_resource_pool

} // namespace uvm
```

10.6.2 get

```
static uvm_resource_pool* get();
```

The member function **get** shall return the singleton handle to the resource pool.

10.6.3 spell_check

```
bool spell_check( const std::string& s );
```

The member function **spell_check** shall invoke the spell checker for the string *s* passed as argument. The universe of correctly spelled strings—i.e. the dictionary—is the name map. NOT IMPLEMENTED

10.6.4 Set interface

10.6.4.1 set

```
void set( uvm_resource_base* rsrc, int override = 0 );
```

The member function **set** shall add a new resource to the resource pool. The resource is inserted into both the name map and type map so it can be located by either.

An object creates a resource and sets it into the resource pool. Later, other objects that want to access the resource must get it from the pool.

Overrides can be specified using this interface. Either a name override, a type override or both can be specified. If an override is specified, then the resource is entered at the front of the queue instead of at the back.

It is not recommended that an application specify the override parameter directly. Instead, an application should use the member functions **set_override**, **set_name_override**, or **set_type_override**.

10.6.4.2 set_override

```
void set_override( uvm_resource_base* rsrc );
```

The member function **set_override** shall override the resource, provided as an argument, in the resource pool both by name and type.

10.6.4.3 set_name_override

```
void set_name_override( uvm_resource_base* rsrc );
```

The member function **set_name_override** shall override the resource, provided as argument *rsrc*, in the resource pool using normal precedence in the type map and will override the name.

10.6.4.4 set_type_override

```
void set_type_override( uvm_resource_base* rsrc );
```

The member function **set_type_override** shall override the resource, provided as argument *rsrc*, in the resource pool using normal precedence in the name map and will override the type.

10.6.5 Lookup

10.6.5.1 lookup_name

```
uvm_resource_types::rsrc_q_t* lookup_name( const std::string& scope,
                                           const std::string& name,
                                           uvm_resource_base* type_handle,
                                           bool rpterr = true );
```

The member function **lookup_name** shall return a queue of resources that match the *name*, *scope*, and *type_handle*, which are passed as arguments. If no resources match the queue is returned empty. If *rpterr* is set, then a warning is issued if no matches are found, and the spell checker is invoked on *name*. If *type_handle* is NULL, then a type check is not made and resources are returned that match only *name* and *scope*.

10.6.5.2 get_highest_precedence

```
uvm_resource_base* get_highest_precedence( uvm_resource_types::rsrc_q_t* q );
MAKE ARGUMENT A REFERENCE?, MAKE CONST
```

The member function **get_highest_precedence** shall traverse the queue passes as argument, *q*, of resources and return the one with the highest precedence. In the case where there exists more than one resource with the highest precedence value, the first one that has that precedence will be the one that is returned.

10.6.5.3 sort_by_precedence

```
static void sort_by_precedence( uvm_resource_types::rsrc_q_t* q );
MAKE ARGUMENT A REFERENCE?
```

The member function **sort_by_precedence** shall sort the resources, passed as argument as a list of resources, in precedence order. The highest precedence resource will be first in the list and the lowest precedence will be last. Resources that have the same precedence and the same name will be ordered by most recently set first.

10.6.5.4 get_by_name

```
uvm_resource_base* get_by_name( const std::string& scope,
                                const std::string& name,
                                uvm_resource_base* type_handle,
                                bool rpterr = true ); MAKE CONST
```

The member function **get_by_name** shall return the resource by using the arguments *name*, *scope*, and *type_handle*. Whether the get succeeds or fails, save a record of the get attempt. If the argument *rpterr* is true, the member function shall report potential errors.

10.6.5.5 lookup_type

```
uvm_resource_types::rsrc_q_t* lookup_type( const std::string& scope,
                                           uvm_resource_base* type_handle ); MAKE CONST
```

The member function **lookup_type** shall return a queue of resources that match the argument *type_handle* and argument *scope*. If no resources match, then the returned queue is empty.

10.6.5.6 get_by_type

```
uvm_resource_base* get_by_type( const std::string& scope,
                                uvm_resource_base* type_handle ); MAKE CONST
```

The member function **get_by_type** shall return the resources that match the argument *type_handle* and argument *scope*. It shall insert a record into the get history list whether or not the get succeeded.

10.6.5.7 lookup_regex_names

```
uvm_resource_types::rsrc_q_t* lookup_regex_names( const std::string& scope,
                                                  const std::string& name,
                                                  uvm_resource_base* type_handle = NULL ); MAKE
CONST
```

The member function **lookup_regex_names** shall return a queue of resources that match the arguments *name*, *scope*, and *type_handle*, where *name* and *scope* may be expressed as a regular expression.

10.6.5.8 lookup_regex

```
uvm_resource_types::rsrc_q_t* lookup_regex( const std::string& re,
                                            const std::string& scope ); MAKE CONST
```

The member function **lookup_regex** shall return a queue of resources that whose name matches the regular expression argument *re* and whose scope matches the specified argument *scope*.

10.6.5.9 lookup_scope

```
uvm_resource_types::rsrc_q_t* lookup_scope( const std::string& scope ); MAKE CONST
```

The member function **lookup_scope** shall return a queue of resources that are visible to a particular *scope*.

NOTE–This member function could be quite computation expensive, as it has to traverse all of the resources in the resource database.

10.6.6 Set priority

10.6.6.1 set_priority_type

```
void set_priority_type( uvm_resource_base* rsrc,
                        uvm_resource_types::priority_e pri );
```

The member function **set_priority_type** shall change the priority of the resource *rsrc* in the resource type map only, based on the value of priority enumeration argument *pri*. The priority in the resource name map remains unchanged.

10.6.6.2 set_priority_name

```
void set_priority_name( uvm_resource_base* rsrc,
                        uvm_resource_types::priority_e pri );
```

The member function **set_priority_name** shall change the priority of the resource *rsrc* in the resource name map only, based on the value of priority enumeration argument *pri*. The priority in the resource type map remains unchanged.

10.6.6.3 set_priority

```
void set_priority( uvm_resource_base* rsrc,
                   uvm_resource_types::priority_e pri );
```

The member function **set_priority** shall change the priority of the resource *rsrc* in the resource name map and type map, based on the value of priority enumeration argument *pri*.

10.6.7 Debug

10.6.7.1 find_unused_resources

```
uvm_resource_types::rsrc_q_t* find_unused_resources() const;
```

The member function **find_unused_resources** shall return a queue of resources that have at least one write and no reads.

NOT IMPLEMENTED YET

10.6.7.2 print_resources

```
void print_resources( uvm_resource_types::rsrc_q_t rq, bool audit = false ) const;
```

The member function **print_resources** shall print the queue of resources passed as argument *rq*. If the argument *audit* is true, the audit trail is printed for each resource along with the name, value, and scope regular expression.

10.6.7.3 dump

```
void dump( bool audit = false ) const;
```

The member function **dump** shall print the entire resource pool. The member function **print_resources** shall be used to initiate the printing. If the argument *audit* is true, the audit trail is printed for each resource along with the name, value, and scope regular expression.

10.7 uvm_resource

The class **uvm_resource** shall provide the interface to read and write to the resource database.

10.7.1 Class definition

```
namespace uvm {
```

```
  template <typename T = int>
  class uvm_resource : public uvm_resource_base
  {
   public:

    // Group: Type Interface
    static uvm_resource<T>* get_type();
    uvm_resource_base* get_type_handle() const;

    // Group: Set/Get Interface
    void set();
    void set_override( uvm_resource_types::override_e override =
                                                    uvm_resource_types::BOTH_OVERRIDE );

    static uvm_resource<T>* get_by_name( const std::string& scope,
                                         const std::string& name,
                                         bool rpterr = true ); MAKE CONST?

    static uvm_resource<T>* get_by_type( const std::string& scope,
                                         uvm_resource_base* type_handle ); MAKE CONST?

    // Group: Read/Write Interface
    T read( uvm_object*& accessor ); MAKE CONST?
    void write( const T& t, uvm_object*& accessor );

    // Group: Priority
    void set_priority( uvm_resource_types::priority_e pri );
    static uvm_resource<T>* get_highest_precedence( uvm_resource_types::rsrc_q_t* q );

  }; // class uvm_resource

} // namespace uvm
```

10.7.2 Template parameter T

The template parameter T specifies the object type of the objects being stored in or retrieved from the resource database.

10.7.3 Type interface

10.7.3.1 get_type

```
static uvm_resource<T>* get_type();
```

The member function **get_type** shall return the static type handle. The return type is the type of the parameterized class.

10.7.3.2 get_type_handle

```
uvm_resource_base* get_type_handle() const;
```

The member function **get_type_handle** shall return the static type handle of this resource in a polymorphic fashion. The return type of **get_type_handle** is **uvm_resource_base**.

NOTE–As the member function is not static, it can only be used by instances of a parameterized resource.

10.7.4 Set/Get interface

10.7.4.1 set

```
void set();
```

The member function **set** shall put the resource into the global resource pool.

10.7.4.2 set_override

```
void set_override( uvm_resource_types::override_e override =
                                                  uvm_resource_types::BOTH_OVERRIDE );
```

The member function **set_override** shall put the resource into the global resource pool as an override. This means it gets put at the head of the list and is searched before other existing resources that occupy the same position in the name map or the type map. The default is to override both the name and type maps. However, using the override argument you can specify that either the name map or type map is overridden.

10.7.4.3 get_by_name

```
static uvm_resource<T>* get_by_name( const std::string& scope,
                                     const std::string& name,
                                     bool rpterr = true );
```

The member function **get_by_name** shall look up a resource by name in the name map. The first resource with the specified name, whose type is the current type, and is visible in the specified scope is returned, if one exists. The rpterr flag indicates whether or not an error should be reported if the search fails. If rpterr is set to one then a failure message is issued, including suggested spelling alternatives, based on resource names that exist in the database, gathered by the spell checker.

10.7.4.4 get_by_type

```
static uvm_resource<T>* get_by_type( const std::string& scope,
                                     uvm_resource_base* type_handle );
```

The member function **get_by_type** shall look up a resource by *type_handle* in the type map. The first resource with the specified *type_handle* that is visible in the specified scope is returned, if one exists. The member function shall return NULL if there is no resource matching the specifications.

10.7.5 Read/Write interface

10.7.5.1 read

```
T read( uvm_object*& accessor );
```

The member function **read** shall return the object stored in the resource container. If an accessor object is supplied then also update the accessor record for this resource.

10.7.5.2 write

```
void write( const T& t, uvm_object*& accessor );
```

The member function **write** shall modify the object stored in this resource container. If the resource is read-only then issue an error message and return without modifying the object in the container. If the resource is not read-only and an accessor object has been supplied then also update the accessor record. Lastly, replace the object value in the container with the value supplied as the argument, t, and release any processes blocked on **uvm_resource_base**::**wait_modified**.

10.7.6 Priority

10.7.6.1 set_priority

```
void set_priority( uvm_resource_types::priority_e pri );
```

The member function **set_priority** shall change the search priority of the resource based on the value of the priority enum argument, pri.

10.7.6.2 get_highest_precedence

```
static uvm_resource<T>* get_highest_precedence( uvm_resource_types::rsrc_q_t* q );
```

The member function **get_highest_precedence** shall locate the first resource, in a queue of resources, with the highest precedence whose type is T.

10.8 uvm_resource_types

The class **uvm_resource_types** shall provide typedefs and enums used throughout the resources facility. This class shall not contain any member function or data members, only typedefs. It's used in lieu of package-scope types.

10.8.1 Class definition

```
namespace uvm {


  class uvm_resource_types
  {
   public:
    typedef sc_dt::sc_bv<2> override_t; DO WE STILL NEED THIS?


    typedef enum { TYPE_OVERRIDE,
                   NAME_OVERRIDE,
                   BOTH_OVERRIDE,
                 } override_e;


    typedef uvm_queue<uvm_resource_base* > rsrc_q_t;
    typedef enum { PRI_HIGH, PRI_LOW } priority_e;
  }; // class uvm_resource_types


} // namespace uvm
```

10.8.2 Type definitions (typedefs)

TODO

10.9 uvm_queue

The class **uvm_queue** shall provide a dynamic queue. Allows queues to be allocated on demand and passed and stored by reference.

10.9.1 Class definition

```
namespace uvm {


  template <typename T = int>
  class uvm_queue : public uvm_object
  {
   public:
    uvm_queue( const std::string& name = "" );
    static uvm_queue<T>* get_global_queue();
    static T get_global( int index );
    virtual T get( int index );
```

```
    virtual int size() const;
    virtual void insert( int index, const T& item ); NOT IMPLEMENTED YET
    virtual void do_delete°( int index = -1 );
    virtual T pop_front();
    virtual T pop_back();
    virtual void push_front( const T& item );
    virtual void push_back( const T& item );

  }; // class uvm_queue

} // namespace uvm
```

10.9.2 Template parameter T

The template parameter T specifies the object type of the objects being stored in or retrieved from the dynamic queue.

10.9.3 Member functions

10.9.3.1 Constructor

```
uvm_queue( const std::string& name = "" );
```

The constructor shall create a new queue with the given *name* as argument.

10.9.3.2 get_global_queue

```
static uvm_queue<T>* get_global_queue();
```

The member function **get_global_queue** shall return the singleton global queue for the item type, T. This allows items to be shared amongst components throughout the verification environment.

10.9.3.3 get_global

```
static T get_global( int index );
```

The member function **get_global** shall return the specified item instance of type T at the given *index* from the global item queue.

10.9.3.4 get

```
virtual T get( int index );
```

The member function **get** shall return the item of type T at the given *index*. If no item exists by that key, a new item is created with that key and returned. CHECK use of key?

10.9.3.5 size

```
virtual int size() const;
```

The member function **size** shall return the number of items stored in the queue.

10.9.3.6 insert

```
virtual void insert( int index, const T& item ); NOT IMPLEMENTED YET
```

The member function **insert** shall insert the *item* at the given *index* in the queue.

10.9.3.7 do_delete° (delete†)

```
virtual void do_delete°( int index = -1 );
```

The member function **do_delete°** shall remove the item at the given *index* from the queue; if *index* is not provided, the entire contents of the queue are deleted.

10.9.3.8 pop_front

```
virtual T pop_front();
```

The member function **pop_front** shall return the first element in the queue (index = 0), or NULL if the queue is empty. TODO: fix return type if queue is empty. Make c++ std::queue compatible?

10.9.3.9 pop_back

```
virtual T pop_back();
```

The member function **pop_back** shall returns the last element in the queue (index = size() - 1), or NULL if the queue is empty. TODO: fix return type if queue is empty. Make c++ std::queue compatible?

10.9.3.10 push_front

```
virtual void push_front( const T& item );
```

The member function **push_front** shall insert the given *item* at the front of the queue.

10.9.3.11 push_back

```
virtual void push_back( const T& item );
```

The member function **push_back** shall insert the given *item* at the back of the queue.

10.10 uvm_pool

The class **uvm_pool** shall provide a dynamic associative array. It shall allow sparse arrays to be allocated on demand, and passed and stored by reference.

10.10.1 Class definition

```
namespace uvm {


  template <typename KEY = int, typename T = uvm_void >
  class uvm_pool : public uvm_object
  {
   public:
    uvm_pool();
    explicit uvm_pool( const std::string& name );
    static uvm_pool<KEY,T>* get_global_pool();
    static T get_global( const KEY& key );
    virtual T get( const KEY& key );
    virtual void add( const KEY& key, const T& item );
    virtual int num() const;
    virtual void do_delete°( const KEY& key );
    virtual bool exists( const KEY& key ) const;
    virtual bool first( KEY& key );
    virtual bool last( KEY& key );
    virtual bool next( KEY& key );
    virtual bool prev( KEY& key );


  }; // class uvm_pool


} // namespace uvm
```

10.10.2 Template parameters

The template parameter KEY and T specifies the key and value type, respectively. The object of type T shall be derived from **uvm_void**. Check: Is the dependency to uvm_void necessary, or can it be any type?

10.10.3 Member functions

10.10.3.1 Constructors

```
uvm_pool();
explicit uvm_pool( const std::string& name );
```

The constructor shall create a new pool with the given *name* as argument.

10.10.3.2 get_global_pool

```
static uvm_pool<KEY,T>* get_global_pool();
```

The member function **get_global_pool** shall return the singleton global pool for the key of type KEY and item of type T. This allows items to be shared amongst components throughout the verification environment.

10.10.3.3 get_global

```
static T get_global( const KEY& key );
```

The member function **get_global** shall return the instance item of type T of the specified *key* of type KEY from the global item pool.

10.10.3.4 get

```
virtual T get( const KEY& key );
```

The member function **get** shall return the item of type T at the given *key*. If no item exists by that key, a new item is created with that key and returned.

10.10.3.5 add

```
virtual void add( const KEY& key, const T& item );
```

The member function **add** shall add the given (*key*, *item*) pair to the pool. If an item already exists at the given *key* it is overwritten with the new *item*.

10.10.3.6 num

```
virtual int num() const;
```

The member function **num** shall return the number of uniquely keyed items stored in the pool.

10.10.3.7 do_delete° (delete†)

```
virtual void do_delete°( const KEY& key );
```

The member function **do_delete**° shall remove the item with the given *key* from the pool.

10.10.3.8 exists

```
virtual bool exists( const KEY& key ) const;
```

The member function **exists** shall return true if an item with the given key exists in the pool; and false otherwise.

10.10.3.9 first

```
virtual bool first( KEY& key );
```

The member function **first** shall returns the key of the first item stored in the pool. If the pool is empty, then *key* is unchanged and false is returned. If the pool is not empty, then *key* is key of the first item and true is returned.

10.10.3.10 last

```
virtual bool last( KEY& key );
```

The member function **last** shall return the key of the last item stored in the pool. If the pool is empty, then false is returned and *key* is unchanged. If the pool is not empty, then *key* is set to the last key in the pool and true is returned.

10.10.3.11 next

```
virtual bool next( KEY& key );
```

The member function **next** shall return the key of the next item in the pool. If the input *key* is the last key in the pool, then *key* is left unchanged and false is returned. If a next key is found, then *key* is updated with that key and true is returned.

10.10.3.12 prev

```
virtual bool prev( KEY& key );
```

The member function **prev** shall return the key of the previous item in the pool. If the input *key* is the first key in the pool, then *key* is left unchanged and false is returned. If the previous key is found, then *key* is updated with that key and true is returned.

11. Phasing and synchronization classes

The phasing and synchronization concept in UVM defines standardized stages called *phases* which are executed in a well defined order. Each UVM component offers dedicated callbacks for each of these phases to implement application-specific behavior. Phases are executed sequentially, but each phase may consist of multiple function calls (of components contributing to that phase) in parallel. Besides standardized common and UVM run-time phases, user-defined phases can be added.

In order to support synchronization during the execution of the run-time phases, which run as concurrent processes, additional methods are available to coordinate the execution of or status of these processes between all UVM components or objects.

The following phasing and synchronization classes are defined:

- **uvm_phase**: The base class for defining a phase's behavior, state, context.
- **uvm_domain**: Phasing schedule node representing an independent branch of the schedule.
- **uvm_bottomup_phase**: A phase implementation for bottom up function phases.
- **uvm_topdown_phase**: A phase implementation for top-down function phases.
- **uvm_process_phase**°: A phase implementation for phases which are launched as spawned processes. In UVM-SystemVerilog, this class was called **uvm_task_phase**†.
- **uvm_objection**: Mechanism to synchronize phases based on passing execution status information between running processes.

11.1 uvm_phase

The class **uvm_phase** shall provide the base class for the UVM phasing mechanism.

11.1.1 Class definition

```
namespace uvm {


  class uvm_phase : public uvm_object
  {
   public:


    // Group: Construction
    explicit uvm_phase( const std::string& name,
                        uvm_phase_type phase_type = UVM_PHASE_SCHEDULE,
                        uvm_phase* parent = NULL );


    uvm_phase_type get_phase_type() const;


    // Group: State
```

```
uvm_phase_state get_state() const;

int get_run_count() const;
uvm_phase* find_by_name( const std::string& name, bool stay_in_scope = true ) const;
uvm_phase* find( const uvm_phase* phase, bool stay_in_scope = true ) const;
bool is( const uvm_phase* phase ) const;
bool is_before( const uvm_phase* phase ) const;
bool is_after( const uvm_phase* phase ) const;

// Group: Callbacks
virtual void exec_func( uvm_component* comp, uvm_phase* phase );
virtual void exec_process°( uvm_component* comp, uvm_phase* phase );

// Group: Schedule
void add( uvm_phase* phase,
          uvm_phase* with_phase = NULL,
          uvm_phase* after_phase = NULL,
          uvm_phase* before_phase = NULL);

uvm_phase* get_parent() const;
virtual const std::string get_full_name() const;
uvm_phase* get_schedule( bool hier = false ) const;
std::string get_schedule_name( bool hier = false ) const;
uvm_domain* get_domain() const;
std::string get_domain_name() const;
uvm_phase* get_imp() const;

// Group: Synchronization
uvm_objection* get_objection() const;
virtual void raise_objection( uvm_object* obj,
                              const std::string& description = "",
                              int count = 1 );

virtual void drop_objection( uvm_object* obj,
                             const std::string& description = "",
                             int count = 1 );
void sync( uvm_domain* target,
```

```
                uvm_phase* phase = NULL,
                uvm_phase* with_phase = NULL );

    void unsync( uvm_domain* target,
                 uvm_phase* phase = NULL,
                 uvm_phase* with_phase = NULL );

    void wait_for_state( uvm_phase_state state, uvm_wait_op op = UVM_EQ );

    // Group: Jumping
    void jump( const uvm_phase* phase );
    uvm_phase* get_jump_target() const;
  }; // class uvm_phase

} // namespace uvm
```

11.1.2 Construction

11.1.2.1 Constructor

```
explicit uvm_phase( const std::string& name,
                    uvm_phase_type phase_type = UVM_PHASE_SCHEDULE,
                    uvm_phase* parent = NULL );
```

The constructor shall create a new phase node, using the arguments *name*, the type name of type *type_name* and optionally the pointer to the parent phase *parent*, as argument.

11.1.2.2 get_phase_type

```
uvm_phase_type get_phase_type() const;
```

The member function **get_phase_type** shall return the phase type as defined by **uvm_phase_type** (see 15.3.6).

11.1.3 State

11.1.3.1 get_state

```
uvm_phase_state get_state() const;
```

The member function **get_state** shall return the current state of this phase.

11.1.3.2 get_run_count

```
int get_run_count() const;
```

The member function **get_run_count** shall return the integer number of times this phase has executed.

11.1.3.3 find_by_name

```
uvm_phase* find_by_name( const std::string& name, bool stay_in_scope = true );
```

The member function **find_by_name** shall locate a phase node with the specified *name* and return its handle. If argument *stay_in_scope* is true, it searches only within this phase's schedule or domain.

11.1.3.4 find

```
uvm_phase* find( const uvm_phase* phase, bool stay_in_scope = true );
```

The member function **find** shall locate the phase node with the specified phase implementation and return its handle. If argument *stay_in_scope* is true, it searches only within this phase's schedule or domain.

11.1.3.5 is

```
bool is( const uvm_phase* phase ) const;
```

The member function **is** shall return true if the containing **uvm_phase** refers to the same phase as the phase argument; otherwise it shall return false.

11.1.3.6 is_before

```
bool is_before( const uvm_phase* phase ) const;
```

The member function **is_before** shall return true if the containing **uvm_phase** refers to a phase that is earlier than the phase argument; otherwise it shall return false.

11.1.3.7 is_after

```
bool is_after( const uvm_phase* phase ) const;
```

The member function **is_after** shall return true if the containing **uvm_phase** refers to a phase that is later than the phase argument; otherwise it shall return false.

11.1.4 Callbacks

11.1.4.1 exec_func

```
virtual void exec_func( uvm_component* comp, uvm_phase* phase );
```

The member function **exec_func** shall implement the functor/delegate functionality for a function phase type comp - the component to execute the functionality upon phase - the phase schedule that originated this phase call.

11.1.4.2 exec_process° (exec_task†)

```
virtual void exec_process( uvm_component* comp, uvm_phase* phase );
```

The member function **exec_process**° shall implement the functor/delegate functionality for a task phase type comp - the component to execute the functionality upon phase - the phase schedule that originated this phase call.

NOTE–The member function was called **exec_task** in UVM in SystemVerilog, but has been renamed in line with SystemC processes.

11.1.5 Schedule

11.1.5.1 add

```
void add( uvm_phase* phase,
          uvm_phase* with_phase = NULL,
          uvm_phase* after_phase = NULL,
          uvm_phase* before_phase = NULL);
```

The member function **add** shall build a schedule structure, inserting phase by phase, specifying linkage. Phases can be added anywhere, in series or parallel with existing nodes. The argument *phase* is the handle of a singleton derived phase implementation containing actual functor. By default the new phase shall be appended to the schedule. When argument *with_phase* is passed, the new phase shall be added in parallel to the actual phase. When argument *after_phase* is passed, the new phase shall be added as successor to the actual phase. When the argument *before_phase* is passed, the new phase shall be added as predecessor to the actual phase.

11.1.5.2 get_parent

```
uvm_phase* get_parent() const;
```

The member function **get_parent** shall return the parent schedule node, if any, for hierarchical graph traversal.

11.1.5.3 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the full path from the enclosing domain down to this node. The singleton phase implementations have no hierarchy.

11.1.5.4 get_schedule

```
uvm_phase* get_schedule( bool hier = false ) const;
```

The member function **get_schedule** shall return the topmost parent schedule node, if any, for hierarchical graph traversal.

11.1.5.5 get_schedule_name

```
std::string get_schedule_name( bool hier = false ) const;
```

The member function **get_schedule_name** shall return the schedule name associated with this phase node.

11.1.5.6 get_domain

```
uvm_domain* get_domain() const;
```

The member function **get_domain** shall return the enclosing domain.

11.1.5.7 get_domain_name

```
std::string get_domain_name() const;
```

The member function **get_domain_name** shall returns the domain name associated with this phase node.

11.1.5.8 get_imp

```
uvm_phase* get_imp() const;
```

The member function **get_imp** shall return the phase implementation for this node. It shall return NULL if this phase type is not a UVM_PHASE_LEAF_NODE.

11.1.6 Synchronization

11.1.6.1 get_objection

```
uvm_objection* get_objection() const;
```

The member function **get_objection** shall return the object of class **uvm_objection** that gates the termination of the phase.

11.1.6.2 raise_objection

```
virtual void raise_objection( uvm_object* obj,
                              const std::string& description = "",
                              int count = 1 );
```

The member function **raise_objection** shall return the object of class **uvm_objection** that gates the termination of the phase.

11.1.6.3 drop_objection

```
virtual void drop_objection( uvm_object* obj,
                             const std::string& description = "",
                             int count = 1 );
```

The member function **drop_objection** shall drop an objection to ending a phase. The drop is expected to be matched with an earlier raise.

11.1.6.4 sync

```
void sync( uvm_domain* target,
```

```
            uvm_phase* phase = NULL,
            uvm_phase* with_phase = NULL );
```

The member function **sync** shall synchronize two domains, fully or partially. The argument *target* is a handle of the target domain to synchronize this one to. The optional argument *phase* is the phase in this domain to synchronize with; otherwise synchronize to all. The optional argument *with_phase* is the target-domain phase to synchronize with; otherwise use *phase* in the target domain.

11.1.6.5 unsync

```
void unsync( uvm_domain* target,
             uvm_phase* phase = NULL,
             uvm_phase* with_phase = NULL );
```

The member function **unsync** shall remove the synchronization between two domains, fully or partially. The argument *target* is a handle of the target domain to remove synchronize from. The optional argument *phase* is the phase in this domain to un-synchronize with; otherwise un-synchronize to all. The optional argument *with_phase* is the target-domain phase to un-synchronize with; otherwise use *phase* in the target domain.

11.1.6.6 wait_for_state

```
void wait_for_state( uvm_phase_state state, uvm_wait_op op = UVM_EQ );
```

The member function **wait_for_state** shall wait until this phase compares with the given state and op operand. For **UVM_EQ** and **UVM_NE** operands, several **uvm_phase_states** can be supplied by their enum constants, in which case the caller will wait until the phase state is any of **UVM_EQ** or none of **UVM_NE** the provided states.

11.1.7 Jumping

11.1.7.1 jump

```
void jump( const uvm_phase* phase );
```

The member function **jump** shall jump to a specified phase. If the destination phase is within the current phase schedule, a simple local jump takes place. If the jump-to phase is outside of the current schedule then the jump affects other schedules which share the phase.

11.1.7.2 get_jump_target

```
uvm_phase* get_jump_target() const;
```

The member function **get_jump_target** shall return the handle to the target phase of the current jump, or NULL if no jump is in progress. This member function shall only be used during the **phase_ended** callback.

11.2 uvm_domain

The class **uvm_domain** shall provide a phasing schedule node representing an independent branch of the schedule.

11.2.1 Class definition

```
namespace uvm {


  class uvm_domain : public uvm_phase
  {
   public:
    explicit uvm_domain( const std::string& name );
    static std::map< std::string, uvm_domain* > get_domains();
    static uvm_phase* get_uvm_schedule();
    static uvm_domain* get_common_domain();
    static void add_uvm_phases( uvm_phase* schedule );
    static uvm_domain* get_uvm_domain();


  }; // class uvm_domain


} // namespace uvm
```

11.2.2 Constructor

```
explicit uvm_domain( const std::string& name );
```

The constructor shall create a new instance of a phase domain with the *name* passed as argument.

11.2.3 Member functions

11.2.3.1 get_domains

```
static std::map< std::string, uvm_domain* > get_domains();
```

The member function **get_domains** shall provide a list of all domains in the provided domains argument.

11.2.3.2 get_uvm_schedule

```
static uvm_phase* get_uvm_schedule();
```

The member function **get_uvm_schedule** shall return the “UVM” schedule, which consists of the run-time phases that all components execute when participating in the “UVM” domain.

11.2.3.3 get_common_domain

```
static uvm_domain* get_common_domain();
```

The member function **get_common_domain** shall return the “common” domain, which consists of the common phases that all components execute in sync with each other. Phases in the “common” domain are build, connect, end_of_elaboration, start_of_simulation, run, extract, check, report, and final.

11.2.3.4 get_uvm_phases

```
static void add_uvm_phases( uvm_phase* schedule );
```

The member function **add_uvm_phases** shall append to the given schedule the built-in UVM phases.

11.2.3.5 get_uvm_domain

```
static uvm_domain* get_uvm_domain();
```

The member function **get_uvm_domain** shall return the handle to the singleton *uvm* domain.

11.3 uvm_bottomup_phase

The class **uvm_bottomup_phase** shall provide the base class for function phases that operate bottom-up. The member function **execute** is called for each component. This is the default traversal so is included only for naming. The bottom-up phase completes when the member function **execute** has been called and returned on all applicable components in the hierarchy.

11.3.1 Class definition

```
namespace uvm {


  class uvm_bottomup_phase : public uvm_phase
  {
   public:
    explicit uvm_bottomup_phase( const std::string& name );


    virtual void traverse( uvm_component* comp,
                           uvm_phase* phase,
                           uvm_phase_state state );


    virtual void execute( uvm_component* comp,
                          uvm_phase* phase );


  }; // class uvm_bottomup_phase


} // namespace uvm
```

11.3.2 Constructor

```
explicit uvm_bottomup_phase( const std::string& name );
```

The constructor shall create a new instance of a bottom-up phase using the *name* passed as argument.

11.3.3 Member functions

11.3.3.1 traverse

```
virtual void traverse( uvm_component* comp,
                       uvm_phase* phase,
                       uvm_phase_state state );
```

The member function **traverse** shall traverse the component tree in bottom-up order, calling member function **execute** for each component.

11.3.3.2 execute

```
virtual void execute( uvm_component* comp,
                      uvm_phase* phase );
```

The member function **execute** shall execute the bottom-up phase *phase* for the component *comp*.

11.4 uvm_topdown_phase

The class **uvm_topdown_phase** shall provide the base class for function phases that operate top-down. The member function **execute** is called for each component. This is the default traversal so is included only for naming. The top-down phase completes when the member function **execute** has been called and returned on all applicable components in the hierarchy.

11.4.1 Class definition

```
namespace uvm {

  class uvm_topdown_phase : public uvm_phase
  {
   public:
    explicit uvm_topdown_phase( const std::string& name );

    virtual void traverse( uvm_component* comp,
                           uvm_phase* phase,
                           uvm_phase_state state );

    virtual void execute( uvm_component* comp,
                          uvm_phase* phase );

  }; // class uvm_topdown_phase
```

```
} // namespace uvm
```

11.4.2 Constructor

```
explicit uvm_topdown_phase( const std::string& name );
```

The constructor shall create a new instance of a top-down phase using the name *name* passed as argument.

11.4.3 Member functions

11.4.3.1 traverse

```
virtual void traverse( uvm_component* comp,
                       uvm_phase* phase,
                       uvm_phase_state state );
```

The member function **traverse** shall traverse the component tree in top-down order, calling member function **execute** for each component.

11.4.3.2 execute

```
virtual void execute( uvm_component* comp,
                      uvm_phase* phase );
```

The member function **execute** shall execute the top-down phase *phase* for the component *comp*.

11.5 uvm_process_phase° (uvm_task_phase†)

The class **uvm_process_phase°** shall provide the base class for all process-oriented phases. It is responsible to create spawned processes as part of the execution of the callback **uvm_phase::exec_process** for each component in the hierarchy. The completion of the execution of this callback does not imply, nor is it required for, the end of phase. Once the phase completes, any remaining spawned processes caused by executing **uvm_phase::exec_process** are forcibly and immediately killed. By default, the way for a process phase to extend over time is if there is at least one component that raises an objection.

11.5.1 Class definition

```
namespace uvm {


  class uvm_process_phase° : public uvm_phase
  {
   public:
    explicit uvm_process_phase°( const std::string& name );
```

```
    virtual void traverse( uvm_component* comp,
                           uvm_phase* phase,
                           uvm_phase_state state );

    virtual void execute( uvm_component* comp,
                          uvm_phase* phase );

  }; // class uvm_process_phase

} // namespace uvm
```

11.5.2 Member functions

11.5.2.1 traverse

```
virtual void traverse( uvm_component* comp,
                       uvm_phase* phase,
                       uvm_phase_state state );
```

The member function **traverse** shall traverse the component tree in bottom-up order, calling member function **execute** for each component.

NOTE–The actual order for process-based phases does not really matter, as each component process is executed in a separate process whose starting order is not deterministic.

11.5.2.2 execute

```
virtual void execute( uvm_component* comp,
                      uvm_phase* phase );
```

The member function **execute** shall spawn a process of phase *phase* for the component *comp*.

11.6 uvm_objection

The class **uvm_objection** shall provide a facility for coordinating status information between two or more participating components, objects, and even module-based IP.

11.6.1 Class definition

```
namespace uvm {

  class uvm_objection : public uvm_object
  {
   public:
```

```
// Constructors
uvm_objection();
uvm_objection( const std::string& name );

// Group: Objection control
virtual void clear( uvm_object* obj = NULL );
bool trace_mode( int mode = -1 );

virtual void raise_objection( uvm_object* obj,
                              const std::string& description = "",
                              int count = 1 );

virtual void drop_objection( uvm_object* obj,
                             const std::string& description = "",
                             int count = 1 );

void set_drain_time( uvm_object* obj = NULL,
                     const sc_core::sc_time& drain = sc_core::SC_ZERO_TIME );

// Group: Callback hooks
virtual void raised( uvm_object* obj,
                     uvm_object* source_obj,
                     const std::string& description,
                     int count );

virtual void dropped( uvm_object* obj,
                      uvm_object* source_obj,
                      const std::string& description,
                      int count );

virtual void all_dropped( uvm_object* obj,
                          uvm_object* source_obj,
                          const std::string& description,
                          int count );

// Group: Objection status
void get_objectors( std::vector<uvm_object*>& objlist ) const;
```

```
    void wait_for( uvm_objection_event objt_event,
                   uvm_object* obj = NULL );

    int get_objection_count( uvm_object* obj = NULL ) const;
    int get_objection_total( uvm_object* obj = NULL ) const;

    const sc_core::sc_time get_drain_time( uvm_object* obj = NULL ) const;

    void display_objections( uvm_object* obj = NULL,
                             bool show_header = true ) const;

  }; // class uvm_objection

} // namespace uvm
```

11.6.2 Constructor

```
uvm_objection();
uvm_objection( const std::string& name );
```

The constructor shall create a new objection instance with name *name*, if specified.

11.6.3 Objection control

11.6.3.1 clear

```
virtual void clear( uvm_object* obj = NULL );
```

The member function **clear** shall clear the objection state immediately. All counts are cleared and any processes that called **wait_for**(**UVM_ALL_DROPPED**, **uvm_top**) are released An application should pass ‘this’ to the *obj* argument for record keeping. Any configured drain times are not affected.

11.6.3.2 trace_mode

```
bool trace_mode( int mode = -1 );
```

The member function **trace_mode** shall set or get the trace mode for the objection object. If no argument is specified (or an argument other than 0 or 1) the current trace mode is unaffected. A trace_mode of 0 turns tracing off. A trace mode of 1 turns tracing on. The return value is the mode prior to being reset.

11.6.3.3 raise_objection

```
virtual void raise_objection( uvm_object* obj,
                              const std::string& description = "",
```

```
                                int count = 1 );
```

The member function **raise_objection** shall increase the number of objections for the source object by *count*, which defaults to 1. The object is usually the current (‘this’) handle of the caller. If an object is not specified or NULL, the implicit top-level component, **uvm_root**, is chosen.

Raising an objection shall cause the following.

- The source and total objection counts for object are increased by *count*.
- The member function **raised** is called, which calls the member function **uvm_component**::**raised** for all of the components up the hierarchy.

The description is a string that marks a specific objection and is used in tracing or debug.

11.6.3.4 drop_objection

```
virtual void drop_objection( uvm_object* obj,
                             const std::string& description = "",
                             int count = 1 );
```

The member function **drop_objection** shall decrease the number of objections for the source object by *count*, which defaults to 1. The object is usually the current handle (‘this’) of the caller. If object is not specified or NULL, the implicit top-level component, **uvm_root**, is chosen.

Dropping an objection shall cause the following:

- The source and total objection counts for object are decreased by *count*. It shall be an error to drop the objection count for object below zero.
- The member function **dropped** is called, which calls the member function **uvm_component**::**dropped** for all of the components up the hierarchy.

If the total objection count has not reached zero for the object, then the drop is propagated up the object hierarchy as with **raise_objection**. Then, each object in the hierarchy will have updated their source counts--objections that they originated--and total counts--the total number of objections by them and all their descendants.

If the total objection count reaches zero, propagation up the hierarchy is deferred until a configurable drain-time has passed and the **uvm_component**::**all_dropped** callback for the current hierarchy level has returned.

For each instance up the hierarchy from the source caller, a process is forked in a non-blocking fashion, allowing the **drop** call to return. The forked process then does the following:

- If a drain time was set for the given object, the process waits for that amount of time.
- The objection’s virtual member function **all_dropped** is called, which calls the **uvm_component**::**all_dropped** method (if object is a component).
- The process then waits for the **all_dropped** callback to complete.
- After the drain time has elapsed and the **all_dropped** callback has completed, propagation of the dropped objection to the parent proceeds as described in **raise_objection**, except as described below.

If a new objection for this object or any of its descendents is raised during the drain time or during execution of the **all_dropped** callback at any point, the hierarchical chain described above is terminated and the **dropped** callback

does not go up the hierarchy. The raised objection will propagate up the hierarchy, but the number of raised propagated up is reduced by the number of drops that were pending waiting for the **all_dropped**/drain time completion. Thus, if exactly one objection caused the count to go to zero, and during the drain exactly one new objection comes in, no raises or drops are propagated up the hierarchy,

As an optimization, if the object has no drain-time set and no registered callbacks, the forked process can be skipped and propagation proceeds immediately to the parent as described.

11.6.3.5 set_drain_time

```
void set_drain_time( uvm_object* obj = NULL,
                     const sc_core::sc_time& drain = sc_core::SC_ZERO_TIME );
```

The member function **set_drain_time** shall set the drain time on the given object to drain. The drain time is the amount of time to wait once all objections have been dropped before calling the **all_dropped** callback and propagating the objection to the parent. If a new objection for this object or any of its descendents is raised during the drain time or during execution of the **all_dropped** callbacks, the drain_time/all_dropped execution is terminated.

11.6.4 Callback hooks

11.6.4.1 raised

```
virtual void raised( uvm_object* obj,
                     uvm_object* source_obj,
                     const std::string& description,
                     int count );
```

The member function **raised** shall be called when a **raise_objection** has reached *obj*. The default implementation shall call **uvm_component**::**raised** (see 7.1.6.1).

11.6.4.2 dropped

```
virtual void dropped( uvm_object* obj,
                      uvm_object* source_obj,
                      const std::string& description,
                      int count );
```

The member function **dropped** shall be called when a **drop_objection** has reached *obj*. The default implementation shall call **uvm_component**::**dropped** (see 7.1.6.2).

11.6.4.3 all_dropped

```
virtual void all_dropped( uvm_object* obj,
                          uvm_object* source_obj,
                          const std::string& description,
                          int count );
```

The member function **all_dropped** shall be called when a **drop_objection** has reached *obj*, and the total count for *obj* goes to zero. This callback is executed after the drain time associated with *obj*. The default implementation shall call **uvm_component**::**all_dropped** (see 7.1.6.3).

11.6.5 Objections status

11.6.5.1 get_objectors

```
void get_objectors( std::vector<uvm_object*>& objlist ) const;
```

The member function **get_objectors** shall return the current list of objecting objects (objects that raised an objection but have not dropped it).

11.6.5.2 wait_for

```
void wait_for( uvm_objection_event objt_event,
               uvm_object* obj = NULL );
```

The member function **wait_for** shall wait for the raised, dropped, or all_dropped event to occur in the given object *obj*. The member function returns after all corresponding callbacks for that event have been executed.

11.6.5.3 get_objection_count

```
int get_objection_count( uvm_object* obj = NULL ) const;
```

The member function **get_objection_count** shall return the current number of objections raised by the given object *obj*.

11.6.5.4 get_objection_total

```
int get_objection_total( uvm_object* obj = NULL ) const;
```

The member function **get_objection_total** shall return the current number of objections raised by the given object *obj* and all descendants.

11.6.5.5 get_drain_time

```
const sc_core::sc_time get_drain_time( uvm_object* obj = NULL ) const;
```

The member function **get_drain_time** shall return the current drain time set for the given object *obj*. The default drain time shall be set to **sc_core**::**SC_ZERO_TIME**.

11.6.5.6 display_objections

```
void display_objections( uvm_object* obj = NULL,
                         bool show_header = true ) const;
```

The member function **display_objections** shall display objection information about the given object *obj*. If object is not specified or NULL, the implicit top-level component, **uvm_root**, is chosen. The argument *show_header* allows control of whether a header is output.

11.7 uvm_callback

The class **uvm_callback** shall provide the base class for user-defined callback classes. Typically, the component developer defines an application-specific callback class that extends from this class. In it, he defines one or more virtual member functions, called a callback interface, that represent the hooks available for user override.

The member functions intended for optional override should not be declared pure virtual. Usually, all the callback member functions are defined with empty implementations so users have the option of overriding any or all of them. The prototypes for each hook member function are completely application specific with no restrictions.

11.7.1 Class definition

```
namespace uvm {


  class uvm_callback : public uvm_object
  {
   public:
    uvm_callback( const std::string& name = "uvm_callback" );
    bool callback_mode( int on = -1 );
    bool is_enabled();
    virtual const std::string get_type_name() const;


  }; // class uvm_callback


} // namespace uvm
```

11.7.1.1 Constructor

```
uvm_callback( const std::string& name = "uvm_callback" );
```

The constructor shall create a new object of type **uvm_callback**, giving it an optional name.

11.7.2 Member functions

11.7.2.1 callback_mode

```
bool callback_mode( int on = -1 );
```

The member function **callback_mode** shall enable or disable callbacks. If argument *on* is set 1, callbacks are enabled. If argument *on* is set 0, callbacks are disabled.

11.7.2.2 is_enabled

```
bool is_enabled();
```

The member function **is_enabled** shall return 1 if the callback is enabled, otherwise it shall return 0.

11.7.2.3 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of this callback object.

11.8 uvm_callback_iter

The class **uvm_callback_iter** is an iterator class for iterating over callback queues of a specific callback type.

11.8.1 Class definition

```
namespace uvm {


  template < typename T = uvm_object, typename CB = uvm_callback>
  class uvm_callback_iter
  {
   public:
    uvm_callback_iter( T* obj );
    CB* first();
    CB* last();
    CB* next();
    CB* prev();
    CB* get_cb();


  }; // class uvm_callback


} // namespace uvm
```

11.8.2 Template parameter T

The template parameter T specifies the base object type with which the callback objects CB will be registered. This object must be a derivative of class **uvm_object**.

11.8.3 Template parameter CB

The template parameter T specifies the base callback type that will be managed by this callback class. The template parameter CB is optional. If not specified, the parameter is assigned the type **uvm_callback**.

11.8.4 Constructor

```
uvm_callback_iter( T* obj );
```

The constructor shall create a new callback iterator object. It is required that the object context be provided.

11.8.5 Member functions

11.8.5.1 first

```
CB* first();
```

The member function **first** shall return the first valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If the queue is empty, then NULL is returned.

11.8.5.2 last

```
CB* last();
```

The member function **last** shall return the last valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If the queue is empty, then NULL is returned.

11.8.5.3 next

```
CB* next();
```

The member function **next** shall return the next valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If there are no more valid callbacks in the queue, then NULL is returned.

11.8.5.4 prev

```
CB* prev();
```

The member function **prev** shall return the previous valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If there are no more valid callbacks in the queue, then NULL is returned.

11.8.5.5 get_cb

```
CB* get_cb();
```

The member function **get_cb** shall return the last callback accessed via the call **first** or **next**.

11.9 uvm_callbacks

The class **uvm_callbacks** shall provide a base class for implementing callbacks, which are typically used to modify or augment component behavior without changing the component class. To work effectively, the developer of the component class defines a set of "hook" methods that enable users to customize certain behaviors of the component in a manner that is controlled by the component developer. The integrity of the component's overall behavior is intact, while still allowing certain customizable actions by the user.

To enable compile-time type-safety, the class is parameterized on both the user-defined callback interface implementation as well as the object type associated with the callback. The object type-callback type pair are associated together using the macro **UVM_REGISTER_CB** to define a valid pairing; valid pairings are checked when a user attempts to add a callback to an object (see 13.4.2).

To provide the most flexibility for end-user customization and reuse, it is recommended that the component developer also define a corresponding set of virtual method hooks in the component itself. This affords users the

ability to customize via inheritance/factory overrides as well as callback object registration. The implementation of each virtual method would provide the default traversal algorithm for the particular callback being called. Being virtual, an application can define subtypes that override the default algorithm, perform tasks before and/or after calling the base class to execute any registered callbacks, or to not call the base implementation, effectively disabling that particular hook.

11.9.1 Class definition

```
namespace uvm {


  template <typename T = uvm_object, typename CB = uvm_callback>
  class uvm_callbacks : public uvm_typed_callbacks<T>
  {
   public:
    uvm_callbacks(); make private?


    // Group: Add/delete inteface
    static void add( T* obj, uvm_callback* cb, uvm_apprepend ordering = UVM_APPEND );


    static void add_by_name( const std::string& name,
                             uvm_callback* cb,
                             uvm_component* root,
                             uvm_apprepend ordering = UVM_APPEND );


    static void do_delete°( T* obj, uvm_callback* cb );


    static void delete_by_name( const std::string& name,
                                uvm_callback* cb,
                                uvm_component* root );


    // Group: Iterator Interface
    static CB* get_first( int& itr, T* obj );
    static CB* get_last( int& itr, T* obj );
    static CB* get_next( int& itr, T* obj );
    static CB* get_prev( int& itr, T* obj );


    // Group: Debug
    static void display( T* obj = NULL );
```

```
  }; // class uvm_callbacks


} // namespace uvm
```

11.9.2 Template parameter T

The template parameter T specifies the base object type with which the callback objects CB will be registered. This object must be a derivative of class **uvm_object**.

11.9.3 Template parameter CB

The template parameter CB specifies the base callback type that will be managed by this callback class.The template parameter CB is optional. If not specified, the parameter is assigned the type **uvm_callback**.

11.9.4 Constructor

```
uvm_callbacks(); make private?
```

The constructor shall create a new object of type **uvm_callbacks**<T, CB>.

11.9.5 Add/delete interface

11.9.5.1 add

```
static void add( T* obj, uvm_callback* cb, uvm_apprepend ordering = UVM_APPEND );
```

The member function **add** shall register the given callback object, *cb*, with the given handle *obj*. The object handle can be NULL, which allows registration of callbacks without an object context. If ordering is **UVM_APPEND** (default), the callback will be executed after previously added callbacks, else the callback will be executed ahead of previously added callbacks. The argument *cb* is the callback handle; it must be non-NULL, and if the callback has already been added to the object instance then a warning shall be issued.

11.9.5.2 add_by_name

```
static void add_by_name( const std::string& name,
                         uvm_callback* cb,
                         uvm_component* root,
                         uvm_apprepend ordering = UVM_APPEND );
```

The member function **add_by_name** shall register the given callback object, ***cb***, with one or more components of type **uvm_component**. The components must already exist and must be type T or a derivative. As with add the CB parameter is optional. Argument *root* specifies the location in the component hierarchy to start the search for *name*. See **uvm_root**::**find_all** (see 4.3.3.2) for more details on searching by name.

11.9.5.3 do_delete° (delete†)

```
    static void do_delete°( T* obj, uvm_callback* cb );
```

The member function **do_delete**° shall delete the given callback object, *cb*, from the queue associated with the given object handle *obj*. The object handle can be NULL, which allows de-registration of callbacks without an object context. The argument *cb* is the callback handle; it must be non-NULL, and if the callback has already been removed from the object instance then a warning is issued.

11.9.5.4 delete_by_name

```
static void delete_by_name( const std::string& name,
                            uvm_callback* cb,
                            uvm_component* root );
```

The member function **delete_by_name** shall remove the given callback object, *cb*, associated with one or more **uvm_component** callback queues. Argument *root* specifies the location in the component hierarchy to start the search for name. See **uvm_root**::**find_all** for more details on searching by name (see 4.3.3.2).

11.9.6 Iterator interfaces

This set of member functions shall provide an iterator interface for callback queues. A facade class, **uvm_callback_iter** is also available, and is the generally preferred way to iterate over callback queues. (See 11.8).

11.9.6.1 get_first

```
static CB* get_first( int& itr, T* obj );
```

The member function **get_first** shall return the first enabled callback of type CB which resides in the queue for object *obj*. If object *obj* is NULL, then the typewide queue for T is searched. Argument *itr* is the iterator; it will be updated with a value that can be supplied to get_next to get the next callback object. If the queue is empty, then NULL is returned. The iterator class **uvm_callback_iter** may be used as an alternative, simplified, iterator interface.

11.9.6.2 get_last

```
static CB* get_last( int& itr, T* obj );
```

The member function **get_last** shall return the last enabled callback of type CB which resides in the queue for object *obj*. If object *obj* is NULL, then the typewide queue for T is searched. Argument *itr* is the iterator; it will be updated with a value that can be supplied to get_prev to get the previous callback object. If the queue is empty then NULL is returned. The iterator class **uvm_callback_iter** may be used as an alternative, simplified, iterator interface.

11.9.6.3 get_next

```
static CB* get_next( int& itr, T* obj );
```

The member function **get_next** shall return the next enabled callback of type CB which resides in the queue for object *obj*, using iterator *itr* as the starting point. If object *obj* is NULL, then the typewide queue for T is searched.

The iterator will be updated with a value that can be supplied to get_next to get the next callback object. If no more callbacks exist in the queue, then NULL is returned. The member function **get_next** will continue to return NULL in this case until member function **get_first** or **get_last** has been used to reset the iterator. The iterator class **uvm_callback_iter** may be used as an alternative, simplified, iterator interface.

11.9.6.4 get_prev

```
static CB* get_prev( int& itr, T* obj );
```

The member function **get_prev** shall return the previous enabled callback of type CB which resides in the queue for object *obj*, using iterator *itr* as the starting point. If object *obj* is NULL, then the typewide queue for T is searched. The iterator will be updated with a value that can be supplied to member function **get_prev** to get the previous callback object. If no more callbacks exist in the queue, then NULL is returned. The member function **get_prev** will continue to return NULL in this case until member function **get_first** or **get_last** has been used to reset the iterator. The iterator class **uvm_callback_iter** may be used as an alternative, simplified, iterator interface.

11.9.7 Debug

11.9.7.1 display

```
static void display( T* obj = NULL );
```

The member function **display** shall display callback information for object *obj*. If object *obj* is NULL, then it displays callback information for all objects of type T, including typewide callbacks.

12. Reporting classes

The UVM-SystemC reporting classes provide an additional facility for issuing reports with consistent formatting. Users can configure what actions to take and what files to send output to based on report severity, ID, or both severity and ID. Users can also filter messages based on their verbosity settings. It supports a component-level reporting mechanism by setting the severity level on a per-instance basis. In addition, some convenience macros are available for the reporting of information, warnings, errors, or fatal errors.

SystemC has already an extensive and highly configurable message-reporting mechanism using the **sc_core**::**sc_report_handler** class and **sc_core**::**sc_report** objects. An application may also use this native SystemC global-level reporting mechanism where appropriate.

The following reporting classes are defined:

- **uvm_report_object**: The base class which provides the interface to the UVM reporting mechanism.
- **uvm_report_handler**: The class which acting as implementation for the member functions defined in the class **uvm_report_object**.
- **uvm_report_server**: The class acting as global server that processes all of the reports generated by the class **uvm_report_handler**.
- **uvm_report_catcher**: The class which captures and counts all reports issued by the class **uvm_report_server**.

The primary interface to the UVM reporting facility is the class **uvm_report_object** from which class **uvm_component** is derived. The class **uvm_report_object** delegates most tasks to its internal **uvm_report_handler**. If the report handler determines the report is not filtered based the configured verbosity setting, it sends the report to the central **uvm_report_server** for formatting and processing.

12.1 uvm_report_object

The class **uvm_report_object** shall provide the primary interface to the UVM reporting facility. Through this interface, components issue the various messages that occur during simulation. An application can configure what actions are taken and what file(s) are output for individual messages from a particular component or for all messages from all components in the environment. Defaults are applied where there is no explicit configuration.

A report consists of an id string, severity, verbosity level, and the textual message itself. They may optionally include the filename and line number from which the message came. If the verbosity level of a report is greater than the configured maximum verbosity level of its report object, it is ignored. If a report passes the verbosity filter in effect, the report's action is determined. If the action includes output to a file, the configured file descriptor(s) are determined.

- *Actions* can be set for (in increasing priority) severity, id, and (severity, id) pair. They include output to the screen or log file (**UVM_DISPLAY** or **UVM_LOG** respectively), whether the message counters should be incremented (**UVM_COUNT**), whether a simulation should be finished (**UVM_EXIT**) or stopped (**UVM_STOP**). The action can also specify if a specific callback should be called as soon as the reporting occurs (**UVM_CALL_HOOK**).
  Actions are of type **uvm_action** and can take the value **UVM_NO_ACTION**, or it can be a bitwise OR of any combination of **UVM_DISPLAY**, **UVM_LOG**, **UVM_COUNT**, **UVM_STOP**, **UVM_EXIT**, and **UVM_CALL_HOOK** (see 15.3.1).

- *Default actions*: The following provides the default actions assigned to each severity. These can be overridden by any of the member function **set_report_id_action**.

Severity	Default action(s)
UVM_INFO	**UVM_DISPLAY**
UVM_WARNING	**UVM_DISPLAY**, **UVM_COUNT**
UVM_ERROR	**UVM_DISPLAY**, **UVM_COUNT**
UVM_FATAL	**UVM_DISPLAY**, **UVM_COUNT**, **UVM_EXIT**

- *File descriptors*: These can be set by (in increasing priority) default, severity level, an id, or (severity, id) pair. File descriptors are of type **UVM_FILE**. They may refer to more than one file. It is the application's responsibility to open and close the files.
- *Default file handle*: The default file handle is 0, which means that reports are not sent to a file even if a **UVM_LOG** attribute is set in the action associated with the report. This can be overridden by the member function **set_report_default_file**, **set_report_severity_file**, **set_report_id_file** or **set_report_severity_id_file**. As soon as the file descriptor is set and the action **UVM_LOG** is set, the report will be sent to its associated file descriptor.

12.1.1 Class definition

```
namespace uvm {


  class uvm_report_object : public uvm_object
  {
   public:
    // Constructors
    uvm_report_object();
    explicit uvm_report_object( const std::string& name );


    // Group: Reporting
    bool uvm_report_enabled( int verbosity,
                             uvm_severity_type severity = UVM_INFO,
                             const std::string& id = "" );


    virtual void uvm_report_info( const std::string& id,
                                  const std::string& message,
                                  int verbosity = UVM_MEDIUM,
                                  const std::string& filename = "",
                                  int line = 0 ) const;
```

```
virtual void uvm_report_warning( const std::string& id,
                                 const std::string& message,
                                 int verbosity = UVM_MEDIUM,
                                 const std::string& filename = "",
                                 int line = 0 ) const;

virtual void uvm_report_error( const std::string& id,
                               const std::string& message,
                               int verbosity = UVM_LOW,
                               const std::string& filename = "",
                               int line = 0 ) const;

virtual void uvm_report_fatal( const std::string& id,
                               const std::string& message,
                               int verbosity = UVM_NONE,
                               const std::string& filename = "",
                               int line = 0 ) const;

// Group: Verbosilty Configuration
int get_report_verbosity_level( uvm_severity_type severity = UVM_INFO,
                                const std::string& id = "" ) const;
void set_report_verbosity_level( int verbosity_level );
void set_report_id_verbosity( const std::string& id, int verbosity );
void set_report_severity_id_verbosity( uvm_severity severity,
                                       const std::string& id,
                                       int verbosity );

// Action configuration
int get_report_action( uvm_severity severity,
                       const std::string& id ) const;
void set_report_severity_action( uvm_severity severity,
                                 uvm_action action );
void set_report_id_action( const std::string& id,
                           uvm_action action );
void set_report_severity_id_action( uvm_severity severity,
                                    const std::string& id,
                                    uvm_action action );
```

```
    // File configuration
    UVM_FILE get_report_file_handle( uvm_severity severity,
                                     const std::string& id ) const;
    void set_report_default_file( UVM_FILE file );
    void set_report_id_file( const std::string& id, UVM_FILE file );
    void set_report_severity_file( uvm_severity severity, UVM_FILE file );
    void set_report_severity_id_file( uvm_severity severity,
                                      const std::string& id,
                                      UVM_FILE file);

    // Override Configuration
    void set_report_severity_override( uvm_severity cur_severity,
                                       uvm_severity new_severity );

    void set_report_severity_id_override( uvm_severity cur_severity,
                                          const std::string& id,
                                          uvm_severity new_severity );

    // Group: Report Handler Configuration
    void set_report_handler( uvm_report_handler* handler );
    uvm_report_handler* get_report_handler() const;
    void reset_report_handler();

  }; // class uvm_report_object

} // namespace uvm
```

12.1.2 Constructors

```
uvm_report_object();
explicit uvm_report_object( const std::string& name );
```

The constructors shall create a new report object with the given name. This member function shall also create a new **uvm_report_handler** object to which most tasks are delegated.

12.1.3 Reporting

The member functions **uvm_report_info**, **uvm_report_warning** and **uvm_report_fatal** are the primary reporting methods in UVM. They ensure a consistent output and central control over where output is directed and any actions that result. All reporting member functions have the same arguments, although each has a different default verbosity:

- *id*: a unique id of type std::string for the report or report group that can be used for identification and therefore targeted filtering. An application can configure an individual report's actions and output file(s) using this id.
- *message*: the message body, preformatted to a single string of type std::string.
- *verbosity*: the verbosity of the message, indicating its relative importance. The verbosity shall be specified as an enumeration of type **uvm_verbosity**. If the equivalent verbosity value is less than or equal to the effective verbosity level (see **set_report_verbosity_level**), then the report is issued, subject to the configured action and file descriptor settings. Verbosity is ignored for warnings, errors, and fatals. However, if a warning, error or fatal is demoted to an info message using the **uvm_report_catcher**, then the verbosity is taken into account.
  The predefined **uvm_verbosity** values are **UVM_NONE**, **UVM_LOW**, **UVM_MEDIUM**, **UVM_HIGH**, and **UVM_FULL**.
- *filename* (optional): The file from which the report was issued. An application can use the predefined macros __FILE__ and __LINE__. If specified, it is displayed in the output.
- *line* (optional): The location from which the report was issued. An application can use the predefined macro __LINE__. If specified, it is displayed in the output.

12.1.3.1 uvm_report_enabled

```
bool uvm_report_enabled( int verbosity,
                         uvm_severity_type severity = UVM_INFO,
                         const std::string& id = "" );
```

The member function **uvm_report_enabled** shall return true if the configured verbosity for this severity/id is greater than or equal to the given argument *verbosity*; otherwise it shall return false.

12.1.3.2 uvm_report_info

```
virtual void uvm_report_info( const std::string& id,
                              const std::string& message,
                              int verbosity = UVM_MEDIUM,
                              const std::string& filename = "",
                              int line = 0 ) const;
```

The member function **uvm_report_info** shall issue an info message using the current messages report object.

12.1.3.3 uvm_report_warning

```
virtual void uvm_report_warning( const std::string& id,
                                 const std::string& message,
                                 int verbosity = UVM_MEDIUM,
                                 const std::string& filename = "",
                                 int line = 0 ) const;
```

The member function **uvm_report_warning** shall issue a warning message using the current messages report object.

12.1.3.4 uvm_report_error

```
virtual void uvm_report_error( const std::string& id,
                               const std::string& message,
                               int verbosity = UVM_LOW,
                               const std::string& filename = "",
                               int line = 0 ) const;
```

The member function **uvm_report_error** shall issue an error message using the current messages report object.

12.1.3.5 uvm_report_fatal

```
virtual void uvm_report_fatal( const std::string& id,
                               const std::string& message,
                               int verbosity = UVM_NONE,
                               const std::string& filename = "",
                               int line = 0 ) const;
```

The member function **uvm_report_fatal** shall issue a fatal message using the current messages report object.

12.1.4 Verbosity configuration

12.1.4.1 get_report_verbosity_level

```
int get_report_verbosity_level( uvm_severity_type severity = UVM_INFO,
                                const std::string& id = "" ) const;
```

The member function **get_report_verbosity_level** shall get the verbosity level in effect for this object. Reports issued with verbosity greater than this will be filtered out. The severity and tag arguments check if the verbosity level has been modified for specific severity/tag combinations.

12.1.4.2 set_report_verbosity_level

```
void set_report_verbosity_level( int verbosity_level );
```

The member function **set_report_verbosity_level** shall set the maximum verbosity level for reports for this component. Any report from this component whose verbosity exceeds this maximum will be ignored.

12.1.4.3 set_report_id_verbosity

```
void set_report_id_verbosity( const std::string& id, int verbosity );
```

The member function **set_report_id_verbosity** shall associate the specified verbosity with reports of the given id. A verbosity associated with a particular id takes precedence over a verbosity associated with a severity.

12.1.4.4 set_report_severity_id_verbosity

```
void set_report_severity_id_verbosity( uvm_severity severity,
                                       const std::string& id,
                                       int verbosity );
```

The member function **set_report_severity_id_verbosity** shall associate the specified verbosity with reports of the given severity-id pair. A verbosity associated with a particular severity-id pair takes precedence over an verbosity associated with id, which take precedence over an verbosity associated with a severity.

12.1.5 Action configuration

12.1.5.1 get_report_action

```
int get_report_action( uvm_severity severity,
                       const std::string& id ) const;
```

The member function **get_report_action** shall get the action associated with reports having the given *severity* and *id*.

12.1.5.2 set_report_severity_action

```
void set_report_severity_action( uvm_severity severity,
                                 uvm_action action );
```

The member function **set_report_severity_action** shall associate the specified action or actions with the given severity. An action associated with a particular severity-id pair or id, using the member functions **set_report_severity_id_action** or **set_report_id_action** respectively, shall take precedence over the association set by this member function.

12.1.5.3 set_report_id_action

```
void set_report_id_action( const std::string& id,
                           uvm_action action );
```

The member function **set_report_id_action** shall associate the specified action or actions with the given id. An action associated with a particular severity-id pair, using the member functions **set_report_severity_id_action**, shall take precedence over the association set by this member function.

12.1.5.4 set_report_severity_id_action

```
void set_report_severity_id_action( uvm_severity severity,
                                    const std::string& id,
                                    uvm_action action );
```

The member function **set_report_severity_id_action** shall associate the specified action or actions with the given id. An action associated with a particular severity-id pair shall take precedence over an action associated with id, which takes precedence over an action associated with a severity.

12.1.6 File configuration

12.1.6.1 get_report_file_handle

```
UVM_FILE get_report_file_handle( uvm_severity severity,
                                 const std::string& id ) const;
```

The member function **get_report_file_handle** shall get the file descriptor associated with reports having the given *severity* and *id*. Returned int in UVM-SystemVerilog

12.1.6.2 set_report_default_file

```
void set_report_default_file( UVM_FILE file );
```

The member function **set_report_default_file** shall configure the report handler to direct some or all of its output to the default *file* descriptor of type **UVM_FILE**. A file associated with a particular severity-id pair shall take precedence over a FILE associated with id, which shall take precedence over a file associated with a severity, which shall takes precedence over the association set by this member function.

12.1.6.3 set_report_id_file

```
void set_report_id_file( const std::string& id, UVM_FILE file );
```

The member function **set_report_id_file** shall configure the report handler to direct reports of the given *id* to the *file* descriptor of type **UVM_FILE**. A file associated with a particular severity-id shall take precedence over the association set by this member function.

12.1.6.4 set_report_severity_file

```
void set_report_severity_file( uvm_severity severity, UVM_FILE file );
```

The member function **set_report_severity_file** shall configure the report handler to direct reports of the given *severity* to the *file* descriptor of type **UVM_FILE**. A file associated with a particular severity-id or associated with a specific id, shall take precedence over the association set by this member function.

12.1.6.5 set_report_severity_id_file

```
void set_report_severity_id_file( uvm_severity severity,
                                  const std::string& id,
                                  UVM_FILE file);
```

The member function **set_report_severity_id_file** shall configure the report handler to direct reports of the given *severity-id* pair to the given *file* descriptor of type **UVM_FILE**. A file associated with a particular *severity-id* pair shall take precedence over a file associated with *id*, which shall take precedence over a file associated with a *severity*, which takes precedence over the default file descriptor.

12.1.7 Override configuration

12.1.7.1 set_report_severity_override

```
void set_report_severity_override( uvm_severity cur_severity,
                                   uvm_severity new_severity );
```

The member function **set_report_severity_override** shall provide the ability to upgrade or downgrade a message in terms of severity given *severity*. An upgrade or downgrade for a specific id, using member function **set_report_severity_id_override,** shall take precedence over an upgrade or downgrade set by this member function.

12.1.7.2 set_report_severity_id_override

```
void set_report_severity_id_override( uvm_severity cur_severity,
                                      const std::string& id,
                                      uvm_severity new_severity );
```

The member function **set_report_severity_id_override** shall provide the ability to upgrade or downgrade a message in terms of severity given *severity*. An upgrade or downgrade for a specific *id* takes precedence over an upgrade or downgrade associated with a *severity*.

12.1.8 Report handler configuration

12.1.8.1 set_report_handler

```
void set_report_handler( uvm_report_handler* handler );
```

The member function **set_report_handler** shall set the report handler, overwriting the default instance. This allows more than one component to share the same report handler.

12.1.8.2 get_report_handler

```
uvm_report_handler* get_report_handler() const;
```

The member function **get_report_handler** shall return the underlying report handler to which most reporting tasks are delegated.

12.1.8.3 reset_report_handler

```
void reset_report_handler();
```

The member function **reset_report_handler** shall reset the underlying report handler to its default settings. This clears any settings made with the member functions **set_report_***. FULL NAMES

12.2 uvm_report_handler

The class **uvm_report_handler** is the class to which most methods in **uvm_report_object** delegate. It stores the maximum verbosity, actions, and files that affect the way reports are handled.

The report handler is not intended for direct use. See **uvm_report_object** for information on the UVM reporting mechanism.

The relationship between class **uvm_report_object**, which is a base class for **uvm_component**, and class **uvm_report_handler** is typically one to one, but it can be many to one if several objects of type **uvm_report_object** are configured to use the same **uvm_report_handler**.

See **uvm_report_object**::**set_report_handler**.

The relationship between an object of type **uvm_report_handler** and an object of type **uvm_report_server** is many to one.

12.2.1 Class definition

```
namespace uvm {


  class uvm_report_handler
  {
   public:
    uvm_report_handler();


    int get_verbosity_level( uvm_severity severity = UVM_INFO,
                             const std::string& id = "" );


    uvm_action get_action( uvm_severity severity,
                           const std::string& id );


    UVM_FILE get_file_handle( uvm_severity severity,
                              const std::string& id );


    virtual void report( uvm_severity severity,
                         const std::string& name,
                         const std::string& id,
                         const std::string& message,
                         int verbosity_level = UVM_MEDIUM,
                         const std::string& filename = "",
                         int line = 0,
                         uvm_report_object* client = NULL );


    std::string format_action( uvm_action action );


  }; // class uvm_report_handler
```

```
} // namespace uvm
```

12.2.2 Constructor

```
uvm_report_handler();
```

The constructor shall create and initialize a new handler object.

12.2.3 Member functions

12.2.3.1 get_verbosity_level

```
int get_verbosity_level( uvm_severity severity = UVM_INFO,
                         const std::string& id = "" );
```

The member function **get_verbosity_level** shall return the verbosity associated with the given *severity* and *id*.

First, if there is a verbosity associated with the pair (*severity*, *id*), return that. Else, if there is a verbosity associated with the *id*, return that. Else, return the maximum verbosity setting.

12.2.3.2 get_action

```
uvm_action get_action( uvm_severity severity,
                       const std::string& id );
```

The member function **get_action** shall return the action associated with the given *severity* and *id*. First, if there is an action associated with the pair (*severity*, *id*), return that. Else, if there is an action associated with the *id*, return that. Else, if there is an action associated with the *severity*, return that. Else, return the default action associated with the severity.

12.2.3.3 get_file_handle

```
UVM_FILE get_file_handle( uvm_severity severity,
                          const std::string& id );
```

The member function **get_file_handle** shall return the file descriptor **UVM_FILE** associated with the given *severity* and *id*. First, if there is a file handle associated with the pair (*severity*, *id*), return that. Else, if there is a file handle associated with the *id*, return that. Else, if there is an file handle associated with the *severity*, return that. Else, return the default file handle.

12.2.3.4 report

```
virtual void report( uvm_severity severity,
                     const std::string& name,
                     const std::string& id,
                     const std::string& message,
```

```
                    int verbosity_level = UVM_MEDIUM,
                    const std::string& filename = "",
                    int line = 0,
                    uvm_report_object* client = NULL );
```

The member function **report** shall be used by the four core reporting methods, **uvm_report_error**, **uvm_report_info**, **uvm_report_warning**, **uvm_report_fatal**, of class **uvm_report_object**.

12.2.3.5 format_action

```
std::string format_action( uvm_action action );
```

The member function **format_action** shall return a string representation of the action, e.g., “DISPLAY”.

12.3 uvm_report_server

The class **uvm_report_server** shall act as a global server that processes all of the reports generated by a **uvm_report_handler**. None of its member functions are intended to be called by normal testbench code, although in some circumstances the virtual member functions **process_report** and/or **compose_uvm_info** may be overloaded in a subclass.

12.3.1 Class definition

```
namespace uvm {


  class uvm_report_server : public uvm_object
  {
   public:
    uvm_report_server();


    static void set_server( uvm_report_server* server );
    static uvm_report_server* get_server();


    void set_max_quit_count( int count, bool overridable = true );
    int get_max_quit_count() const;
    void set_quit_count( int quit_count );
    int get_quit_count() const;
    void incr_quit_count();
    void reset_quit_count();
    bool is_quit_count_reached();


    void set_severity_count( uvm_severity severity, int count );
```

```
    int get_severity_count( uvm_severity severity ) const;
    void incr_severity_count( uvm_severity severity );
    void reset_severity_counts();

    void set_id_count( const std::string& id, int count );
    int get_id_count( const std::string& id ) const;
    void incr_id_count( const std::string& id );

    virtual void process_report( uvm_severity severity,
                                 const std::string& name,
                                 const std::string& id,
                                 const std::string& message,
                                 uvm_action action,
                                 UVM_FILE file,
                                 const std::string& filename,
                                 int line,
                                 const std::string& composed_message,
                                 int verbosity_level,
                                 uvm_report_object* client );

    virtual std::string compose_message( uvm_severity severity,
                                         const std::string& name,
                                         const std::string& id,
                                         const std::string& message,
                                         const std::string& filename,
                                         int line ) const;

    virtual void report_summarize( UVM_FILE file = 0 );

    void dump_server_state() const;

  }; // class uvm_report_server

} // namespace uvm
```

12.3.2 Constructor

```
uvm_report_server();
```

The constructor shall create a **uvm_report_server** object, if not already created. Else, it does nothing.

12.3.3 Member functions

12.3.3.1 set_server

```
static void set_server( uvm_report_server* server );
```

The member function **set_server** shall set the global report server to use for reporting. The report server is responsible for formatting messages.

12.3.3.2 get_server

```
static uvm_report_server* get_server();
```

The member function **get_server** shall get the global report server. This member function will always return a valid handle to a report server.

12.3.3.3 set_max_quit_count

```
void set_max_quit_count( int count, bool overridable = true );
```

The member function **set_max_quit_count** shall set the maximum number of COUNT actions that can be tolerated before a **UVM_EXIT** action is taken. The default is 0, which specifies no maximum.

12.3.3.4 get_max_quit_count

```
int get_max_quit_count() const;
```

The member function **get_max_quit_count** shall get the maximum number of COUNT actions that can be tolerated before a **UVM_EXIT** action is taken. The member function shall return 0 if no maximum is set.

12.3.3.5 set_quit_count

```
void set_quit_count( int quit_count );
```

The member function **set_quit_count** shall set the quit count, i.e., the number of COUNT actions, to the value *quit_count*.

12.3.3.6 get_quit_count

```
int get_quit_count() const;
```

The member function **get_quit_count** shall get the quit count, i.e., the number of COUNT actions.

12.3.3.7 incr_quit_count

```
void incr_quit_count();
```

The member function **incr_quit_count** shall increase the quit count, i.e., the number of COUNT actions. WITH ONE?

12.3.3.8 reset_quit_count

```
void reset_quit_count();
```

The member function **reset_quit_count** shall reset the quit count, i.e., the number of COUNT actions, to 0.

12.3.3.9 is_quit_count_reached

```
bool is_quit_count_reached();
```

The member function **is_quit_count_reached** shall return *true* when the quit counter has reached the maximum.

12.3.3.10 set_severity_count

```
void set_severity_count( uvm_severity severity, int count );
```

The member function **set_severity_count** shall set the counter for the given *severity* to counter value *count*.

12.3.3.11 get_severity_count

```
int get_severity_count( uvm_severity severity ) const;
```

The member function **get_severity_count** shall get the counter value for the given *severity*.

12.3.3.12 incr_severity_count

```
void incr_severity_count( uvm_severity severity );
```

The member function **incr_severity_count** shall increase the counter value for the given *severity*. WITH ONE?

12.3.3.13 reset_severity_counts

```
void reset_severity_counts();
```

The member function **reset_severity_counts** shall reset all severity counters to 0.

12.3.3.14 set_id_count

```
void set_id_count( const std::string& id, int count );
```

The member function **set_id_count** shall set the counter for reports with the given *id*.

12.3.3.15 get_id_count

```
int get_id_count( const std::string& id ) const;
```

The member function **get_id_count** shall get the counter for reports with the given *id*.

12.3.3.16 incr_id_count

```
void incr_id_count( const std::string& id );
```

The member function **incr_id_count** shall increase the counter for reports with the given *id*. WITH ONE?

12.3.3.17 process_report

```
virtual void process_report( uvm_severity severity,
                             const std::string& name,
                             const std::string& id,
                             const std::string& message,
                             uvm_action action,
                             UVM_FILE file,
                             const std::string& filename,
                             int line,
                             const std::string& composed_message,
                             int verbosity_level,
                             uvm_report_object* client );
```

The member function **process_report** shall call the member function **compose_message** to construct the actual message to be output. It then takes the appropriate action according to the value of action and file. This member function can be overloaded by an application to customize the way the reporting system processes reports and the actions enabled for them.

12.3.3.18 compose_message

```
virtual std::string compose_message( uvm_severity severity,
                                     const std::string& name,
                                     const std::string& id,
                                     const std::string& message,
                                     const std::string& filename,
                                     int line ) const;
```

The member function **compose_message** shall construct the actual string sent to the file or command line from the *severity*, component *name*, report *id*, and the *message* itself. An application can overload this member function to customize report formatting.

12.3.3.19 report_summarize

```
virtual void report_summarize( UVM_FILE file = 0 ); make const?
```

The member function **report_summarize** shall output statistical information issued by this central report server. This information is sent to the standard output (stdout) if there is no argument specified or if the argument *file* is 0; otherwise the information is send to a file using the argument *file* as file handle. The member function **uvm_root**::**run_test** shall call this member function at the end of simulation.

12.3.3.20 dump_server_state

```
void dump_server_state() const;
```

The member function **dump_server_state** shall print server state information. Add: The actual information is implementation defined?

12.4 uvm_report_catcher

The class **uvm_report_catcher** shall be used to catch messages issued by the **uvm report server**. Catchers are objects of type **uvm_callbacks**<**uvm_report_object**, **uvm_report_catcher**>, so all facilities in the classes **uvm_callback** and **uvm_callbacks**<**T**, **CB**> are available for registering catchers and controlling catcher state.

Multiple report catchers can be registered with a report object. The catchers can be registered as default catchers which catch all reports on all reporters of type **uvm_report_object**, or catchers can be attached to specific report objects (i.e. components).

User extensions of uvm_report_catcher must implement the catch method in which the action to be taken on catching the report is specified. The catch method can return CAUGHT, in which case further processing of the report is immediately stopped, or return THROW in which case the (possibly modified) report is passed on to other registered catchers. The catchers are processed in the order in which they are registered.

On catching a report, the catch method can modify the severity, id, action, verbosity or the report string itself before the report is finally issued by the report server. The report can be immediately issued from within the catcher class by calling the issue method.

The catcher maintains a count of all reports with severity **UVM_FATAL**, **UVM_ERROR** or **UVM_WARNING** severity and a count of all reports with severity **UVM_FATAL**, **UVM_ERROR** or **UVM_WARNING** whose severity was lowered. These statistics are reported in the summary of the **uvm_report_server**.

12.4.1 Class definition

```
namespace uvm {


  class uvm_report_catcher : public uvm_callback
  {
   public:
    typedef enum { UNKNOWN_ACTION, THROW, CAUGHT} action_e;


    uvm_report_catcher( const std::string& name = "uvm_report_catcher" );


    // Group: Current Message State
    uvm_report_object* get_client() const;
    uvm_severity get_severity() const;
    int get_verbosity() const;
    std::string get_id() const;
```

```
  std::string get_message() const;
  uvm_action get_action() const;
  std::string get_fname() const;
  int get_line() const;

  // Group: Change Message State
 protected:
  void set_severity( uvm_severity severity );
  void set_verbosity( int verbosity );
  void set_id( const std::string& id );
  void set_message( const std::string& message );
  void set_action( uvm_action action );

  // Group: Debug
  static uvm_report_catcher* get_report_catcher( const std::string& name );
  static void print_catcher( UVM_FILE file = 0 );

  // Group: Callback interface
  virtual action_e do_catch°() = 0;

  // Group: Reporting
 protected:
  void uvm_report_fatal( const std::string& id,
                         const std::string& message,
                         int verbosity,
                         const std::string& fname = "",
                         int line = 0 );

  void uvm_report_error( const std::string& id,
                         const std::string& message,
                         int verbosity,
                         const std::string& fname = "",
                         int line = 0 );

  void uvm_report_warning( const std::string& id,
                           const std::string& message,
                           int verbosity,
```

```
                              const std::string& fname = "",
                              int line = 0 );

    void uvm_report_info( const std::string& id,
                          const std::string& message,
                          int verbosity,
                          const std::string& fname = "",
                          int line = 0 );

    void issue();
    static void summarize_report_catcher( UVM_FILE file );

  }; // class uvm_report_catcher

} // namespace uvm
```

12.4.2 Constructor

```
uvm_report_catcher( const std::string& name = "uvm_report_catcher" );
```

The constructor shall create a new report catcher object. The argument *name* is optional, but should generally be provided to aid in debugging.

12.4.3 Current message state

12.4.3.1 get_client

```
uvm_report_object* get_client() const;
```

The member function **get_client** shall return the **uvm_report_object** that has generated the message that is currently being processed.

12.4.3.2 get_severity

```
uvm_severity get_severity() const;
```

The member function **get_severity** shall return the **uvm_severity** of the message that is currently being processed. If the severity was modified by a previously executed report object (which re-threw the message), then the returned severity is the modified value.

12.4.3.3 get_verbosity

```
int get_verbosity() const;
```

The member function **get_verbosity** shall return the verbosity of the message that is currently being processed. If the verbosity was modified by a previously executed report object (which re-threw the message), then the returned verbosity is the modified value.

12.4.3.4 get_id

```
std::string get_id() const;
```

The member function **get_id** shall return the string id of the message that is currently being processed. If the id was modified by a previously executed report object (which re-threw the message), then the returned id is the modified value.

12.4.3.5 get_message

```
std::string get_message() const;
```

The member function **get_message** shall return the string message of the message that is currently being processed. If the message was modified by a previously executed report object (which re-threw the message), then the returned message is the modified value.

12.4.3.6 get_action

```
uvm_action get_action() const;
```

The member function **get_action** shall return the **uvm_action** of the message that is currently being processed. If the action was modified by a previously executed report object (which re-threw the message), then the returned action is the modified value.

12.4.3.7 get_fname

```
std::string get_fname() const;
```

The member function **get_fname** shall return the file name of the message.

12.4.3.8 get_line

```
int get_line() const;
```

The member function **get_line** shall return the line number of the message.

12.4.4 Change message state

12.4.4.1 set_severity

```
void set_severity( uvm_severity severity );
```

The member function **set_severity** shall change the severity of the message to *severity*. Any other report catchers will see the modified value.

12.4.4.2 set_verbosity

```
void set_verbosity( int verbosity );
```

The member function **set_severity** shall change the verbosity of the message to *verbosity*. Any other report catchers will see the modified value.

12.4.4.3 set_id

```
void set_id( const std::string& id );
```

The member function **set_id** shall change the id of the message to *id*. Any other report catchers will see the modified value.

12.4.4.4 set_message

```
void set_message( const std::string& message );
```

The member function **set_message** shall change the text of the message to *message*. Any other report catchers will see the modified value.

12.4.4.5 set_action

```
void set_action( uvm_action action );
```

The member function **set_action** shall change the action of the message to *action*. Any other report catchers will see the modified value.

12.4.5 Debug

12.4.5.1 get_report_catcher

```
static uvm_report_catcher* get_report_catcher( const std::string& name );
```

The member function **get_report_catcher** shall return the first report catcher that has name.

12.4.5.2 print_catcher

```
static void print_catcher( UVM_FILE file = 0 );
```

The member function **print_catcher** shall print information about all of the report catchers that are registered. For finer grained detail, the member function **uvm_callbacks**<T,CB>::**display** can be used by calling **uvm_report_cb**::**display**(**uvm_report_object**).

12.4.6 Callback interface

12.4.6.1 do_catch° (catch†)

```
virtual action_e do_catch°() = 0
```

The member function **do_catch**° shall be called for each registered report catcher. The member functions in the Current Message State interface (see 12.4.3) can be used to access information about the current message being processed.

12.4.7 Reporting

12.4.7.1 uvm_report_fatal

```
void uvm_report_fatal( const std::string& id,
                       const std::string& message,
                       int verbosity,
                       const std::string& fname = "",
                       int line = 0 );
```

The member function **uvm_report_fatal** shall issue a fatal message using the current messages report object. This message will bypass any message catching callbacks.

12.4.7.2 uvm_report_error

```
void uvm_report_error( const std::string& id,
                       const std::string& message,
                       int verbosity,
                       const std::string& fname = "",
                       int line = 0 );
```

The member function **uvm_report_error** shall issue an error message using the current messages report object. This message will bypass any message catching callbacks.

12.4.7.3 uvm_report_warning

```
void uvm_report_warning( const std::string& id,
                       const std::string& message,
                       int verbosity,
                       const std::string& fname = "",
                       int line = 0 );
```

The member function **uvm_report_warning** shall issue a warning message using the current messages report object. This message will bypass any message catching callbacks.

12.4.7.4 uvm_report_info

```
void uvm_report_info( const std::string& id,
                       const std::string& message,
                       int verbosity,
                       const std::string& fname = "",
```

```
                          int line = 0 );
```

The member function **uvm_report_info** shall issue an info message using the current messages report object. This message will bypass any message catching callbacks.

12.4.7.5 issue

```
void issue();
```

The member function **issue** shall immediately issue the message which is currently being processed. This is useful if the message is being CAUGHT but should still be emitted. Issuing a message will update the report_server stats, possibly multiple times if the message is not CAUGHT.

12.4.7.6 summarize_report_catcher

```
static void summarize_report_catcher( UVM_FILE file );
```

The member function **summarize_report_catcher** shall print the statistics for the active catchers. It shall be called automatically by the member function **uvm_report_server**::**summarize**.

13. Macros

UVM-SystemC defines macros for the following functions:

- Component and object registration
- Reporting
- Sequence execution
- Callbacks

13.1 Component and object registration macros

These macros shall register components and objects with the **uvm_factory**, using the component registry **uvm_component_registry** or **uvm_object_registry**, respectively. In addition, they shall implement the member functions **get_type** and **get_type_name** to facilitate debugging and factory configuration or overrides.

13.1.1 Macro definitions

```
namespace uvm {


  #define UVM_OBJECT_UTILS( implementation-defined ) implementation-defined
  #define UVM_OBJECT_PARAM_UTILS( implementation-defined ) implementation-defined
  #define UVM_COMPONENT_UTILS( implementation-defined ) implementation-defined
  #define UVM_COMPONENT_PARAM_UTILS( implementation-defined ) implementation-defined


} // namespace uvm
```

13.1.2 UVM_OBJECT_UTILS, UVM_OBJECT_PARAM_UTILS

```
#define UVM_OBJECT_UTILS( implementation-defined ) implementation-defined
#define UVM_OBJECT_PARAM_UTILS( implementation-defined ) implementation-defined
```

The macros **UVM_OBJECT_UTILS** and **UVM_OBJECT_PARAM_UTILS** shall implement the following functionality:

- Implement the virtual member function **get_type_name** with the following signature:
  `virtual const std::string get_type_name() const;`
  This member function shall return the name of the class, which is provided as argument to this macro, as string.
- Implement the static member function **get_type** with the following signature:
  `static uvm_object_registry< classname >* get_type();`
  This member function shall return the factory proxy object as pointer of type **uvm_object_registry**.
- Register the class with the factory.

NOTE–An implementation may use the concept of variadic macros to be able to accept a variable number of macro arguments.

13.1.3 UVM_COMPONENT_UTILS, UVM_COMPONENT_PARAM_UTILS

```
#define UVM_COMPONENT_UTILS( implementation-defined ) implementation-defined
#define UVM_COMPONENT_PARAM_UTILS( implementation-defined ) implementation-defined
```

The macros **UVM_COMPONENT_UTILS** and **UVM_COMPONENT_PARAM_UTILS** shall implement the following functionality:

- Implement the virtual member function **get_type_name** with the following signature:
  `virtual const std::string get_type_name() const;`
  This member function shall return the name of the class, which is provided as argument to this macro, as string.
- Implement the static member function **get_type** with the following signature:
  `static uvm_component_registry< classname >* get_type();`
  This member function shall return the factory proxy object as pointer of type **uvm_component_registry**.
- Register the class with the factory

NOTE–An implementation may use the concept of variadic macros to be able to accept a variable number of macro arguments.

13.2 Reporting macros

The report macros shall provide additional functionality to the UVM reporting classes to facilitate efficient filtering messages based on verbosity, id and severity information, as well as annotating file and line number information to the reported messages.

13.2.1 Macro definitions

```
namespace uvm {

  #define UVM_INFO( ID, MSG, VERBOSITY ) implementation-defined
  #define UVM_WARNING( ID, MSG ) implementation-defined
  #define UVM_ERROR( ID, MSG ) implementation-defined
  #define UVM_FATAL( ID, MSG ) implementation-defined

} // namespace uvm
```

13.2.2 UVM_INFO

```
#define UVM_INFO( ID, MSG, VERBOSITY ) implementation-defined
```

The macro **UVM_INFO** shall only call member function **uvm_report_info** if argument VERBOSITY is lower than the configured verbosity of the associated reporter. Argument ID is given as the message tag and argument MSG is given as the message text. The file and line number are also sent to the member function **uvm_report_info** by means of using the predefined macros __FILE__ and __LINE__.

13.2.3 UVM_WARNING

```
#define UVM_WARNING( ID, MSG ) implementation-defined
```

The macro **UVM_WARNING** shall call the member function **uvm_report_warning** with a verbosity of **UVM_NONE**. The message cannot be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. Argument ID is given as the message tag and argument MSG is given as the message text. The file and line number are also sent to the member function **uvm_report_warning** by means of using the predefined macros __FILE__ and __LINE__.

13.2.4 UVM_ERROR

```
#define UVM_ERROR( ID, MSG ) implementation-defined
```

The macro **UVM_ERROR** shall call the member function **uvm_report_error** with a verbosity of **UVM_NONE**. The message cannot be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. Argument ID is given as the message tag and argument MSG is given as the message text. The file and line number are also sent to the member function **uvm_report_error** by means of using the predefined macros __FILE__ and __LINE__.

13.2.5 UVM_FATAL

```
#define UVM_FATAL( ID, MSG ) implementation-defined
```

The macro **UVM_FATAL** shall call member function **uvm_report_fatal** with a verbosity of **UVM_NONE**. The message cannot be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. Argument ID is given as the message tag and argument MSG is given as the message text. The file and line number are also sent to the member function **uvm_report_fatal** by means of using the predefined macros __FILE__ and __LINE__.

13.3 Sequence execution macros

The sequence execution macros are shall provide a convenience layer to start sequences or sequence items on a default sequencer, if not specified, or on another sequencer if specified.

NOTE–It is strongly recommended not to use the sequence execution macros in an application. Instead, for a sequence item to start, it is recommended to use the member functions **start_item** (see 9.3.6.2) and **finish_item** (see 9.3.6.3). To start a sequence, it is recommended to use the member function **start** (see 9.3.4.1).

13.3.1 Macro definitions

```
namespace uvm {


  #define UVM_DO( SEQ_OR_ITEM ) implementation-defined
  #define UVM_DO_PRI( SEQ_OR_ITEM, PRIORITY ) implementation-defined
  #define UVM_DO_WITH( SEQ_OR_ITEM, CONSTRAINTS ) implementation-defined NOT IMPLEMENTED
  #define UVM_DO_PRI_WITH( SEQ_OR_ITEM, PRIORITY, CONSTRAINTS ) \
    implementation-defined NOT IMPLEMENTED
```

```
  #define UVM_DO_ON( SEQ_OR_ITEM, SEQR ) implementation-defined
  #define UVM_DO_ON_PRI( SEQ_OR_ITEM, SEQR, PRIORITY ) implementation-defined
  #define UVM_DO_ON_WITH( SEQ_OR_ITEM, SEQR, CONSTRAINTS )
    implementation-defined NOT IMPLEMENTED
  #define UVM_DO_ON_PRI_WITH( SEQ_OR_ITEM, SEQR, PRIORITY, CONSTRAINTS ) \
    implementation-defined NOT IMPLEMENTED

  #define UVM_CREATE( SEQ_OR_ITEM ) implementation-defined
  #define UVM_CREATE_ON( SEQ_OR_ITEM, SEQR ) implementation-defined

  #define UVM_DECLARE_P_SEQUENCER( SEQR ) implementation-defined

} // namespace uvm
```

13.3.2 UVM_DO

```
#define UVM_DO( SEQ_OR_ITEM ) implementation-defined
```

The macro **UVM_DO** shall start the execution of a sequence or sequence item. It takes as an argument *SEQ_OR_ITEM*, which is an object of type **uvm_sequence_item** or object of type **uvm_sequence**.

In the case of a sequence, the sub-sequence shall be started using member function **uvm_sequence_base**::**start** with argument *call_pre_post* set to false. In the case of a sequence item, the item shall be sent to the driver through the associated sequencer.

NOTE–Randomization is not yet implemented as part of the UVM_DO macro.

13.3.3 UVM_DO_PRI

```
#define UVM_DO_PRI( SEQ_OR_ITEM, PRIORITY ) implementation-defined
```

The macro **UVM_DO_PRI** shall implement the same functionality as **UVM_DO**, except that the sequence item or sequence is executed with the priority specified in the argument *PRIORITY*.

13.3.4 UVM_DO_WITH

```
#define UVM_DO_WITH( SEQ_OR_ITEM, CONSTRAINTS ) implementation-defined NOT IMPLEMENTED
```

The macro **UVM_DO_WITH** shall implement the same functionality as **UVM_DO**, except that the constraint block as second argument is applied to the item or sequence in a randomize with statement before execution.

13.3.5 UVM_DO_PRI_WITH

```
#define UVM_DO_PRI_WITH( SEQ_OR_ITEM, PRIORITY, CONSTRAINTS ) \
```

```
  implementation-defined NOT IMPLEMENTED
```

The macro **UVM_DO_PRI_WITH** shall implement the same functionality as **UVM_DO_PRI**, except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

13.3.6 UVM_DO_ON

```
#define UVM_DO_ON( SEQ_OR_ITEM, SEQR ) implementation-defined
```

The macro **UVM_DO_ON** shall implement the same functionality as **UVM_DO**, except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified argument *SEQR*.

13.3.7 UVM_DO_ON_PRI

```
#define UVM_DO_ON_PRI( SEQ_OR_ITEM, SEQR, PRIORITY ) implementation-defined
```

The macro **UVM_DO_ON_PRI** shall implement the same functionality as **UVM_DO_PRI**, except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified argument *SEQR*.

13.3.8 UVM_DO_ON_WITH

```
#define UVM_DO_ON_WITH( SEQ_OR_ITEM, SEQR, CONSTRAINTS )
  implementation-defined NOT IMPLEMENTED
```

The macro **UVM_DO_ON_PRI** shall implement the same functionality as **UVM_DO_WITH**, except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified argument *SEQR*.

13.3.9 UVM_DO_ON_PRI_WITH

```
#define UVM_DO_ON_PRI_WITH( SEQ_OR_ITEM, SEQR, PRIORITY, CONSTRAINTS ) \
  implementation-defined NOT IMPLEMENTED
```

The macro **UVM_DO_ON_PRI_WITH** shall implement the same functionality as **UVM_DO_PRI_WITH**, except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified argument *SEQR*.

13.3.10 UVM_CREATE

```
#define UVM_CREATE( SEQ_OR_ITEM ) implementation-defined
```

The macro **UVM_CREATE** shall create and register the sequence item or sequence using the factory. It intentionally does not start the execution.

NOTE–After calling this member function, an application can manually set values and start the execution.

13.3.11 UVM_CREATE_ON

```
#define UVM_CREATE_ON( SEQ_OR_ITEM, SEQR ) implementation-defined
```

The macro **UVM_CREATE_ON** shall implement the same functionality as UVM_CREATE, except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified argument *SEQR*.

13.3.12 UVM_DECLARE_P_SEQUENCER

```
#define UVM_DECLARE_P_SEQUENCER( SEQR ) implementation-defined
```

The macro **UVM_DECLARE_P_SEQUENCER** shall declare a variable *p_sequencer* whose type is specified by The argument *SEQUENCER*.

13.4 Callback macros

The callback macros shall register and execute callbacks which are derived from class **uvm_callbacks**.

13.4.1 Macro definitions

```
namespace uvm {


  #define UVM_REGISTER_CB( T, CB ) implementation-defined
  #define UVM_DO_CALLBACKS( T, CB, METHOD ) implementation-defined


} // namespace uvm
```

13.4.2 UVM_REGISTER_CB

```
#define UVM_REGISTER_CB( T, CB ) implementation-defined
```

The macro **UVM_REGISTER_CB** shall register the given callback type *CB* with the given object type *T*. If a type-callback pair is not registered, then a warning is issued if an attempt is made to use the pair (add, delete, etc.).

13.4.3 UVM_DO_CALLBACKS

```
#define UVM_DO_CALLBACKS( T, CB, METHOD ) implementation-defined
```

The macro **UVM_DO_CALLBACKS** shall call the given *METHOD* of all callbacks of type *CB* registered with the calling object (i.e. this object), which is or is based on type *T*.

This macro executes all of the callbacks associated with the calling object (i.e. this object). The macro takes three arguments:

- *CB* is the class type of the callback objects to execute. The class type must have a function signature that matches the argument *METHOD*.

- *T* is the type associated with the callback. Typically, an instance of type T is passed as one the arguments in the *METHOD* call.
- *METHOD* is the method call to invoke, with all required arguments as if they were invoked directly.

14. TLM interfaces

The TLM interfaces of UVM-SystemC shall be derived from the SystemC TLM interface definitions as defined in IEEE Std. 1666-2011. As communication between UVM components is primarily based on TLM-1 message passing semantics, dedicated ports and exports are defined compliant with these semantics.

The TLM interfaces are described in section 14.1.

NOTE–UVM-SystemC does not yet define the TLM-2.0 blocking and non-blocking transport interfaces, direct memory interface (DMI), nor a debug transport interface.

14.1 TLM-1 interfaces

The following TLM-1 interfaces are defined in UVM-SystemC:

- Blocking interface classes
  - **uvm_blocking_put_port**
  - **uvm_blocking_get_port**
  - **uvm_blocking_peek_port**
  - **uvm_blocking_get_peek_port**
- Non-blocking interface classes TODO
  - **uvm_nonblocking_put_port**
  - **uvm_nonblocking_get_port**
  - **uvm_nonblocking_peek_port**
  - **uvm_nonblocking_get_peek_port**
- Analysis interface port and export classes:
  - **uvm_analysis_port**
  - **uvm_analysis_export**
  - **uvm_analysis_imp**
- Request-response channel class:
  - **uvm_tlm_req_rsp_channel**.
- Transport channel class:
  - **uvm_tlm_transport_channel**. TODO

NOTE–There are no dedicated TLM-1 FIFO and FIFO interface classes defined in UVM-SystemC. Instead, the use the SystemC FIFO base classes **tlm**::**tlm_fifo**<T> or **tlm**::**tlm_analysis_fifo**, or FIFO interfaces **tlm**::**tlm_fifo_debug_if**, **tlm**::**tlm_fifo_put_if**, and **tlm**::**tlm_fifo_get_if** is recommended.

14.2 uvm_blocking_put_port

The class **uvm_blocking_put_port** offers a convenience layer for UVM users to access the SystemC TLM-1 blocking interface **tlm**::**tlm_blocking_put_if**. As this port class shall be derived from class **uvm_port_base**, it inherits the UVM specific member functions **connect**, **get_name**, **get_full_name** and **get_type_name**.

14.2.1 Class definition

```
namespace uvm {


  template <typename T>
  class uvm_blocking_put_port : public uvm_port_base< tlm::tlm_blocking_put_if<T> >
  {
   public:
    uvm_blocking_put_port();
    uvm_blocking_put_port( const std::string& name );
    virtual const std::string get_type_name() const;
    virtual void put( const T& val );


  }; // class uvm_blocking_put_port


} // namespace uvm
```

14.2.2 Template parameter T

The template parameter T specifies the type of transaction to be communicated by the port.

14.2.3 Constructor

```
uvm_blocking_put_port();
uvm_blocking_put_port( const std::string& name );
```

The constructor shall create a new port with TLM-1 blocking put interface semantics. If specified, the argument *name* shall define the name of the port. Otherwise, the name of the port is implementation-defined.

14.2.4 Member functions

14.2.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_blocking_put_port**".

14.2.4.2 put

```
virtual void put( const T& val );
```

The member function **put** shall send the transaction of type T to the recipient. It shall call the member function **put** of the associated interface which is bound to this port.

According to the TLM-1 blocking put semantics, the member function **put** shall not return until the recipient has indicated that the transaction object has been processed, by calling member function **get** or **peek**. Subsequent calls to the member function **put** shall be treated as distinct transaction instances, regardless of whether or not the same transaction object or message is passed.

14.3 uvm_blocking_get_port

The class **uvm_blocking_get_port** offers a convenience layer for UVM users to access the SystemC TLM-1 blocking interface **tlm**::**tlm_blocking_get_if**. As this port class shall be derived from class **uvm_port_base**, it inherits the UVM specific member functions **connect**, **get_name**, **get_full_name** and **get_type_name**.

14.3.1 Class definition

```
namespace uvm {


  template <typename T>
  class uvm_blocking_get_port : public uvm_port_base< tlm::tlm_blocking_get_if<T> >
  {
   public:
    uvm_blocking_get_port();
    uvm_blocking_get_port( const std::string& name );
    virtual const std::string get_type_name() const;
    virtual void get( T& val );


  }; // class uvm_blocking_get_port


} // namespace uvm
```

14.3.2 Template parameter T

The template parameter T specifies the type of transaction to be received by the port.

14.3.3 Constructor

```
uvm_blocking_get_port();
uvm_blocking_get_port( const std::string& name );
```

The constructor shall create a new port with TLM-1 blocking get interface semantics. If specified, the argument *name* shall define the name of the port. Otherwise, the name of the port is implementation-defined.

14.3.4 Member functions

14.3.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_blocking_get_port**".

14.3.4.2 get

```
virtual void get( T& val );
```

The member function **get** shall retrieve a transaction of type T from the sender. It shall call the member function **get** of the associated interface which is bound to this port.

According to the TLM-1 blocking get semantics, the member function **get** shall not return until a transaction object has been delivered by the sender by means of its member function **put**. Subsequent calls to the member function **get** shall return a different transaction object. This actually means that a call to **get** shall consume the transaction from the sender.

14.4 uvm_blocking_peek_port

The class **uvm_blocking_peek_port** offers a convenience layer for UVM users to access the SystemC TLM-1 blocking interface **tlm**::**tlm_blocking_peek_if**. As this port class shall be derived from class **uvm_port_base**, it inherits the UVM specific member functions **connect**, **get_name**, **get_full_name** and **get_type_name**.

14.4.1 Class definition

```
namespace uvm {


  template <typename T>
  class uvm_blocking_peek_port : public uvm_port_base< tlm::tlm_blocking_peek_if<T> >
  {
   public:
    uvm_blocking_peek_port();
    uvm_blocking_peek_port( const std::string& name );
    virtual const std::string get_type_name() const;
    virtual void peek( T& val ) const;


  }; // class uvm_blocking_peek_port


} // namespace uvm
```

14.4.2 Template parameter T

The template parameter T specifies the type of transaction to be received by the port.

14.4.3 Constructor

```
uvm_blocking_peek_port();
uvm_blocking_peek_port( const std::string& name );
```

The constructor shall create a new port with TLM-1 blocking peek interface semantics. If specified, the argument *name* shall define the name of the port. Otherwise, the name of the port is implementation-defined.

14.4.4 Member functions

14.4.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_blocking_peek_port**".

14.4.4.2 peek

```
virtual void peek( T& val ) const;
```

The member function **peek** shall retrieve a transaction of type T from the sender. It shall call the member function **peek** of the associated interface which is bound to this port.

According to the TLM-1 blocking peek semantics, the member function **peek** shall not return until a transaction object has been delivered by the sender by means of its member function **put**. Subsequent calls to the member function **peek** shall return exactly the same transaction object. This actually means that a call to **peek** shall not consume the transaction from the sender. A transaction shall only be consumed by means of a call to **get**.

14.5 uvm_blocking_get_peek_port

The class **uvm_blocking_get_peek_port** offers a convenience layer for UVM users to access the SystemC TLM-1 blocking interface **tlm**::**tlm_blocking_get_peek_if**. As this port class shall be derived from class **uvm_port_base**, it inherits the UVM specific member functions **connect**, **get_name**, **get_full_name** and **get_type_name**.

14.5.1 Class definition

```
namespace uvm {


  template <typename T>
  class uvm_blocking_get_peek_port : public uvm_port_base< tlm::tlm_blocking_get_peek_if<T> >
  {
   public:
    uvm_blocking_get_peek_port();
    uvm_blocking_get_peek_port( const std::string& name );
    virtual const std::string get_type_name() const;
    virtual void get( T& val );
```

```
    virtual void peek( T& val ) const;

  }; // class uvm_blocking_get_peek_port

} // namespace uvm
```

14.5.2 Template parameter T

The template parameter T specifies the type of transaction to be received by the port.

14.5.3 Constructor

```
uvm_blocking_get_peek_port();
uvm_blocking_get_peek_port( const std::string& name );
```

The constructor shall create a new port with TLM-1 blocking get and peek interface semantics. If specified, the argument *name* shall define the name of the port. Otherwise, the name of the port is implementation-defined.

14.5.4 Member functions

14.5.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_blocking_get_peek_port**".

14.5.4.2 get

```
virtual void get( T& val );
```

The member function **get** shall retrieve a transaction of type T from the sender. It shall call the member function **get** of the associated interface which is bound to this port.

According to the TLM-1 blocking get semantics, the member function **get** shall not return until a transaction object has been delivered by the sender by means of its member function **put**. Subsequent calls to the member function **get** shall return a different transaction object. This actually means that a call to **get** shall consume the transaction from the sender.

14.5.4.3 peek

```
virtual void peek( T& val ) const;
```

The member function **peek** shall retrieve a transaction of type T from the sender. It shall call the member function **peek** of the associated interface which is bound to this port (see member function **connect**).

According to the TLM-1 blocking peek semantics, the member function **peek** shall not return until a transaction object has been delivered by the sender by means of its member function **put**. Subsequent calls to the member function **peek** shall return exactly the same transaction object. This actually means that a call to **peek** shall not consume the transaction from the sender. A transaction shall only be consumed by means of a call to **get**.

14.6 uvm_analysis_port

The class **uvm_analysis_port** offers a convenience layer for UVM users and is compatible with the SystemC **tlm**::**tlm_analysis_port**, since it shall be derived from this class. Primary reason to introduce this derived port class is to offer the member function **connect** as alternative to the SystemC **bind** and **operator**() to connect analysis ports with exports.

14.6.1 Class definition

```
namespace uvm {


  template <typename T>
  class uvm_analysis_port : public tlm::tlm_analysis_port<T>
  {
   public:
    uvm_analysis_port();
    uvm_analysis_port( const std::string& name );
    virtual const std::string get_type_name() const;
    virtual void connect( tlm::tlm_analysis_if<T>& _if );


  }; // class uvm_analysis_port


} // namespace uvm
```

14.6.2 Template parameter T

The template parameter T specifies the type of transaction to be communicated by the analysis port.

14.6.3 Constructor

```
uvm_analysis_port();
uvm_analysis_port( const std::string& name );
```

The constructor shall create a new analysis port. If specified, the argument *name* shall define the name of the port. Otherwise, the name of the port is implementation-defined.

NOTE–UVM-SystemC does not define, in contrast to UVM-SystemVerilog, the constructor arguments *min_size* and *max_size* to specify the minimum and maximum number of interfaces, respectively, that must have been connected to this port by the end of elaboration.

14.6.4 Member functions

14.6.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_analysis_port**".

14.6.4.2 connect

```
virtual void connect( tlm::tlm_analysis_if<T>& _if );
```

The member function **connect** shall register the subscriber passed as an argument, so that any call to the member function **write** of such analysis port instance shall be passed on to the registered subscriber. Multiple subscribers may be registered with a single analysis port instance.

NOTE 1–The member function **connect** implements the same functionality as the SystemC member function **bind**.

NOTE 2–There may be zero subscribers registered with any given analysis port instance, in which case calls to the member function **write** shall not be propagated.

14.7 uvm_analysis_export

The class **uvm_analysis_export** offers a convenience layer for UVM users and is compatible with the SystemC export type **sc_core**::**sc_export** < **tlm**::**tlm_analysis_if** <T> > since it shall be derived from this class. Primary reason to introduce this export class is to offer the member function **connect** as alternative to the SystemC **bind** and **operator**() to connect analysis ports with exports.

14.7.1 Class definition

```
namespace uvm {


  template <typename T>
  class uvm_analysis_export : public sc_core::sc_export< tlm::tlm_analysis_if<T> >
  {
   public:
    uvm_analysis_export();
    uvm_analysis_export( const std::string& name );
    virtual const std::string get_type_name() const;
    virtual void connect( tlm::tlm_analysis_if<T>& _if );


  }; // class uvm_analysis_export


} // namespace uvm
```

14.7.2 Template parameter T

The template parameter T specifies the type of transaction to be communicated by the analysis port.

14.7.3 Constructor

```
uvm_analysis_export();
```

```
uvm_analysis_export( const std::string& name );
```

The constructor shall create a new analysis export. If specified, the argument *name* shall define the name of the export. Otherwise, the name of the export is implementation-defined.

NOTE–UVM-SystemC does not define, in contrast to UVM-SystemVerilog, the constructor arguments *min_size* and *max_size* to specify the minimum and maximum number of interfaces, respectively, that must have been connected to this port by the end of elaboration.

14.7.4 Member functions

14.7.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_analysis_export**".

14.7.4.2 connect

```
virtual void connect( tlm::tlm_analysis_if<T>& _if );
```

The member function **connect** shall register the subscriber passed as an argument, so that any call to the member function **write** of such analysis export instance shall be passed on to the registered subscriber. Multiple subscribers may be registered with a single analysis export instance.

NOTE 1–The member function **connect** implements the same functionality as the SystemC member function **bind**.

NOTE 2–There may be zero subscribers registered with any given analysis export instance, in which case calls to the member function **write** shall not be propagated.

14.8 uvm_tlm_req_rsp_channel

The class **uvm_tlm_req_rsp_channel** offers a convenience layer for UVM users and is compatible with the SystemC **tlm**::**tlm_req_rsp_channel**, since it shall be derived from this class. It offers some UVM additional capabilities such as the analysis ports for request and response monitoring.

The class **uvm_tlm_req_rsp_channel** contains a request FIFO of default type **tlm**::**tlm_fifo**<REQ> and a response FIFO of default type **tlm**::**tlm_fifo**<RSP>. These FIFOs can be of any size. This channel is particularly useful for dealing with pipelined protocols where the request and response are not tightly coupled.

14.8.1 Class definition

```
namespace uvm {


  template < typename REQ,
             typename RSP = REQ,
             typename REQ_CHANNEL = tlm::tlm_fifo<REQ>,
             typename RSP_CHANNEL = tlm::tlm_fifo<RSP> >
    class uvm_tlm_req_rsp_channel
```

```
    : public tlm::tlm_req_rsp_channel<REQ, RSP, REQ_CHANNEL, RSP_CHANNEL>
  {
   public:

    // ports and exports
    uvm_analysis_port<REQ> request_ap;
    uvm_analysis_port<RSP> response_ap;
    sc_core::sc_export< tlm::tlm_fifo_put_if<REQ> > put_request_export;
    sc_core::sc_export< tlm::tlm_fifo_put_if<RSP> > put_response_export;
    sc_core::sc_export< tlm::tlm_fifo_get_if<REQ> > get_request_export;
    sc_core::sc_export< tlm::tlm_fifo_get_if<RSP> > get_response_export;
    sc_core::sc_export< tlm::tlm_fifo_get_if<REQ> > get_peek_request_export;
    sc_core::sc_export< tlm::tlm_fifo_get_if<RSP> > get_peek_response_export;
    sc_core::sc_export< tlm::tlm_master_if<REQ, RSP> > master_export;
    sc_core::sc_export< tlm::tlm_slave_if<REQ, RSP> >  slave_export;

    // constructors
    uvm_tlm_req_rsp_channel( int req_size = 1 , int rsp_size = 1 );
    uvm_tlm_req_rsp_channel( uvm_component_name name, int req_size = 1, int rsp_size = 1 );

  }; // class uvm_tlm_req_rsp_channel

} // namespace uvm
```

14.8.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. The template parameters REQ_CHANNEL and RSP_CHANNEL specify the type of the request and response FIFO, respectively. If parameters REQ_CHANNEL or RSP_CHANNEL are not specified, the interface will use FIFOs of type **tlm**::**tlm_fifo**.

14.8.3 Ports and exports

14.8.3.1 request_ap

```
uvm_analysis_port<REQ> request_ap;
```

The analysis port **request_ap** shall send the request transactions, which are passed via the member function **put** or **nb_put** (via any port connected to the export **put_request_export**), via its member function **write**, to all connected analysis exports and imps.

14.8.3.2 response_ap

```
uvm_analysis_port<RSP> response_ap;
```

The analysis port **response_ap** shall send the response transactions, which are passed via the member function **put** or **nb_put** (via any port connected to the export **put_response_export**), via its member function **write**, to all connected analysis exports and imps.

14.8.3.3 put_request_export

```
sc_core::sc_export< tlm::tlm_fifo_put_if<REQ> > put_request_export;
```

The export **put_request_export** shall provide both the blocking and non-blocking **put** interface member functions to the request FIFO based on interface **tlm**::**tlm_fifo_put_if**, being member functions **put**, **nb_put** and **nb_can_put**. Any put port variant can connect and send transactions to the request FIFO via this export, provided the transaction types match.

14.8.3.4 put_response_export

```
sc_core::sc_export< tlm::tlm_fifo_put_if<RSP> > put_response_export;
```

The export **put_response_export** shall provide both the blocking and non-blocking **put** interface member functions to the response FIFO based on interface **tlm**::**tlm_fifo_put_if**, being **put**, **nb_put** and **nb_can_put**. Any put port variant can connect and send transactions to the response FIFO via this export, provided the transaction types match.

14.8.3.5 get_request_export

```
sc_core::sc_export< tlm::tlm_fifo_get_if<REQ> > get_request_export;
```

The export **get_request_export** shall provide both the blocking and non-blocking **get** and **peek** interface member functions to the request FIFO based on interface **tlm**::**tlm_fifo_get_if**, being **get**, **nb_get**, **nb_can_get**, **peek**, **nb_peek** and **nb_can_peek**. Any put port variant can connect and send transactions to the request FIFO via this export, provided the transaction types match.

NOTE–This member function is functionally equivalent to **get_peek_request_export**.

14.8.3.6 get_response_export

```
sc_core::sc_export< tlm::tlm_fifo_get_if<RSP> > get_response_export;
```

The export **get_response_export** shall provide both the blocking and non-blocking **get** and **peek** interface member functions to the response FIFO based on interface **tlm**::**tlm_fifo_get_if**, being **get**, **nb_get**, **nb_can_get**, **peek**, **nb_peek** and **nb_can_peek**. Any put port variant can connect and send transactions to the response FIFO via this export, provided the transaction types match.

NOTE–This member function is functionally equivalent to **get_peek_response_export**.

14.8.3.7 get_peek_request_export

```
sc_core::sc_export< tlm::tlm_fifo_get_if<REQ> > get_peek_request_export;
```

The export **get_peek_request_export** shall provide both the blocking and non-blocking **get** and **peek** interface member functions to the request FIFO based on interface **tlm**::**tlm_fifo_get_if**, being **get**, **nb_get**, **nb_can_get**, **peek**, **nb_peek** and **nb_can_peek**. Any put port variant can connect and send transactions to the request FIFO via this export, provided the transaction types match.

NOTE–This member function is functionally equivalent to **get_request_export**.

14.8.3.8 get_peek_response_export

```
sc_core::sc_export< tlm::tlm_fifo_get_if<RSP> > get_peek_response_export;
```

The export **get_peek_response_export** shall provide both the blocking and non-blocking **get** and **peek** interface member functions to the response FIFO based on interface **tlm**::**tlm_fifo_get_if**, being **get**, **nb_get**, **nb_can_get**, **peek**, **nb_peek** and **nb_can_peek**. Any put port variant can connect and send transactions to the response FIFO via this export, provided the transaction types match.

NOTE–This member function is functionally equivalent to **get_response_export**.

14.8.3.9 master_export

```
sc_core::sc_export< tlm::tlm_master_if<REQ, RSP> > master_export;
```

The export **master_export** shall provide a single interface that allows a master to put requests and get or peek responses. It is a combination of the functionality offered by the exports **put_request_export** and **get_peek_response_export**.

14.8.3.10 slave_export

```
sc_core::sc_export< tlm::tlm_slave_if<REQ, RSP> > slave_export;
```

The export **slave_export** shall provide a single interface that allows a slave to get or peek requests and to put responses. It is a combination of the functionality offered by the exports **get_peek_request_export** and **put_response_export**.

14.8.4 Constructor

```
uvm_tlm_req_rsp_channel( int req_size = 1 , int rsp_size = 1 );
uvm_tlm_req_rsp_channel( uvm_component_name name, int req_size = 1, int rsp_size = 1 );
```

The constructor shall create a new TLM-1 interface containing a request and response FIFO. The argument *req_size* specifies the size of the request FIFO. The argument *rsp_size* specifies the size of the response FIFO. If not specified, default size of these FIFOs is 1. If specified, the argument *name* shall define the name of the interface. Otherwise, the name of the interface is implementation-defined.

15. Global defines, typedefs and enumerations

This section lists the global defines, types and enumerations used in UVM-SystemC.

15.1 Global defines

15.1.1 UVM_MAX_STREAMBITS

The definition **UVM_MAX_STREAMBITS** shall be used to set the maximum size for integer types. If not defined, a default size of 64 is used.

UVM-SystemVerilog defines a max size of 4096, but that is not supported by sc_int<T>??

15.1.2 UVM_DEFAULT_TIMEOUT

The definition **UVM_DEFAULT_TIMEOUT** shall be used as default timeout for the run phases. If not defined, a default timeout of 9200 seconds shall be used. The timeout can be overridden by using the member function **uvm_root**::**set_timeout** (see 4.3.2.3).

15.2 Typedefs

15.2.1 uvm_bitstream_t

The typedef **uvm_bitstream_t** shall define an integer type with a size defined by **UVM_MAX_STREAMBITS**. An application can use this type in member functions such as **uvm_printer**::**print_field** (see 5.2.2.1), **uvm_recorder**::**record_field** (see x.x.x), **uvm_packer**::**pack_field** (see 5.1.2.1) and **uvm_packer**::**unpack_field** (see 5.1.3.3).

15.2.2 uvm_integral_t

The typedef **uvm_bitstream_t** shall define an integer type with a size of 64 bits. An application can use this type in member functions such as **uvm_printer::print_field_int** (see 5.2.2.2), **uvm_recorder::record_field_int** (see x.x.x), **uvm_packer::pack_field_int** (see 5.1.2.2) and **uvm_packer::unpack_field_int** (see 5.1.3.2).

15.2.3 UVM_FILE

The typedef **uvm_file** shall define the file descriptor which supports output streams.

15.2.4 uvm_report_cb

The typedef **uvm_report_cb** is the alias for **uvm_callbacks**<**uvm_report_object**, **uvm_report_catcher**>.

15.2.5 uvm_config_int

The typedef **uvm_config_int** is the alias for **uvm_config_db**<**uvm_bitstream_t**>.

15.2.6 uvm_config_string

The typedef **uvm_config_string** is the alias for **uvm_config_db**<**std::string**>.

15.2.7 uvm_config_object

The typedef **uvm_config_object** is the alias for **uvm_config_db**<**uvm_object***>.

15.2.8 uvm_config_wrapper

The typedef **uvm_config_wrapper** is the alias for **uvm_config_db**<**uvm_object_wrapper***>.

15.3 Enumeration

15.3.1 uvm_action

The enumeration type **uvm_action** shall define all possible values for report actions. Each report is configured to execute one or more actions, determined by the bitwise OR of any or all of the following enumeration constants.

- **UVM_NO_ACTION**: No action is taken.
- **UVM_DISPLAY**: Sends the report to the standard output.
- **UVM_LOG**: Sends the report to the file(s) for this (severity, id) pair.
- **UVM_COUNT**: Counts the number of reports with the COUNT attribute. When this value reaches **max_quit_count**, the simulation terminates.
- **UVM_EXIT**: Terminates the simulation immediately.
- **UVM_CALL_HOOK**: Callback the report hook methods.
- **UVM_STOP**: Causes the simulator to stop, enabling continuation as interactive session.

15.3.2 uvm_severity

The enumeration type **uvm_severity** shall define all possible values for report severity:

- **UVM_INFO**: Informative message.
- **UVM_WARNING**: Indicates a potential problem.
- **UVM_ERROR**: Indicates a real problem. Simulation continues subject to the configured message action.
- **UVM_FATAL**: Indicates a problem from which simulation cannot recover. The simulation will be terminated immediately.

15.3.3 uvm_verbosity

The enumeration type **uvm_verbosity** shall define standard verbosity levels for reports.

- **UVM_NONE**: Report is always printed. Verbosity level setting cannot disable it.
- **UVM_LOW**: Report is issued if configured verbosity is set to **UVM_LOW** or above.
- **UVM_MEDIUM**: Report is issued if configured verbosity is set to **UVM_MEDIUM** or above.
- **UVM_HIGH**: Report is issued if configured verbosity is set to **UVM_HIGH** or above.
- **UVM_FULL**: Report is issued if configured verbosity is set to **UVM_FULL** or above.

15.3.4 uvm_active_passive_enum

The enumeration type **uvm_active_passive_enum** shall define whether a component, usually an agent, is in "active" mode or "passive" mode.

- **UVM_ACTIVE**: **uvm_agent** is in "active" mode, which means that the sequencer, driver and monitor are enabled.
- **UVM_PASSIVE**: **uvm_agent** is in "passive" mode, which means that only the monitor is enabled.

15.3.5 uvm_sequence_state_enum

The enumeration type **uvm_sequence_state_enum** shall define the current sequence state.

- **CREATED**: The sequence has been allocated.
- **PRE_START**: The sequence is started and the callback **uvm_sequence_base**::**pre_start** is being executed.
- **PRE_BODY**: The sequence is started and the callback **uvm_sequence_base**::**pre_body** is being executed.
- **BODY**: The sequence is started and the callback **uvm_sequence_base**::**body** is being executed.
- **ENDED**: The sequence has completed the execution of the callback **uvm_sequence_base**::**body**.
- **POST_BODY**: The sequence is started and the callback **uvm_sequence_base**::**post_body** is being executed.
- **POST_START**: The sequence is started and the callback **uvm_sequence_base**::**post_start** is being executed.
- **STOPPED**: The sequence has been forcibly ended by issuing a **uvm_sequence_base**::**kill** on the sequence.
- **FINISHED**: The sequence is completely finished executing.

15.3.6 uvm_phase_type

The typedef **uvm_phase_type** shall define an enumeration list which defines the phase type.

- **UVM_PHASE_IMP**: The phase object is used to traverse the component hierarchy and call the component phase method as well as the callbacks **phase_started** and **phase_ended**.
- **UVM_PHASE_NODE**: The object represents a simple node instance in the graph. These nodes will contain a reference to their corresponding IMP object.
- **UVM_PHASE_SCHEDULE**: The object represents a portion of the phasing graph, typically consisting of several NODE types, in series, parallel, or both.
- **UVM_PHASE_TERMINAL**: This internal object serves as the termination NODE for a SCHEDULE phase object.
- **UVM_PHASE_DOMAIN**: This object represents an entire graph segment that executes in parallel with the run phase. Domains may define any network of NODEs and SCHEDULEs. The built-in domain called *uvm* consists of a single schedule of all the run-time phases, starting with **pre_reset** and ending with **post_shutdown**.

16. Annex A: UVM-SystemVerilog features not included in UVM-SystemC

(Informative)

The following is a list of major UVM-SystemVerilog features not available in UVM-SystemC. However, future UVM-SystemC implementations may address these topics. Note that this is not an exhaustive list.

16.1 No field macros

UVM in SystemVerilog provides field macros to set up automation of fields inside a **uvm_object**. The automation of the fields means that the fields automatically get an implementation for all of the mandatory member functions of **uvm_object**. It is recommended not to use these field automation macros, because their implementation has impact on simulation performance and gives intransparent results for e.g. debugging. Therefore UVM-SystemC will not implement these field automation macros. As a consequence, an application needs to implement each of the mandatory member functions of **uvm_object**: **do_print**, **do_pack**, **do_unpack**, **do_copy**, and **do_compare**.

16.2 No automated configuration

UVM-SystemC does not define automated configuration through the member function **apply_config_settings**. All configuration needs to be explicitly retrieved using **get_config_int**, **get_config_string**, or **get_config_object**.

16.3 No transaction recording

UVM-SystemC does not define a transaction recording mechanism aligned with that of UVM-SystemVerilog. An application may use the existing transaction recording mechanism available in the SystemC Verification library (SCV) where appropriate.

16.4 No register abstraction layer

UVM-SystemC does not define any register layer classes to create registers and memories.

16.5 No constraint randomization and coverage classes

SystemVerilog offers the API for constraint randomization and coverage classes. Classes for constraint randomization of parameters and usage of coverage classes are not yet available in UVM-SystemC. Future extensions may integrate the SystemC Verification library (SCV) or CRAVE library to introduce randomization and a constraint solver.

16.6 No assertions

Although not part of the UVM, but native functionality in SystemVerilog are assertions. These elements are not part of the SystemC language and therefore not supported in the UVM-SystemC implementation.

17. Annex B: Deprecated Cadence UVM-SC functionality

(Informative)

This annex contains a list of deprecated features compared to the initial Cadence UVM-SC 1.0 release, June 2011.

17.1 Global functions

This section lists the global functions which have been deprecated.

17.1.1 Simulator stop and timeout functions

The global functions **uvm_set_stop_mode**, **uvm_set_global_timeout**, **uvm_set_global_stop_timeout**, and **uvm_stop_request** are not compatible with the UVM 1.1 standard and thus removed from the UVM namespace. In a future release of the UVM-SC standard, the functionality becomes available via the command line interface.

17.1.2 Print configurations

The global function **uvm_print_config_matches** is not compatible with the UVM 1.1 standard and thus removed from the UVM namespace. Instead, the member function **print_config_matches** which is part of the class **uvm_component** shall be used.

17.2 Factory

The following macros and functions which could be used in combination with the **uvm_factory** have been deprecated.

17.2.1 Registration macros

The factory registration macros **UVM_*_REGISTER*** for objects derived from **uvm_object** and components derived from **uvm_component** are not compatible with the UVM standard. Instead, registration of these elements shall be done via the macro **UVM_OBJECT_UTILS** or **UVM_COMPONENT_UTILS**.

For template classes with one or more arguments, the macros **UVM_OBJECT_PARAM_UTILS** and **UVM_COMPONENT_PARAM_UTILS** can be used.

17.2.2 Global create functions

The global convenience functions **uvm_create_object** and **uvm_create_component** are not compatible with the UVM standard. Instead, the standardized UVM factory static member functions **type_id::create** shall be used for the creation and instantiation of objects which are registered in the UVM factory. Refer to Section 6.3.2 for a description of these standardized **uvm_factory** member functions.

17.2.3 Global type and instance override functions

The global convenience functions **uvm_set_type_override** and **uvm_set_inst_override** are not compatible with the UVM standard. Instead, the member functions, which are part of the class **uvm_component** shall be used.

18. Annex C: Renamed functions UVM-SystemC versus UVM-SystemVerilog

(Informative)

Classes or member functions marked with symbol **°** are renamed in UVM-SystemC compared to the UVM 1.1 standard implemented in SystemVerilog, due to the incompatibility in case of reserved keywords in C/C++ or an inappropriate name in the context of SystemC base class or member function definitions. Table C.1 below shows the renamed classes and member functions and also the reference to the original UVM 1.1 name is given.

Class name in UVM-SystemC	**Class name in UVM-SystemVerilog**	**Member function in UVM-SystemC**	**Method in UVM-SystemVerilog**	**Section**
uvm_process_phase	**uvm_task_phase**			2.11
uvm_factory		**do_register**	**register**	6.4.2.1
uvm_queue		**do_delete**	**delete**	10.9.3.7
uvm_pool		**do_delete**	**delete**	10.10.3.7
uvm_phase		**exec_process**	**exec_task**	11.1.4.2
uvm_callbacks		**do_delete**	**delete**	11.9.5.3
uvm_report_catcher		**do_catch**	**catch**	12.4.6.1

19. Annex D: Terminology

(Informative)

19.1 Definitions

agent: An abstract container used to emulate and verify DUT devices; agents encapsulate a driver, sequencer, and monitor.

application: A C++ program, written by an end user.

blocking: An interface where tasks block execution until they complete. See also: non blocking.

callback: A member function overridden within a class in the component hierarchy that is called back by the kernel at certain fixed points during elaboration and simulation. UVM defines pre-defined callback functions as part of the phasing mechanism, such as **end_of_elaboration_phase**, **build_phase**, **connect_phase**, **run_phase**, etc. In addition, UVM supports the creation of user-defined callback classes and functions.

child:: An instance that is within a given component. Component A is a child of component B if component A is within component B. See also: parent.

component: A piece of VIP that provides functionality and interfaces. Also referred to as a transactor.

configuration: Ability to change the properties of components or objects independent from the component hierarchy and composition. Configuration parameters can be stored in and retrieved from a central database, which can be accessed at any place in the verification environment, and at any time during the simulation.

consumer: A verification component that receives transactions from another component.

driver: A component responsible for executing or otherwise processing transactions, usually interacting with the device under test (DUT) to do so.

environment: The container object that defines the testbench topology.

export: A transaction level modeling (TLM) interface that provides the implementation of methods used for communication. Used in UVM to connect to a port.

factory method: A classic software design pattern used to create generic code by deferring, until run time, the exact specification of the object to be created.

fifo: An instance of a primitive channel that models a first-in-first-out buffer.

foreign methodology: A verification methodology that is different from the methodology being used for the majority of the verification environment.

generator: A verification component that provides transactions to another component. Also referred to as a producer.

implementation: A specific concrete implementation of the UVM-SystemC class library as defined in this standard. It only implements the public shell which need be exposed to the application (for example, parts may be precompiled and distributed as object code by a tool vendor). See also: kernel.

interface: A class derived, directly or indirectly, from class **sc_core**::**sc_interface**. An interface proper is an interface, and in the object-oriented sense a channel is also an interface. However, a channel is not an interface proper.

interface proper: An abstract class derived, directly or indirectly, from class **sc_core**::**sc_interface** but not derived from class **sc_core**::**sc_object**. An interface proper declares the set of methods to be implemented within a channel and to be called through a port. An interface proper contains pure virtual function declarations, but typically it contains no function definitions and no data members.

kernel: The core of any UVM-SystemC implementation including the underlying elaboration and simulation engines. The kernel honors the semantics defined by this standard but may also contain implementation-specific functionality outside the scope of this standard. See also: implementation.

master: This term has no precise technical definition in this standard, but it is used to mean a module or port that can take control of a memory-mapped bus in order to initiate bus traffic, or a component that can execute an autonomous software thread and thus initiate other system activity. Generally, a bus master would be an initiator.

member function: A function declared within a class definition, excluding friend functions. Outside of a constructor or member function of the class or of any derived class, a non-static member function can only be accessed using the dot . and arrow -> operators. See also: method.

method : A function that implements the behavior of a class. This term is synonymous with the C++ term member function. In UVM-SystemC, the term method is used in the context of an interface method call. Throughout this standard, the term member function is used when defining C++ classes (for conformance to the C++ standard), and the term method is used in more informal contexts and when discussing interface method calls.

monitor: A passive entity that samples DUT signals, but does not drive them.

non blocking: A call that returns immediately. See also: blocking.

parent:: The inverse relationship to child. Component A is the parent of component B if component B is a child of component A.

parent sequence: A sequence which contains one or more child sequences.

port: A TLM interface that defines the set of methods used for communication. Used in UVM to connect to an export.

primary (host) methodology: The methodology that manages the top-level operation of the verification environment and with which the user/integrator is presumably more familiar.

process: A process instance belongs to an implementation-defined class derived from class **uvm_object**. Each process instance has an associated function that represents the behavior of the process. A process may be a static or a dynamic (e.g., spawned) process. See also: spawned process.

recipient: The component that implements a callback or function that receives and processes a transaction. See also: sender.

request: A transaction that provides information to initiate the processing of a particular operation.

response: A transaction that provides information about the completion or status of a particular operation.

root sequence: A sequence which has no parent sequence.

scoreboard: The mechanism used to dynamically predict the response of the design and check the observed response against the predicted response. Usually refers to the entire dynamic response-checking structure.

sender: The component that implements a callback or function that initiates the transmission of a transaction. See also: recipient.

sequence: A UVM object that procedurally defines a set of transactions to be executed and/or controls the execution of other sequences.

sequencer: An advanced stimulus generator which executes sequences that define the transactions provided to the driver for execution.

spawned process: A process instance that is dynamically created by calling the SystemC function **sc_core**::**sc_spawn**. See also: process.

test: Specific customization of an environment to exercise required functionality of the DUT.

testbench: The structural definition of a set of verification components used to verify a DUT. Also referred to as a verification environment.

transaction: A class instance that encapsulates information used to communicate between two or more components.

transactor: See component.

verification environment: See environment.

virtual sequence: A conceptual term for a sequence that controls the execution of sequences on other sequencers.

19.2 Acronyms and Abbreviations

AMS	analog mixed signal
API	application programming interface
CDV	coverage-driven verification
CBCL	common base class library
CLI	command line interface
DUT	device under test
DUV	device under verification
EDA	electronic design automation
FIFO	first-in, first-out
HDL	hardware description language
HVL	high-level verification language
IP	intellectual property
OSCI	Open SystemC Initiative
SC	SystemC
SCV	SystemC Verification library
SV	SystemVerilog
TLM	transaction level modeling
UVC	UVM Verification Component

UVM	Universal Verification Methodology
VIP	verification intellectual property

20. Index

~

~uvm_component_name, destructor, 37

A

abstract, data member
  class uvm_comparer, 55
  class uvm_packer, 44
action configuration
  class uvm_report_object, 183
add, member function
  class uvm_callbacks, 174
  class uvm_phase, 157
  class uvm_pool, 151
add_by_name, member function
  class uvm_callbacks, 174
adjust_name, member function
  class uvm_printer, 50
agent. definition, 227
all_dropped, member function
  class uvm_component, 84
  class uvm_objection, 168
analysis_export, export
  class tuvm_subscriber, 96
application, definition, 227

B

big_endian, data member
  class uvm_packer, 44
blocking, definition, 227
body, member function
  class uvm_sequence_base, 118
build_phase, member function
  class uvm_component, 78

C

callback hooks
  class uvm_objection, 168
callback macros, 205
callback, definition, 227
callback_mode, member function
  class uvm_callback, 170
callbacks
  class uvm_phase, 156
check_phase, member function
  class uvm_component, 81
check_type, data member
  class uvm_comparer, 55
child, definition, 227
clear, member function
  class uvm_objection, 166
clear_response_queue, member function
  class uvm_sequence_base, 123
clone, member function
  class uvm_object, 26
compare, member function
  class uvm_object, 28
compare_field, member function
  class uvm_comparer, 52
compare_field_int, member function
  class uvm_comparer, 52
compare_field_real, member function
  class uvm_comparer, 53
compare_object, member function
  class uvm_comparer, 53
compare_string, member function
  class uvm_comparer, 53
comparing
  class uvm_object, 28
component and object registration, 200
component, definition, 227
compose_message, member function
  class uvm_report_server, 192
configuration
  class uvm_object, 31
configuration, definition, 227
configuration_phase, member function
  class uvm_component, 80
connect, member function
  class uvm_analysis_export, 215
  class uvm_analysis_port, 214
  class uvm_port_base, 36
connect_phase, member function
  class uvm_component, 78
construction
  class uvm_phase, 155
construction interface
  class uvm_component, 74

constructor
class uvm_analysis_export, 215
class uvm_analysis_port, 214
class uvm_blocking_get_peek_port, 212
class uvm_blocking_get_port, 209
class uvm_blocking_peek_port, 211
class uvm_blocking_put_port, 208
class uvm_report_catcher, 195
class uvm_report_handler, 187
class uvm_report_server, 189
class uvm_tlm_req_rsp_channel, 219
constructors
class uvm_report_object, 180
consumer, definition, 227
convert2string, member function
class uvm_object, 27
copy, member function
class uvm_object, 28
copying
class uvm_object, 28
create, member function
class uvm_component_registry, 62
class uvm_object, 26
class uvm_object_registry, 59
create_component, member function
class uvm_component, 85
class uvm_component_registry, 61
class uvm_object_wrapper, 58
create_component_by_name, member function
class uvm_factory, 67
create_component_by_type, member function
class uvm_factory, 67
create_item, member function
class uvm_sequence_base, 121
create_object, member function
class uvm_component, 85
class uvm_object_registry, 59
class uvm_object_wrapper, 58
create_object_by_name, member function
class uvm_factory, 67
create_object_by_type, member function
class uvm_factory, 66
creation
class uvm_factory, 66
class uvm_object, 26
current_grabber, member function
class uvm_sequencer_base, 101

D

debug
class uvm_factory, 68
class uvm_resource_pool, 143
debug_create_by_name, member function
class uvm_factory, 68
debug_create_by_type, member function
class uvm_factory, 68
define_domain, member function
class uvm_component, 83
delete_by_name, member function
class uvm_callbacks, 175
die, member function
class uvm_root, 32
display, member function
class uvm_callbacks, 176
display_objections, member function
class uvm_objection, 169
do_catch, member function
class uvm_report_catcher, 197
do_compare, member function
class uvm_object, 28
do_copy, member function
class uvm_object, 28
do_delete, member function
class uvm_callbacks, 174
class uvm_pool, 151
class uvm_queue, 149
do_kill, member function
class uvm_sequence_base, 120
do_pack, member function
class uvm_object, 29
do_print, member function
class uvm_object, 27
class uvm_resource_base, 137
do_record, member function
class uvm_object, 27
do_register, member function
class uvm_factory, 65
do_unpack, member function
class uvm_object, 30
driver, definition, 227
drop_objection, member function
class uvm_objection, 167
class uvm_phase, 158
dropped, member function

class uvm_component, 84
class uvm_objection, 168
dump, member function
class uvm_resource_pool, 143
dump_server_state, member function
class uvm_report_server, 193

E

emit, member function
class uvm_printer, 49
enable_print_topology, data member
class uvm_root, 33
end_of_elaboration_phase, member function
class uvm_component, 78
environment, definition, 227
exec_func, member function
class uvm_phase, 156
exec_process, member function
class uvm_phase, 156
exec_task. *See* exec_process
execute, member function
class uvm_bottomup_phase, 162
class uvm_process_phase, 164
class uvm_topdown_phase, 163
exists, member function
class uvm_pool, 151
uvm_config_db, 128
export, definition, 227
extract_phase, member function
class uvm_component, 81

F

factory
class uvm_component, 85
factory method, definition, 227
fifo, definition, 227
file configuration
class uvm_report_object, 184
final_phase, member function
class uvm_component, 82
find, member function
class uvm_phase, 156
class uvm_root, 33
find_all, member function
class uvm_root, 33
find_by_name, member function
class uvm_phase, 156
find_override_by_name, member function
class uvm_factory, 69
find_override_by_type, member function
class uvm_factory, 68
find_unused_resources, member function
class uvm_resource_pool, 143
finish_item, member function
class uvm_sequence_base, 121
finish_on_completion, data member
class uvm_root, 33
first, member function
class uvm_callback_iter, 172
class uvm_pool, 151
foreign methodology, definition, 227
format_action, member function
class uvm_report_handler, 188
format_footer, member function
class uvm_printer, 50
format_header, member function
class uvm_printer, 49
format_row, member function
class uvm_printer, 49

G

generator, definition, 227
get, member function
class uvm_blocking_get_peek_port, 212
class uvm_blocking_get_port, 210
class uvm_component_registry, 62
class uvm_object_registry, 59
class uvm_pool, 151
class uvm_queue, 148
class uvm_resource_pool, 139
class uvm_sequencer, 105
class uvm_sqr_if_base, 108
uvm_config_db, 128
get_action, member function
class uvm_report_catcher, 196
class uvm_report_handler, 187
get_by_name, member function
class uvm_resource, 145
class uvm_resource_db, 130
class uvm_resource_pool, 141
get_by_type, member function
class uvm_resource, 146

class uvm_resource_db, 130
class uvm_resource_pool, 142
get_cb, member function
class uvm_callback_iter, 172
get_child, member function
class uvm_component, 75
get_children, member function
class uvm_component, 75
get_client, member function
class uvm_report_catcher, 195
get_common_domain, member function
class uvm_domain, 160
get_current_item, member function
class uvm_sequencer_param_base, 103
uvm_sequence, 125
get_depth, member function
class uvm_component, 76
class uvm_sequence_item, 114
get_domain, member function
class uvm_component, 83
class uvm_phase, 158
get_domain_name, member function
class uvm_phase, 158
get_domains, member function
class uvm_domain, 160
get_drain_time, member function
class uvm_objection, 169
get_file_handle, member function
class uvm_report_handler, 187
get_first, member function
class uvm_callbacks, 175
get_first_child, member function
class uvm_component, 75
get_fname, member function
class uvm_report_catcher, 196
get_full_name, member function
class uvm_component, 75
class uvm_object, 25
class uvm_phase, 157
class uvm_port_base, 35
get_global, member function
class uvm_pool, 151
class uvm_queue, 148
get_global_pool, member function
class uvm_pool, 151
get_global_queue, member function
class uvm_queue, 148
get_highest_precedence, member function
class uvm_resource, 146
class uvm_resource_pool, 141
get_id, member function
class uvm_report_catcher, 196
get_id_count, member function
class uvm_report_server, 191
get_imp, member function
class uvm_phase, 158
get_inst_count, member function
class uvm_object, 25
get_inst_id, member function
class uvm_object, 25
get_is_active, member function
class uvm_agent, 92
get_jump_target, member function
class uvm_phase, 159
get_last, member function
class uvm_callbacks, 175
get_line, member function
class uvm_report_catcher, 196
get_max_quit_count, member function
class uvm_report_server, 190
get_message, member function
class uvm_report_catcher, 196
get_name, member function
class uvm_object, 24
class uvm_port_base, 35
get_next, member function
class uvm_callbacks, 175
get_next_child, member function
class uvm_component, 75
get_next_item, member function
class uvm_sequencer, 105
class uvm_sqr_if_base, 107
get_num_children, member function
class uvm_component, 75
get_object_type, member function
class uvm_object, 25
get_objection, member function
class uvm_phase, 158
get_objection_count, member function
class uvm_objection, 169
get_objection_total, member function
class uvm_objection, 169
get_objectors, member function
class uvm_objection, 169

get_packet_size, member function
class uvm_packer, 43
get_parent, member function
class uvm_component, 75
class uvm_phase, 157
class uvm_port_base, 35
get_parent_sequence, member function
class uvm_sequence_item, 113
get_peek_request_export, export
class uvm_tlm_req_rsp_channel, 218
get_peek_response_export, export
class uvm_tlm_req_rsp_channel, 218
get_phase_type, member function
class uvm_phase, 155
get_prev, member function
class uvm_callbacks, 176
get_priority, member function
class uvm_sequence_base, 119
get_quit_count, member function
class uvm_report_server, 190
get_report_action, member function
class uvm_report_object, 183
get_report_catcher, member function
class uvm_report_catcher, 197
get_report_file_handle, member function
class uvm_report_object, 184
get_report_handler, member function
class uvm_report_object, 185
get_report_verbosity_level, member function
class uvm_report_object, 182
get_request_export, export
class uvm_tlm_req_rsp_channel, 218
get_response, member function
uvm_sequence, 125
get_response_export, export
class uvm_tlm_req_rsp_channel, 218
get_response_queue_depth, member function
class uvm_sequence_base, 123
get_response_queue_error_report_disabled, member function
class uvm_sequence_base, 123
get_root_sequence, member function
class uvm_sequence_item, 114
get_root_sequence_name, member function
class uvm_sequence_item, 114
get_run_count, member function
class uvm_phase, 155
get_schedule, member function
class uvm_phase, 157
get_schedule_name, member function
class uvm_phase, 157
get_scope, member function
class uvm_resource_base, 136
get_sequence_path, member function
class uvm_sequence_item, 114
get_sequence_state, member function
class uvm_sequence_base, 117
get_sequencer, member function
class uvm_sequence_item, 113
get_server, member function
class uvm_report_server, 190
get_severity, member function
class uvm_report_catcher, 195
get_severity_count, member function
class uvm_report_server, 191
get_state, member function
class uvm_phase, 155
get_transaction_id, member function
class uvm_transaction, 111
get_type, member function
class uvm_object, 25
class uvm_resource, 145
get_type_handle, member function
class uvm_resource, 145
class uvm_resource_base, 136
get_type_name, member function
class uvm_agent, 92
class uvm_analysis_export, 215
class uvm_analysis_port, 214
class uvm_blocking_get_peek_port, 212
class uvm_blocking_get_port, 210
class uvm_blocking_peek_port, 211
class uvm_blocking_put_port, 208
class uvm_callback, 171
class uvm_component_registry, 62
class uvm_driver, 91
class uvm_env, 93
class uvm_monitor, 91
class uvm_object, 26
class uvm_object_registry, 59
class uvm_object_wrapper, 58
class uvm_port_base, 35
class uvm_scoreboard, 95
class uvm_subscriber, 96

class uvm_test, 94
get_use_response_handler, member function
class uvm_sequence_base, 122
get_use_sequence_info, member function
class uvm_sequence_item, 113
get_uvm_domain, member function
class uvm_domain, 161
get_uvm_phases, member function
class uvm_domain, 161
get_uvm_schedule, member function
class uvm_domain, 160
get_verbosity, member function
class uvm_report_catcher, 195
get_verbosity_level, member function
class uvm_report_handler, 187
grab, member function
class uvm_sequence_base, 119
class uvm_sequencer_base, 100

H

has_child, member function
class uvm_component, 76
has_do_available, member function
class uvm_sequencer, 106
class uvm_sequencer_base, 101
has_lock, member function
class uvm_sequence_base, 120
class uvm_sequencer_base, 100
hierarchical reporting interface
class uvm_component, 87
hierarchy interface
class uvm_component, 74
host methodology, definition, 228

I

identification
class uvm_object, 24
implementation, definition, 228
incr_id_count, member function
class uvm_report_server, 191
incr_quit_count, member function
class uvm_report_server, 190
incr_severity_count, member function
class uvm_report_server, 191
init_access_record, member function
class uvm_resource_base, 138
insert, member function
class uvm_queue, 149
interface proper, definition, 228
interface, definition, 228
is, member function
class uvm_phase, 156
is_after, member function
class uvm_phase, 156
is_auditing, member function
class uvm_resource_options, 134
is_before, member function
class uvm_phase, 156
is_blocked, member function
class uvm_sequence_base, 120
class uvm_sequencer_base, 99
is_child, member function
class uvm_sequencer_base, 98
is_enabled, member function
class uvm_callback, 170
is_grabbed, member function
class uvm_sequencer_base, 101
is_item, member function
class uvm_sequence_item, 114
is_null, member function
class uvm_packer, 42
is_quit_count_reached, member function
class uvm_report_server, 191
is_read_only, member function
class uvm_resource_base, 136
is_relevant, member function
class uvm_sequence_base, 119
is_tracing, member function
class uvm_resource_db_options, 133
issue, member function
class uvm_report_catcher, 199
item_done, member function
class uvm_sequencer, 105
class uvm_sqr_if_base, 108

J

jump, member function
class uvm_phase, 159
jumping
class uvm_phase, 159

K

kernel, definition, 228
kill, member function
  class uvm_sequence_base, 120
knobs, data member
  class uvm_printer, 50

L

last, member function
  class uvm_callback_iter, 172
  class uvm_pool, 152
lock, member function
  class uvm_sequence_base, 119
  class uvm_sequencer_base, 100
lookup
  class uvm_resource_pool, 141
lookup, member function
  class uvm_component, 76
lookup_name, member function
  class uvm_resource_pool, 141
lookup_regex, member function
  class uvm_resource_pool, 142
lookup_regex_names, member function
  class uvm_resource_pool, 142
lookup_scope, member function
  class uvm_resource_pool, 142
lookup_type, member function
  class uvm_resource_pool, 141

M

macros
  class uvm_component, 89
  class uvm_object, 31
main_phase, member function
  class uvm_component, 80
master, definition, 228
master_export, export
  class uvm_tlm_req_rsp_channel, 218
match_scope, member function
  class uvm_resource_base, 137
member function, definition, 228
method, definition, 228
mid_do, member function
  class uvm_sequence_base, 118
miscompares, data member
  class uvm_comparer, 54
monitor, definition, 228

N

next, member function
  class uvm_callback_iter, 172
  class uvm_pool, 152
non blocking, definition, 228
notification
  class uvm_resource_base, 136
num, member function
  class uvm_pool, 151

O

objection control
  class uvm_objection, 166
objection interface
  class uvm_component, 84
objection status
  class uvm_objection, 169
operator const char*(), operator
  class uvm_component_name, 37
override configuration
  class uvm_report_object, 185

P

pack, member function
  class uvm_object, 28
pack_bytes, member function
  class uvm_object, 29
pack_field, member function
  class uvm_packer, 41
pack_field_int, member function
  class uvm_packer, 41
pack_ints, member function
  class uvm_object, 29
pack_object, member function
  class uvm_packer, 41
pack_real, member function
  class uvm_packer, 41
pack_string, member function
  class uvm_packer, 41
pack_time, member function
  class uvm_packer, 41
packing

class uvm_object, 28
parent sequence, definition, 228
parent, definition, 228
peek, member function
class uvm_blocking_get_peek_port, 213
class uvm_blocking_peek_port, 211
class uvm_sequencer, 105
class uvm_sqr_if_base, 109
phase_ended, member function
class uvm_component, 82
phase_ready_to_end, member function
class uvm_component, 82
phase_started, member function
class uvm_component, 82
phasing interface
class uvm_component, 76
physical, data member
class uvm_comparer, 55
class uvm_packer, 43
policy, data member
class uvm_comparer, 54
pop_back, member function
class uvm_queue, 149
pop_front, member function
class uvm_queue, 149
port, definition, 228
post_body, member function
class uvm_sequence_base, 118
post_configuration_phase, member function
class uvm_component, 80
post_do, member function
class uvm_sequence_base, 118
post_main_phase, member function
class uvm_component, 80
post_reset_phase, member function
class uvm_component, 79
post_shutdown_phase, member function
class uvm_component, 81
post_start, member function
class uvm_sequence_base, 118
post-run phases
class uvm_component, 77
pre_abort, member function
class uvm_component, 89
pre_body, member function
class uvm_sequence_base, 117
pre_configuration_phase, member function
class uvm_component, 79
pre_do, member function
class uvm_sequence_base, 118
pre_main_phase, member function
class uvm_component, 80
pre_reset_phase, member function
class uvm_component, 79
pre_shutdown_phase, member function
class uvm_component, 81
pre_start, member function
class uvm_sequence_base, 117
pre-run phases
class uvm_component, 76
prev, member function
class uvm_callback_iter, 172
class uvm_pool, 152
primary methodology, definition, 228
print, member function
class uvm_factory, 69
class uvm_object, 26
print_accessors, member function
class uvm_resource_base, 137
print_array_footer, member function
class uvm_printer, 50
print_array_header, member function
class uvm_printer, 50
print_array_range, member function
class uvm_printer, 50
print_catcher, member function
class uvm_report_catcher, 197
print_double, member function
class uvm_printer, 48
print_field, member function
class uvm_printer, 47
print_field_int, member function
class uvm_printer, 47
print_generic, member function
class uvm_printer, 49
print_msg, member function
class uvm_comparer, 54
print_object, member function
class uvm_printer, 48
print_object_header, member function
class uvm_printer, 48
print_override_info, member function
class uvm_component, 87
print_real, member function

class uvm_printer, 47
print_resources, member function
class uvm_resource_pool, 143
print_string, member function
class uvm_printer, 48
print_time, member function
class uvm_printer, 49
print_topology, member function
class uvm_root, 33
printing
class uvm_object, 26
priority
class uvm_resource, 146
class uvm_resource_base, 137
process control interface
class uvm_component, 83
process, definition, 228
process_report, member function
class uvm_report_server, 192
push_back, member function
class uvm_queue, 149
push_front, member function
class uvm_queue, 149
put, member function
class uvm_blocking_put_port, 209
class uvm_sequencer, 105
class uvm_sqr_if_base, 109
put_request_export, export
class uvm_tlm_req_rsp_channel, 217
put_response_export, export
class uvm_tlm_req_rsp_channel, 217

R

raise_objection, member function
class uvm_objection, 166
class uvm_phase, 158
raised, member function
class uvm_component, 84
class uvm_objection, 168
read only interface
class uvm_resource_base, 136
read, member function
class uvm_resource, 146
read/write interface
class uvm_resource, 146
read_by_name, member function
class uvm_resource_db, 131
read_by_type, member function
class uvm_resource_db, 131
recipient, definition, 228
record, member function
class uvm_object, 27
record_read_access, member function
class uvm_resource_base, 137
record_write_access, member function
class uvm_resource_base, 137
recording
class uvm_object, 27
recording interface
class uvm_component, 89
register. *See* do_register
registering types
class uvm_factory, 65
report handler configuration
class uvm_report_object, 185
report, member function
class uvm_report_handler, 187
report_phase, member function
class uvm_component, 82
report_summarize, member function
class uvm_report_object, 192
reporting
class uvm_report_object, 180
reporting macros, 201
request, definition, 229
request_ap, port
class uvm_tlm_req_rsp_channel, 217
reset_phase, member function
class uvm_component, 79
reset_quit_count, member function
class uvm_report_server, 191
reset_report_handler, member function
class uvm_report_object, 185
reset_severity_counts, member function
class uvm_report_server, 191
resolve_bindings, member function
class uvm_component, 83
resource database interface
class uvm_resource_base, 136
response interface
class uvm_sequence_base, 122
response, definition, 229
response_ap, port

class uvm_tlm_req_rsp_channel, 217
response_handler, member function
class uvm_sequence_base, 122
result, data member
class uvm_comparer, 55
resume, member function
class uvm_component, 84
root sequence, definition, 229
run_phase, member function
class uvm_component, 78
run_test, global function, 21
run_test, member function
class uvm_root, 32
run-time phases
class uvm_component, 77

S

schedule
class uvm_phase, 157
scope interface
class uvm_resource_base, 136
scoreboard, definition, 229
seeding
class uvm_object, 24
send_request, member function
class uvm_sequencer_base, 102
class uvm_sequencer_param_base, 103
uvm_sequence, 125
sender, definition, 229
seq_item_export, data member
class uvm_sequencer, 105
seq_item_port, data member
class uvm_driver, 90
sequence control
class uvm_sequence_base, 119
sequence execution
class uvm_sequence_base, 117
sequence execution macros, 202
sequence item execution
class uvm_sequence_base, 121
sequence, definition, 229
sequencer, definition, 229
set
class uvm_resource_pool, 140
set priority
class uvm_resource_pool, 142
set, member function
class uvm_resource, 145
class uvm_resource_db, 131
class uvm_resource_pool, 140
uvm_config_db, 127
set/get interface
class uvm_resource, 145
set_action, member function
class uvm_report_catcher, 197
set_anonymous, member function
class uvm_resource_db, 131
set_arbitration, member function
class uvm_sequencer_base, 101
set_default, member function
class uvm_resource_db, 130
set_depth, member function
class uvm_sequence_item, 113
set_domain, member function
class uvm_component, 82
set_drain_time, member function
class uvm_objection, 168
set_id, member function
class uvm_report_catcher, 197
set_id_count, member function
class uvm_report_server, 191
set_id_info, member function
class uvm_sequence_item, 113
set_inst_override, member function
class uvm_component, 86
class uvm_component_registry, 62
class uvm_object_registry, 60
set_inst_override_by_name, member function
class uvm_factory, 65
set_inst_override_by_type, member function
class uvm_component, 86
class uvm_factory, 65
set_max_quit_count, member function
class uvm_report_server, 190
set_message, member function
class uvm_report_catcher, 197
set_name, member function
class uvm_object, 24
set_name_override, member function
class uvm_resource_pool, 140
set_override, member function
class uvm_resource, 145
class uvm_resource_pool, 140

set_parent_sequence, member function
class uvm_sequence_item, 113
set_phase_imp, member function
class uvm_component, 83
set_priority, member function
class uvm_resource, 146
class uvm_resource_base, 137
class uvm_resource_pool, 143
class uvm_sequence_base, 119
set_priority_name, member function
class uvm_resource_pool, 143
set_priority_type, member function
class uvm_resource_pool, 142
set_quit_count, member function
class uvm_report_server, 190
set_read_only, member function
class uvm_resource_base, 136
set_report_default_file, member function
class uvm_report_object, 184
set_report_default_file_hier, member function
class uvm_component, 88
set_report_handler, member function
class uvm_report_object, 185
set_report_id_action, member function
class uvm_report_object, 183
set_report_id_action_hier, member function
class uvm_component, 87
set_report_id_file, member function
class uvm_report_object, 184
set_report_id_file_hier, member function
class uvm_component, 88
set_report_id_verbosity, member function
class uvm_report_object, 182
set_report_id_verbosity_hier, member function
class uvm_component, 87
set_report_severity_action, member function
class uvm_report_object, 183
set_report_severity_action_hier, member function
class uvm_component, 87
set_report_severity_file, member function
class uvm_report_object, 184
set_report_severity_file_hier, member function
class uvm_component, 88
set_report_severity_id_action, member function
class uvm_report_object, 183
set_report_severity_id_action_hier, member function
class uvm_component, 88
set_report_severity_id_file, member function
class uvm_report_object, 184
set_report_severity_id_file_hier, member function
class uvm_component, 88
set_report_severity_id_override, member function
class uvm_report_object, 185
set_report_severity_id_verbosity, member function
class uvm_report_object, 183
set_report_severity_id_verbosity_hier, member function
class uvm_component, 87
set_report_severity_override, member function
class uvm_report_object, 185
set_report_verbosity_level, member function
class uvm_report_object, 182
set_report_verbosity_level_hier, member function
class uvm_component, 89
set_request, member function
class uvm_sequence_base, 122
set_response_queue_depth, member function
class uvm_sequence_base, 123
set_response_queue_error_report_disabled, member function
class uvm_sequence_base, 123
set_scope, member function
class uvm_resource_base, 136
set_sequencer, member function
class uvm_sequence_item, 113
set_server, member function
class uvm_report_server, 190
set_severity, member function
class uvm_report_catcher, 196
set_severity_count, member function
class uvm_report_server, 191
set_timeout, member function
class uvm_root, 32
set_transaction_id, member function
class uvm_transaction, 111
set_type_override, member function
class uvm_component, 86
class uvm_component_registry, 62
class uvm_object_registry, 60
class uvm_resource_pool, 140
set_type_override_by_name, member function
class uvm_factory, 66
set_type_override_by_type, member function
class uvm_component, 85

class uvm_factory, 66
set_use_sequence_info, member function
class uvm_sequence_item, 112
set_verbosity, member function
class uvm_report_catcher, 197
sev, data member
class uvm_comparer, 54
show_max, data member
class uvm_comparer, 54
shutdown_phase, member function
class uvm_component, 81
size, member function
class uvm_queue, 149
slave_export, export
class uvm_tlm_req_rsp_channel, 219
sort_by_precedence, member function
class uvm_resource_pool, 141
spawned process, definition, 229
spell_check, member function
class uvm_resource_pool, 140
sprint, member function
class uvm_object, 26
start, member function
class uvm_sequence_base, 117
start_item, member function
class uvm_sequence_base, 121
start_of_simulation_phase, member function
class uvm_component, 78
start_phase_sequence, member function
class uvm_sequencer_base, 99
starting_phase, member function
class uvm_sequence_base, 124
state
class uvm_phase, 155
stop_sequences, member function
class uvm_sequencer, 106
class uvm_sequencer_base, 100
summarize_report_catcher, member function
class uvm_report_catcher, 199
suspend, member function
class uvm_component, 83
sync, member function
class uvm_phase, 158
synchronization
class uvm_phase, 158

T

template parameter CB
class uvm_callback_iter, 171
class uvm_callbacks, 174
template parameter IF
class uvm_ port_base, 35
template parameter T
class uvm_analysis_export, 215
class uvm_analysis_port, 213
class uvm_blocking_get_peek_port, 212
class uvm_blocking_get_port, 209
class uvm_blocking_peek_port, 211
class uvm_blocking_put_port, 208
class uvm_callback_iter, 171
class uvm_callbacks, 174
class uvm_component_registry, 61
class uvm_config_db, 127
class uvm_object_registry, 59
class uvm_queue, 148
class uvm_resource, 144
class uvm_resource_db, 130
template parameters
class uvm_driver, 90
class uvm_pool, 150
class uvm_sequence, 124
class uvm_sequencer, 104
class uvm_sequencer_param_base, 103
class uvm_sqr_if_base, 107
class uvm_tlm_req_rsp_channel, 217
test, definition, 229
testbench, definition, 229
trace_mode, member function
class uvm_objection, 166
transaction, definition, 229
transactor. *See* component
traverse, member function
class uvm_bottomup_phase, 162
class uvm_process_phase, 164
class uvm_topdown_phase, 163
try_next_item, member function
class uvm_sequencer, 105
class uvm_sqr_if_base, 107
turn_off_auditing, member function
class uvm_resource_options, 134
turn_off_tracing, member function
class uvm_resource_db_options, 133

turn_on_auditing, member function
class uvm_resource_options, 134
turn_on_tracing, member function
class uvm_resource_db_options, 133
type and instance overrides types
class uvm_factory, 65
type interface
class uvm_resource, 145

U

ungrab, member function
class uvm_sequence_base, 120
class uvm_sequencer_base, 100
unlock, member function
class uvm_sequence_base, 120
class uvm_sequencer_base, 100
unpack, member function
class uvm_object, 29
unpack_bytes, member function
class uvm_object, 30
unpack_field, member function
class uvm_packer, 42
unpack_field_int, member function
class uvm_packer, 42
unpack_ints, member function
class uvm_object, 30
unpack_object, member function
class uvm_packer, 43
unpack_real, member function
class uvm_packer, 42
unpack_string, member function
class uvm_packer, 42
unpack_time, member function
class uvm_packer, 42
unpacking
class uvm_object, 29
unsync, member function
class uvm_phase, 159
use_metadata, data member
class uvm_packer, 44
use_response_handler, member function
class uvm_sequence_base, 122
user_priority_arbitration, member function
class uvm_sequencer_base, 98
utility functions
class uvm_resource_base, 137
uvm_action, enum, 221
uvm_active_passive_enum, enum, 222
uvm_agent
class, 92
class definition, 92
constructor, 92
uvm_analysis_export
class, 214
class definition, 214
uvm_analysis_port
class, 213
class definition, 213
uvm_bitstream_t, typedef, 220
uvm_blocking_get_peek_port
class, 211
uvm_blocking_get_port
class, 209
class definition, 209
uvm_blocking_peek_port
class, 210
class definition, 210, 211
uvm_blocking_put_port
class, 208
class definition, 208
uvm_bottomup_phase
class, 161
class definition, 161
constructor, 161
overview, 17
uvm_callback
class, 170
class definition, 170
constructor, 170
overview, 18
uvm_callback_iter
class, 171
class definition, 171
constructor, 171
overview, 18
uvm_callbacks
class, 172
class definition, 173
constructor, 174
overview, 18
uvm_comparer
class, 51
class definition, 51

overview, 15
uvm_component
class, 70
class definition, 70
constructor, 74
uvm_component_name
class, 36
class definition, 36
constructor, 37
overview, 15
UVM_COMPONENT_PARAM_UTILS, macro, 201
uvm_component_registry
class, 60
class definition, 60
overview, 16
UVM_COMPONENT_UTILS, macro, 201
uvm_config_db
class, 126
class definition, 126
constraints on usage, 127
overview, 17
uvm_config_int, typedef, 220
uvm_config_object, typedef, 221
uvm_config_string, typedef, 221
uvm_config_wrapper, typedef, 221
UVM_CREATE, macro, 204
UVM_CREATE_ON, macro, 205
UVM_DECLARE_P_SEQUENCER
macro, 106
UVM_DECLARE_P_SEQUENCER, macro, 205
uvm_default_comparer, default policy object, 56
uvm_default_line_printer, default policy object, 56
uvm_default_packer, default policy object, 56
uvm_default_printer, default policy object, 56
uvm_default_recorder, default policy object, 56
uvm_default_table_printer, default policy object, 55
UVM_DEFAULT_TIMEOUT, global define, 220
uvm_default_tree_printer, default policy object, 55
UVM_DO, macro, 203
UVM_DO_CALLBACKS, macro, 205
UVM_DO_ON, macro, 204
UVM_DO_ON_PRI, macro, 204
UVM_DO_ON_PRI_WITH, macro, 204
UVM_DO_ON_WITH, macro, 204
UVM_DO_PRI, macro, 203
UVM_DO_PRI_WITH, macro, 203
UVM_DO_WITH, macro, 203
uvm_domain
class, 159
class definition, 160
constructor, 160
overview, 17
uvm_driver
class, 89
class definition, 90
constructor, 90
uvm_env
class, 93
class definition, 93
constructor, 93
UVM_ERROR, macro, 202
uvm_factory
class, 63
class definition, 63
overview, 16
UVM_FATAL, macro, 202
UVM_INFO, macro, 201
uvm_integral_t, typedef, 220
UVM_MAX_STREAMBITS, global define, 220
uvm_monitor
class, 91
class definition, 91
constructor, 91
uvm_object
class, 22
class definition, 22
constructor, 24
overview, 15
UVM_OBJECT_PARAM_UTILS, macro, 200
uvm_object_registry
class, 58
class definition, 58
overview, 16
UVM_OBJECT_UTILS, macro, 200
uvm_object_wrapper
class, 57
class definition, 57
overview, 16
uvm_objection
class, 164
class definition, 164
constructor, 166
overview, 17
uvm_packer

class, 38
class definition, 38
overview, 15
uvm_phase
class, 153
class definition, 153
constructor, 155
overview, 17
uvm_phase_type, enum, 222
uvm_pool
constructor, 150
uvm_pool
class, 149
class definition, 150
uvm_port_base
class, 34
class definition, 34
constructor, 35
overview, 15
uvm_printer
class, 45
class definition, 45
overview, 15
uvm_process_phase
class, 163
class definition, 163
overview, 17
uvm_queue
class, 147
class definition, 147
constructor, 148
UVM_REGISTER_CB, macro, 205
uvm_report_catcher
class, 193
class definition, 193
overview, 18
uvm_report_cb, typedef, 220
uvm_report_enabled, member function
class uvm_report_object, 181
uvm_report_error, member function
class uvm_report_catcher, 198
class uvm_report_object, 182
uvm_report_fatal, member function
class uvm_report_catcher, 198
class uvm_report_object, 182
uvm_report_handler
class, 185
class definition, 186
overview, 18
uvm_report_info, member function
class uvm_report_catcher, 199
class uvm_report_object, 181
uvm_report_object
class, 177
class definition, 178
overview, 18
uvm_report_server
class, 188
class definition, 188
overview, 18
uvm_report_warning, member function
class uvm_report_catcher, 198
class uvm_report_object, 181
uvm_resource
class, 143
class definition, 143
overview, 17
uvm_resource_base
class, 134
class definition, 134
constructor, 135
overview, 17
uvm_resource_db
class, 128
class definition, 128
overview, 17
uvm_resource_db_options
class, 132
class definition, 132
uvm_resource_options
class, 133
class definition, 133
overview, 17
uvm_resource_pool
class, 138
class definition, 138
overview, 17
uvm_resource_types
class, 146
class definition, 147
uvm_root
class, 31
class definition, 31
overview, 15

uvm_scoreboard
  class, 94
  class definition, 94
  constructor, 95
uvm_sequence
  class, 124
  class definition, 124
  constructor, 125
  overview, 17
uvm_sequence_base
  class, 114
  class definition, 114
  constructor, 116
  overview, 17
uvm_sequence_item
  class, 111
  class definition, 111
  constructor, 112
  overview, 17
uvm_sequence_state_enum, enum, 222
uvm_sequencer
  class, 103
  class definition, 103
  constructor, 104
  macros, 106
  overview, 16
uvm_sequencer_base
  class, 97
  class definition, 97
  constructor, 98
  overview, 16
uvm_sequencer_param_base
  class, 102
  class definition, 102
  constructor, 103
  overview, 16
uvm_set_config_int, global function, 21
uvm_set_config_string, global function, 21
uvm_severity, enum, 221
uvm_srq_if_base
  class, 106
  class definition, 106
  overview, 16
uvm_subscriber
  class, 95
  class definition, 95
  constructor, 96
uvm_task_phase. *See* uvm_process_phase
uvm_test
  class, 93
  class definition, 93
  constructor, 94
uvm_tlm_req_rsp_channel
  class, 216
  class definition, 216
uvm_top, data member
  class uvm_root, 34
uvm_topdown_phase
  class, 162
  class definition, 162
  constructor, 163
  overview, 17
uvm_transaction
  class, 110
  class definition, 110
  constructor, 111
  overview, 17
uvm_verbosity, enum, 221
uvm_void
  class, 22
  class definition, 22
  overview, 15
UVM_WARNING, macro, 202

V

verbosity configuration
  class uvm_report_object, 182
verbosity, data member
  class uvm_comparer, 54
verification environment. *See* environment
virtual sequence, definition, 229

W

wait_for, member function
  class uvm_objection, 169
wait_for_grant, member function
  class uvm_sequence_base, 121
  class uvm_sequencer_base, 99
wait_for_item_done, member function
  class uvm_sequence_base, 122
  class uvm_sequencer_base, 99
wait_for_sequences, member function
  class uvm_sequencer, 106

class uvm_sequencer_base, 101
wait_for_state, member function
class uvm_phase, 159
wait_modified, member function
uvm_config_db, 128
wait_modified, member function
class uvm_resource_base, 136
write, member function
class uvm_resource, 146
write_by_name, member function
class uvm_resource_db, 132
write_by_type, member function
class uvm_resource_db, 132