25. Messages

25.1 Overview

The messaging feature is a centralized and flexible mechanism used to send messages to various destinations, such as log files, display, waveforms or databases. It lets a developer easily insert formatted messages into code and provides the user with powerful and flexible controls to selectively enable or disable groups of messages.

The three most typical uses for messages are the following:

a) Summaries—Writing summary information at the beginning or end of significant chunks of activity.
b) Tracing—Writing detailed trace messages during the simulation upon interesting events.
c) Debugging—Writing detailed debug messages during the run to help the user or developer debug unexplained behaviors.

Messages are different from plain **out()** and **outf()** calls (see 29.7); they have an optional standard-format prefix and their actions can be disabled or redirected. Messages are also different from **dut_error()** calls (see 17.2.2); they do not signify failure, increment error counters, or increment warning counters.

25.2 Message model

There are two kinds of message actions in the ***e*** language: structured debug messages (SDM) and regular messages.

Structured debug messages have a standard pre-defined structure, arguments, and attributes. Each kind of SDM has a specific purpose and is used for reporting specific kind of events, such as the beginning or the end of a transaction spanning over time. Some kinds of SDMs sample data objects given as arguments which can further be used for data flow analysis (e.g. using a waveform viewer). The sampling of the arguments of SDM actions is related to transaction recording.

Regular messages are used to report “general” events. They do not have specific structure, but define a text string to be printed or recorded.

Upon execution, the message action (SDM or regular) creates a message and sends it to the *context unit* for further handling. Each unit can be configured to filter messages in various ways, format the enabled messages in various ways (adding the time, name of the unit, etc.), and send them to various destinations (such as files and the screen).

25.3 Message execution

When a message action (SDM or regular) is executed, the following happens:

— For SDM messages, if an action-block exists, it gets executed. This typically contains assignments to message instance optional parameters. The specific parameters differ between the various SDM kinds (see 25.4).
— The message body is created as follows:

  — For a SDM, the message body consists of the body text, appended to a predefined prefix that contains information on the SDM kind and id. The body text is determined by the **body_text** parameter assigned in the action block, if any. If no **body_text** is assigned, the default text specific to each SDM kind is used.

— For regular messages, the message body is created by appending expressions, similarly to **out()** or **outf()**.

Then, if an *action-block* exists, it gets executed. It typically contains further output-producing actions, calls to reporting methods, etc. The output of all of those is added, as a list of string, to the message body.

For **message()**, the message body is created by appending all of the expressions, like **out()** does.

For **messagef()**, the message body is created using the *format-exp*, similar to **outf()**.

— **messagef()** does not automatically add a newline (`\n`) to the message string. Therefore, if the optional *action-block* requires a newline to be written before it is executed, terminate the *format-exp* using `\n`.

— If the fully composed message string – including that portion written by the optional *action-block* – is not terminated by a newline, a newline is appended. **messagef()** also allows appending of the *action-block* output to the **messagef()** header output.

The context unit of the message is the unit instance in the context of which the message action is being executed. If the message action resides in the context of a unit type, that is the context unit. If it resides in the context of a struct type, the context unit is the parent unit of the struct instance (see 7.5.1).

According to the current message selection settings of the context unit, and according to the message tag, the list of destinations to which the message has to be sent is determined. If there are no destinations, the processing of the message ends here.

The message is sent for each destination as follows:

— For a text destination, the message is formatted by calling **create_formatted_message()**. The body text created above is passed to the buffer parameter of that method. The current message format settings of the context unit are used. If there are no extensions of **create_formatted_message()**, the default formatting is used according to the format settings of the context unit.

The resulting message text is sent to the destination accordingly.

— For a non-text destination, such as a wave form or a database, the message is sent or handled according to the nature of that destination. This behavior is implementation-dependent, and various tools may handle it differently. For example, a wave form can display matching pairs of **msg_started** and **msg_ended** messages (with the same message id and data item) as a transaction.

Some messages may not be handled at all by some destinations. For example, regular (non-SDM) messages may not be handled by a transaction database.

Message code shall not modify the flow of the simulation in any way. Time-consuming operations in message headers or action blocks are strictly disallowed.

25.4 Structured debug messages

The ***e*** language provides the following methods for defining structured debug message actions:

— **msg_started()** — Marks the beginning of a sequence of events that can be logged as a transaction.

— **msg_ended()** — Marks the end of a sequence of events that can be logged as a transaction.

— **msg_transformed()** — Marks a data transformation. It can be used to link transactions that have been indicated by structured debug messages.

— **msg_changed()** — Marks a change to an object, such as a state variable changing from transmit to idle. This message action marks a one-time event. It is logged as a transaction with a single attribute, the result of the state expression.

— **msg_info()** — Reports any other kind of significant event in the environment. Events reported by **msg_info()** cannot be logged as transactions.

SDM actions have the generic format described in Table 39. The specifics for each of the SDMs are described in separate/dedicated sections.

Table 39—SDM Action Generic Format

<table>
<tr><td>Purpose</td><td colspan="2">Reports the start of a transaction</td></tr>
<tr><td>Syntax</td><td colspan="2">msg_<<i>sdm specifier</i>>([<i>tag</i>,]<i>verbosity, msg-id</i>, <<i>sdm-specific-arguments</i>>) [{<i>action-block</i>}].
Note: <i>sdm specifier</i> can be any of the following: started (see 25.4.1), ended (see 25.4.2), transformed (see 25.4.3), changed (see 25.4.4), info (see 25.4.5). It determine two aspects:
— List of arguments
— Message instance parameters that can be accessed within the action block scope</td></tr>
<tr><td rowspan="5">Parameters</td><td>tag</td><td>A constant of type <code>message_tag</code>, either NORMAL or a user-defined tag (see 25.6). If no tag is specified, NORMAL is assumed by default.</td></tr>
<tr><td>verbosity</td><td>A constant of type <code>message_verbosity</code>, one of the following: NONE, LOW, MEDIUM, HIGH, or FULL (see 25.7).</td></tr>
<tr><td>msg_id</td><td>Message ID. A string expression that identifies the specific occurrence reported by the message (i.e., message ID uniquely identifies a transaction stream).
When a literal string is provided (as opposed to a string expression that is computed at runtime), the text can be used for static message filtering.</td></tr>
<tr><td>SDM specific arguments</td><td>These are determined by each SDM action and they are described in the sections 25.4.1 through 25.4.5, for each of the SDM actions.
Usually these are objects that will be sampled for later analysis purposes.</td></tr>
<tr><td>action_block</td><td>A list of zero or more actions separated by semicolons and enclosed in curly braces. Syntax: {action;...}
The action block may be or not executed depending on the configuration and this aspect is implementation dependent.
In the scope of this action-block, the pseudo-variable it refers to an object of type sdm_handler (see 25.8) and specifically to its concrete subclass according to the SDM kind (for example, sdm_started_handler in case of msg_started(), and so on).
The primary use of this action-block is to initialize configurable message instance parameters (described below) to be stored in the fields of it.
Example:
<pre>msg_started(HIGH," monitoring transfer",cur_trans) {
    it.parent = cur_burst;
};</pre>The following sdm_handler fields that can be set in the action block are common to all SDM actions; the ones specific to each SDM kind (fields of subclasses of sdm_handler) are described in the specific sections:
<table>
<tr><td>scope</td><td>Identifies the unit context where the action occurs. This can be used, for example, to hide the actual unit and use an enclosing unit as the message scope.
If scope is not assigned, the default is used. If the message action resides in the context of a unit type, that is the context unit. If it resides in the context of a struct type, the context unit is the parent unit of the struct instance (see 7.5.1).</td></tr>
<tr><td>body_text</td><td>Defines a text string to be displayed with the message. Used to override the message's default text, which depends on the specific SDM kind.</td></tr>
</table></td></tr>
</table>

The developer can configure transaction recording process and specify what to sample from a transaction object and when to sample it. For more information see 25.9.2.1, 25.9.2.2 and 25.9.2.3.

25.4.1 msg_started()

Purpose	Reports the start of a transaction		
Category	Action		
Syntax	**msg_started(**[*tag*,]*verbosity*, *msg-id*, *data-item***)** [{*action-block*}]		
Parameters	*tag, verbosity, msg_id*	See Table 39, “SDM Action Generic Format”.	
	data_item	Struct that contains the data that is being processed	
	action_block	See the *action-block* description in Table 39, “SDM Action Generic Format”. For **msg_started()**, the following fields of **sdm_started_handler** can also be set, besides the ones presented in Table 39:	
		parent	Identifies the higher-level (parent) transaction containing the current transaction. If specified, the struct assigned to the **.parent** field becomes the parent transaction, and the data item of the current transaction becomes the child transaction. This can be useful, for example, for showing to which burst a set of packets belongs. (Transactions are usually used to model “packets,” and bursts are the children of “transfers”; thus, the parent attribute for each packet points to the “burst” message, and parent of the burst points to a transfer.) If both matching **msg_started** and **msg_ended** actions assign a parent, which is not the same, the behavior is undefined.
		body_text	The default contains a hyperlink to the transaction data item

Syntax example:

```
on burst_started {
      msg_started(LOW,"monitoring burst", driven_burst);
};
```

25.4.2 msg_ended()

<table>
<tr><td>Purpose</td><td colspan="2">Reports the end of a transaction</td></tr>
<tr><td>Category</td><td colspan="2">Action</td></tr>
<tr><td>Syntax</td><td colspan="2">msg_ended([tag,]verbosity, msg-id, data-item) [{action-block}]</td></tr>
<tr><td rowspan="5">Parameters</td><td>tag, verbosity, msg_id</td><td>See Table 39, “SDM Action Generic Format”.</td></tr>
<tr><td>data_item</td><td>Struct that contains the data that is being processed</td></tr>
<tr><td>action_block</td><td>See the action-block description in Table 39, “SDM Action Generic Format”. For msg_ended(), the following fields of sdm_started_handler can also be set, besides the ones presented in Table 39:</td></tr>
<tr><td></td><td>parent</td><td>Identifies the higher-level (parent) transaction containing the current transaction.
If specified, the struct assigned to the .parent field becomes the parent transaction, and the data item of the current transaction becomes the child transaction.
This can be useful, for example, for showing to which burst a set of packets belongs. (Transactions are usually used to model “packets,” and bursts are the children of “transfers”; thus, the parent attribute for each packet points to the “burst” message, and parent of the burst points to a transfer.)
If both matching msg_started and msg_ended actions assign a parent, which is not the same, the behavior is undefined.</td></tr>
<tr><td></td><td>start_time</td><td>The time at which this transaction started (by default UNDEF, which indicates that the starting of the transaction was already reported by a msg_started() action). Must be a value of type time. If assigned, no corresponding msg_started action is considered to indicate the transaction start.</td></tr>
<tr><td></td><td>body_text</td><td>The default contains a hyperlink to the transaction data item</td></tr>
</table>

It reports the end of each transaction that you want to track. Unless the sample points are specified with the recording configuration API, data is sampled as follows:

— When a **msg_ended()** action has a corresponding **msg_started()** action, data is sampled at both the beginning and end of the transaction.

— When a **msg_ended()** action has no corresponding **msg_started()** action, the start time can be specified in the body of the **msg_ended()** action. In this case, data is sampled at the end of the transaction.

— When a **msg_ended()** action has no corresponding **msg_started()** action, and no start time is set in the action body, a 0-time transaction is created, and data is sampled at the end of the transaction.

Syntax example:

```
on burst_ended {
      msg_ended(LOW,"monitoring burst", driven_burst);
      burst_ended_o$.write(driven_burst);
```

```
};
```

25.4.3 msg_transformed()

Purpose	Reports the transformation of an existing data item or items, or the outcome of a relationship between data items	
Category	Action	
Syntax	**msg_transformed(**[*tag*,]*verbosity*, *msg-id*, *from-item*, *to-item***)** [{*action-block*}]	
Parameters	*tag*, *verbosity*, *msg_id*	See Table 39, “SDM Action Generic Format”.
	from_item	Struct that contains the data that is being processed.
	to_item	Struct that contains the data after transformation.
	action_block	See the *action-block* description in Table 39, “SDM Action Generic Format”. For **msg_transformed()**, the following field of **sdm_started_handler** can also be set, besides the ones presented in Table 39:
		body_text — The default contains a hyperlink to both transaction data items.

Syntax example:

```
match_write(m:burst) is {
    if not exp_items.is_empty() {
        var exp_i:=exp_items.pop0();
        msg_transformed(MEDIUM, "Matching bursts", exp_i, m);
    };
};
```

25.4.4 msg_changed()

Purpose	Reports a significant state change taking place in this scope	
Category	Action	
Syntax	**msg_changed(**[*tag*,]*verbosity*, *msg-id*, *new_state_exp***)** [{*action-block*}]	
Parameters	*tag*, *verbosity*, *msg_id*	See Table 39, “SDM Action Generic Format”.
	new_state_exp	Text string describing the new state this unit assumes.
	action_block	See the *action-block* description in Table 39, “SDM Action Generic Format”. For **msg_changed()**, the following field of **sdm_started_handler** can also be set, besides the ones presented in Table 39:
		body_text — The default contains the new state string.

Syntax example:

```
drive_burst(new_burst:burst)@clk is {
    driven_burst = new_burst;
    emit burst_started;
    msg_changed(MY_TAG, HIGH, "burst state", "started");
};
```

25.4.5 msg_info()

Purpose	Reports a significant event in the environment, that occurs at a certain point in time, possibly related to the provided data items or items, and which is not applicable to the other kinds of structured debug messages.	
Category	Action	
Syntax	**msg_info(**[*tag*,]*verbosity*, *msg-id*, [*data-item1*, [*data-item2*]]**)** [{*action-block*}]	
Parameters	*tag*, *verbosity*, *msg_id*	See Table 39, “SDM Action Generic Format”.
	data_item1, *data_item2*	References to data items involved in the reported event. Up to two data items can be specified (both are optional).
	to_item	Struct that contains the data after transformation.
	action_block	See the *action-block* description in Table 39, “SDM Action Generic Format”. For **msg_info()**, the following field of **sdm_started_handler** can also be set, besides the ones presented in Table 39:
		body_text — The default contains a hyperlink to the data items, if any.

Syntax example:

```
body()@driver.clock is only {
    do s1;
    wait [10]; driv-er.drop_objection(TEST_DONE);
    msg_info(LOW,"end of test ", s2);
};
```

25.5 message and messagef

Purpose	Create a text message and send it to the one or more destinations
Category	Action
Syntax	**message (**[*tag*]**,** *verbosity***,** *exp***,** ...**)** [*action_block*] **messagef (**[*tag*]**,** *verbosity***,** *format_exp***,** [*exp***,** ...]**)** [*action_block*]

Parameters	*tag*	A constant of type `message_tag`, either **NORMAL** or a user-defined tag (see 25.6). If no tag is specified, **NORMAL** is assumed by default.
	verbosity	A constant of type `message_verbosity`: one of **NONE**, **LOW**, **MEDIUM**, **HIGH**, or **FULL** (see 25.7).
	exp	Value(s) to write.
	action_block	A block of actions to perform, the output of which is appended to the message body. **Note:** If the action block has any side effects, other than text output, the behavior is undefined and implementation dependent. Depending on whether or not there are text destinations to which the message is being sent, and whether there are more than one such destinations, the action block may not be actually executed (if there are no text destinations), or it may be executed more than once.
	format_exp	For **messagef()**, an **outf()**-style format string for the output.

Syntax examples:

```
message(HIGH, "Master ", me, " has received ", the_packet) {
    write the_packet
};
    -- Output this message and write the packet, at verbosity HIGH.

message(VR_XBUS_FILE, MEDIUM, "Packet ", num, " sent: ", data)
    -- Output this message at verbosity MEDIUM.
    -- Use VR_XBUS_FILE as the message-tag.
```

25.6 Tag

All kinds of messages have an optional first parameter of type `message_tag`, which is initially defined as:

```
type message_tag : [NORMAL]
```

This can be extended, e.g.,

```
extend message_tag : [VR_XBUS_PACKET]
```

If a *tag* is not specified [i.e., the first parameter of a message is a legal value for verbosity], then the value `NORMAL` is prepended. Thus, the following two lines are the same:

```
message(        MEDIUM, "Packet done: ", packet);
message(NORMAL, MEDIUM, "Packet done: ", packet)
```

Message tags are used for associating specific message actions with a class of actions or an aspect. This gives you more flexibility when it comes to determine the behavior of the message actions.

25.7 Verbosity

The *verbosity* parameter can be set to **NONE**, **LOW**, **MEDIUM**, **HIGH**, or **FULL** (from lowest to highest). Since a lower verbosity setting means fewer messages are shown, important messages should be assigned a lower *verbosity* parameter value.

Table 46 shows the recommended usage of verbosity. Each level can assume that all lower levels are also writing (thus, there is no need to repeat them).

Table 46—Verbosity Levels

Level	Recommended use	Examples
NONE	Critical messages.	`"WARNING: Running in reduced mode"`
LOW	Messages that happen once per run or once per reset.	`"Master M3 was instantiated"` `"Device D6 got out of reset"`
MEDIUM	Short messages that happen once per data item or sequence.	`"Packet-@36 was sent to port 7"` `"A write request to pci bus 2 with address=0xf2223, data=0x48883"`
HIGH	More detailed per-data-item information, including: — Actual value of the packet — Sub-transaction details	`"Full details for packet-@36: len=5 kind=small ..."`
FULL	Anything else, including writing by using specific methods (just to follow the algorithm of that method).	

25.8 Predefined type sdm_handler

The predefined type **sdm_handler** is used to represent the specific properties of a given SDM at run time. It provides API that can be used to query specific information about a given SDM, in the context of an extension of **create_formatted_message()** hook method (see 25.9.2.14.1). In addition, some of its fields can be assigned in the context of an SDM optional *action-block* (Table 39).

sdm_handler has several subclasses, used to represent the different SDM kinds.

25.8.1 sdm_handler

This struct provides API fields and methods common to all SDM kinds.

— **sdm_handler.scope**: any_unit

Holds a reference to the unit instance to which the message belongs. This can be assigned in the SDM *action-block*. If not assigned, it holds a reference to the context unit, or to the owing unit of the context struct.

— **sdm_handler.id_str**: string

Holds the *msg-id* string, as specified in the SDM action.

— **sdm_handler.body_text**: string

Holds a text string to be displayed with the message. This can be assigned in the SDM *action-block*. If not assigned, it holds an empty string.

— **sdm_handler.get_kind_string()**: string

Returns a string that represents the SDM kind: “started” for **msg_started**, “changed” for **msg_changed**, and so on.

— **sdm_handler.get_attribute_string(***inst:* any_struct**)**: string

Returns the string that displays the registered text attributes for the specified data object *inst*, or an empty string if there are no text attributes. Text attributes are those for which **set_text_attribute()** or (in case of **msg_changed()**) **set_text_state_var()** was called after registering them (see 25.9.2.1.13 and 25.9.2.1.14).

— **sdm_handler.collect_text_attributes(***inst:* any_struct**,** *names:* list of string**,** *values:* list of string**)**

Collects the names and printed string values of the registered text attributes for the specified data object *inst*, and adds them to the two provided lists, *names* and *values* respectively. Both lists are cleaned before collecting attributes, and any items present in them prior to calling this method are removed. Text attributes are those for which **set_text_attribute()** or (in case of **msg_changed()**) **set_text_state_var()** was called after registering them (see 25.9.2.1.13 and 25.9.2.1.14).

25.8.2 sdm_started_handler

This struct *like*-inherits from **sdm_handler** (see 25.8.1). It provides API fields specific to **msg_started()** (see 25.4.1).

— **sdm_started_handler.data_item**: any_struct

Holds a reference to the *data-item* struct, as specified in the **msg_started()** action.

— **sdm_started_handler.parent**: any_struct

Holds a reference to a struct that represents the parent transaction of the current transaction. This can be assigned in the *action-block* of the **msg_started**. If not assigned, it holds NULL.

25.8.3 sdm_ended_handler

This struct *like*-inherits from **sdm_handler** (see 25.8.1). It provides API fields specific to **msg_ended()** (see 25.4.2).

— **sdm_ended_handler.data_item**: any_struct

Holds a reference to the *data-item* struct, as specified in the **msg_ended()** action.

— **sdm_ended_handler.parent**: any_struct

Holds a reference to a struct that represents the parent transaction of the current transaction. This can be assigned in the *action-block* of the **msg_ended**. If not assigned, it holds NULL.

— **sdm_ended_handler.start_time**: time

Holds the time at which this transaction started. This can be assigned in the *action-block* of the **msg_ended**. If not assigned, it holds UNDEF.

25.8.4 sdm_transformed_handler

This struct *like*-inherits from **sdm_handler** (see 25.8.1). It provides API fields specific to **msg_transformed()** (see 25.4.3).

— **sdm_transformed_handler.from_item**: any_struct

Holds a reference to the *from-item* struct, as specified in the **msg_transformed()** action.

— **sdm_transformed_handler.to_item**: any_struct

Holds a reference to the *to-item* struct, as specified in the **msg_transformed()** action.

25.8.5 sdm_changed_handler

This struct *like*-inherits from **sdm_handler** (see 25.8.1). It provides API fields specific to **msg_changed()** (see 25.4.4).

— **sdm_changed_handler.new_state**: string

Holds the string value of *new-state-exp*, as specified in the **msg_changed()** action.

25.8.6 sdm_info_handler

This struct *like*-inherits from **sdm_handler** (see 25.8.1). It provides API fields specific to **msg_info()** (see 25.4.5).

— **sdm_info_handler.item1**: any_struct

Holds a reference to the *data-item-1* struct, as specified in the **msg_info()** action. If none specified, holds NULL.

— **sdm_info_handler.item2**: any_struct

Holds a reference to the *data-item-2* struct, as specified in the **msg_info()** action. If none specified, holds NULL.

25.9 Messages Interface

25.9.1 Message configuration and customization

25.9.1.1 Initial pre-supplied Default Message Settings

Every message is assigned a verbosity level and a tag. By default, every unit instance is configured to send to the screen those messages that have both a NORMAL tag and a verbosity that is at or below the LOW level. Furthermore, by default:

— no destinations other than screen are initially set.

— no handling of messages with a tag other than NORMAL is defined.

— the default format used for all messages is **short**.

25.9.1.2 Modifying Initial Default Message Settings

To modify the defaults from these initial settings, the **set_..._messages()** and/or **set_message_format()** methods (see 25.9.2.5 and 25.9.2.5.7) are used. Normally these methods are used within extensions of **post_generate()**.

Example

```
extend my_env {
    post_generate() is also {
        message_manager.set_screen_messages(me, NORMAL, MEDIUM);
        message_manager.set_screen_messages(me.agent.monitor, NORMAL, HIGH);
    };
};
```

25.9.2 Predefined Types and methods

25.9.2.1 recording_config

The recording configuration API allows to control the attributes and state variables that are reported by messages.

The **recording_config** predefined struct encapsulates transaction recording configuration for a unit or for numerous units at once. The configuration is determined procedurally through API calls.

Predefined methods of **recording_config** are described in the following subsections.

25.9.2.1.1 register_all_field_attributes()

Purpose	Defines all user-defined public fields to be recorded as transaction attributes for data-items of the specified type	
Category	Predefined method	
Syntax	**register_all_field_attributes(***type-name*: string**)**	
Parameters	*type-name*	Any struct type name, including “when” subtypes and template instances

25.9.2.1.2 register_callback_attribute()

Purpose	Defines an attribute for transactions of the specified type, the value of which is determined dynamically by the hook method **tr_get_attribute_value()** (see 25.9.2.3.1).	
Category	Predefined method	
Syntax	**register_callback_attribute(***type-name*: string**,** *attr-name*: string**)**	
Parameters	*type-name*	Any struct type name, including “when” subtypes and template instances
	attr-name	Attribute name to be associated with the struct

Whenever a transaction of the given type needs to be sampled, the hook method is called on the data-item, with the scope unit passed as parameter. If this call returns an empty string, the same method is called on the scope unit, with the data-item passed as parameter. The value returned from either of the calls becomes the value of the attribute.

The callback method default implementation returns an empty string.

Example

In this example, the attribute *destination* is calculated by the monitor when the message is recorded.

```
extend frame_monitor {
    connect_pointers() is also {
        var tr_cfg:recording_config = new;
        tr_cfg.register_field_attributes("frame",{"addr";} );
        tr_cfg.register_callback_attribute("frame","destination");
        assign_recording_config(tr_cfg)
    }

    tr_get_attribute_value(inst:any_struct,name:string):string is also {
        if inst is a frame (f) and name == "destination" then {
            result = append(me.base_addr + f.addr)
        }
    }
}
```

25.9.2.1.3 register_callback_attributes()

Purpose	Defines a set of callback attributes for transactions of the specified type. Calling this method is equivalent to calling "register_callback_attribute()" for each of the given names.	
Category	Predefined method	
Syntax	**register_callback_attributes(***type-name*: string**,** *attr-names*: list of string**)**	
Parameters	*type-name*	Any struct type name, including "when" subtypes and template instances
	attr-names	List of attribute names to be associated with the struct

25.9.2.1.4 register_callback_state_var()

Purpose	Defines a state variable, the value of which is determined dynamically by the unit hook method **tr_get_state_var_value()** (see 25.9.2.2.3). The value is sampled by calling the hook method upon the execution of **msg_changed()** when the scope unit is of the specified type.	
Category	Predefined method	
Syntax	**register_callback_state_var(***unt-name*: string**,** *var-name*: string**)**	
Parameters	*unit-name*	Name of a unit type (including "when" subtypes and template instances)
	var-name	State variable name to be associated with the unit

25.9.2.1.5 register_field_attribute()

Purpose	Defines the specified data item field to be recorded as a transaction attribute. The field is sampled upon the execution of structured messages of certain kinds with given data-items of the specified type as parameters.	
Category	Predefined method	
Syntax	**register_field_attribute(***type-name*: string**,** *field-name*: string**)**	
Parameters	*type-name*	Any struct type name, including "when" subtypes and template instances
	field-name	Name of a field declared for the struct

Example

This example specifies that the field *addr* of the frame should be recorded, in all messages issued by the xbus_monitor.

```
extend xbus_monitor {
    connect_pointers() is also {
        var tr_cfg : recording_config = new;
        tr_cfg.register_field_attribute("frame", "addr");
        assign_recording_config(tr_cfg)
    }
}
```

25.9.2.1.6 register_field_attributes()

Purpose	Defines a set of field attributes for transactions of the specified type. Calling this method is equivalent to calling "register_field_attribute()" for each of the given field names.	
Category	Predefined method	
Syntax	**register_field_attributes(***type-name*: string**,** *field-names*: list of string**)**	
Parameters	*type-name*	Any struct type name, including "when" subtypes and template instances
	field-names	List of field names declared for the struct

Example

This example specifies that the field addr and data of the frame should be recorded, in all messages issued by the xbus_monitor.

```
extend xbus_monitor {
    connect_pointers() is also {
        var tr_cfg : recording_config = new;
        tr_cfg.register_field_attributes("frame", {"addr"; "data"});
        assign_recording_config(tr_cfg)
    }
}
```

25.9.2.1.7 register_field_state_var()

Purpose	Defines the specified field to be recorded as a state variable. The field is sampled on the execution of **msg_changed()** when the scope unit is of the specified type.	
Category	Predefined method	
Syntax	**register_field_state_var(***unt-name*: string**,** *field-name*: string**)**	
Parameters	*unit-name*	Name of a unit type (including "when" subtypes and template instances)
	field-name	Name of a field declared for the unit

25.9.2.1.8 register_method_attribute()

Purpose	Defines an attribute for transactions of the specified type, the value of which is determined dynamically by calling the specified hook method of the specified type.	
Category	Predefined method	
Syntax	**register_method_attribute(***type-name*: string**,** *method-name*: string**)**	
Parameters	*type-name*	Any struct type name, including "when" subtypes and template instances
	method-name	Name of a method declared for the struct

Notes

- Whenever a transaction of the given type needs to be sampled, the hook method by the specified name is called on the data-item, possibly with the scope unit passed as parameter. The value returned from the call becomes the value of the attribute.
- If the struct does not have a declared method by the specified name, the behavior is undefined.
- The callback method must meet the following conditions:
  - It must not be a time-consuming method.
  - It must have a return type.
  - It must either have no parameters, or have exactly one parameter of type any_unit which is not passed by reference.

  If any of these conditions is not met, the behavior is undefined

Example

In this example, the attribute *destination* is calculated by a hook method

```
extend frame_monitor {
    connect_pointers() is also {
        var tr_cfg:recording_config = new;
        tr_cfg.register_field_attributes("frame",{"addr";} );
        tr_cfg.register_method_attribute("frame","destination");
        assign_recording_config(tr_cfg)
    }
}

extend frame {
    destination(scope: any_unit): uint is {
        if scope is a frame_monitor (m) then {
            result = append(m.base_addr + me.addr)
        }
    }
}
```

25.9.2.1.9 register_method_state_var()

Purpose	Defines a state variable, the value of which is determined dynamically by the specified unit hook method. The value is sampled by calling the hook method upon the execution of **msg_changed()** when the scope unit is of the specified type.	
Category	Predefined method	
Syntax	**register_method_state_var(***unit-name*: string**,** *method-name*: string**)**	
Parameters	*unit-name*	Name of a unit type, including “when” subtypes and template instances
	method-name	Name of a method declared for the unit

Notes

- Upon the execution of **msg_changed()**, the hook method by the specified name is called on the scope unit. The value returned from the call becomes the value of the state variable.
- If the unit does not have a declared method by the specified name, the behavior is undefined.
- The callback method must meet the following conditions:

— It must not be a time-consuming method.
— It must have a return type.
— It must have no parameters.

If any of these conditions is not met, the behavior is undefined.

25.9.2.1.10 set_attribute_format()

Purpose	Determines the textual format in which an already registered attribute is to be displayed.	
Category	Predefined method	
Syntax	**set_attribute_format(***type-name*: string**,** *attr-name*: string**,** *format*: string**)**	
Parameters	*type-name*	Any struct type name, including “when” subtypes and template instances
	attr-name	Registered attribute name for the given struct type. The behavior is not defined for the case that attribute *attr-name* is not registered within this **recording_config** object.
	format	Format string to be used.

Notes

— When an attribute format is set using this method, the attribute value is formatted using the specified format string prior to being displayed at destinations.
— The string specified by *format* must be a valid format string that contains exactly one “**%...**” parameter applicable to the attribute type (see 29.7.3). If this condition is not met, the behavior is undefined.

Example

In this example, the attribute *addr* is set to be displayed in the hexadecimal format.

```
extend frame_monitor {
    connect_pointers() is also {
        var tr_cfg:recording_config = new;
        tr_cfg.register_field_attribute("frame","addr");
        tr_cfg.set_attribute_format("frame","addr","%#x");
        assign_recording_config(tr_cfg)
    }
}
```

25.9.2.1.11 set_attribute_sampling()

Purpose	Determines that an already registered attribute is to be sampled and recorded at the specified sampling points.
Category	Predefined method
Syntax	**set_attribute_sampling(***type-name*: string**,** *attr-name*: string**,** *points*: list of tr_sampling_point_t**)**

Parameters	*type-name*	Any struct type name, including “when” subtypes and template instances
	attr-name	Registered attribute name for the given struct type. The behavior is not defined for the case that attribute *attr-name* is not registered within this **recording_config** object.
	points	A list of zero or more values representing structured message sampling points. The sampling points correspond to different roles of the data item within structured messages. The setting for the specified attribute overrides the previous setting. The possible sampling points are item of predefined type **tr_sampling_point_t** (see 25.9.2.4). The default setting for all attributes is the element list {**STARTED**; **ENDED**}.

25.9.2.1.12 set_label_attribute()

Purpose	Sets an already registered attribute as the transaction label for transaction of the given type.	
Category	Predefined method	
Syntax	**set_label_attribute(***type-name*: string**,** *attr-name*: string**)**	
Parameters	*type-name*	Any struct type name, including “when” subtypes and template instances
	attr-name	Registered attribute name for the given struct type. The behavior is not defined for the case that attribute *attr-name* is not registered within this **recording_config** object.

Note: The actual effect of this setting is implementation dependent. Typically it only affects the implementation-specific transaction database (see 25.9.2.5.3).

25.9.2.1.13 set_text_attribute()

Purpose	Determines that an already registered attribute is to be written to text destinations (screen and log files).	
Category	Predefined method	
Syntax	**set_text_attribute(***type-name*: string**,** *attr-name*: string**)**	
Parameters	*type-name*	Any struct type name, including “when” subtypes and template instances
	attr-name	Registered attribute name for the given struct type. The behavior is not defined for the case that attribute *attr-name* is not registered within this **recording_config** object.

Note: This setting does not affect destinations other than screen and log files.

25.9.2.1.14 set_text_state_var()

Purpose	Determines that an already registered state variable is written to text destinations (screen and log files).	
Category	Predefined method	
Syntax	**set_text_state_var(***unit-name*: string**,** *var-name*: string**)**	
Parameters	*unit-name*	Name of a unit type (including “when” subtypes and template instances)
	var-name	Registered state variable name for the given unit type. The behavior is not defined for the case that state variable *var-name* is not registered within this **recording_config** object.

Note: This setting does not affect destinations other than screen and log files.

25.9.2.2 any_unit Recording Configuration API extensions

This section contains the Recording Configuration methods belonging to **any_unit**.

25.9.2.2.1 assign_recording_config()

Purpose	Assigns the given **recording_config** object to this unit instance. This also affects the configuration of all descendant units that have not been assigned a **recording_config** object explicitly, or associated with one through a closer parent.	
Category	Predefined method	
Syntax	**assign_recording_config(***rec*: recording_config**)**	
Parameters	*rec*	The recording_config object to be assigned to the unit instance tree under and including the current unit.

25.9.2.2.2 get_recording_config()

Purpose	Returns the **recording_config** object that is associated with this unit instance. A unit is associated with a **recording_config** object either by explicit assignment using **assign_recording_config()** (see 25.9.2.2.1), or otherwise inherits the association from its parent unit (recursively). Units with no parent unit(e.g., sys) are associated by default with an empty **recording_config** object.
Category	Predefined method
Syntax	**get_recording_config()**: recording_config
Return value	The **recording_config** object that is associated with this unit instance.

25.9.2.2.3 tr_get_state_var_value()

Purpose	Callback method that returns the value of a registered callback state variable for this unit instance. It is invoked on scope units for which the state variable was registered upon execution of **msg_changed()**. (see "register_callback_state_var()"). The value returned by it is written to the message destination.
Category	Predefined method
Syntax	**tr_get_state_var_value(***attr-name*: string**)**: string
Parameters	*attr-name* Name of the registered attribute
Return value	Value of a registered state variable.

Note: The default value returned by this method is "" (an empty string).

25.9.2.3 any_struct Recording Configuration API extensions

This section contains the Recording Configuration methods belonging to **any_struct**.

25.9.2.3.1 tr_get_attribute_value()

Purpose	Callback method that returns the value of a registered callback attribute. When called on a non-unit struct with a unit parameter, it returns the value of the callback attribute for this data-item instance in the given scope unit. When called on a unit with a non-unit struct parameter, it returns the value of the callback attribute for the given data-item instance in this scope unit. (It is never called on a non-unit struct with a non-unit struct parameter.) It is invoked on data-item instances for which the attribute was registered at the appropriate sampling points by calling **register_callback_attribute()** (see "register_callback_attribute()").
Category	Predefined method
Syntax	**tr_get_attribute_value(***inst*: any_struct**,** *attr-name*: string**)**: string
Parameters	*inst* Depending on usage, either of the following: — Reference to the data item instance that is being considered (when called on a scope unit) — Reference to the unit in the scope where the attribute should be evaluated (when called on a data item instance).
	attr-name Name of the registered attribute.
Return value	String value of a registered attribute specified by *attr-name*.

Note: The default value returned by this method is "" (an empty string).

Example 1

In this example, the **tr_get_attribute_value()** callback method is extended to calculate and return the frame address when the message is emitted by the monitor:

```
extend frame {
  tr_get_attribute_value(scope: any_struct, attr_name: string): string is also
   {
     if scope is a monitor (m) {
         if attr_name == "address" then {
             result = append(m.base_addr + header.addr);
         };
         if attr_name == "direction" then {
             result = append(header.dir);
         };
     };
  };
};
```

Example 2

In this example, the attribute *destination* is calculated by the monitor when the message is recorded.

```
extend frame_monitor {
    tr_get_attribute_value(inst: any_struct, name: string): string is also {
        if inst is a frame (f) and name == "destination" then {
            result = append(me.base_addr + f.addr);
        };
    };
};
```

25.9.2.4 tr_sampling_point_t

This enumerated type is used by **set_attribute_sampling()** (see 25.9.2.1.11) to define sampling points for data item attributes. Possible sampling points are:

- **STARTED** – The data-item argument of **msg_started()**
- **ENDED** – The data-item argument of **msg_ended()**
- **TRANSFORMED** – The from-item (first) or to-item (second) argument of **msg_transformed()**
- **CHANGED** – State variables of the scope unit, used with **msg_changed()**
- **INFO** – The data-item arguments of **msg_info()**

25.9.2.5 message_manager API

All methods presented in this section belong to the **message_manager** predefined type. There is a singleton object of this type, the instance of which is under **global**.

25.9.2.5.1 set_screen_messages

Purpose	Selects which messages from the unit and its subtree with the specified tag will be sent to the screen destination. The selection is done according to the verbosity, modules, and text-pattern parameters
Category	Predefined method
Syntax	**set_screen_messages(***root-unit*: any_unit**,** *tag*: message_tag**,** *verbosity*: message_verbosity [**,** *modules*: string [**,** *text_pattern*: string [**,** *rec*: bool]]]**)**

Parameters	*root-unit*	Unit to which the new setting will be applied.
	tag	Message tag to which the new setting applies. The new setting will only affect messages of the specified tag.
	verbosity	Highest verbosity level for messages. A given message is issued to the screen only if its verbosity is equal to or lower than the specified verbosity. Valid values are: **NONE**, **LOW**, **MEDIUM**, **HIGH**, and **FULL**.
	modules	String pattern used for matching module names (wild cards permitted). A given message is issued to the screen only if it is defined in the specified module(s). Default value is "*" which matches any module name.
	text_pattern	String pattern used for matching the message text (default = "..." which matches any string). A given message is issued to the screen only if its message string matches the specified text-pattern string (see 4.11). For SDMs, the ID string is considered as the message string for this purpose.
	rec	Recursion indicator (default = **TRUE**): — If **TRUE**, the new setting will apply to all units in the unit subtree under the given unit. — If **FALSE**, the new setting will apply to the given unit instance only, and will not apply to other units in its unit subtree.

Example

```
extend my_env {
    post_generate() is also {
        message_manager.set_screen_messages(me.the_agent.the_monitor, NORMAL,
        HIGH);
    };
};
```

25.9.2.5.2 set_screen_messages_off

Purpose	Disables the sending of messages with the specified tag to the screen destination	
Category	Predefined method	
Syntax	**set_screen_messages_off(***root-unit*: any_unit**,** *tag*: message_tag [**,** *rec*: bool]**)**	
Parameters	*root-unit*	Unit to which the new setting will be applied.
	tag	Message tag to which the new setting applies. The new setting will only affect messages of the specified tag.
	rec	Recursion indicator (default = **TRUE**): — If **TRUE**, the new setting will apply to all units in the unit subtree under the given unit. — If **FALSE**, the new setting will apply to the given unit instance only, and will not apply to other units in its unit subtree.

Example

```
extend my_env {
    post_generate() is also {
        message_manager.set_screen_messages_off(me, NORMAL);
    };
};
```

25.9.2.5.3 set_transaction_messages

Purpose	Selects which SDM transaction messages from the unit and its subtree with the specified tag will be sent to an implementation specific transaction database. The selection is done according to the verbosity, modules, and text-pattern parameters	
Category	Predefined method	
Syntax	**set_transaction_messages(***root-unit*: any_unit**,** *tag*: message_tag**,** *verbosity*: message_verbosity [**,** *modules*: string [**,** *text_pattern*: string [**,** *rec*: bool]]]**)**	
Parameters	*root-unit*	Unit to which the new setting will be applied.
	tag	Message tag to which the new setting applies. The new setting will only affect messages of the specified tag.
	verbosity	Highest verbosity level for messages. A given message is issued to the transaction database only if its verbosity is equal to or lower than the specified verbosity. Valid values are: **NONE**, **LOW**, **MEDIUM**, **HIGH**, and **FULL**.
	modules	String pattern used for matching module names (wild cards permitted). A given message is issued to the transaction database only if it is defined in the specified module(s). Default value is "*" which matches any module name.
	text_pattern	String pattern used for matching the message text (default = "..." which matches any string). A given message is issued to the transaction database only if its message string matches the specified text-pattern string (see 4.11). For SDMs, the ID string is considered as the message string for this purpose.
	rec	Recursion indicator (default = **TRUE**): — If **TRUE**, the new setting will apply to all units in the unit subtree under the given unit. — If **FALSE**, the new setting will apply to the given unit instance only, and will not apply to other units in its unit subtree.

Example

```
extend my_env {
    post_generate() is also {
        message_manager.set_transaction_messages(me.the_agent.the_monitor,
        NORMAL, HIGH);
    };
};
```

25.9.2.5.4 set_transaction_messages_off

Purpose	Disables the sending of messages with the specified tag to an implementation specific transaction database.

Category	Predefined method	
Syntax	**set_transaction_messages(***root-unit*: any_unit**,** *tag*: message_tag [**,** *rec*: bool]**)**	
Parameters	*root-unit*	Unit to which the new setting will be applied.
	tag	Message tag to which the new setting applies. The new setting will only affect messages of the specified tag.
	rec	Recursion indicator (default = **TRUE**): — If **TRUE**, the new setting will apply to all units in the unit subtree under the given unit. — If **FALSE**, the new setting will apply to the given unit instance only, and will not apply to other units in its unit subtree.

Example

```
extend my_env {
    post_generate() is also {
        message_manager.set_transaction_messages_off(me, NORMAL);
    };
};
```

25.9.2.5.5 set_file_messages

Purpose	Selects which messages from the unit and its subtree with the specified tag will be sent to the specified log file. The selection is done according to the verbosity, modules, and text-pattern parameters.
Category	Predefined method
Syntax	**set_file_messages(***file-name*: string**,** *root-unit*: any_unit**,** *tag*: message_tag**,** *verbosity*: message_verbosity [**,** *modules*: string [**,** *text_pattern*: string [**,** *rec*: bool]]]**)**

Parameters	*file-name*	Log file to which the new setting will be applied.
	root-unit	Unit to which the new setting will be applied.
	tag	Message tag to which the new setting applies. The new setting will only affect messages of the specified tag.
	verbosity	Highest verbosity level for messages. A given message is issued to the screen only if its verbosity is equal to or lower than the specified verbosity. Valid values are: **NONE**, **LOW**, **MEDIUM**, **HIGH**, and **FULL**.
	modules	String pattern used for matching module names (wild cards permitted). A given message is issued to the file only if it is defined in the specified module(s). Default value is "*" which matches any module name.
	text_pattern	String pattern used for matching the message text (default = "..." which matches any string). A given message is issued to the file only if its message string matches the specified text-pattern string (see 4.11). For SDMs, the ID string is considered as the message string for this purpose.
	rec	Recursion indicator (default = **TRUE**): — If **TRUE**, the new setting will apply to all units in the unit subtree under the given unit. — If **FALSE**, the new setting will apply to the given unit instance only, and will not apply to other units in its unit subtree.

25.9.2.5.6 set_file_messages_off

Purpose	Disables the sending of messages with the specified tag to the log file.	
Category	Predefined method	
Syntax	**set_file_messages_off(***file-name*: string**,** *root-unit*: any_unit**,** *tag*: message_tag [**,** *rec*: bool]**)**	
Parameters	*file-name*	Log file to which the new setting will be applied.
	root-unit	Unit to which the new setting will be applied.
	tag	Message tag to which the new setting applies. The new setting will only affect messages of the specified tag.
	rec	Recursion indicator (default = **TRUE**): — If **TRUE**, the new setting will apply to all units in the unit subtree under the given unit. — If **FALSE**, the new setting will apply to the given unit instance only, and will not apply to other units in its unit subtree.

25.9.2.5.7 set_message_format

Purpose	Modifies the format of messages that are issued by the specified unit or unit subtree with the specified tag when sending them to the specified text destination(s).
Category	Predefined method

Syntax	**set_message_format(***root-unit*: any_unit**,** *tag*: message_tag**,** *file-names*: list of string**,** *format*: message_format [**,** *rec*: bool]**)**	
Parameters	*root-unit*	Unit to which the new setting will be applied.
	tag	Message tag to which the new setting applies. The new setting will only affect messages of the specified tag.
	file-names	List of file names to which the new format settings will be applied. An empty string in the list denotes the screen. If the entire list is empty, the new settings will be applied to all text destinations.
	format	Format to be used for messages. Valid predefined values: **none**, **short**, and **long**. Other values can be added by extending the message_format type (see 25.9.2.6). **none** specifies no additions to the bare message text. Any styles implied by the other formats are implementation-dependent.
	rec	Recursion indicator (default = **TRUE**): — If **TRUE**, the new setting will apply to all units in the unit subtree under the given unit. — If **FALSE**, the new setting will apply to the given unit instance only, and will not apply to other units in its unit subtree.

Example

```
unit my_env {
    post_generate() is also {
        message_manager.set_message_format(me, NORMAL, {}, long);
    };
};
```

25.9.2.6 message_format

This enumerated type defines possible message formats, used when messages are sent to text destinations. It is used in **set_message_format()** to set the format to be used by messages from a given unit or unit tree with a given tag. It is also used in **create_formatted_message()** hook method to identify which formatting option should be applied to the current message.

Predefined values are **short**, **long** and **none**. One can extend this type to include specific user-defined formatting.

25.9.2.7 message_action

The predefined struct type **message_action** represents a specific actual message that occurs during a run. It is used in **create_formatted_message()** and provides information about the current message.

Predefined methods of **message_action** are described in the following subsections.

25.9.2.7.1 get_id()

Description	Get the message's unique id
Category	Predefined method

Syntax	**get_id()**: int
Return value	Unique id of this message action.

25.9.2.7.2 get_tag()

Description	Get the message tag.
Category	Predefined method
Syntax	**get_tag()**: message_tag
Return value	Tag of this message.

25.9.2.7.3 get_verbosity()

Description	Get the message verbosity.
Category	Predefined method
Syntax	**get_verbosity()**: message_verbosity
Return value	Verbosity of this message.

25.9.2.7.4 get_source_method_layer()

Description	Get the source method layer of a message.
Category	Predefined method
Syntax	**get_source_method_layer()**: rf_ method_layer
Return value	Method layer in which this message action resides.

25.9.2.7.5 get_source_line_num()

Description	Get the source line number of a message.
Category	Predefined method
Syntax	**get_source_line_num()**: int
Return value	Source line number in which this message action resides.

25.9.2.7.6 get_source_struct()

Description	Get the actual struct instance that issued the message.

Category	Predefined method
Syntax	**get_source_struct()**: any_struct
Return value	Struct instance that issued the message.

25.9.2.7.7 get_time()

Description	Get the message time.
Category	Predefined method
Syntax	**get_time()**: string
Return value	Properly formatted string for the current value of **sys.time**.

25.9.2.7.8 get_format()

Description	Get the message format.
Category	Predefined method
Syntax	**get_format()**: message_format
Return value	Format being used for this message.

25.9.2.7.9 get_sdm_handler()

Description	Get the SDM handler.
Category	Predefined method
Syntax	**get_sdm_handler()**: sdm_handler
Return value	The **sdm_handler** object for this message, or NULL if this message is not SDM (see 25.9.2.8).

25.9.2.8 sdm_handler

The predefined struct type **sdm_handler** represents SDM-specific data on messages. It is used in **create_formatted_message()** and provides additional information about the current message.

sdm_handler is an abstract class, and each SDM kind is represented by its subtype. Predefined fields and methods of **sdm_handler**, common to all SDM kinds, are described in the following subsections.

25.9.2.8.1 Predefined fields

— **scope**: any_unit

The unit instance to which the message belongs. By default, it is the unit in the context of which the message is executed; this can be modified in the message's optional action block (see Table 39).

— **id_str**: string

The **msg-id** string, as specified in the SDM action.

— **body_text**: string

The text assigned in the optional action block (see Table 39), or an empty string if none.

25.9.2.8.2 get_kind_string()

Description	Get the SDM handler.
Category	Predefined method
Syntax	**get_kind_string()**: string
Return value	String representing the SDM kind. For example, “started” for **msg_started**, “changed” for **msg_changed**.

25.9.2.8.3 get_attribute_string()

Description	Get string representation for text attributes.
Category	Predefined method
Syntax	**get_attribute_string(***inst*: any_struct**)**: string
Return value	The string to display the registered text attributes for the specified data object (or an empty string if there are no text attributes) after registering them. Text attributes are those for which **set_text_attribute()** was called (see 25.9.2.1.13); or in case of **msg_changed** those for which **set_text_state_var()** was called (see 25.9.2.1.14).

25.9.2.8.4 collect_text_attributes()

Description	Collect text attributes and their values.
Category	Predefined method
Syntax	**collect_text_attributes(***inst*: any_struct**,** *names*: list of string**,** *values*: list of string**)**

This method collects the names and printed string values of the text attributes for the specified data object *inst*, and adds them to the two provided lists, *names* and *values* respectively. Both lists are cleaned before collecting attributes, and any items present in them prior to calling this method are removed.

Note: Text attributes are those for which **set_text_attribute()** was called (see 25.9.2.1.13); or in case of **msg_changed** those for which **set_text_state_var()** was called (see 25.9.2.1.14).

25.9.2.9 sdm_started_handler

This inherits from **sdm_handler** (see 25.9.2.8) and represents **msg_started actions**.

25.9.2.9.1 Predefined fields

— **data_item**: any_struct

Data item object specified in the **msg_started** action.

— **parent**: any_struct
Parent transaction object specified in the optional action block, or NULL if none.

25.9.2.10 sdm_ended_handler

This inherits from **sdm_handler** (see 25.9.2.8) and represents **msg_ended actions**.

25.9.2.10.1 Predefined fields

— **data_item**: any_struct
Data item object specified in the **msg_ended** action.
— **parent**: any_struct
Parent transaction object specified in the optional action block, or NULL if none.
— **start_time**: time
Start time specified in the optional action block, or UNDEF if none.

25.9.2.11 sdm_transformed_handler

This inherits from **sdm_handler** (see 25.9.2.8) and represents **msg_transformed actions**.

25.9.2.11.1 Predefined fields

— **from_item**: any_struct
First data item object specified in the **msg_transformed** action.
— **to_item**: any_struct
Second data item object specified in the **msg_transformed** action.

25.9.2.12 sdm_changed_handler

This inherits from **sdm_handler** (see 25.9.2.8) and represents **msg_changed actions**.

25.9.2.12.1 Predefined fields

— **new_state**: string
State string specified in the **msg_changed** action.

25.9.2.13 sdm_info_handler

This inherits from **sdm_handler** (see 25.9.2.8) and represents **msg_transformed actions**.

25.9.2.13.1 Predefined fields

— **item1**: any_struct
First data item object specified in the **msg_info** action, or NULL if none.
— **item2**: any_struct
Second data item object specified in the **msg_info** action, or NULL if none.

25.9.2.14 any_unit message API

25.9.2.14.1 create_formatted_messsage()

Description	This method is a hook (callback) predefined method used for implementing user-defined formatting on message output to text destinations.	
Category	Predefined method	
Syntax	**create_formatted_message(***message*: message_action**,** *buffer*: list of string**)**	
Parameters	*message*	Object that represents that current message being issued. (see 25.9.2.7)
	buffer	Buffer for the formatted message. At the start of this method, *buffer* contains the base text of the message before formatting.

Notes:

— By default this method does not modify the original content of *buffer*, when **get_format()** of *message* is none or a user-defined value. When **get_format()** of *message* is **short** or **long**, the result is implementation-dependent.

— There are no guarantees on the number of times this method is actually called for a given message. Thus, if an extension of this method produces any side effects (other than the actual message formatting), the behavior is undefined.

— If a user extension of **create_formatted_message()** uses **is also**, the content of *buffer* at the beginning of the user's extension contains the default formatting. If it uses is only, the default formatting is not performed, and the content of *buffer* is just the base text of the message.

Example

```
extend message_format: [MY_FORMAT};
unit env_u {
    post_generate() is also {
        message_manager.set_message_format(me, MY_TAG, {},MY_FORMAT);
    };
};

extend any_unit {
    create_formatted_message(message: message_action, buffer: list of string)
    is also {
        if message.get_format() == MY_FORMAT {
            var src_module: rf_module =
            message.get_source_method_layer().get_module();
            var src: string = append("at line ",
            message.get_source_line_num(),
            " in @",
            src_module.get_name()); outf("sys.time is %s\n", get_time());
            buffer.add0(src); buffer.add0(append("sys.time is %s\n",
            message.get_time()));
        };
    };
};
```