

5. Units

This clause describes the constructs used to define units and their use in a modular verification methodology (see also Clause 4).

5.1 Overview

Units are the basic structural blocks for creating verification modules (verification cores) which can easily be integrated together to test larger and larger portions of an HDL design as it develops. Units, like structs, are compound data types that contain data fields, procedural methods, and other members. Unlike structs, however, a unit instance is bound to a particular component in the DUT (an HDL path). Furthermore, each unit instance has a unique and constant place (an *e* path) in the runtime data structure of an *e* program. Both the *e* path and the complete HDL path associated with a unit instance are determined prior to generation.

The basic runtime data structure of an *e* program is a tree of unit instances whose root is **sys**, the only pre-defined unit in *e*. Additionally there are structs that are dynamically bound to unit instances. The runtime data structure of a typical *e* program is similar to that of the `XYZ_router` program shown in Figure 1.

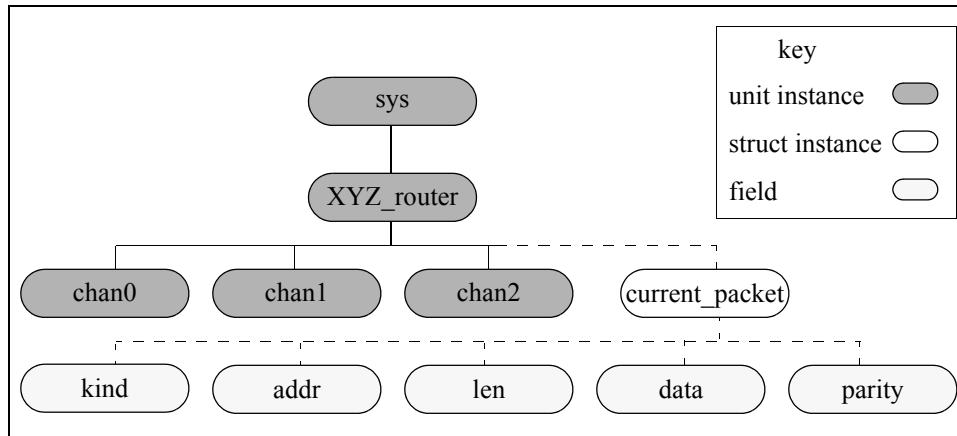


Figure 1—Runtime data structure of the XYZ_Router

Each unit instance in the unit instance tree of the `XYZ_router` matches a module instance in the Verilog DUT, as shown in Figure 2. The one-to-one correspondence in this particular design between *e* unit instances and DUT module instances is not required for all designs. In more complex designs, there may be several levels of DUT hierarchy corresponding to a single level of hierarchy in the tree of *e* unit instances.

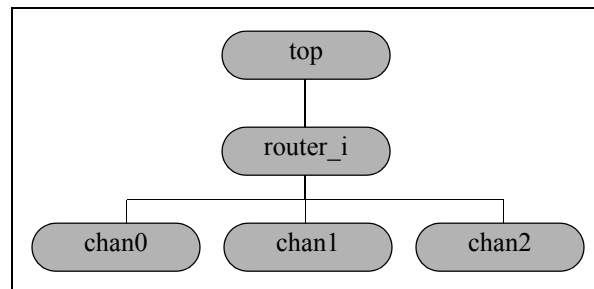


Figure 2—DUT router hierarchy

By binding an *e* unit instance to a particular component in the DUT hierarchy, signals can be referenced within that DUT component using relative HDL path names. When the units are integrated into a unit instance tree during pre-run generation, the complete path name for each referenced HDL object is determined by concatenating the complete HDL path of the parent unit to the path of the unit containing the referenced object. This ability to use relative path names to reference HDL objects means the combination of verification cores can be changed as the HDL design and the verification environment evolve; regardless of where the DUT component is instantiated in the final integration, the HDL path names in the verification environment remain valid.

5.1.1 Units vs. structs

Modeling a DUT component with a unit or a struct often depends on which verification strategy is employed. Compelling reasons for using a unit instead of a struct include:

- The DUT component will be tested both as a standalone and integrated into a larger system. Modeling the DUT component with a unit allows the use relative path names when referencing HDL objects. When integrating the component with the rest of the design, simply change the HDL path associated with the unit instance and all the HDL references it contains are updated to reflect the component's new position in the design hierarchy. This methodology eliminates the need for computed HDL names (e.g., `'(path_str).sig'`), which can impact runtime performance.
- Methods that access many signals at runtime will be used. The correctness of all signal references within units is determined and checked during pre-run generation. If an *e* program does not contain user-defined units, the absolute HDL references within structs are also checked during pre-run generation. However, if it does contain user-defined units, the relative HDL references within structs are checked at runtime. In this case, using units rather than structs can enhance runtime performance.

On the other hand, using a struct to model abstract collections of data, such as packets, allows more flexibility in generating the data. With structs, the data can be generated during pre-run generation, at runtime, or on the fly (possibly in response to conditions in the DUT). Unit instances, however, can only be generated during pre-run generation, because each unit instance has a unique and constant place (an *e* path) in the runtime data structure of an *e* program, just as an HDL component instance has a constant place in the DUT hierarchical tree. Thus, the unit tree cannot be modified by generating unit instances on the fly.

NOTE—An allocated struct instance automatically establishes a reference to its parent unit. If this struct is a generated during pre-run generation it inherits the parent unit of its parent struct. If the struct is dynamically allocated by the **new** or **gen** action, then the parent unit is inherited from the struct belonging to the enclosing method.

5.1.2 HDL paths and units

Relative HDL paths are essential in creating a verification module that can be used to test a DUT component. Once an *e* unit instance is bound to a particular component in the DUT hierarchy, regardless of where the DUT component is instantiated in the final integration, the HDL path names are still valid.

Example

The `XYZ_router` (shown in Figure 1) can be bound to the DUT router (shown in Figure 2) as follows.

- a) Use the **hdl_path()** method within a **keep** constraint to associate a unit or unit instance with a DUT component. The following code extends **sys** by creating an instance of the `XYZ_router` unit and binds the unit instance to the `router_i` instance in the DUT.

```

    extend sys {
        unit_core: XYZ_router is instance;
        keep unit_core.hdl_path() == "top.router_i";
    };

```

- b) Similarly, the following code creates three instances of `XYZ_channel` in `XYZ_router` and constrains the HDL path of the instances to be `chan0`, `chan1`, and `chan2`. These are the names of the channel instances in the DUT relative to the `router_i` instance.

```

    unit XYZ_router {
        channels: list of XYZ_channel is instance;
        keep channels.size() == 3;
        keep for each in channels { .hdl_path() ==
            append("chan", index); };
    };

```

The full path for a unit instance is used to resolve any internal HDL object references that contain relative HDL paths. It is determined during generation by appending the HDL path of the child unit instance to the full path of its parent, starting with `sys`. `sys` has the empty full path `""`. The full path for the `XYZ_router` instance is `top.router_i` and that for the first channel instance is `top.router_i.chan0`.

NOTE—By default, the `do_print()` method of any unit prints two predefined lines, as well as the user-defined fields. The predefined lines display the *e* path and the full HDL path for that unit. The *e* path line contains a hyperlink to the parent unit.

5.1.3 Methodology recommendations and limitations

Fields of type `unit` can be generated dynamically. However, the field shall be constrained to only refer to an existing unit instance.

The following limitations are implied by the nature of unit instances and fields of type `unit`.

- Unit instances cannot be the object of a **new** or **gen** action or a call to **copy()**.
- Unit instances cannot be placed on the left-hand-side of the assignment operator.
- List methods which alter the original list, like **list.add()** or **list.pop()** cannot be applied to lists of unit instances.
- Units are intended to be used as structural components and not as data carriers. Therefore, using physical fields in unit instances, as well as packing or unpacking into unit instances, is not recommended. Unpacking into a field of type `unit` when the field is `NULL` shall cause a runtime error.
- All instances of the same unit type shall be bound to the same kind of HDL component.

To create a modular verification environment, the following recommendations are also important:

- a) Avoid setting global configuration options with **set_config()**. Instead, for numeric settings, use **set_config_max()**.
- b) Avoid global changes to the default packing options. Instead, define unit-specific options in the top-level unit and access them from lower-level units with **get_enclosing_unit()**.
- c) Place references to HDL objects in unit methods. To access HDL objects from struct methods, declare additional methods in a unit. When these access methods are one line of *e* code, declare them as **inline** methods for maximum efficiency.
- d) Use computed path names in structs that may be dynamically associated with more than one unit.
- e) Pre-run generation is performed before creating the stubs file. To minimize the time required to create a stubs file, move any pre-run generation that is not related to building the tree of unit instances into the procedural code, preferably as an extension of the **run()** method of the appropriate structs.

5.2 Defining units and fields of type unit

The following sections describe the constructs for defining units and fields of type unit.

5.2.1 unit

Purpose	Define a data struct associated with an HDL component or block
Category	Statement
Syntax	unit <i>unit-type</i> [like <i>base-unit-type</i>] { [<i>unit-member</i> ; ...]}
Parameters	<i>unit-type</i> The type of the new unit.
	<i>base-unit-type</i> The type of the unit from which the new unit inherits its members.
	<i>unit-member</i> ; ... The contents of the unit. Like structs, units can have the following types of members: — data fields for storing data — methods for procedures — events for defining temporal triggers — coverage groups for defining coverage points — when , for specifying inheritance subtypes — declarative constraints for describing relations between data fields — on , for specifying actions to perform upon event occurrences — expect , for specifying temporal behavior rules Unlike structs, units can also have verilog members, so Verilog stub files can be created for modular designs. A unit can be empty, containing no members.

Units are the basic structural blocks for creating verification modules (verification cores) that can easily be integrated together to test larger designs. Units are a special kind of struct, with two important properties:

- Units or unit instances can be bound to a particular component in the DUT (an HDL path).
- Each unit instance has a unique and constant parent unit (an *e* path). Unit instances create a static tree, determined during pre-run generation, in the runtime data structure of an *e* program.

Because the base unit type (**any_unit**) is derived from the base struct type (**any_struct**), user-defined units have the same predefined methods. In addition, units can have **verilog** members and have several specialized predefined methods.

A unit type can be extended or used as the basis for creating unit subtypes. Extended unit types or unit subtypes inherit the base type's members and contain additional members (see also 5.1.1).

Syntax example:

```
unit XYZ_channel {
    event external_clock;
    event packet_start is rise('valid_out')@sim;
    event data_passed;

    verilog variable 'valid_out' using wire;

    data_checker() @external_clock is {
        while 'valid_out' == 1 {
```

```

        wait cycle;
        check that 'data_out' == 'data_in';
    };
emit data_passed;
};

on packet_start {
    start data_checker();
};
};

```

5.2.2 field: unit-type is instance

Purpose	Define a unit instance field
Category	Unit member
Syntax	<i>field-name</i> [: <i>unit-type</i>] is instance
Parameters	<i>field-name</i> The name of the unit instance being defined.
	<i>unit-type</i> The name of a unit type. If the field name is the same as an existing type, the “: <i>unit-type</i> ” part of the field definition can be omitted. Otherwise, the type specification is required.

This defines a field of a unit to be an instance of a unit type. Units can be instantiated within other units, thus creating a unit tree. The root of the unit tree is **sys**, the only predefined unit in *e*. The do-not-generate operator (!) is not allowed with fields of type unit instance.

A unit instance has to be bound to a particular component in the DUT (an HDL path). Each unit instance also has a unique and constant place (an *e* path) in the runtime data structure of an *e* program that is determined during pre-run generation. Unit instances can be created only during pre-run generation. Instantiating a unit in a struct shall cause a compile-time error; units can only be instantiated within another unit.

NOTE—It is not recommended to use the physical field operator (%) with fields of type unit instance.

Syntax example:

```
cpu: XYZ_cpu is instance;
```

5.2.3 field: unit-type

Purpose	Define a field of type unit	
Category	Struct or unit member	
Syntax	[!] <i>field-name</i> [: <i>unit-type</i>]	
Parameters	!	Denotes an ungenerated field. If this field is generated on the fly, it needs to be constrained to an existing unit instance or a runtime error shall occur.
	<i>field-name</i>	The name of the field being defined.
	<i>unit-type</i>	The name of a unit type. If the field name is the same as an existing type, the “: <i>unit-type</i> ” part of the field definition can be omitted. Otherwise, the type specification is required.

This defines a field of unit type. A field of unit type is always either NULL or a reference to a unit instance of a specified unit type. If a field of type unit is generated, it shall be constrained to an existing unit instance.

NOTE—It is not recommended to use the physical field operator (%) with fields of type unit.

Syntax example:

```

extend XYZ_router{
    !current_chan: XYZ_channel;
};

```

5.2.4 field: list of unit instances

Purpose	Define a list field of unit instances	
Category	Struct or unit member	
Syntax	<i>name</i> :[<i>length-exp</i>]: list of unit-type is instance	
Parameters	<i>name</i>	The name of the list being defined.
	<i>length-exp</i>	An expression that gives the initial size for the list.
	<i>unit-type</i>	A unit type.
	is instance	Creates a list of unit instances.

This defines a list field of unit instances. A list of unit instances can only be created during pre-run generation and cannot be modified after it is generated. List operations, such as **list.add()** or **list.pop()**, that alter the list created during pre-run generation are not allowed for lists of unit instances.

NOTE—It is not recommended to use the physical field operator (%) with lists of unit instance.

Syntax example:

```

channels: list of XYZ_channel is instance;

```

5.2.5 field: list of unit-type

Purpose	Define a list field of type unit
Category	Struct or unit member
Syntax	[!]name:[length-exp]: list of unit-type
Parameters	! Do not generate this list.
	name The name of the list being defined.
	length-exp An expression that gives the initial size for the list.
	unit-type A unit type.

This defines a list field of type unit.

NOTE—It is not recommended to use the physical field operator (%) with lists of unit type.

Syntax example:

```
var currently_valid_channels: list of XYZ_channel;
```

5.3 Predefined methods for any unit

There is a predefined generic type **any_unit**, which is derived from **any_struct**. **any_unit** is the base type implicitly used in user-defined unit types, so all predefined methods for **any_unit** are available for any user-defined unit. The predefined methods for **any_struct** are also available for any user-defined unit.

The predefined methods for **any_unit** include:

- hdl_path()
- full_hdl_path()
- e_path()
- agent()
- get_parent_unit()

5.3.1 hdl_path()

Purpose	Return a relative HDL path for a unit instance
Category	Predefined pseudo-method for any unit
Syntax	[unit-exp.]hdl_path(): string
Parameters	unit-exp An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed.

Returns the HDL path of a unit instance. The most important role of this method is to bind a unit instance to a particular component in the DUT hierarchy. Binding an *e* unit or unit instance to a DUT component

enables the referencing of signals within that component using relative HDL path names. All instances of the same unit shall be bound to the same kind of HDL components.

Regardless of where the DUT component is instantiated in the final integration, the HDL path names are still valid. The binding of unit instances to HDL components is a part of the pre-run generation process and needs to be done by using **keep** constraints. The HDL path for **sys** cannot be constrained.

This method always returns an HDL path exactly as it was specified in constraints, e.g., if a macro is used in a constraint string, then **hdl_path()** returns the original string and not the substituted string.

NOTE—Although absolute HDL paths are allowed, relative HDL paths are recommended for using a modular verification strategy.

Syntax example:

```

extend dut_error_struct {
  write() is first {
    var channel: XYZ_channel =
      source_struct().try_enclosing_unit(XYZ_channel);
    if (channel != NULL) {
      out("Error in XYZ channel: instance ",
        channel.hdl_path());
    };
  };
};

```

5.3.2 full_hdl_path()

Purpose	Return an absolute HDL path for a unit instance
Category	Predefined method for any unit
Syntax	[<i>unit-exp.</i>] full_hdl_path() : string
Parameters	<i>unit-exp</i> An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed.

Returns the absolute HDL path for the specified unit instance. This method is used mainly in informational messages. Like the **hdl_path()** method (see 5.3.1), this method returns the path as originally specified in the **keep** constraint, without making any macro substitutions.

Syntax example:

```

out ("Mutex violation in ", get_unit().full_hdl_path());};

```

5.3.3 e_path()

Purpose	Returns the location of a unit instance in the unit tree
Category	Predefined method for any unit
Syntax	<code>[unit-exp].e_path(): string</code>
Parameters	<i>unit-exp</i> An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed.

Returns the location of a unit instance in the unit tree. This method is used mainly in informational messages.

Syntax example:

```
out("Started checking ", get_unit().e_path());
```

5.3.4 agent()

Purpose	Maps the DUT's HDL partitions into <i>e</i> code
Category	Predefined pseudo-method for any unit
Syntax	<code>keep [unit-exp].agent() == string;</code>
Parameters	<i>unit-exp</i> An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed.
	<i>string</i> One of the following predefined agent names: verilog , vhdl , mti_vlog , mti_vhdl , ncvlog and ncvhdl . Specifying the agent name as verilog or vhdl is preferred because it makes the <i>e</i> code portable between simulators. In contrast, if a unit is bound to a specific agent, for example to mti_vhdl , an error shall be issued if it is ported to NC Simulator. The predefined names are case-insensitive; in other words, verilog is the same as Verilog .

Specifying an agent identifies the simulator that is used to simulate the corresponding DUT component. Once a unit instance has an explicitly specified agent name then all other unit instances instantiated within it are implicitly bound to the same agent name, unless another agent is explicitly specified.

An agent name can be omitted in a single-HDL environment, but it shall be defined implicitly or explicitly in a mixed HDL environment for each unit instance that is associated with a non-empty **hdl_path()**. If an agent name is not defined for a unit instance with a non-empty **hdl_path()** in a mixed HDL environment, an error message shall be issued.

Given the **hdl_path()** and **agent()** constraints, a correspondence map is established between the unit instance HDL path and its agent name. Any HDL path below the path in the map is associated with the same agent unless otherwise specified. This map is further used internally to pick the right adapter for each accessed HDL object.

1 It is possible to access Verilog signals from a VHDL unit instance code and vice-versa. Every signal is mapped to its HDL domain according to its full path, regardless of the specified agent of the unit that the signal is accessed from.

5 NOTE—Agents are bound to unit instances during the generation phase. Consequently, there is no way to map between HDL objects and agents before generation. As a result of this limitation, HDL objects in a mixed Verilog/VHDL environment cannot be accessed before generation from `sys.setup()`.

10 When the `agent()` method is called procedurally, it returns the agent of the unit. The spelling of the agent string is exactly as specified in the corresponding constraint. An unsupported agent name shall cause an error message during the test phase.

15 5.3.5 `get_parent_unit()`

Purpose	Return a reference to the unit containing the current unit instance
Category	Predefined method for any unit
Syntax	<code>[unit-exp].get_parent_unit():</code> unit type
Parameters	<i>unit-exp</i> An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed.

25 Returns a reference to the unit containing the current unit instance.

Syntax example:

```
30 out(sys.unit_core.channels[0].get_parent_unit())
    XYZ_router-@2
```

35 5.4 Unit-related predefined methods for any struct

The predefined methods for any struct include:

- `get_unit()`
- `get_enclosing_unit()`
- `try_enclosing_unit()`
- `set_unit()`

40 5.4.1 `get_unit()`

Purpose	Return a reference to the unit
Category	Predefined method of any struct
Syntax	<code>[exp].get_unit():</code> unit type
Parameters	<i>exp</i> An expression that returns a unit or a struct. If no expression is specified, the current struct or unit is assumed.

45 When applied to an allocated struct instance, this method returns a reference to the parent unit—the unit to which the struct is bound. When applied to a unit, it returns the unit itself.

Any allocated struct instance automatically establishes a reference to its parent unit. If this struct is generated during pre-run generation it inherits the parent unit of its parent struct. If the struct is dynamically allocated by the **new** or **gen** action, then the parent unit is inherited from the struct to which the enclosing method belongs.

This method is useful to determine the parent unit instance of a struct or a unit. It can also be used to access predefined unit members, such as **hdl_path()** or **full_hdl_path()**. To access user-defined unit members, use 5.4.2.

Syntax example:

```
out ("Mutex violation in ", get_unit().full_hdl_path());};
```

5.4.2 get_enclosing_unit()

Purpose	Return a reference to nearest unit of specified type	
Category	Predefined pseudo-method of any struct	
Syntax	[<i>exp.</i>]get_enclosing_unit(<i>unit-type</i> : <i>exp</i>): unit instance	
Parameters	<i>exp</i>	An expression that returns a unit or a struct. If no expression is specified, the current struct or unit is assumed. NOTE—If get_enclosing_unit() is called from within a unit of the same type as <i>exp</i> , it returns the present unit instance and not the parent unit instance.
	<i>unit-type</i>	The name of a unit type or unit subtype.

Returns a reference to the nearest higher-level unit instance of the specified type, so fields of the parent unit can be accessed in a typed manner. The parent unit can be used to store shared data or options such as packing options that are valid for all its associated subunits or structs.

The unit type is recognized according to the same rules used for the **is a** operator, e.g., if a base unit type is specified and an instance of a unit subtype exists, the unit subtype is found. If a unit instance of the specified type is not found, a runtime error shall be issued.

****Add x-ref to the is a operator****

Syntax example:

```
unpack(p.get_enclosing_unit(XYZ_router).pack_config,
      'data', current_packet);
```

5.4.3 try_enclosing_unit()

Purpose	Return a reference to nearest unit of specified type or NULL
Category	Predefined method of any struct
Syntax	[<i>exp.</i>]try_enclosing_unit(<i>unit-type</i> : <i>exp</i>): unit instance
Parameters	<i>exp</i> An expression that returns a unit or a struct. If no expression is specified, the current struct or unit is assumed. NOTE—If try_enclosing_unit() is called from within a unit of the same type as <i>exp</i> , it returns the present unit instance and not the parent unit instance.
	<i>unit-type</i> The name of a unit type or unit subtype.

Like **get_enclosing_unit()** (see 5.4.2), this method returns a reference to the nearest higher-level unit instance of the specified type, so fields of the parent unit can be accessed in a typed manner.

Unlike **get_enclosing_unit()**, this method does not issue a runtime error if no unit instance of the specified type is found. Instead, it returns NULL. This feature makes the method suitable for use in extensions to global methods, such as **dut_error_struct.write()**, which may be used with more than one unit type.

Syntax example:

```
var MIPS := source_struct().try_enclosing_unit(MIPS);
```

5.4.4 set_unit()

Purpose	Change the parent unit of a struct
Category	Predefined method of any struct
Syntax	[<i>struct-exp.</i>]set_unit(<i>parent</i> : <i>exp</i>)
Parameters	<i>struct-exp</i> An expression that returns a struct. If no expression is specified, the current struct is assumed.
	<i>parent</i> An expression that returns a unit instance.

Changes the parent unit of a struct to the specified unit instance.

NOTE—This method does not exist for units because the unit tree cannot be modified.

Syntax example:

```
p.set_unit(sys.unit_core)
```

5.5 Unit-related predefined routines

The predefined routines that are useful for units include **set_config_max()** and **get_all_units()**.

5.5.1 `set_config_max()`

Purpose	Increase values of numeric global configuration parameters	1
Category	Predefined routine	5
Syntax	<code>set_config_max(category: keyword, option: keyword, value: exp [, option: keyword, value: exp...])</code>	10
Parameters	<i>category</i> Is one of the following: cover , gen , memory , and run .	
	<i>option</i> The valid cover option is: <code>absolute_max_buckets</code> . The valid generate options are: — <code>absolute_max_list_size</code> — <code>max_depth</code> — <code>max_structs</code> The valid memory options are: — <code>gc_threshold</code> — <code>gc_increment</code> — <code>max_size</code> — <code>absolute_max_size</code> The valid run option is: <code>tick_max</code> All these options are described in 24.7.1.	15
	<i>value</i> The valid values are different for each option; see 24.7.1.	25

This routine sets the numeric options of a particular category to the specified maximum values.

NOTE—When working a modular verification environment, it is recommended to use `set_config_max()` instead of `set_config()` in order to avoid possible conflicts that can happen in an integrated environment. For example, if two units are instantiated and both of them attempt to enlarge `absolute_max_size`'s configuration value, use `set_config_max()` to access it, so that no unit decrements the value set by another one.

Syntax example:

```
set_config_max(memory, gc_threshold, 100m);
```

5.5.2 `get_all_units()`

Purpose	Return a list of instances of a specified unit type	40
Category	Routine	
Syntax	<code>get_all_units(unit-type: exp): list of unit instances</code>	45
Parameters	<i>unit-type</i> The name of a unit type. The type needs to be defined or an error shall occur.	

This routine receives a unit type as a parameter and returns a list of instances of this unit type, as well as any unit instances contained within each instance.

Syntax example:

```
print get_all_units(XYZ_channel);
```

1

5

10

15

20

25

30

35

40

45

50

55