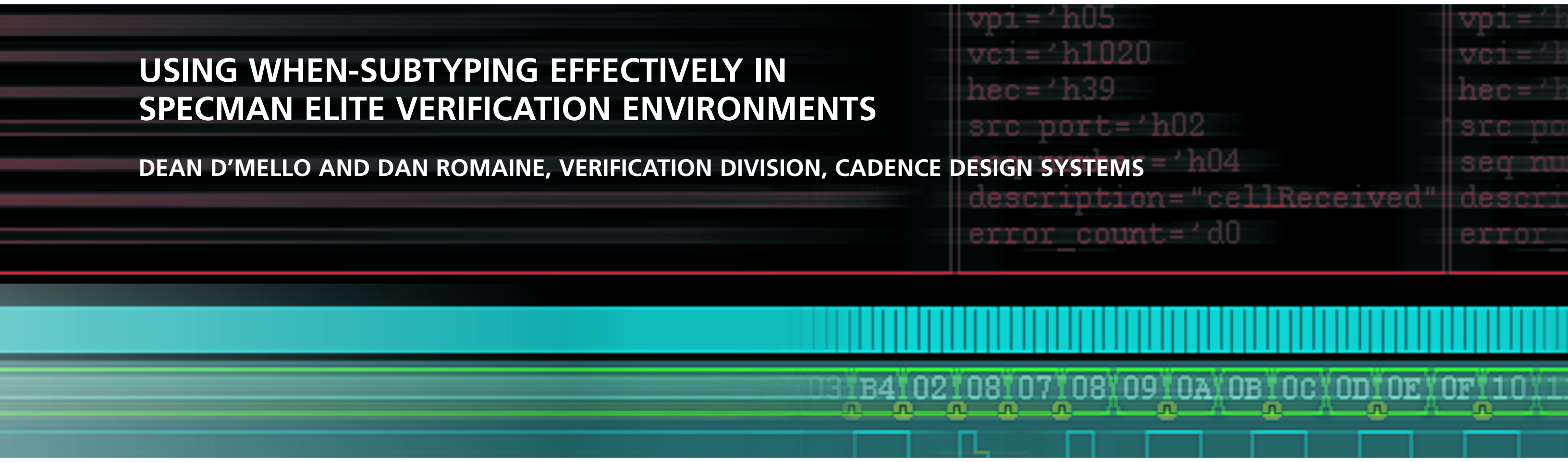
INCISIVE VERIFICATION ARTICLE
MAY 2006



USING WHEN-SUBTYPING EFFECTIVELY IN SPECMAN ELITE VERIFICATION ENVIRONMENTS

DEAN D'MELLO AND DAN ROMAINE, VERIFICATION DIVISION, CADENCE DESIGN SYSTEMS

INTRODUCTION

Developers and users of today's verification environments face huge challenges. The code for these environments must be reusable in multiple simulations for the same or for different designs; adaptable for creating the numerous simulation scenarios required to verify these designs; and easy to use without knowing low-level details of the verification environment. When-subtyping, also known as when-inheritance, is a unique feature of the *e* language that helps verification engineers address these issues and maximize the benefits of Incisive® Enterprise Specman Elite® generation, checking, and coverage engines.

This article will briefly describe when-subtyping and its uses and benefits; offer tips and guidelines for developers and users of Specman Elite environments to help them maximize the benefits of when-subtyping; and address some common pitfalls associated with using this feature.

OVERVIEW OF WHEN-SUBTYPING

Objects in an *e* verification environment are built out of structs and units. Structs and units allow developers to model data structures for generating stimulus and checking the behavior of the design, and are similar to classes in C++ or modules in Verilog®.

When-subtyping provides a means to model different behaviors of these objects without creating new types. A struct or unit can have different struct-members (fields, methods, events; and so on) based on the value of a Boolean or enumerated-type field, known as the "when-determinant" field.

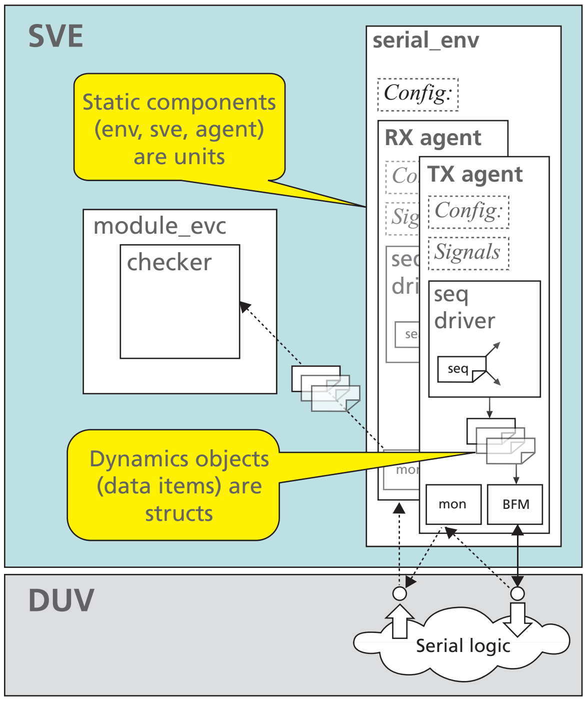
```
struct vehicle_s {
  kind: [AUTOMOBILE, BIKE];
  when AUTOMOBILE vehicle_s {
    num_doors : uint [2..5];
  };
}
```

In this example, the vehicle_s object is defined. Based on the value of the when-determinant kind field, it can conditionally have a data field num_doors. If the kind field has a value of AUTOMOBILE, then the num_doors field exists and can be accessed. When an object of type vehicle_s is instantiated, it can be generated to be any possible subtype of vehicle_s. In other words, it can be an AUTOMOBILE, BIKE, or one of any additional subtypes defined in code extensions after creating the instance.

WHEN-SUBTYPING IN VERIFICATION ENVIRONMENTS

In verification environments, dynamic objects such as stimulus data items (packets, bus transactions, instructions) are implemented as structs, while the static components that comprise the environment (such as agents, bus functional models (BFMs), stimulus sequence drivers, and monitors) are built from units. In this article, we give examples of how when-subtyping can be useful for modeling with structs and units in the verification environment.

To provide context for the discussions that follow, the figure below shows a block diagram of an example verification environment designed according to the *e* Reuse Methodology (*e*RM), in which the simulation verification environment (SVE) is the highest-level unit that encapsulates the verification components used in simulations for the design under verification (DUV).



MODELING VARIATIONS OF DATA ITEMS

One of the most common uses of when-subtyping in Specman Elite environments is to model variations of data items. This provides high-level control knobs to simplify the tasks of test writers as they create code to vary stimulus.

```
extend vehicle_s {
  speed_cap   :[SLOW, AVERAGE, FAST];
  top_speed   :uint;

  when SLOW vehicle_s { keep top_speed in [20..50]; };
  when AVERAGE vehicle_s { keep top_speed in [40..90]; };
  when FAST vehicle_s { keep top_speed in [60..150]; };
};
```

Continuing the vehicle code example presented earlier, in the code example above, the speed_cap field is a control knob for the top_speed field. In this case, the knowledge of top_speed has been abstracted away for easy use of the vehicle_s object. The code below shows how the speed_cap knob is used to get SLOW or FAST vehicles only in a specific simulation.

```
extend vehicle_s {
  keep speed_cap in [SLOW, FAST];
};
```

An additional note on syntax: the code below shows an alternate way of specifying subtypes. You will often see when-subtyping coded as shown below; it is the recommended way to code conditional struct members when a reader's understanding of the code does not require viewing code for different subtypes together. Use the when-block style shown above in cases where keeping together code for different subtypes will help readers of the code (for physical fields and method extensions).

```
extend vehicle_s {
  speed_cap   :[SLOW, AVERAGE, FAST];
  top_speed   :uint;
};

  extend SLOW vehicle_s {
    keep top_speed in [20..50];
};

// Similar extensions for AVERAGE, FAST vehicles
```

CONFIGURING VERIFICATION ENVIRONMENTS

When-subtyping also provides a way of modeling verification environment components that are required to behave differently in different simulations. For example, a verification environment might require an active driving component or a passive monitor for differing device architectures. By building in when-subtypes, the same code can be used for both cases, reducing duplication of code. (The *e*RM architecture standard requires that *e* verification components (*e*VCs) provide an ACTIVE/PASSIVE when-determinant field.)

This same concept can be applied to enable/disable checking or coverage in all or some of the environment, or to configure individual instances of an *e*VC to behave differently. The benefit of this approach is that a user of the *e*VC can choose the environment they would like to simulate by controlling these when-determinant fields.

In the code example below, several uses of subtyped verification environment components are demonstrated.

```
// provided in an earlier file
extend vr_pcie_env_name : [LINK_0, LINK_1];

extend sys {
  link_0 : vr_pcie_single_link_env is instance;
    keep link_0.name == LINK_0;

  link_1 : vr_pcie_single_link_env is instance;
    keep link_1.name == LINK_1;
};

// Simulation config code
extend LINK_0 vr_pcie_single_link_env {
  keep has_active_rc;
  keep has_passive_ep;

  keep env_config.has_pl;
  keep env_config.has_dll;
  keep env_config.has_tl;
};
```

First, the basic environments—instantiated under sys—are subtyped using appropriate names. This enables instantiating multiple vr_pcie_single_link_env units under sys and controlling them independently, as well as implementing different constraint blocks or procedural code unique to each of the subtypes. A user can extend the LINK_0 vr_pcie_single_link_env and write code that applies only to instances of that subtype. This is demonstrated with the constraints in the subtype extension.

Second, several Boolean when-determinant fields are constrained to affect the verification component being built inside this vr_pcie_single_link_env through when-subtypes of other components of the environment. Note that no inheritance tree is developed (just descriptions of when-inherited functionality) and the types of the instances are not being changed; rather, through when-subtyping, the behavior of existing instances is being modified.

DEFINING AND USING STIMULUS SEQUENCES

Another common use of when-subtyping in verification environments is in the creation and use of sequences of stimulus using *e*RM sequences. An environment developer uses the sequence capability of the *e*RM evc_util library to create a sequence struct with a when-determinant field and a few pre-defined subtypes. Sequences of stimuli are created by implementing additional subtypes of the sequence struct. The pre-defined and user-defined subtypes are then used to implement stimulus scenarios that use the power of the Specman generator with a standard user interface for specifying constraints as well as sequential behavior. This is an example of an *e*RM sequence:

```
extend vr_pcie_tl_seq_kind: [SMALL_PACKET];

extend SMALL_PACKET vr_pcie_tl_seq {
  !req_pkt : MWR MEM REQ vr_pcie_tl_pkt;

  body() @driver.clock is {
    do req_pkt keeping {
      .tlp_data.size() == 1;
      .first_addr == 0xff;
    };
  };
};
```

In this example, we have created a new when-subtype of the existing sequence struct vr_pcie_tl_seq called SMALL_PACKET. The **body**() time-consuming method (TCM) of this new sequence sends a single packet by activating a data-item field. To use this subtype, any sequence field can be generated to be a SMALL_PACKET sequence. Sequence fields in pre-existing code will not need any additional modification to use the new sequence, as traditional object-oriented inheritance would require. Subtyping enables the *e*RM sequence implementation to provide a standard user interface for creating stimulus for all interfaces of the DUV.

A COMMON PITFALL: ACCESSING CONDITIONAL STRUCT-MEMBERS

As seen in the previous sections, when-subtyping provides a powerful means to model and use stimulus objects as well as verification environment components.

New and advanced users of Specman Elite occasionally run into difficulties when coding and using when-subtypes. In this section, we look at the most common pitfall—accessing conditional struct members—and recommend some techniques for avoiding and addressing the issue. The code example below continues the vehicle example presented earlier and shows the error resulting from an attempt to access the conditional field num_doors.

```
extend sys {
  vehicle : vehicle_s;
  keep vehicle.kind == AUTOMOBILE;
  keep vehicle.num_doors == 3;
};

Loading vehicle.e ...

  *** Error: 'vehicle' (of type 'vehicle_s')
does not have 'num_doors' field though its
subtypes do.
To access the field use 'is a' or 'as_a':
For example: vehicle.as_a(AUTOMOBILE vehicle_s).num_doors
         at line 63 in vehicle.e
  keep vehicle.num_doors == 3;
```

Here are a couple of ways to address this issue, depending on the user's intent:

a) Whenever possible, declare fields (or variables) as a specific subtype, as shown in the code below. This allows accessing conditional fields without additional code and is likely the most appropriate solution when the when-determinant field is constrained to be a single value, as in the code presented above.

```
extend sys {
  vehicle : AUTOMOBILE vehicle_s;
  keep vehicle.num_doors == 3;
};
```

b) If you want the generator to select other values for the when-determinant field (resulting in different subtypes each time the field is generated), then other solutions might be preferable. In the alternate code example below, extending the specific subtype of the struct allows accessing its conditional fields without additional code. In this case the vehicle field of sys can be either AUTOMOBILE or BIKE, but all AUTOMOBILE vehicles will be generated with num_doors = 3.

```
extend sys {
  vehicle : vehicle_s;  // can be AUTOMOBILE or BIKE
};

extend AUTOMOBILE vehicle_s {
  keep num_doors == 3;
};
```

In the next section, we look at some additional techniques for addressing the issue of accessing conditional struct members for cases where the above solutions might not apply.

SOME GENERAL GUIDELINES FOR ADDRESSING THE ISSUE

1. To minimize the likelihood of issues with accessing conditional struct members, it is recommended to implement as conditional struct members only those fields, methods, and events that are truly specific to each subtype. Note that partial or default definitions can be placed in the base struct or unit definition and modified appropriately in subtype-specific extensions, allowing subtype-specific behavior without conditional struct members, as shown below for the sound_horn() method, which can be invoked for any field of type vehicle_s, regardless of its subtype.

```
struct vehicle_s {
  kind   : [AUTOMOBILE, BIKE];

  sound_horn() is empty;  // default definition

  when AUTOMOBILE vehicle_s {
    sound_horn() is only {  // subtype-specific
    extension
      out("BEEP, BEEP!");
    };
  };

  when BIKE vehicle_s {
    sound_horn() is only {
      out("RING, RING!");
    };
  };

};
```

2. When using sequences, a special capability is provided to alleviate the issue of accessing conditional fields. It allows declaring a single field of the base sequence or data-item struct, then activating the same field as one of its subtypes using the "**do**" action, and freely constraining the conditional fields in the **keeping** block. So, the sequence example introduced earlier could be rewritten as:

```
extend vr_pcie_tl_seq_kind: [SMALL_PACKET];

  extend SMALL_PACKET vr_pcie_tl_seq {
    !req_pkt : vr_pcie_tl_pkt;

      body() @driver.clock is {
        do MWR MEM REQ req_pkt keeping {
          .tlp_data.size() == 1;
          .first_addr == 0xff;
        };
      };
  };
```

Specifying the subtype (MWR MEM REQ) in the "**do**" action allows the conditional field first_addr to be accessed in the keeping block even though the field being activated is of the base struct type. The ability to specify the subtype in the "**do**" action allows activating any subtype of the sequence (or its data item) without needing to pre-declare fields of every subtype or adding typecasting code. Without the subtype name (when-determinant values) after the "**do**" action, this code would produce the same "Does not have field…" error message as above.

3. The techniques described above can help in many cases, but several cases remain where it is advantageous to have an instance or variable of the base type and generate it as the required subtype using constraints. A classic example is the instantiation of an agent unit in a verification environment, which is generated as ACTIVE or PASSIVE according to constraints in a configuration file. In such cases, some form of typecasting is required to access conditional struct members in constraints and procedural code, and it is here that Specman users often run into difficulty.

   The *e* language provides two constructs to help with typecasting of subtypes: "**is a**" and "**as_a()**." The "**is a**" construct can be used to create a Boolean expression to test the subtype of a struct expression and optionally perform an automatic typecast to access conditional struct members. The "**as_a()**" pseudo-method attempts to perform a typecast of a struct expression and returns NULL if the struct expression does not match the requested subtype.

   The "**as_a()**" pseudo-method will affect generation order when used in constraints, causing unintended issues when simply trying to access conditional struct members. Additionally, there is a need to test its return value to ensure that it is not NULL, or other runtime errors might result. For these reasons it is recommended to use "**is a**" for handling when-subtypes whenever possible, and "**as_a()**" for type conversions between scalar and list types. The generation order is not affected by "**is a**" and procedural code implemented using "**is a**" will typically be more elegant and maintainable than the equivalent functionality implemented using "**as_a()**."

   Here is an example of how to use the "**is a**" construct in both a constraint and in procedural code:

```
unit bfm_u {
  interface : [MII, GMII];
  rerun() is {
    //...
  };
};

unit agent_u {
  active_passive : erm_active_passive_t //
[ACTIVE, PASSIVE];
  when ACTIVE agent_u {  // bfm is in the
active-subtype
    bfm : bfm_u is instance;
  };
};

unit env_u like any_env {

  agent : agent_u is instance;

  // Constraint Example using "is a" to access
  conditional-field
  keep agent is a ACTIVE agent_u (a) =>
a.bfm.interface == GMII;

  event reset;
  on reset {  // rerun the BFM at each reset
    // Procedural Example using "is a" to access
    cond field
    if agent is a ACTIVE agent_u (active_agent)
{
      active_agent.bfm.rerun();
    };
  };

};
```

In this example, at the env_u level of hierarchy, a constraint has been written for the bfm_u interface field. Since the BFM is only instantiated in the ACTIVE subtype of the agent_u, and for reuse purposes the agent field cannot be declared as a specific subtype, we need some additional syntax to get access to the BFM. Using the "**is a**" construct in an implication constraint and specifying an identifier (a) provides a typecasted identifier, allowing conditional struct members of the ACTIVE subtype to be accessed within the scope of the constraint.

Upon detecting a reset at the env level, we want to call the BFM's predefined **rerun**() method. Again, we need to access a conditional struct member of the ACTIVE subtype, this time from procedural code. The use of the "**is a**" construct with the automatic typecast allows this to be implemented in a manner that is elegant and easier to maintain than "**as_a**()," especially if there is a need for multiple accesses to conditional struct members. This style also allows the handling of exceptions in an optional else block (not shown in this example).

CONCLUSION

When-subtyping helps implement powerful verification environments that are adaptable for creating different simulation scenarios and usable with minimal knowledge of the low-level implementation details of the code.

This article described the unique capability of when-subtyping in the *e* language, discussed the benefits it provides for verification in the modeling and use of Specman environments, and provided some tips for using when-subtyping effectively. There are several other aspects of when-subtyping that might be interesting to readers. The section below identifies some sources of additional information in Specman Elite documentation.

We invite readers to provide feedback on this article and initiate follow-up discussions on the *e* discussion forum on the Cadence user website: cdnusers.org

WHERE TO GET MORE INFORMATION

The following sections of Specman Elite documentation might be interesting to users and developers of Specman environments who would like to learn more about when-subtyping. Section numbers refer to the documentation provided with the Specman Elite 5.0.2 release.

In the *e* Language Reference Version 5.0.2:

- 2.14.1 is [not] a
- 3.1.5 Struct Subtypes
- 3.1.5.1 Referring to a Struct Subtype
- 3.1.5.2 Using Extend, When, and Like with Struct Subtypes
- 3.1.5.3 Referring to Conditional Fields in When-Constructs
- 3.8.2 as_a()
- 4.7 Creating Subtypes with When
- 4.8 Extending When-Subtypes
- 4.8.1 Coverage and When-Subtypes
- 4.8.2 Extending Methods in When-Subtypes
- 4.10 Comparison of When and Like Inheritance

IN THE *e* REUSE METHODOLOGY (*e*RM) DEVELOPER MANUAL, VERSION 2.1:

- 5.8.1 Specifying Subtype in Do Actions