Modelling e Verification Language Testbenches inUML 2.0 with Theme and MARTE

Éamonn Linehan
Lero – The Irish Software Engineering Research Centre
Trinity College Dublin, Ireland

Siobhán Clarke
Lero – The Irish Software Engineering Research Centre
Trinity College Dublin, Ireland

23rd August 2010

Contact

Address	Lero International Science Centre University of Limerick Ireland
Phone	+353 61 233782
Fax	+353 61 213036
E-Mail	info@lero.ie
Website	http://www.lero.ie/

This work is partially supported by Science Foundation Ireland
under grant no. 03/CE2/I303-1

Version History

Version 3.0 (23 August 2010)

- Reformatted for inclusion in Lero technical report series.
- Introduction written for document.

Version 2.0 (13 July 2010)

- Added documentation for additional profile elements (Macro, Comment & Load order)

Version 1.0 (29 March 2010)

- First version of document complete.

Abstract

This document presents a metamodel for the *e* Hardware Verification Language through example. The *e* metamodel is captured as a UML 2 profile (UML's mechanism for lightweight extension). Stereotypes from this profile can be used at the modeling level as annotations on models of hardware verification testbenches.

This document is organised into sections corresponding to the main constructs of the *e* Hardware Verification Language. For each of these constructs we present:

1) a snippet of code showing their usage;
2) a UML model showing the profile elements;
3) a sample model annotated with the stereotypes from the profile (corresponding to the code snippet);
4) and an excerpt from the XPAND code generation templates that converts the UML model back into *e* source code.

This document concludes with an overview of the specific extensions to the use of Theme/UML required to accommodate the aspect-oriented constructs in e.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current publications and the latest revision of this technical report can be found in the Lero Technical Report Archive at http://www.lero.ie/.

This document is a stable document and may be used as reference material or cited from another document.

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre (www.lero.ie)

Related Documents

Darren Galpin and Cormac Driver and Siobhán Clarke, *Modelling Hardware Verification Concerns Specified in the e Language: An Experience Report*, International Conference on Aspect-Oriented Software Development (AOSD) Industry Track, Charlottesville, Virginia, USA, 2009, mar, TCD-CS-2009-08 , https://www.cs.tcd.ie/publications/tech-reports/reports.09/TCD-CS-2009-08.pdf

Cormac Driver and Vinny Cahill and Siobhán Clarke, *Separation of Distributed Real-Time Embedded Concerns with Theme/UML*, The Fifth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES), Budapest, Hungary, 2008, Ricardo J. Machado and Joao M. Fernades and Flavio R. Wagner and Rick Kazman, pp. 27--33, apr, IEEE Computer Society, ISBN 0-7695-3104-0, TCD-CS-2008-40, https://www.cs.tcd.ie/publications/tech-reports/reports.08/TCD-CS-2008-40.pdf

References

Object Management Group Inc. (OMG), *Unified Modelling Language (UML)*, Version 2.0, June 2003, http://www.uml.org/#UML2.0

Object Management Group Inc. (OMG), *A UML Profile for MARTE: Modeling and analysis of real-time embedded systems*, beta 2, ptc/08-06-09, June 2008, http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf

Verisity Design, Inc, *e Language Reference Manual*, online; Accessed 8th February 2010, February 2002, http://www.ieee1647.org/downloads/prelim_e_lrm.pdf

IEEE Computer Society, *IEEE standard for the Functional Verification Language e*, Standard IEEE Std 1647-2008, IEEE, NY, USA, August 2008

Table of Contents

1 Profile Relationships

The *e* UML profile inherits features from both Theme/UML and the OMG UML profile for Modeling and Analysis of Real-time and Embedded systems (MARTE). MARTE is a UML profile that supports specification of real-time and embedded systems. In addition to functional design, this profile adds constructs to describe the hardware and software (for example, OS services) resources and defines specific properties to enable designers to perform timing and power consumption analysis. MARTE packages features into individual sub-profiles allowing one to import only the parts of the profile that are required. The *e* profile extends and reuses elements from the Time and Value Specification Language (VSL). These constructs are reused to specify constructs such as concurrency and synchronisation and to attach quality attributes and model simulator clocks.

The *e* profile itself is divided into three packages: the Core package contains model elements corresponding to *e* language constructs; the Verilog package; and the VHDL package contain simulation related constructs (statements or unit members that expose functionality simulator).

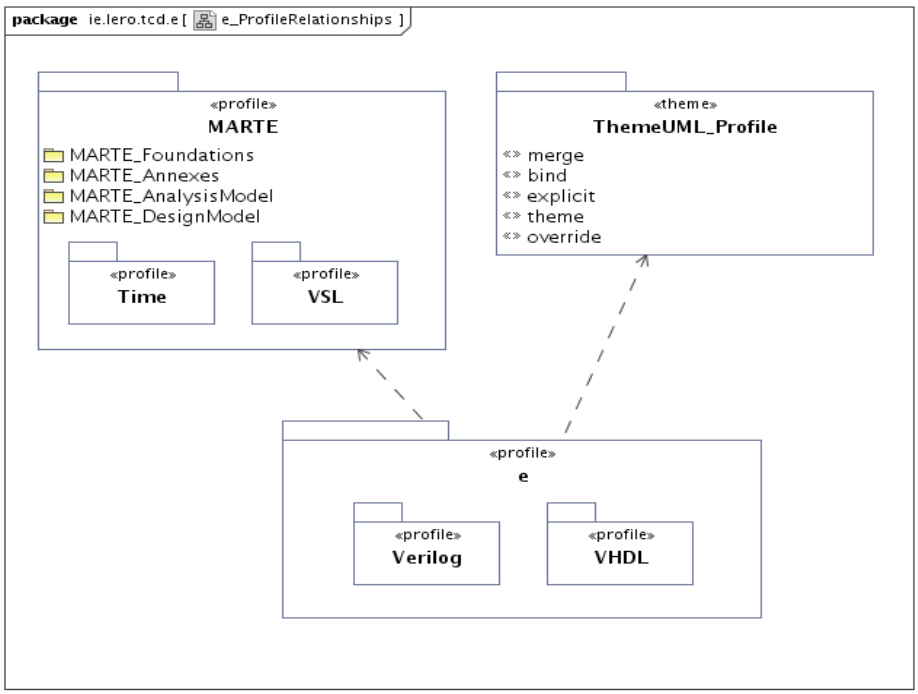

Figure 1 - UML 2.0 Profile Relationships

2 Structure and Code Organisation

Code organisation in *e* is not specified as part of the standard but it is common for files to contain a single code segment containing a module or modules related to a single concern where a 'concern' is the most important way of organising and viewing code. This concept is based on the work of Robinson, “Aspect-Oriented Programming with the *e* Verification Language”, Chapter 3: Using AOP to Organise your Code, pp. 67.

2.1 Sample *e* Code

```
<'
        import verification.e;

        struct Packet {

        };

'>
```

2.2 UML Profile Elements

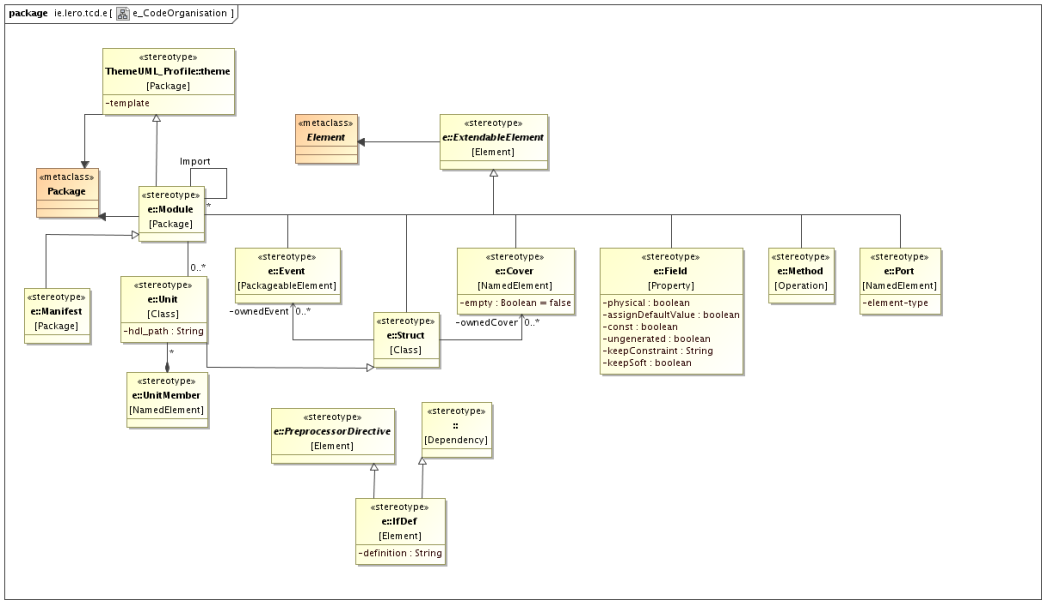

Figure 2 - Primary Structural Profile Elements

2.3 Sample Model

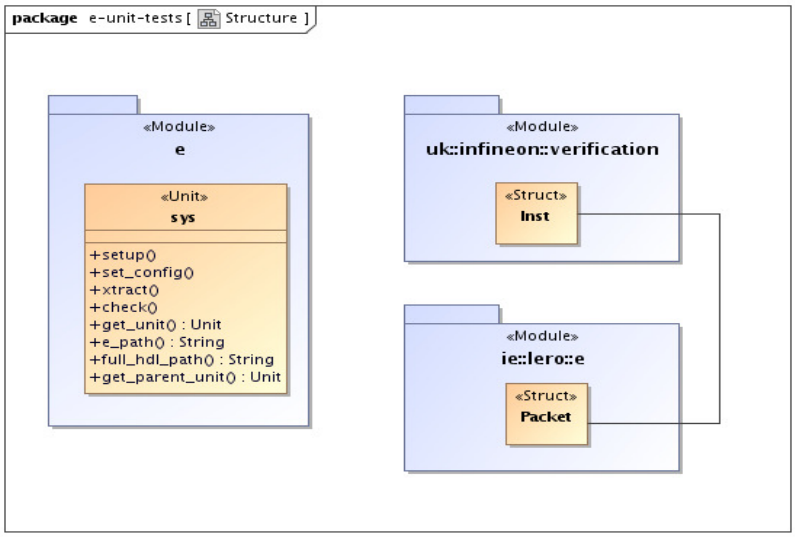

Figure 3 - Sample model demonstrating use of structural profile elements.

2.4 Code Generation

```
«DEFINE PackageRoot FOR e::Module»
  «FILE fileName(this)»
/*******************************************************************************
* PROJECT:   «this.getModel().name»
* AUTHOR:    Eamonn Linehan
* DATE:      «getModuleHeaderTime()»
*
* MODULE:    «fileName(this)»
*******************************************************************************/
<'
        «EXPAND Import FOREACH getRelatedModules(this)»

        «EXPAND Types FOREACH ownedType.typeSelect(uml::DataType)»
        «EXPAND Struct FOREACH ownedType.typeSelect(e::Struct)»
        // End Module «name».e
'>
  «ENDFILE»
«ENDDEFINE»
```

3 Load Order

The order that import statements appear in a module determine the order in which those files are loaded into the execution environment. The profile includes elements that allow this order to be captured in the model.

3.1 Sample *e* Code

```
import ../sri_evc/sri_evc_types.e;
import ../sri_evc/sri_evc_units.e;
import ../sri_evc/sri_evc_trans.e;
import ../sri_evc/sri_evc_bus;
import ../sri_evc/sri_evc_log;
import ../sri_evc/sri_evc_slave_agent;
import ../sri_evc/sri_evc_error;
import ../sri_evc/sri_evc_macros;
```

3.2 UML Profile Elements

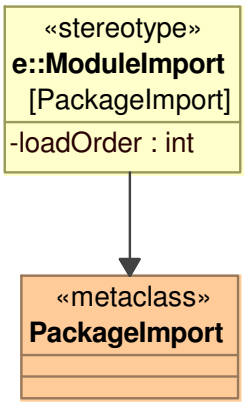

Figure 4 - Module Import Profile Elements

3.3 Sample Model

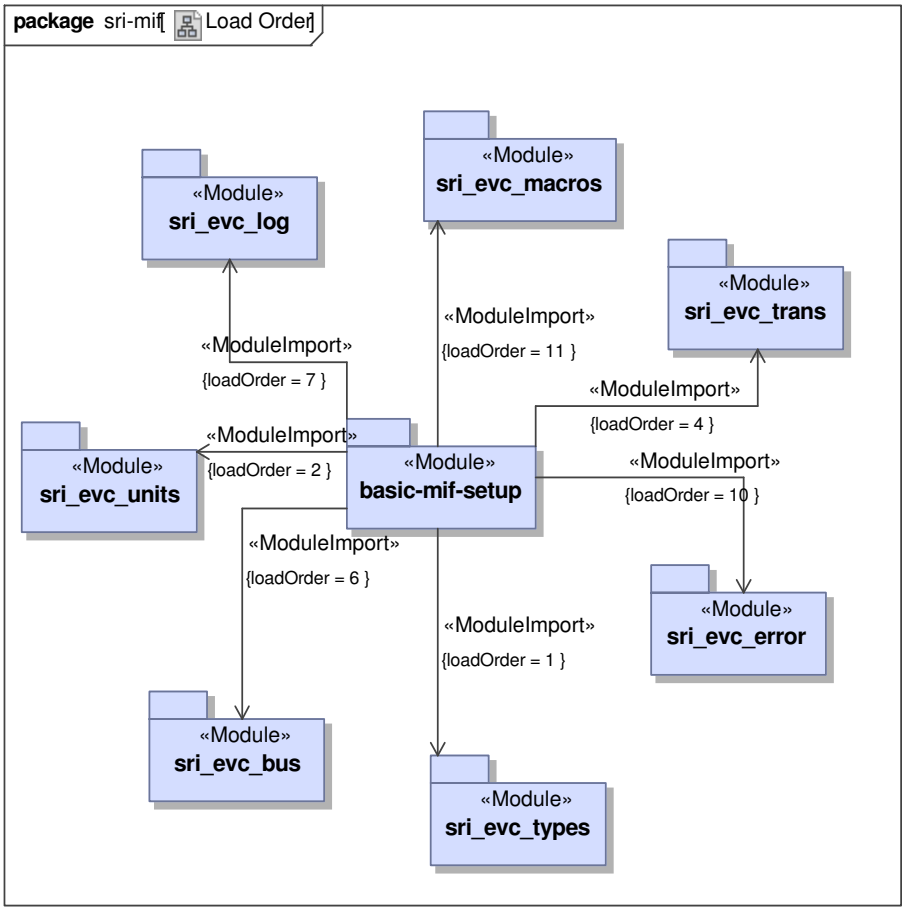

Figure 5 - Specifying Load Order.

3.4 Code Generation

```
«EXPAND Import FOREACH (List[e::Module]) getRelatedModules(this)»

«REM» Each module relationship should result in an import statement «ENDREM»
«DEFINE Import FOR uml::Package»«IF hasStereotypeApplied(this, e::Module)»
        import «fileName((e::Module)this)»;«ENDIF»
«ENDDEFINE»
```

A Java extension enforces the defined order by returning a list ordered by loadOrder.

```
        // Sort ModuleImport objects by load order
        Collections.sort(importedModules, new Comparator<ModuleImport>() {
                @Override
                public int compare(ModuleImport mi0, ModuleImport mi1) {
```

```
            return mi0.getLoadOrder() - mi1.getLoadOrder();
        }
    });
```

4 Data Types

The *e* language has a number of predefined data types including the integer and Boolean scalar types common to most programming languages. In addition, you can create new scalar data types (enumerated types) that are appropriate for programming, modeling hardware, and interfacing with hardware simulators. The *e* language also provides a powerful mechanism for defining object-oriented hierarchical data structures (structs) and ordered collections of elements of the same type (lists).

The following numeric and boolean primitive types are modelled as data types in the *e* profile.

Type Name	Function	Size
int	Represents numeric data, both negative and non-negative integers.	32 bits
uint	Represents unsigned numeric data, non-negative integers only.	32 bits
bit	An unsigned integer in the range 0–1.	1 bit
byte	An unsigned integer in the range 0–255.	8 bits
time	An unsigned integer in the range 0–263-1.	64 bits
bool	Represents truth values, TRUE(1) and FALSE(0).	1 bit

4.1 Scalar Subtypes

You can create a scalar subtype by using a scalar modifier to specify the range or bit width of a scalar type. You can also specify a name for the scalar subtype if you plan to use it repeatedly.

4.1.1 Sample *e* Code

```
<'

    type int_count: [0..99] (bits: 7);

    struct Counter {
        count: int_count;
    };

'>
```

4.1.2 UML Profile Elements

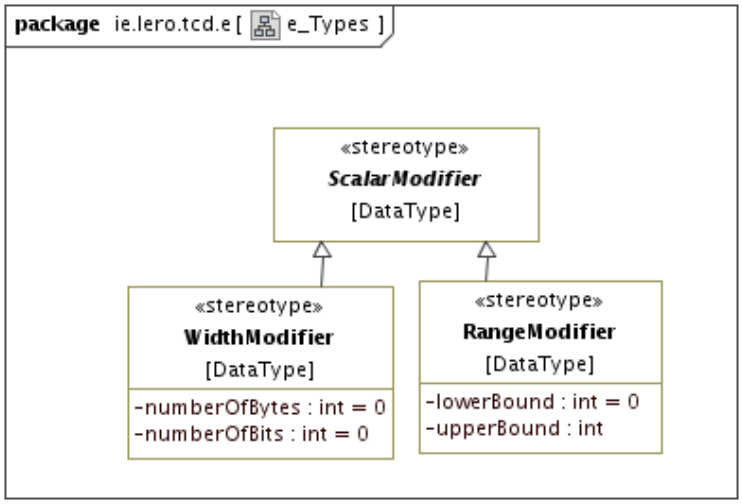

Figure 6 - Scalar Modifier Profile Elements

4.1.3 Sample Model

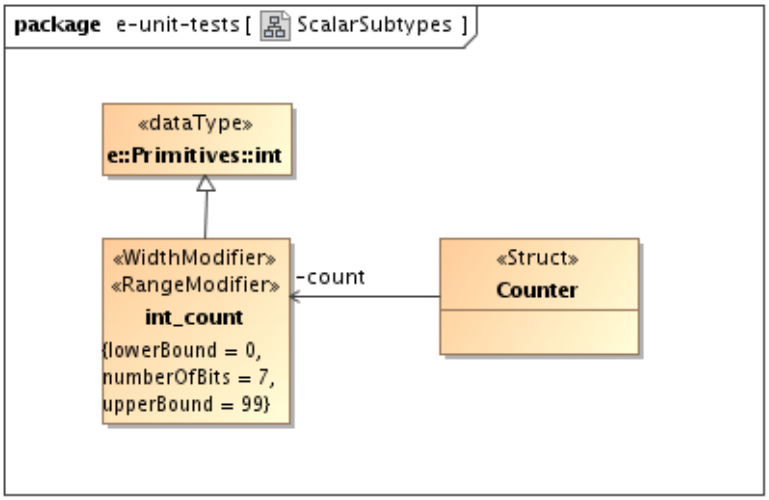

Figure 7 - Example usage of scalar modifiers

4.1.4 Code Generation

```
«DEFINE Types FOR uml::DataType»
        type «this.name»: «getScalarModifiers(this)»;
«ENDDEFINE»
```

4.2 Enumerated Types

You can define the valid values for a field as a list of symbolic constants.

4.2.1 Sample *e* Code

```
<'
        type  NetworkType :[ IP=0x0800, ARP=0x8060 ] (bits:16);
        type PacketKind:  [ETH, ATM](bits: 8);

        struct Packet {
                kind: PacketKind;
        };
'>
```

4.2.2 UML Profile Elements

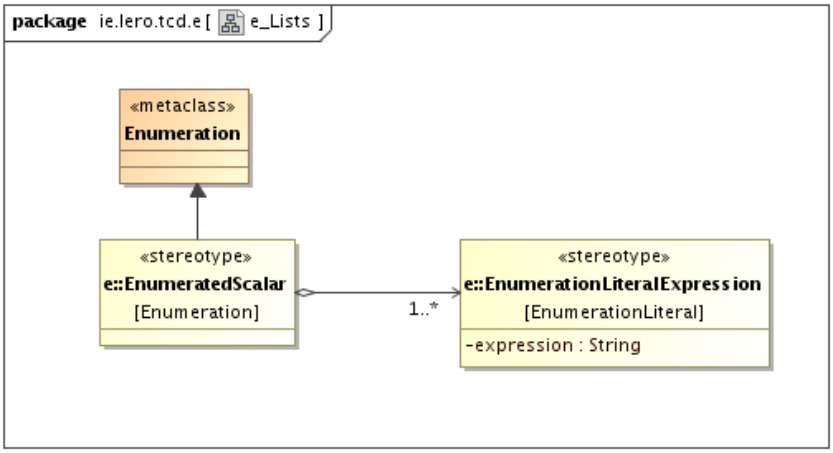

Figure 8 - Enumerated Type Profile Elements

4.2.3 Sample Model

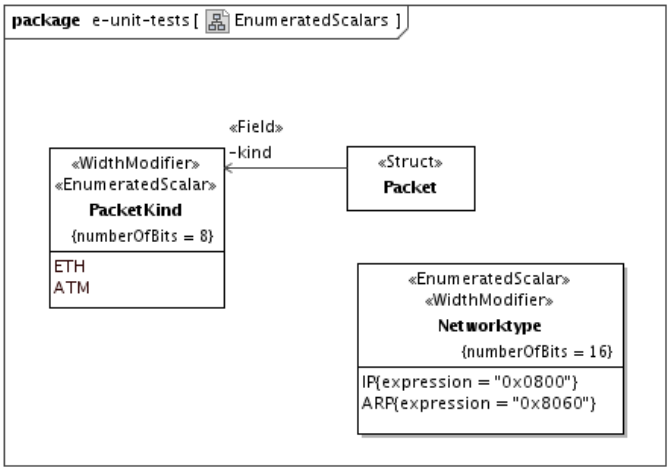

Figure 9 - Modelling using enumerated types.

4.2.4 Code Generation

```
«DEFINE Types FOR uml::DataType»
        type «this.name»:
        «IF hasStereotypeApplied(this, e::EnumeratedScalar)»
                [«getEnumerationLiterals(this)»]
        «ENDIF»
        «getScalarModifiers(this)»;
«ENDDEFINE»
```

5 Structs

Note: Struct & Method definitions omitted as they do not change from standard UML except for the addition of e::Struct and e::Method Stereotypes to uml::Class and uml::Operation.

5.1 Like Inheritance

Like inheritance is the classical, single inheritance familiar to users of all object-oriented languages and is specified with the like clause in new struct definitions.

5.1.1 Sample *e* Code

```
        <'
        import uk/infineon/verification.e;

        struct Packet {
```

```
    kind: PacketKind;
    len: int;
};

struct TxPacket like Packet {
        keep len > 10;
};

'>
```

5.1.2 Sample Model

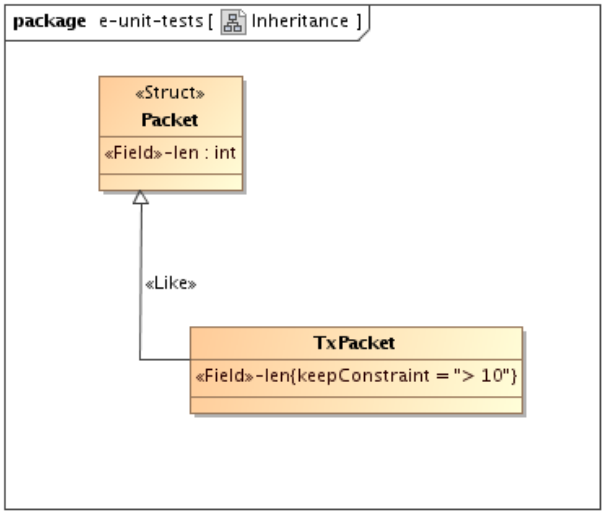

Figure 10 - Like Inheritance Example

5.1.3 Code Generation

```
struct «name» «IF hasLikeInheritance(this)»like
«getLikeInheritanceParent(this).name»«ENDIF» { ...};
```

5.2 When Inheritance

When inheritance is a concept unique to *e* and is specified by defining subtypes with when struct members. When inheritance provides the following advantages compared to like inheritance:

- Ability to have explicit reference to the when fields
- Ability to have multiple, orthogonal subtypes
- Ability to extend the struct later

5.2.1 Sample *e* Code

```
struct packet {
```

```
        good: bool;
        when FALSE'good packet {
                pkt_msg() is {
                        out("bad packet");
                };
        };
};
```

5.2.2 UML Profile Elements

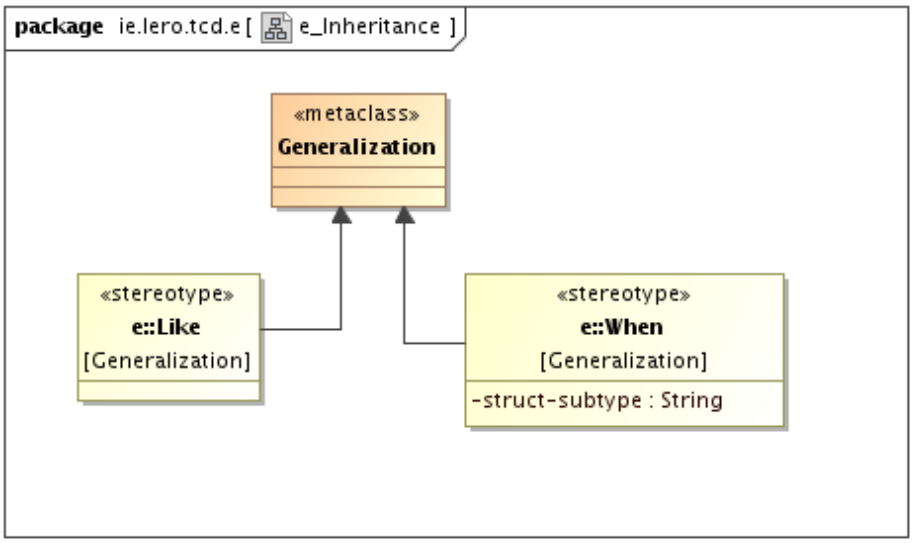

Figure 11 - When Inheritance Profile Elements

5.2.3 Sample Model

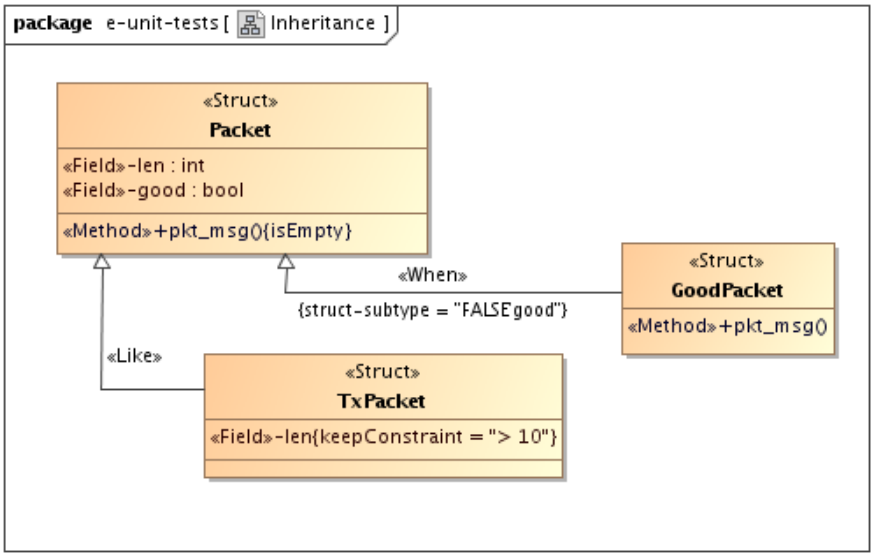

Figure 12 - Modelling When Inheritance

5.2.4 Code Generation

```
«DEFINE When FOR e::When»when «this.structsubtype» «getBaseStruct().name» {
        «EXPAND Operations FOREACH
getSpecificStruct().allOwnedElements().typeSelect(e::Method)»
                };
«ENDDEFINE»
```

6 Events

All *e* temporal language features depend on the occurrence of events, which are used to synchronize activity within the simulation environment.

6.1 Sample *e* Code

```
event sim_ready is change('top.ready') @sim;
```

6.2 UML Profile Elements

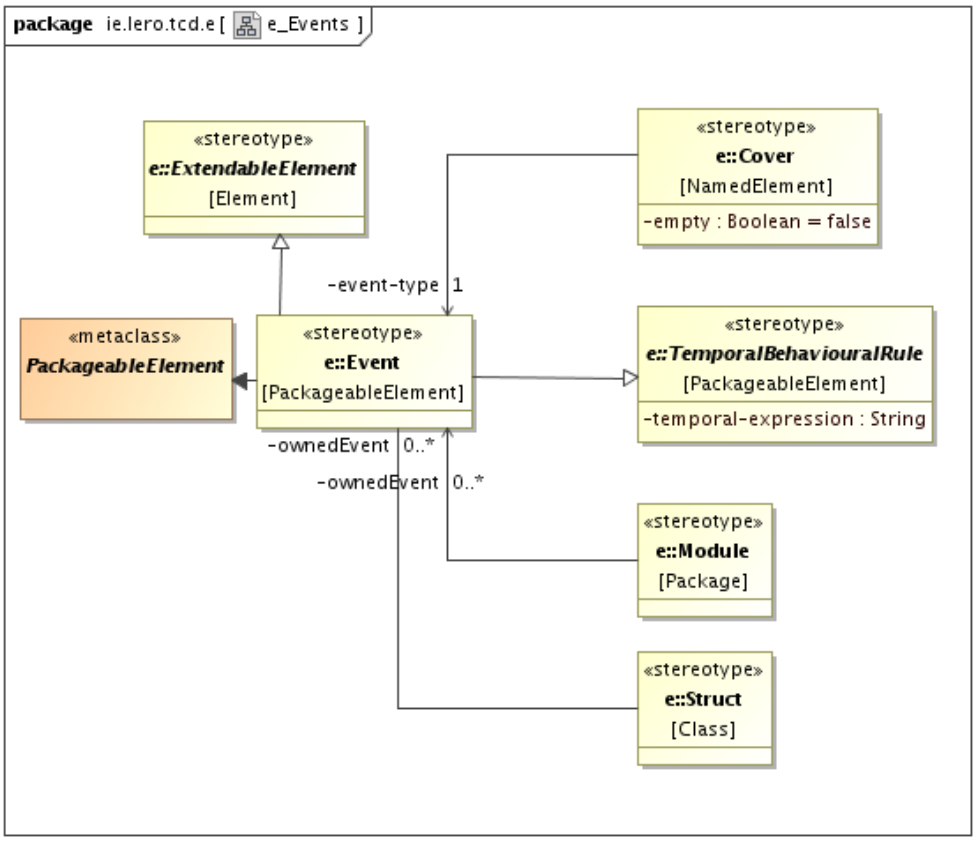

Figure 13 - Event Profile Elements

6.3 Sample Model

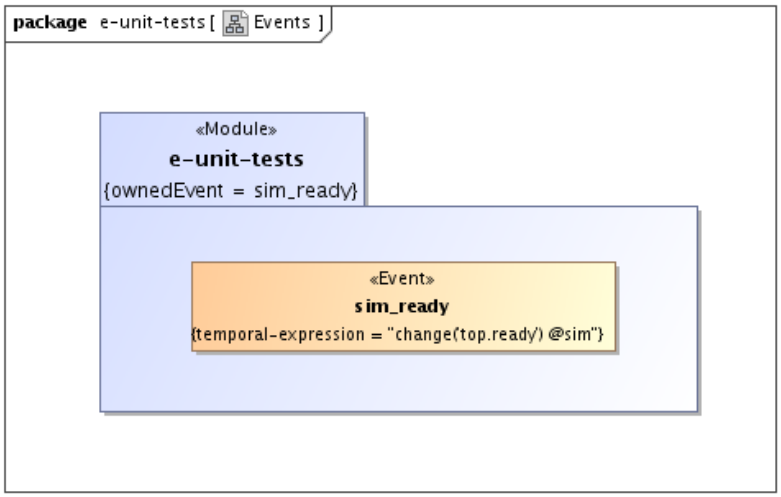

Figure 14 - Example addition of event to a module.

6.4 Code Generation

```
«DEFINE Event FOR e::Event»
        event «this.name» is «this.temporalexpression»;
«ENDDEFINE»
```

7 Coverage

A coverage group is struct member that contains a list of data items for which data is collected over time. Once coverage items have been defined in a coverage group, you can use them to define special coverage group items called transition and cross items. Coverage groups can be extended.

7.1 Sample *e* Code

```
struct Packet {
        len: int;
        event sim_ready is change('top.ready') @sim;
        cover sim_ready is {
                item len;
        };
};
```

7.2 UML Profile Elements

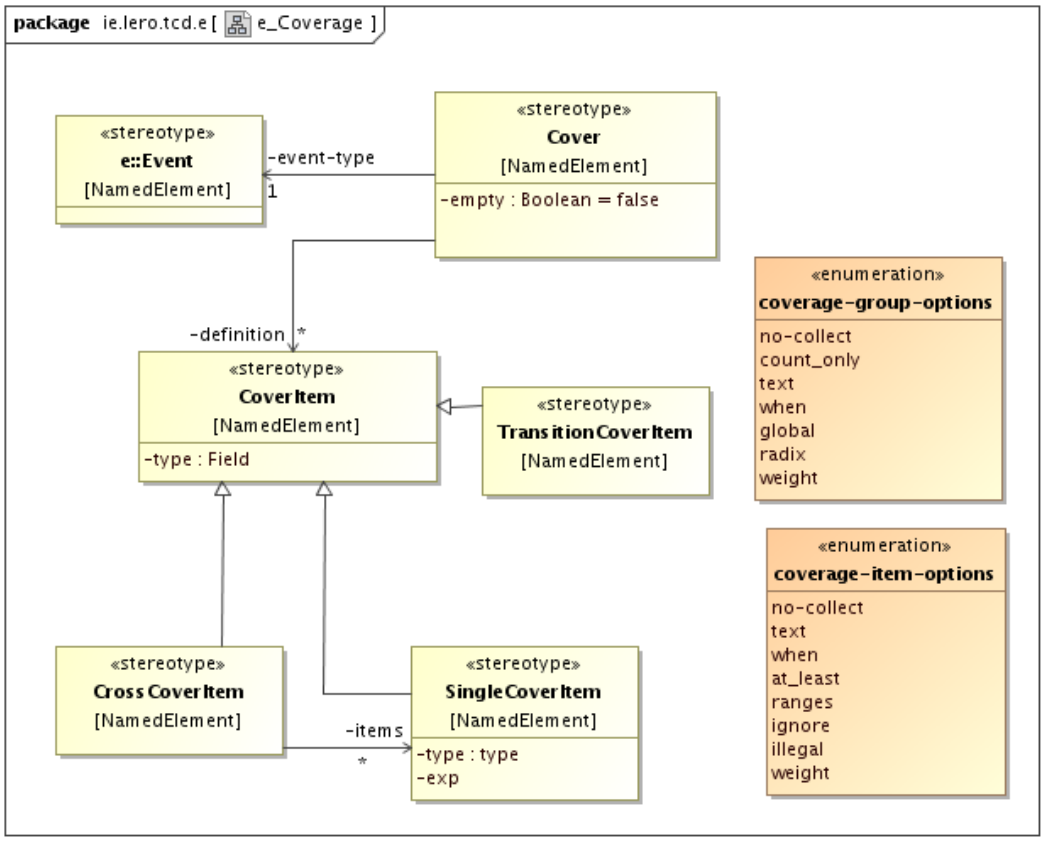

Figure 15 - Coverage Profile Elements

7.3 Sample Model

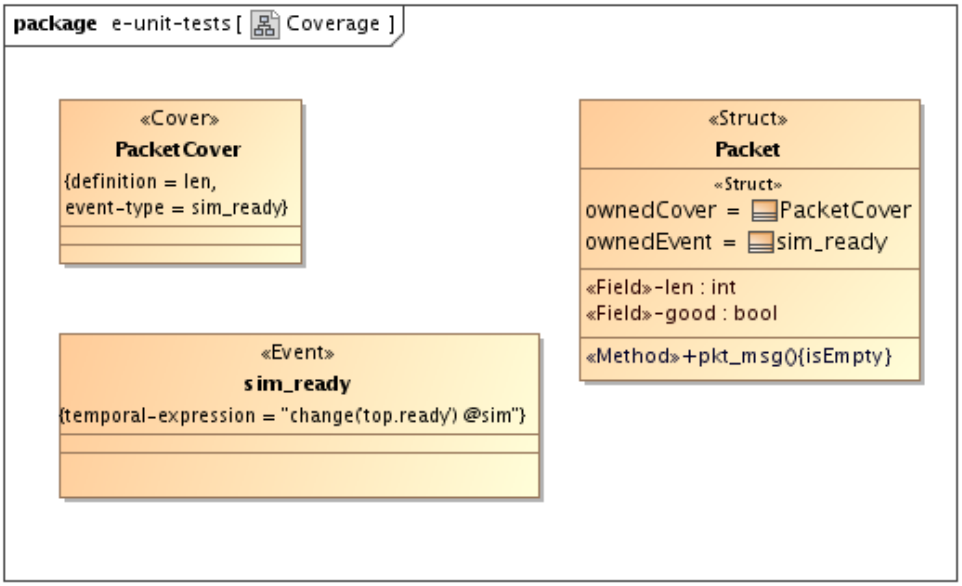

Figure 16 - Modelling of Coverage

7.4 Code Generation

```
«DEFINE Cover FOR e::Cover»
                cover «this.eventtype.name» is {«EXPAND CoverItem FOREACH
this.definition.typeSelect(e::CoverItem)»
                }
«ENDDEFINE»

«DEFINE CoverItem FOR e::CoverItem»
                        item «this.name»;
«ENDDEFINE»
```

8 Aspect-Oriented Extensions

8.1 Method Extension

Methods defined in one module can later be overwritten, modified or enhanced in subsequent modules using the extend mechanism. This mechanism replaces or extends the action block in the original method declaration with the specified action block.

8.1.1 Sample *e* Code

```
/**********************************************
* MODULE:    uk/infineon/verification/net/core.e
**********************************************/
```

```
<'
        struct Packet {
                pkt_msg() is { .. };
                checksum() is { .. };
                run() is { .. };
        };
'>

/**********************************************
* MODULE:    uk/infineon/verification/net/ethernet.e
**********************************************/
<'
        import uk/infineon/verification/net/core.e;

        struct Packet {
                pkt_msg() is also { .. };
                checksum() is only { .. };
                run() is first { .. };
        };
'>
```

8.1.2 UML Profile Elements

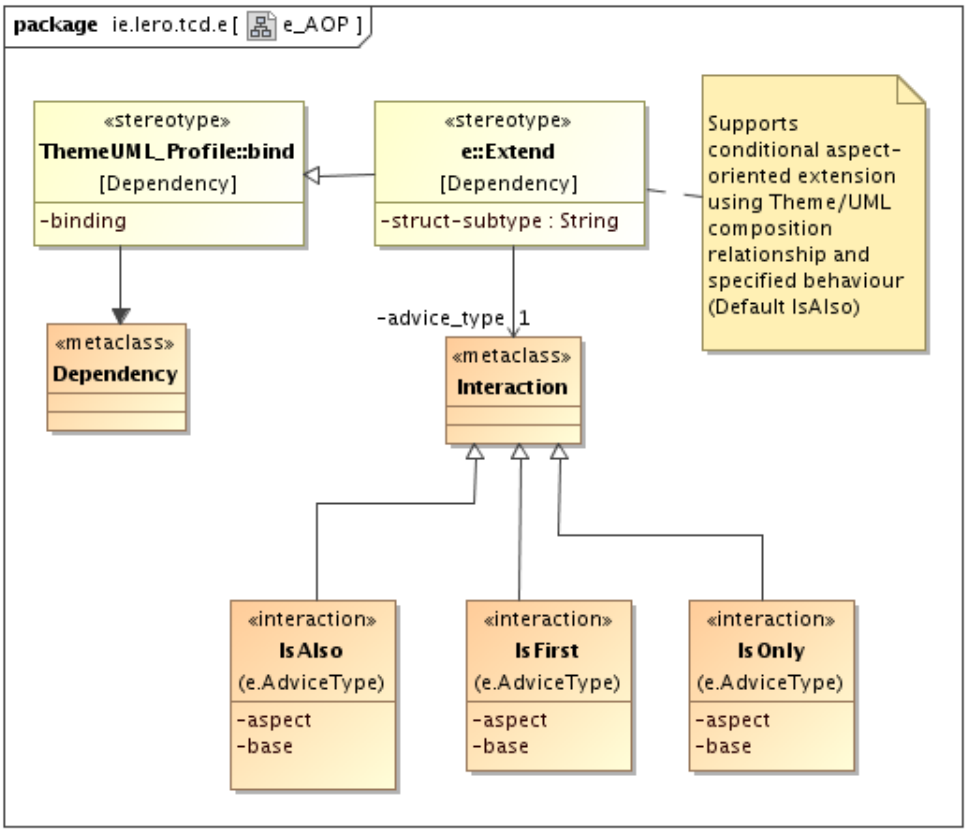

Figure 17 - Aspect Oriented Profile Elements

8.1.3 Sample Model

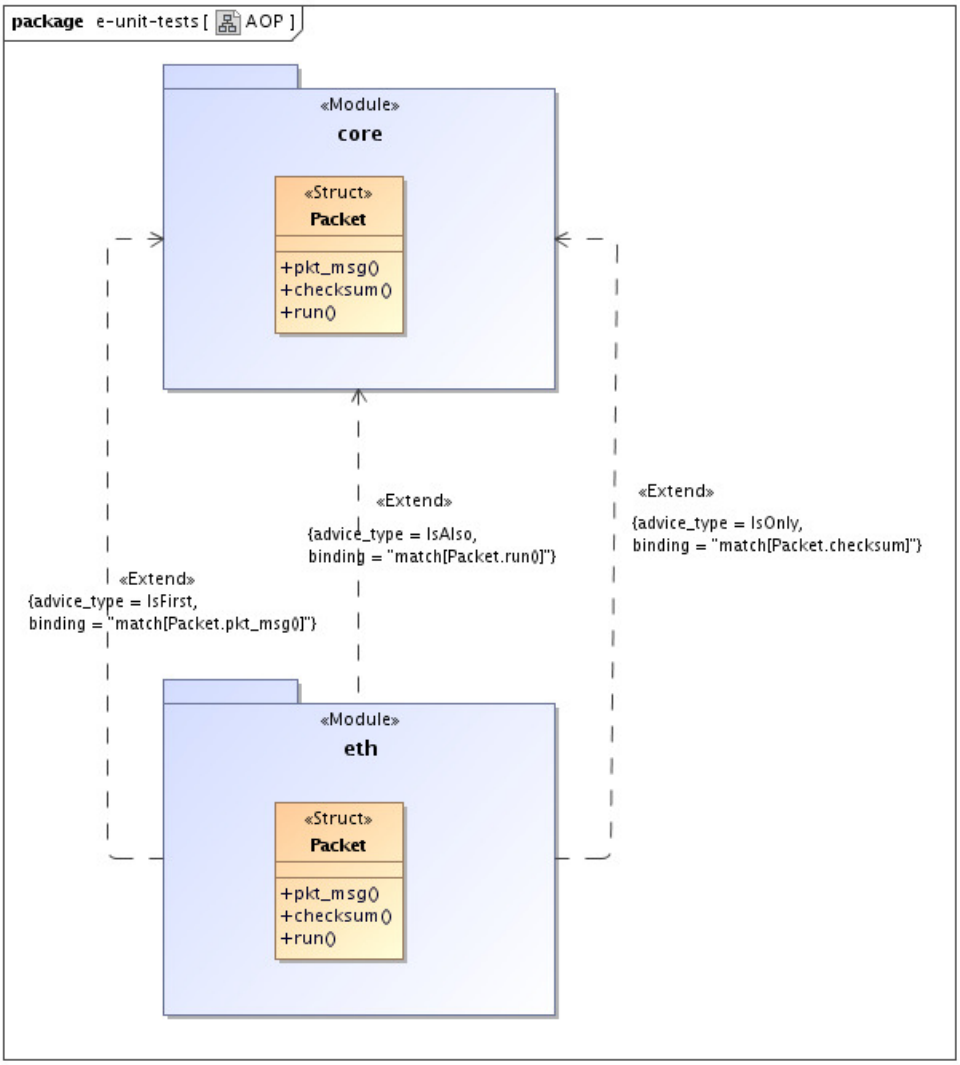

Figure 18 - Usage of Aspect-Oriented Profile Eleemnts

8.1.4 Code Generation

«**IF** isExtension()»extend«**ELSE**»struct«**ENDIF**» «name»

8.2 Extending Enumerated Types

Extends the specified enumerated scalar type to include the names or name-value pairs you specify.

8.2.1 Sample *e* Code

```
type PacketKind : [ATM, FDDI] (bits: 8);

extend PacketKind : [ETH];
```

8.2.2 Sample Model

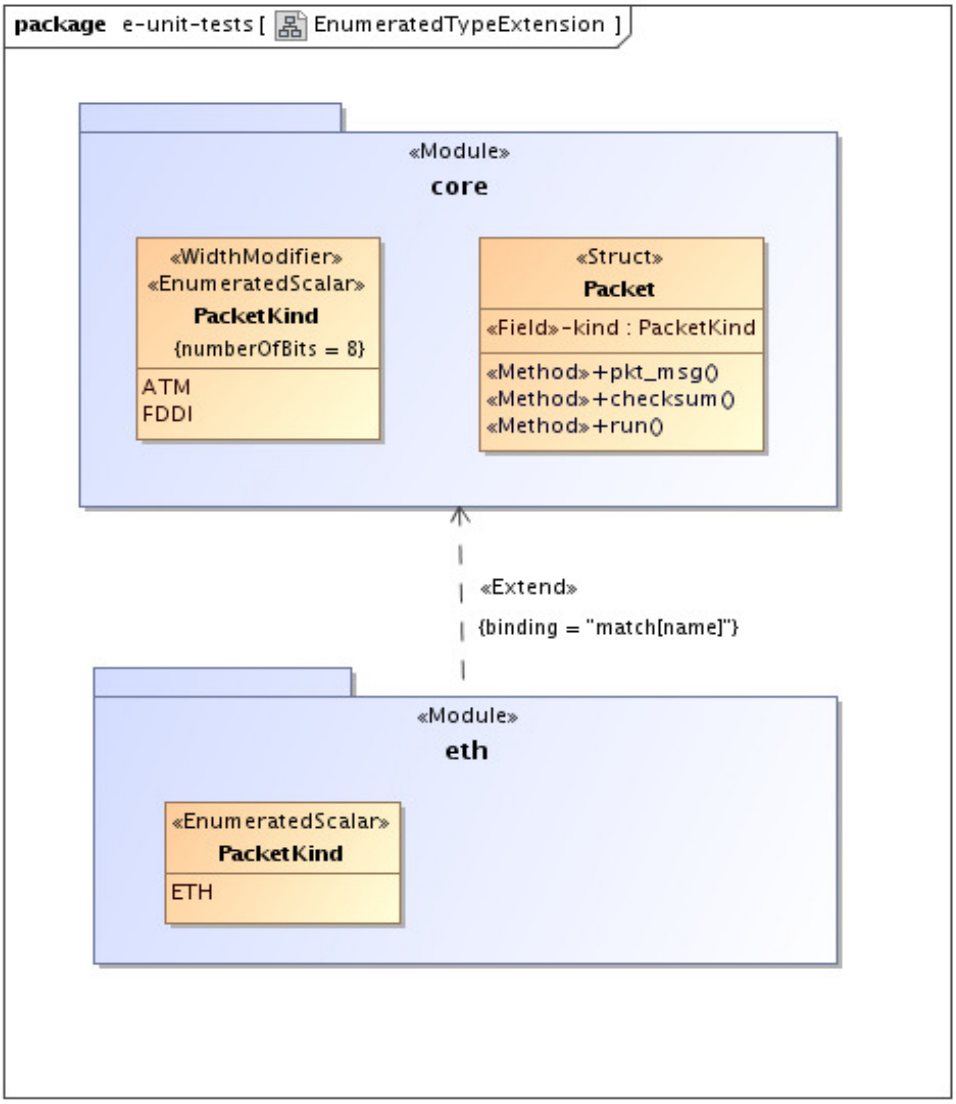

Figure 19 - Aspect-Oriented Extension Example

8.2.3 Code Generation

«**IF** isExtension()»extend«**ELSE**»type«**ENDIF**» «name»: ...

8.3 Cover Group Extension

The is also clause adds new items to a previously defined coverage group, or can be used to change the options for previously defined items.

8.3.1 Sample *e* Code

```
struct Packet {
        len: int;
        event sim_ready is change('top.ready') @sim;
        cover sim_ready is {
                item len;
        };
};

extend Packet {
        proto_version: int;
        cover sim_ready is also {
                item proto_version;
        };
};
```

8.3.2 Sample Model

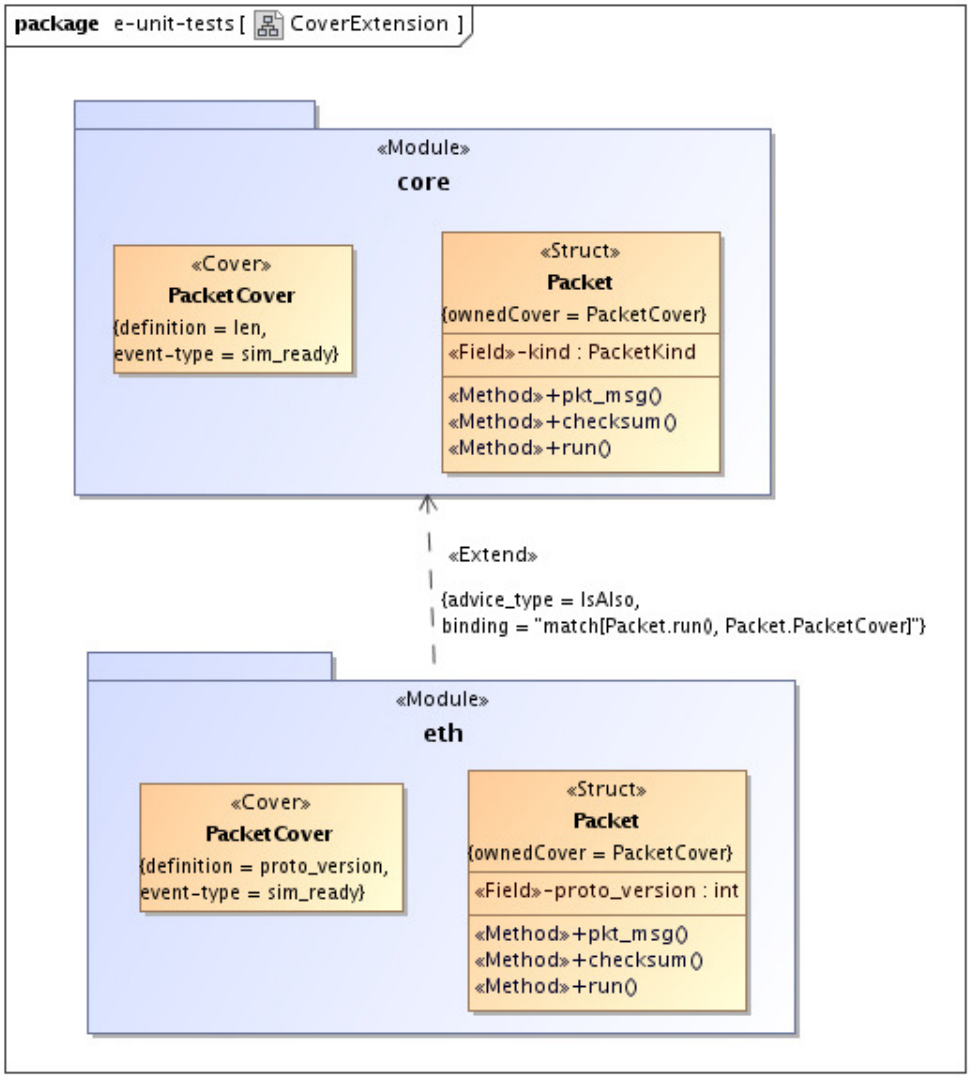

Figure 20 - Example of coverage extension

8.3.3 Code Generation

```
«DEFINE Cover FOR e::Cover»
            cover «this.eventtype.name» is «getMatchingAdviceType()» {«EXPAND
CoverItem FOREACH this.definition.typeSelect(e::CoverItem)»
            };
«ENDDEFINE»
```

8.4 Conditional Extension

Conditional Extension can be achieved using When inheritance or by extending a Struct subtype. A struct subtype is an instance of the struct in which one of its fields has a

particular value. The metamodel supports only the use of when inheritance.

8.4.1 Sample *e* Code

```
type PacketKind : [ATM, FDDI];

struct Packet {
        kind: PacketKind;
};

extend ATM Packet {
        virtual_channel_identifier: int (bits:16);
};
```

8.4.2 Sample Model

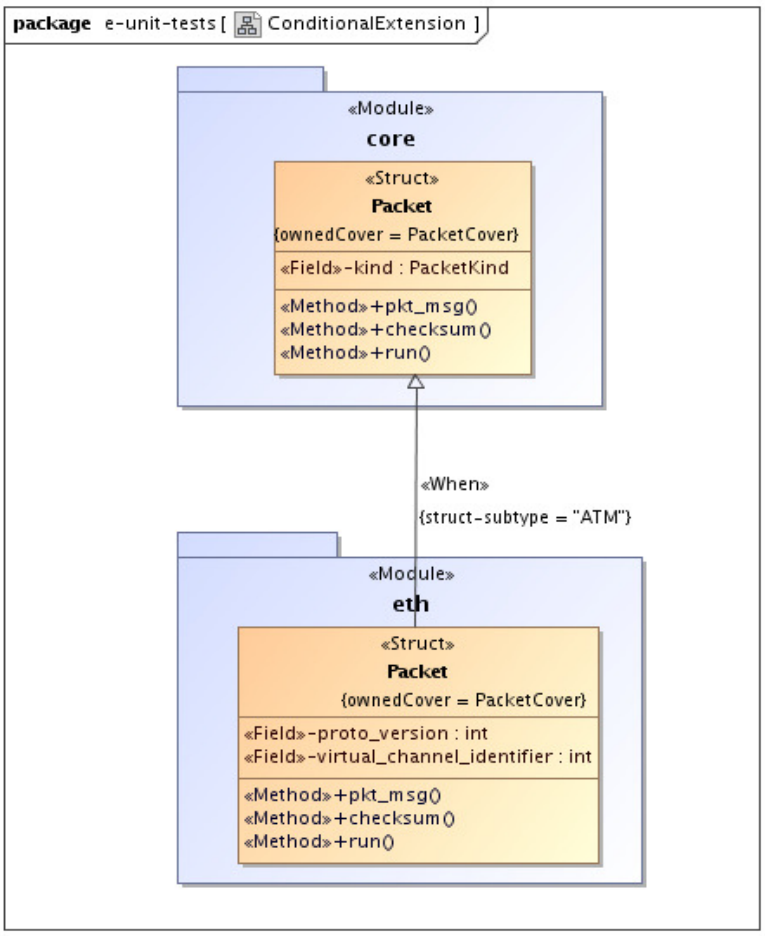

Figure 21 - Example of Conditional Extension

8.4.3 Code Generation

«**DEFINE** Struct **FOR** e::Struct»

```
«IF !(isWhenSpecificStruct() && isSameModule(this, getWhenTargetStruct(this)))»
        «IF isExtension()»extend«getStructSubtype()»«ELSE»struct«ENDIF» «name» {
                ...
        };
«ENDIF»
«ENDDEFINE»
```

9 Comments

Comments can appear in *e* within any code segment. For this reason comments are modelled using the standard uml::comment element that can be contained within any other element.

9.1 Sample *e* Code

```
-- create a coverage event and coverage group for this master
```

9.2 Sample Model

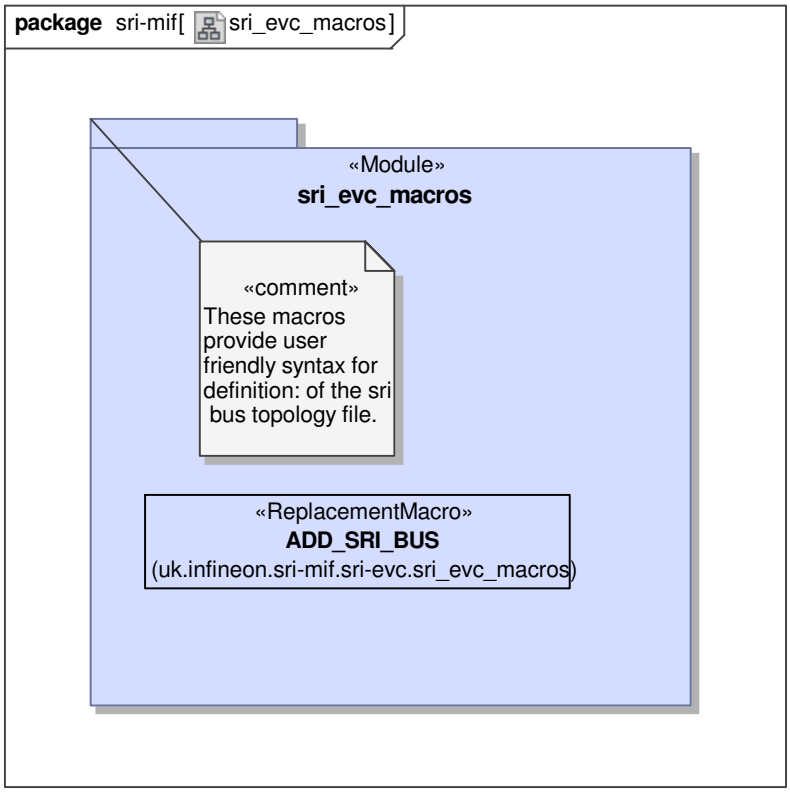

Figure 22 - Example addition of a comment.

9.3 Code Generation

```
«EXPAND Comment FOREACH allOwnedElements().typeSelect(uml::Comment)-»


«DEFINE Comment FOR uml::Comment-»
        «log('uml::Comment ' + this.body)-»
        «IF this.body.length > 80-»
                -- «EXPAND MultiLineComment FOREACH this.splitCommentString(80)-»
        «ELSE»-- «this.body-»               «ENDIF»
«ENDDEFINE»

«DEFINE MultiLineComment FOR String-»«this»«ENDDEFINE»
```

10 Macros

Macro definitions specify a name or a pattern that is to be replaced by *e* code text.

10.1 Sample *e* Code

```
define <ADD_SRI_BUS'statement> "ADD_SRI_BUS <bus_name'name>
<addr_width'num> <data_width'num> <constraints'block>" as {

   extend sri_evc_bus_name_t : [<bus_name'name>];

   extend <bus_name'name> sri_evc_bus_u {
      keep addr_bus_width == <addr_width'num>;
      keep data_bus_width == <data_width'num>;
      keep all of <constraints'block>;
   };

   -- define a constant for the data bus width of this bus - this will be used
   -- to control illegal coverage values
   define SRI_EVC_DATA_WIDTH_<bus_name'name> <data_width'num>;

}; -- define <ADD_SRI_BUS'statement>


  ADD_SRI_BUS SRI0 32 64 {

        -- eVC will drive bus into reset for first 5 clock cycles of simulation
   keep soft reset_period = 5;
        -- Reset will be re-asserted once after start of test
   keep soft num_resets = 1;
        -- Delay before reset is re-asserted
   keep soft reset_delay = 350;
        -- eVC is not supplying the SRI clock
   keep soft clock_half_period = 0;
   keep soft hdl_path() = "~/mif_tb0:";
```

```
};
```

10.2 UML Profile Elements

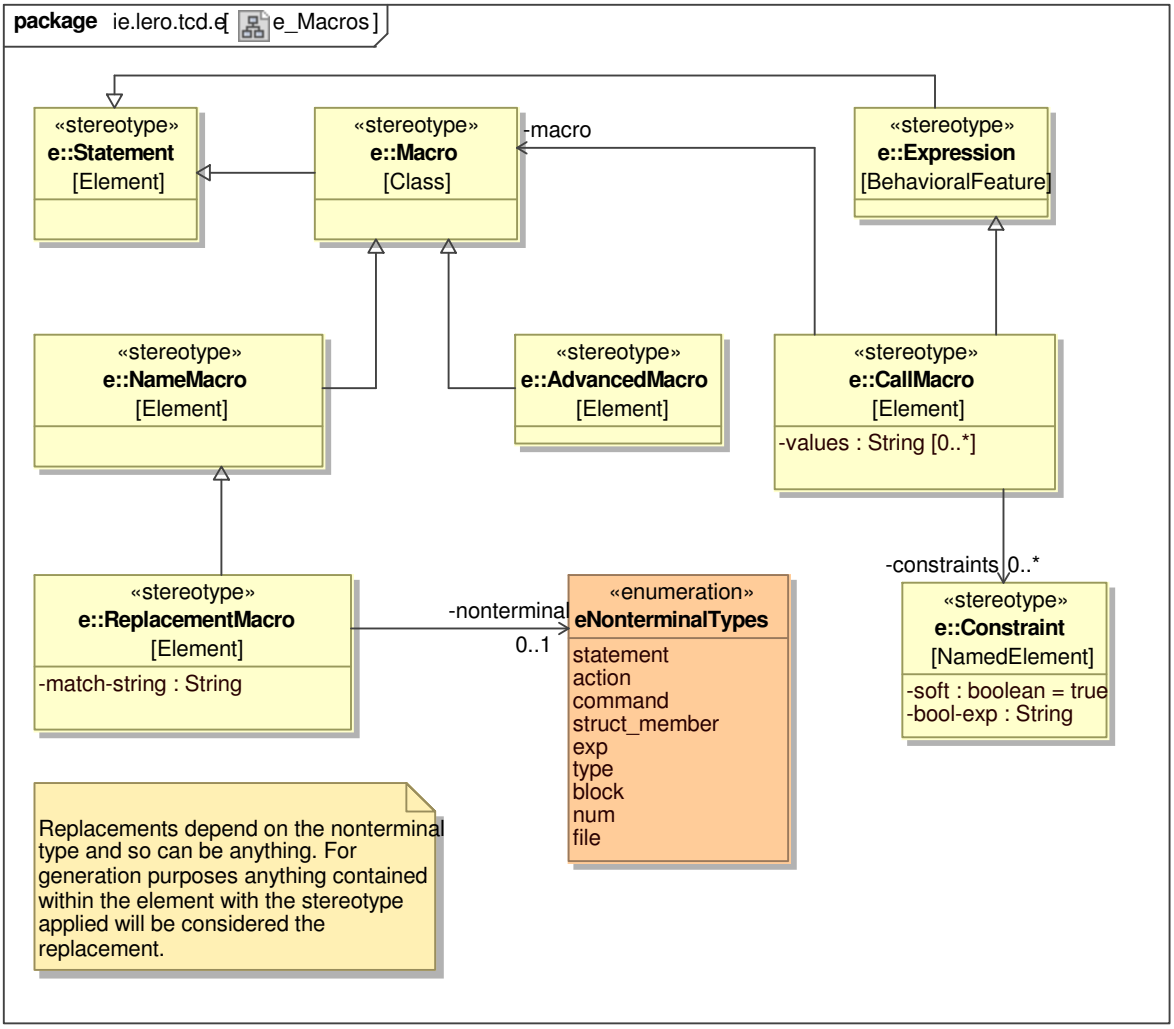

Figure 23 - Macro Profile Elements

10.3 Sample Model

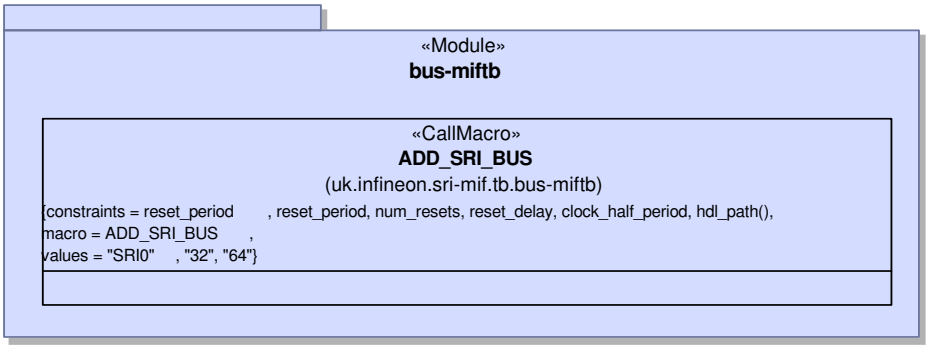

Figure 24 - Example of modelling a macro.

10.4 Code Generation

```
«DEFINE Statements FOR e::ReplacementMacro»
  «EXPAND Comment FOREACH allOwnedElements().typeSelect(uml::Comment)-»
  define <«this.name»«IF this.nonterminal !=
null»'«this.nonterminal.name»«ENDIF»> "«this.matchstring»" as {
    «EXPAND Statements FOREACH this.allOwnedElements()»
  }; -- define <«this.name»«IF this.nonterminal !=
null»'«this.nonterminal.name»«ENDIF»>
«ENDDEFINE»
```

11 Constraints

A keep struct member specifies a limitation, or restriction, on the values that are generated for an item.

11.1 Sample *e* Code

```
-- eVC will drive bus into reset for first 5 clock cycles
keep soft reset_period = 5;
```

11.2 UML Profile Elements

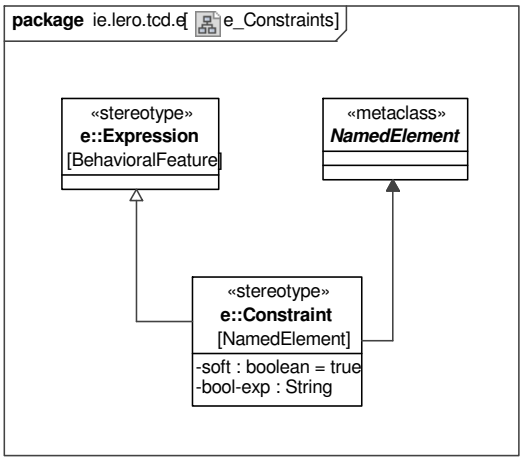

Figure 25 – Constraint Profile Elements

11.3 Code Generation

```
«EXPAND Constraint FOREACH (Collection[e::Constraint]) this.constraints-»

«REM»Constraints inside replacement macro or ...«ENDREM»
«DEFINE Constraint FOR e::Constraint-»
   «EXPAND Comment FOREACH ownedElement.typeSelect(uml::Comment)-»
   keep «IF this.soft»soft «ENDIF»«this.name» = «this.boolexp»;
«ENDDEFINE»
```

12 Extensions to Theme/UML

To facilitate the modelling of aspect-oriented constructs in e, Theme/UML has been extended in the following ways:

1. Composition Relationships are represented by curved, dashed lines between elements to be composed. All composition relationships are directional (using theme precedence inferred from relationship).
2. Override integration must be specified explicitly in the same way as merge integration.
3. Each element named in a match composition must appear in both modules and be of the same type.
4. Matching elements in a composition are specified as part of the composition using a list syntax (similar to template and binding specification) that supports multiple element specification using groups and wildcards.
5. Theme composition relationship is extended to incorporate an 'advice_type' tag that specifies and instance of a uml::Interaction. This interactions specifies the behaviour of the composition in the same way that Theme/UML uses sequence diagrams within a theme to specify the behaviour of the composition process for the themes templated elements.

13 Conclusion

This document has presented a UML 2.0 Profile extension for the *e* Hardware Verification Language. This document is intended to be used as developer reference material for the modeling of hardware verification testbenches in the e verification language. This document is subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find directly to the authors.

This profile is available as a UML 2.0 Profile (EMF UML2 v2.x XMI) or MagicDraw UML 16.8 Shared Module) by contacting the authors.

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre (www.lero.ie)