Verification Plans for a System

The *Verify* notation lets users specify *verification plan*s for each component in a system architecture. Verify also lets users specify a *method registry* that identfies verification methods implemented in different languages and apply to AADL models, detailed design models, and source code.

A verification plan specifies how every requirement of a system component (AADL component type or implementation) is verified, i.e., a system implementation meets the requirements. This is achieved by specifying a set of verification activities that must complete successfully as evidence that a requirement is met. This is done by a claim declaration for each requirement in the verification plan.

A verification activity is performed on an artifact representing the system implementation. This artifact may be an AADL model representing the architecture specification of the system, detailed design models, e.g., Simulink models of physical or control behavior, or the actual implementation, e.g., in form of software that must be executed on a hardware platform. Verification activities may also include review of documents, design, or code by human reviewers.

A verification activity identifies the verification method to be used, and a set of parameters. By default every verification method is assumed to accept a reference to a model element, typically an AADL instance model element. The verification method may accept additional parameters, or it may expect the relevant information to be available as a property on the AADL model. By default the verification method is expected to return a Boolean as result. Verification methods may also return results via Eclipse Markers, in Resolute result format, or in a result report format defined by Alisa.

A verification method registry allows users to specify the type and signature of a verification method as well as a reference to its implementation in an implementation language neutral way. The actual method may take a variety of forms. However, the methods have a common interface to interface with the Alisa incremental assurance engine. Currently we support Osate analysis plug-ins, Resolute, and Java/Xtend based method implementations.

The *Verify* notation supports two file extensions: *verify* for verification plans, and *methodregistry* for verification method specifications.

Verification Plan

Verification plans are defined in files with the extension *verify.* Multiple verification plans can be placed in a single file.

Users define one verification plan for each *system requirements* or *global requirements* declaration. A component type that is an extension of another component type may have a separate *system requirements* or *global requirements* declaration to represent requirements that are specific to the extension. Note that the requirements declared for the original component type are inherited. Similarly, the verification plan of the component type being extended is inherited to complement the verification activities of the extension. Similarly, component implementations inherit the requirements and verification plans of the component types.

The verification plans of a system and its subsystems get combined into an assurance case instance according to an *assurance case* declaration exprssed in the *Alisa* notation (see Alisa.html).

Note: Users may want to follow the convention to name the set of *system requirements* by a <dot.-

separated qualified name path corresponding to the component, whose requirements are specified – and the same name be used for the verification plan.

```
VerificationPlan ::=

verification plan qualifiedname ( : "descriptive title" )?

for <system or global requirements reference>

[

 ( description Description )?

 Claim*

 ( rationale String )?

 ( issues ("explanation")+ )?

]
```

The verification plan specifies a claim for each of the requirements that have been specified for a system.

A verification plan consists of the following:

- *Qualifiedname*: a name that consists of one or more <dot> separated identifiers. This name is globally known and must be unique with respect to other verification plan names. A verification plan is referenced by its name.
- *Title:* a short descriptor of the verification plan. This optional element may be used as more descriptive label than the name.
- *For:* reference to *system requirements* or *global requirements*. The verification plan must have one claim for each requirement in *system requirements* or *global requirements*. The individual requirement can be referenced in a claim by their identifier without qualification by its container.
- *Description:* A textual description of the verification plan. In its most general form this can be a sequence of strings, a reference to the classifier/model element identified by the *for* element (expressed by the keyword *this*), as well as references to Variables defined with requirements.
- *Rationale:* the rationale for a system requirement as a string.
- *Claim:* a set of claims that make up the verification plan, one claim per requirement in the requirement set identified by the **for**.
- *Issues:* allows users to record issues that may be encountered as a set of textual notes (Strings).

Claim

A claim declaration specifies a set of verification activities for a requirement.

```
Claim ::=

claim <requirement reference> ( : "descriptive title" )?

[
```

```
( activities VerificationActivity+ )?

( assert ArgumentExpression }?

( rationale String )?

( weight Integer )?

( Claim )*

( issues ("explanation")+ )?

]
```

A claim declaration consists of the following:

- *Requirement reference*: a reference to a requirement in the requirements set identified in the **for** of the enclosing verification plan. The reference identifies the requirement by its identifier without qualification.
- *Title:* a short descriptor of the claim. This optional element may be used as more descriptive label than the name.
- *Weight:* a weighting factor used in metrics to indicate the importance of different claims as evidence.
- *Activities:* A sequence of verification activity declarations associated with a claim.
- *Assert:* An optional argument expression specifying the argument logic for the claim. By default all verification activities are required to be successful (see below).
- *Claim:* a set of zero or more subclaims. A subclaim is associated with a requirement that is a refinement of the requirement associated with a given claim.
- *Rationale:* the rationale for a system requirement as a string.

Note: A requirement for a system may have been refined into sub-requirements that are verifiable. Typically, only the leaf elements of a requirement refinement hierarchy involve verification activities. Howeever, the notation supports the possibility for a veritication activity for each requirement in the refinement hierarchy.

Verification Activity

A verification activity specifies the which verification method is to be called, the actual parameter values to be passed directly as part of the call, values to be made available to the method via properties associated with the component, and any result values to be bound to **compute** variables defined in requirements. Verification activities can have phase and user defined category labels that will be used for filtered views of an assurance case instance (see Incremental Assurance with Alisa alisa.html).

```
VerificationActivity ::=
activity activityname ( : "descriptive title" )?
  : ( <result parameter list> = )?
    <verification method reference> ( <actual parameter list> )
    ( property values ( <property value list> ) )?
```

```
[
    ( phase phaselabels )?
    ( category categorylabels )?
    ( timeout Integer <time unit> )?
  ( weight Integer )?

]
```

A claim declaration consists of the following:

- *Activityname*: unique name within a verification plan. Verification activities are referenced by this name in argument expressions.
- *Title:* a short descriptor of the verification activity. This optional element may be used as more descriptive label than the name.
- *Result parameter list:* comma separated list of zero or more **compute** variable references to hold any returned result parameter of a verification method.
- *Verification method reference:* reference to a verification method qualified with the method registry name.
- *Actual parameter list:* comma separated list of zero or more actual parameter values. Acceptable parameter values are references to **val** variables, integers and reals with or without measurement unit, strings, and booleans. In the future general expressions will be supported.
- *Property value list:* optional comma separated list of values to be made available to the verification method via an AADL property.
- *Phase:* list of development phases labels (without comma separation) in which this verification activity should be performed. It is used when defining filter criteria for assurance tasks.
- *Category:* list of user defined category labels (without comma separation) in which this verification activity should be performed. It is used when defining filter criteria for assurance tasks.
- *Timeout:* specification of a time limit for the execution of the verification activity. When the time limit is reached the verification activity is aborted with a *timeout* result error indication.
- *Weight:* a weighting factor used in metrics to indicate the importance of different verification activities as evidence.

Note: The intent of result parameters is to provide access to computed results from the execution of a verification method. These results can then be evaluated by the predicate specified as part of the requirement declaration. This predicate evaluation will be supported in the near future.

Argument Expression

The argument expression of a claim assertion specifies the logic for evaluating verification activity results. A verification activity can have *success*, *fail*, or not complete for two reasons: *timeout* or any other reason not to complete indicated by *error.*

```
ArgumentExpr ::=  ElseExpr |
  ElseExpr then ArgumentExpr

ElseExpr ::= SingleElseExpr | CompositeElseExpr

SingleElseExpr ::=   <VerificationActivity>< |
```

```
  VerificationActivity> else  ( ElseExpr |
                            ( ( fail:  ArgumentExpr )?
                                ( error: ArgumentExpr )?
                          ( timeout: ArgumentExpr )?
                            ) )

CompositeElseExpr ::=  CompositeExpr |
  CompositeExpr else ElseExpr

CompositeExpr ::=  ( ArgumentExpr )
  all [ ArgumentExpr ( , ArgumentExpr)* ] |
```

The expression takes the following form:

- *Then:* lets users specify an ordering of verification activities, i.e., the second verification activity is only executed if the first one is successful.
- *Else:* lets users specify an alternate verification activity if the first one returns *fail* or does not complete.
- *Fail, Error, Timeout:* lets users specify a different alternative verification activity for each of the three results of an unsuccessful first verification activity. This is possible only if the first verification activity is a single activity.
- *All:* lets users specify that all of a set of veriifcation activities must be successful. All verification activities will be executed independent of whether they are successful or not. The result of the *all* operator will indicate *success* only if all listed verification activities were successful.

Brackets allow users to specify a change in precedence ordering of argument expression operators.

Registry of Verification Methods

A verification method registry allows users to register implemetnations of different verification methods in an implementation language neutral format. A verification method registry is defined in a separate file with the extension *methodregistry.*

A verification method is invoked when a verification activity is executed. The method analyzes, simulates, or executes an AADL model, detailed design model, or source code. The result may be an evaluation of a predicate that reflects whether a requirement has been met, i.e., returns a *success* or *fail* as result. The verification method may also return the result values of computations.

Note: A verification method will always be passed the component instance it applies to as first parameter. This parameter does not have to be specified in the registry.

```
VerificationMethodRegistry ::=
verification methods qualifiedname ( : "descriptive title" )?
[
  ( description Description )?
  VerificationMethod+
]

VerificationMethod ::=
```

```
method methodname
  ( ( <FormalParameter list> )
    ( properties ( <property list> ) )?
    ( returns ( <result list> ) )?
    ( boolean | report )?
  )?
  ( : "descriptive title" )?
  [
    MethodKind
    ( description Description )?
    VerificationPrecondition?
    VerificationValidation?
    ( category categorylabels )?
    ( quality qualitylabels )?
  ]


MethodKind ::=
  java MethodPath |
  resolute MethodID |
  plugin MethodID |
  manual DialogIdentifier

FormalParameter ::=
  Type ID ( % Unit )?

VerificationPrecondition ::=
precondition <method reference> ( <formal parameter reference list> )

VerificationValidation ::=
validation <method reference> ( <formal parameter reference list> )
```

The verification method registry consists of:

- *Qualifiedname:* a <dot> separated sequence of identifiers. The registry name acts as qualifier for verification methods.
- *Title:* a short descriptor of the verification method registry. This optional element may be used as more descriptive label than the name.
- *Description:* a description of the verification method.

The verification method declaration consists of:

- *Methodname:* identifier for the method that must be unique within the registry.
- *FormalParameter list:* optional comma separated list of formal parameter specifications. The formal parameter specification consists of a type an identifier as name and an optional specification of expected measurement unit for the value. The actual value will be converted into the expected unit. This feature is useful when the method expectes a value without a measurement unit. The following types are currently supported: Java types String, Boolean, Double, double, Long, long; AADL property types StringLiteral, BooleanLiteral, RealLiteral (with a double value and an optional measurement unit), IntegerLiteral. The unit can refer to any of the AADL property units. The specified type must be the type/class name of the Java method without package qualification. NOTE: the first parameter is assumed to be ComponentInstance and does not have to be specified.
- *Property list:* optional comma separated list of AADL property definition references. The

references use the core AADL syntax for property references.

- *Result list:* optional comma separated list of formal parameter specifications to indicate result values.
- *boolean:* indication that the method returns a boolean value to indicate success or fail.
- *report:* indication that the method returns a *result report* in an Alisa specified format. This report can contain a collection of result indicators and result data (see ResultReport.html).
- *Title:* a short descriptor of the verification method registry. This optional element may be used as more descriptive label than the name.
- *MethodKind:* identification of the method implementation and implementation specific reference to the method
- *Java:* a Java method identified by a path to a method inside a class. Note that Xtend methods can be registered this way as they get translated into Java. Java methods can report violation of assertions (predicates) by throwing *AssertionException*. They are mapped into *Fail* results. This is how JUnit and Java8 support assertions. This allows users to register existing JUnit test methods as verification methods. Uncaught runtime exceptions within a Java method are also caught by the Assure execution harness and mapped into *Error* results. Java methods can return a Boolean result, which gets mapped into *Success* and *Fail*. The method can also return a String object. In this case the result is interpreted as *Success* with the text presented as succes message. In this case, AssertionExceptions must be used to report *Fail* results. Finally, Java methods can return result reports in an Alis defined format.
- *Resolute:* a Resolute claim function or computational method identified by the unqualified method name. Results in the Resolute result format are mapped into a common Alisa Result Issue format (see Section Result Issue Report below).
- Plugin: an OSATE analysis plugin method identified by an identifier. Plugin methods are defined in a predeclared method registry (see below). OSAT Eplugins report their results via the Eclipse Marker mechanism. These results are mapped into a common Alisa result issue format for icnlusion in the assurance case result instance.
- *Manual:* the method represents a manual method, i.e., a method that is performed by a human. The person will interactively report the result of the verification, e.g., the result of performing a review.
- *Description*: a description of the verification method.
- *Precondition*: reference to a registered verification method that determines whether the model meets criteria for the verification method to be able to execute. If the method takes parameters their values are those of the verification method formal parameters as a comma separate list. Typically such a method will check that certain model element and expected property values are in place. The verification method in a verification activity will not execute if the precondition is not met.
- *Validation*: reference to a registered verification method that determines the validity of the results from the execution of the verification method. If the method takes parameters their values are those of the verification method formal parameters as a comma separate list. Typically such a method is used when a verification method can operate on a model with incomplete information, i.e., it will indicate how complete the input was. Similarly, the method

can be used to assess whether the results are within a reserve margin, e.g., whether the result has an x% margin from the limits. The verification method may return a success result while the validation may indicate a fail.

- *Quality*: list of quality attribute labels (without comma separation) that this verification method addresses. It is used when defining filter criteria for assurance tasks.
- *Category*: list of user defined category labels (without comma separation) in which this verification activity should be performed. It is used when defining assurance tasks.

Built-in and User-defined Registries

Alisa comes with a built-in registry of verification methods. This registry is available as a project called *AlisaBasics* In the Github repository *osate/alisa-examples*. The registry contains OSATE analysis plugin method declarations.

Users can add currently verification methods to Alisa in two ways. First, users can write claim functions and computational methods in Resolute (see Resolute language specification for details). Users then define their own verification method registry to complement the "predefined" registry. Since Resolute is an interpreted notation, these newly registered methods are immediately available for use. An example can be found in the project *SimpleAlisaExample* found in the Github repository *osate/alisa-examples.*

Second, users can write or make use of verification methods written in Java or Xtend. Users may even write Java wrapper methods to interface with existing tools or external tools, such as *JUnit* or *Simulink*. Since Java methods are called reflectively, a Java method once added to a registry can be used in a verification activity. When writing methods in Java or Xtend, users have available a large collection of methods from OSATE to process AADL models and retrieve AADL property values. These are the same methods users would use when writing a plugin for OSATE.