# Verification Plans for a System

The *Verify* notation lets users specify *verification plan*s for each *system requirement set* of system components as well as for *global requirment sets*. 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 or global requirement set is verified. This is achieved by specifying a set of verification activities that must complete successfully as evidence that a requirement is met. Users specify a claim declaration for each requirement that contains a set of verification activities. Users may also specify an optional condition under which verification activities are performed, e.g., one verification has to be successful for another one to be executed, and a verification activity acts as backup if the original one fails to complete successfully. In addition, verification methods may have specified an optional *precondition* that must be satisfied before the method is executed, and a result *validation* that determines the degree of validity of the result.

A verification activity is performed on an artifact representing the system implementation. This artifact is an AADL model representing the architecture specification of the system, or accessible from the AADL model, such as detailed design models, e.g., Simulink models of physical or control behavior, or the source code. Verification activities may also include manual activities, e.g., 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 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.

We support two types of methods: a predicate method that evaluates the artifact to determine if it meets the requirement (*success*, *fail*); a compute method that computes values (referenced by *computed variables*) that can be used to evaluate the predicate specified as part of the requirement or used as input parameter for a succeeding verification activity.

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 (reporting results via Eclipse markers), Resolute (reporting results in Resolute format), JUnit4 (reporting results via AssertionException and JUnit result format), and Java/Xtend based method implementations (reporting result via AssertionException, return value, or a result format defined by ALISA.

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

## Verification Plan

Users define one verification plan for each *system requirement set* or *global requirement set* declaration. The verification plan must have one claim for each requirement in *system requirements* or *global requirements*. The verification plans of a system and its subsystems get combined into an assurance case expressed in the *Alisa* notation (see Assurance Cases).

A component type that is an extension of another component type may have a separate *system*

*requirement set* declaration to represent requirements that are specific to the extension. Note that the requirements declared for the original component type are inherited by a component type or implementation extension and requirements associated with a component type are inherited by each implementation of that type. Similarly, the verification plan of the component type being extended is inherited to complement the verification activities of the extension and component implementations inherit verification plans of the component types.

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

```
VerificationPlan ::=

verification plan qualifiedname ( : "descriptive title" )?

for <system or global requirement set reference>

[

 ( description Description )?

 Claim*

 ( rationale String )?

 ( issues ("explanation")+ )?

]
```

The verification plan specifies a claim for each of the requirements in the requirement set identified by the *for* reference.

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*.
- *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**. The individual requirement can be referenced in a claim by its identifier without qualification.
- *Issues:* allows users to record issues that may be encountered as a set of textual notes (Strings).

Note: Users may want to follow the convention to name the *system requirement set* and related *verification plan* by qualified name of the component classifier, whose requirements and verification plans are specified.

# Claim

A claim represents the set of verification activities that must be successful for a requirement to be met. An argument expression may be defined on the set of verification activities to specify a condition on the combination of verification activities that must be successful. If no argument expression is specified all verification activities must be successful.

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 )*      // subclaims

( 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.
- *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).
- *Rationale:* the rationale for a system requirement as a string.
- *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.
- *Weight:* a weighting factor used in metrics to indicate the importance of different claims as evidence.
- *Issues:* allows users to record issues that may be encountered as a set of textual notes (Strings).

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. In other words, a claim for a requirement that has been refined only may only contain subclaims that themselves contain verification activities. However, the notation supports the possibility for a verification activity for each claim in the requirement refinement hierarchy.

# Verification Activity

A verification activity specifies the verification method 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).

A verification activity when executed can have *success*, *fail*, or does not complete for two reasons: *timeout* or any other reason (uncaught exception thrown during the execution) not to complete indicated by *error.* This state is maintained in an instance representation of the assurance case (see Assurance Case Instance).

```
VerificationActivity ::=
activity activityname ( : "descriptive title" )?
  : ( <result parameter list> = )?
    <verification method reference> ( <actual parameter list> )
    ( property values ( <property value list> ) )?
    [
      ( category categorylabel* )?
      ( timeout Integer <time unit> )?
      ( weight Integer )?

    ]

CategoryLabel ::=  CategoryTypeID . CategoryLabelID
```

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:* qualified reference to a verification method in the method registry.
- *Actual parameter list:* comma separated list of zero or more actual parameter values. The expected list of parameters and their types are specified as part of the verification method in the method registry. Acceptable parameter values are expressions that include references to constants (val) and computed variables (**compute**), integers and reals with or without measurement unit, strings, booleans, AADL properties and property constants. For details of the expression language see Expression Notation.
- *Property value list:* optional comma separated list of values to be made available to the verification method via an AADL property. The expected set of properties with specific value are specified as part of the verification method in the method registry. Currently the values are limited to references to constant variables (**val**).
- *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 (assurance task).

- *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.

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 or passed as input to succeeding verification activities.

Actual parameter and property value expressions may refer to constants and computed variables. Those can be variables defined in requirements or requirement sets associated with the verification activity, including those of requirements and their containing requirement sets inherited through the component classifier extends hierarchy.

# Argument Expression

Without an argument expression all verfication expressions must be successful for the claim to be met. The arugment expression allows users to specify condition under which verification activities are performed: one verification has to be successful for another one to be executed (**then**), and a verification activity acts as backup if the original one fails to complete successfully (**else**). In the case of backup verification activity, users can specify a single verification activity executed if the original one is not successful, or users can specify separately whether they intend to have a backup verification activity for the original activity return *Fail*, not completing because of a *Timeout*, or because the execution failed to complete for other reasons, e.g., the result of an uncaught exception.

```
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 verification 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 implementations of different verification methods in an implementation language neutral format. Currently OSATE Analysis plug-ins, Java, Resolute, Agree, JUnit, and manual methods are supported.

A verification method registry is defined in a separate file with the extension *methodregistry.*

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

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.

# Verification Method Specification

A verification method is invoked when a verification activity is executed. The method analyzes, simulates, or executes an AADL model, or a detailed design model, or source code associated with a component in the AADL model.

The verification method may evaluate a predicate to indicate whether the verification is successful or failed to be successful, or the verification method may compute one or more values that are turned. In the latter case a predicate defined for the requirement may refer to the computed variable holding the result value, or the result value may be passed as in parameter to a succeeding verification activity.

A verification method is assumed to always be passed the component instance as its first parameter. This parameter does not have to be specified in the registry. The model element type of this parameter must be specified if it is not the component itself, i.e., if it is a *feature*, *connection*, end to end *flow*, or any model *element* in the AADL instance model.

Additional parameters are passed to the method as part of the call as specified by the list of typed formal parameters or through a property associated with the model element being passed as first parameter.

Return values are assigned to *Computed Variables*, whose names match the formal parameter specification of the **returns** clause.

For further details see the description below.

```
VerificationMethod ::=
method methodname
  ( ( ( ModelElementType , )?
      FormalParameter ( , FormalParameter )* )
    ( properties ( Property ( , Property )* ) )?
    ( returns ( FormalParameter ( , FormalParameter )* ) )?
    ( boolean | report )?
  )?
  ( : "descriptive title" )?
  ( for <ComponentClassifier>  | ComponentCategory* )?
  [
    MethodKind
    ( description Description )?
    VerificationPrecondition?
    VerificationValidation?
    ( category categorylabel* )?
  ]


ModelElementType ::=
  component | feature | connection | flow | mode | element

MethodKind ::=
  java MethodPath ( (  JavaParameter ( ,  JavaParameter )* ) )? |
  plugin MethodID |
  resolute MethodID |
  agree ( single | all ) |
  junit ClassPath |
  manual DialogIdentifier

FormalParameter ::=
  ID : TypeSpec ( in Unit )?

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

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

The verification method declaration consists of:

- *Methodname:* identifier for the method that must be unique within the registry.
- *TargetType:* The type of the first parameter passed to the method when invoked. By default the component instance, whose requirement is verified, is passed. Some requirements apply to an element within a component instance, e.g., to a feature. In this case, users must specify the fact that the methods express a feature instance instead of a component instance. Other targets are connection instances, mode instances, and (end to end) flow instances. Element is used to indicate that a InstanceObject is expected. In the case of Java methods, the specified target type must match the class of the first parameter.
- *FormalParameter list:* optional comma separated list of formal parameter specifications. The formal parameter specification consists of an identifier as name, a type, and an optional specification of expected measurement unit for the value if the expected type is numeric without measurement unit. The actual value will be converted into the expected unit. This feature is useful when the method expectes a value without a measurement unit.

- *Property list:* optional comma separated list of AADL property definition references. The references use the core AADL syntax for property references. The properties will be assigned the value specified as part of the method call in a verification activity if no property value is present for the model element. If a property value is present the value must be the same as the value specified in the call.
- *Result list:* optional comma separated list of formal parameter specifications to indicate result values. The result values will be assigned in order to *Computed Variables* specified as part of the method call in a verificaiton activity.
- *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.
- *For:* a restriction on the type of component the verification methods can be applied to. For example, a verificaiton methods may have been developed to only apply to threads, or for specific component classifiers.
- *MethodKind:* identification of the method implementation and implementation specific reference to the method. The following method kinds are supported: OSATE analysis plug-ins, Resolute, Junit4, Java, Agree, Manual. See Kinds of Verification Methods for details.
- *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.
- *Category*: list of quality attribute labels (without comma separation) that this verification method addresses. It is used when defining filter criteria via assurance tasks.

## Supported Verification Method Kinds

The following method types are supported:

- *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 (see below for registering Junit4 test classes).
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* if specified to do so by the **boolean** keyword.
Java methods can return result reports in an ALISA defined format (see Result Report) if the method is registered accordingly (**report**).
Java methods can return a single value as object or a hasmap of multiple values. These values will be assigned to the specified *computed variables*.
The actual Java method may expect Java types that are not part of the ALISA Types. For example, the Java method may expect a l*ong* numeric. An optional parameter list allows users to specify the expected Java type followed by the name of the formal parameter. Currently we map *Stringliteral -> String*, *BooleanLiteral -> Boolean*, *RealLiteral -> double* or *real*, *IntegerLiteral -> long* or *int*.

- *Plugin*: an OSATE analysis plugin method identified by an identifier. Plugin methods are defined in a predeclared method registry (see Predefined Method Registry). OSATE analysis plugins report their results via the Eclipse Marker mechanism. These results are mapped into a ALISA result issue format for inclusion in the assurance case result instance.
- *Resolute:* a Resolute claim function or computational functions identified by the method name. Results in the Resolute result format are mapped into the ALISA Result Issue format. In the case of computational functions the returned value is assigned to the specified *Computed Variable*.
- *Agree:* Agree verification is invoked on the target component for a single layer verification or for verifying all layers. This is equivalent to invoking Agree on a component through the user interface. Note that for execution of Agree the Jkind verifier must be installed (see Agree documentation).
- *JUnit:* The specified JUnit test class is invoked. Results of a JUnit run are mapped back into the ALISA Result Issue format.
- *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.

# Built-in and User-defined Registries

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

The *AlisaPredefined* Project also contains a set of predefined category types and labels. See the ReqSpec documentation for details.

Users can add verification methods to ALISA in several ways.

First, users can write claim functions and computational functions in Resolute (see Resolute language specification in OSATE Help 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 *Simulink* or the execution of source code via JUnit tests. 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. You create verification method implementations in Java or Xtend in a Plugin project, which allows you to easily get access to OSATE plugins.

Third, users can write JUnit tests and register test classes. Those tests can operate on an AADL model or they may execute source code.