Constraints and Tokens

This note summarizes three areas of exploration:

* Product Line Constraint Verification
* Tag Propagation Analysis
* Tag Propagation and Transformation (Error Propagation as Typed Token System)

We will utilize the new type system and expression language. However, until it can be integrated with AV3 we have a locally implemented variant.

We also utilize JGraphT as an infrastructure to process graph structures. JGraphT allows us to provide model elements from the AV3 model – in particular we utilize instance model elements as vertexes and edges where appropriate.

# Product Line Constraints

The basic idea of product line constraints is that component variants are tagged with product line characteristics. Each component variant may have zero, one, or more feature labels. A product line constraint specifies that for every choice point, i.e., component where users can select from a number of choices (variants aka alternative implementations and V3 configurations), the choices must satisfy the constraint. The constraint is expressed as a disjunction of feature label conjunctions.

For every subcomponent in a system model the assigned classifiers, i.e., interface, implementation, and configurations, must satisfy the constraint. Satisfaction of the constraint means that the feature labels of each classifier must satisfy one of the disjunctions, i.e., the set of feature labels of the classifier must contain the feature labels of at least one conjunction.

ProductLineConstraint.exists[disjunction| FeatureLabels.contains(disjunction) ]

A classifier without feature labels always meets the product line constraint.

The product line constraint applies across the whole product or within the scope of a particular subsystem.

The product line constraint is used in two ways:

* To verify that a configured system instance meets the product line constraint,
* To limit the choices in an interactive configuration tool to those that satisfy the constraint.

## The Prototype Implementation

The AV3 Meta model defines the following:

* Literal as a super type for StringLiteral, NumberLiteral (with IntegerLiteral and RealLiteral as subtypes), BooleanLiteral, and TypeReference as Literal. We make use of TypeReference as the elements of FeatureLabel and ProductLineConstraint.
* LCollection as a collection of Literals
* MultiLiteralConstraint as a subtype of LCollection that includes a match operator for its elements. The LOperator includes *any* (interpreted as disjunction) and *all* (interpreted as conjunction).

The property *FeatureLabels* is introduced to represent the feature labels associated with a classifier. The property *FeatureLabels* is defined to accept a list of type references. The referenced types are expected to be a subtype of type *FeatureLabel*.

The product line constraint is a MultiLiteralConstraint whose LOperator is *any* and whose elements are MultiLiteralConstraints with LOperator *all* and elements referring to types.

The *contains* operation on LCollection is used to evaluate the constraint satisfaction. Currently this operation uses *equals* to compare the elements with *equals* defined for TypeReference objects to reference the same Type definition.

The instance configuration declaration in a *WorkingSet* optionally includes an assertion for *#FeatureLabels* to satisfy the product line constraint. When specified this constraint is evaluated after the instance model has been generated and any constraint violations are reported as issues.

The package org.osate.aadlv3.util.ProductLineConstraint contains support methods for evaluating product line constraints. This package makes use of *contains* methods for LCollection, which are defined in org.osate.aadlv3.util.Aadlv3Util. The support methods consist of

* Support methods to retrieve the product line constraint (getProductLineConstraint) of an instance configuration and to retrieve the feature labels of a classifier (getFeatureLabels).
* Support methods to determine whether the product line constraint is satisfied by a component instance for use in validating an instance model, or by a classifier for use in filtering configuration choices.
* A method for retrieving all realizations of a classifier associated with a subcomponent that satisfy the product line constraint.
* Methods to validate the product line constraint on an instance model – one taking the root instance, the other taking individual component instances. Both methods are assumed to be called as part of the instantiation as they utilize the configuredClassifierTypeReferenceCache for component instances. Any issues are reported through a collection of Diagnostic objects. Calls to both methods are commented out in the code. The issues need to be attached to the eResource or directly to the instance model itself.

Note: The FeatureLabels property is retrieved from the classifiers, which is what we want. However, this property is potentially assigned by multiple classifiers, i.e., is considered a conflicting assignment. Either we need to treat this property as a Meta property, or we need to support compositional values for property value collections.

# Use of JGraphT

JGraphT allows us to create graph structures layered on top of the AV3 instance model. This is done by utilizing instance model objects as representation for vertexes and edges. JGraphT provides a large and well maintained library of algorithms to process graphs.

I have created several methods that generate graphs from instance models. I have used JGraphX to visualize the generated graphs. The code can be found in org.osate.graph.util.AIJGraphTUtil

* Component instance hierarchy: Component instances are the vertexes and containment is represented by the default edge representation.
* Connection topology: Leaf component instances are the vertexes and connection instances are the edges.
* Propagation paths: Association instances can be of category connection, binding, or flow. Association instance ends are used as vertex objects, i.e., this can be feature instances or component instances (e.g., for access connections or bindings). Edges use the default representation to represent the association – the reason is that we may need to represent flows that do not have an explicit representation. For incoming features that are not sources of flows we generate edges to all outgoing features. Similarly, for outgoing features that are not the destination of any flow we generate edges from all incoming features. For flow sources the edge goes from the incoming feature instance to the component instance, and for flow sinks from the component instance to the outgoing feature instance. We have methods that can find the association instance from the edge endpoints.
* Behavior Propagation Graphs: This is a graph representation of an instance model that includes behavior rule instances with or without token literals as Meta data. Such graphs are the basis for effect token traces, as found in fault impact analysis, and for cause token traces, which then can be transformed into fault trees or minimal cut sets.

# Tag Propagation Analysis

We use two properties to specify a tag source and a tag sink. Tags are type references.

Tag sources can be associated with a component or a particular outgoing feature. Similarly tag sinks can be associated with components or particular incoming features.

Tags propagate from the tag source to a specified target along all possible paths. When a tag sink is encountered along the path the tag is masked.

We support two variants of tag propagation.

* One variant operates on the connection topology graph. In this case the tag sources and sink are associated with components. Propagations follow all outgoing edges of a component, i.e., all outgoing connections.
* The second variant operates on the propagation path graph. In this case, tag sources and sinks can be associated with the component, i.e., apply to all outgoing/incoming edges, or they can be associated with individual outgoing or incoming features. The edges represent both connections between components and flows from incoming to outgoing features of components.

For analysis of tag propagations, users identify a component instance that is the target of the propagation. The analysis then identifies all components that are tag sources and determines all paths from those sources to the target. For each path the analysis checks whether any component along the path is a tag sink and filters out the appropriate tag. The analysis reports all components that are tag sources and the tags that reach the target component.

# Specification of Behavior Rules

Behavior rules are used to specify behavior of actual processing as well as behavior of Meta data. Examples of Meta data are association of error types and their propagation throughout the system, and association of general Meta tokens and their propagation. The latter can be used to model security concerns, e.g., John Hatcliff’s proposal for token propagation.

Behavior rules can be stateless and stateful.

## Stateless Behavior Rules

Stateless behavior rules consist of conditions on inputs and actions that result in outputs.

Example: behavorrulename: <incoming condition> -> <actions>;

Conditions on inputs can be on the computation, i.e., conditions as found in the current BA, or conditions on Meta data such as error types and general tokens. We currently use an annex declaration such as @EM for error model and @BA for computational behavior to indicate the type of input and output data whose behavior is specified. For error behavior and general token behavior users can specify conditions on incoming token literals.

Token literals can be references to defined types, enumeration literals, string literals or even numeric literals. Condition elements are specification of whether a token of an incoming feature is contained in a specified collection (currently enclosed by “()”) using the ‘in’ operator.

Conditions can be multi literal operations with operators ‘any’, ‘all’, ‘oneof’, k ‘of’, k ‘ormore, k ‘orless’. Input conditions can be specified solely as reference to an incoming features without tokens. In this case, any incoming token is acceptable.

A behavior rule can have zero or more actions. No action indicates that the behavior is that of a sink indicated by the keyword ‘sink’ instead of an action specification. Actions can be computational functions (for BA type of processing) and assignment of values to outgoing features. In the case of EM and token Meta data a specific literal value can be assigned to one or more outgoing features. In the case of token the action may simply be an outgoing feature reference without a token value. In this case, the incoming token(s) are passed on as output.

Note that stateless behavior rules specify flows from one or more incoming feature to specific outgoing features. If the specification does not include Meta data (token literals) then we have the equivalent of V2 flow specifications supporting merge point specification (multiple inputs mapped to outputs).

## Stateful Behavior Rules

Behavior rules can be stateful. States of a state machine are defined as literals of an enumeration type. An instance of a state machine is specified in an annex subclause such as @EM or @BA as ‘uses states’ <enumeration type reference>.

States are referenced in stateful behavior rules.

Example: behaviorrulename: current\_state –[ incoming condition> ]-> target\_state { actions} ;

The full rule specifies given the current state and a condition a transition to the target state occurs and actions are taken.

A minimal rule specifies the current state and actions, which specifies how a state is propagated.

A rule can specify a transition only with a current state, a condition and a target state.

A state may carry one of a set of specified literal tokens (as we do in EMV2 where error states can carry error types).

## Generators

Generators represent sources of data or Meta data. In the case of @EM we use generators to model error events. A generator specification for Meta data indicates a collection of token literals from which a token instance is generated.

# Instantiation of Behavior Rules

The instance model is similar to that of Aadl V2 in stat we have component instances and feature instances with an attribute indicating the category. In V3 we have association instances with categories connection, binding, (and flow for flow specifications not expressed by behavior rules).

Component instances also contain state instances to represent the states expressed by enumeration literals, generator instances to represent generators, behavior rule instances, and action instances. Constrained Instance Objects (CIO) are used to represent instance objects with associated Meta data in the form of literals. Examples are the current and target state of a behavior rule. The CIO of the current state consists of a reference to a state instance and optionally to a collection of literals. The CIO of the target state consists of a reference to a state instance and optionally a single literal. The condition of a behavior rule instance is a copy of the condition in the behavior rule with all condition elements replaced by equivalent CIOs referring to incoming feature instances (or component instances for access or binding) and optionally a literal collection. Actions are recorded a CIO directly in the component instance with one or more behavior rule instances referring to the action CIO. The action CIO consists of a reference to a feature (or component) instance and optionally a single token literal.

# Token Propagation and Transformation

Here we are modeling a generalized version of the error type propagation and transformation supported by EMV2. We do so for forward effect propagation from sources to all effected components, as well as backward cause propagation to identify all contributors to an effect. We support this in several steps: 1) generation of a behavior propagation graph representation expressed in JGraphT, and 2) generation of an effect token trace, or a cause token trace, which can be a graph, and 3) possible transformation of the token trace to a more compact form. In the case of cause token traces, this form can be what is traditionally known as fault tree, or as minimal cut set.

## Behavior Propagation Graph

In the case of an instance model that includes behavior rule instances, the vertexes of the graph are the CIOs, and the edges the paths from action CIOs to condition element CIOs reflecting the endpoints of connections and bindings. Edges also represent paths from condition element CIOs to action CIOs as well as CIOs that represent current and target states.

If components do not include behavior rules then the vertexes and edges are those of the propagation graph with features instances (or component instances for access and binding) from outgoing to incoming corresponding to connections and bindings, as well as edges from incoming to outgoing feature instances to represent flows – if not explicitly specified from each incoming feature to all outgoing features.

Note that the generated graph takes into account constraints on acceptable token literals that can be propagated from actions to condition elements.

## Cause and Effect Token Traces

Token traces have their own Ecore model. A token trace consists of a collection of tokens that are organized into a graph structure. A token consists of an instance object reference (e.g., component or feature instance) and a single token literal. One token is the root token. Other tokens are sub token. Tokens can have operators to reflect to logical operators in multi literal operations that represent conditions or merge points of propagation paths.

Token traces are generated by effectively simulating the forward propagation of token instances from sources (generators) for effect token traces, or the backward propagation from specified endpoints for cause token traces – typically the outgoing features of the top-level component instance (which reflect the effect on the external/operational environment.