SAE AADL AS-5506A Errata Sheet

December 16, 2011

This document lists a set of errata and corrections to the SAE Architecture Analysis & Design Language (AADL) standard published in Jan 2009 under the SAE publication number AS-5506A. These errata have been approved by the committee.

Errata

Location	Errata	Correction	Rationale

4.1 (01), Syntax	An AADL specification consists of many packages and property sets. Current syntax rule allows only one package or property set.	Change the rule to (change shown in italics)
		
		`AADL_specification ::=`
		`  { package_spec | property_set }+`
			An AADL specification can consist of multiple packages and property sets. A tool may limit a file to contain only one package or one property set.

4.2 (02)	Missing rule: The component category in an alias declaration must match the category of the referenced component type.	The component category in an alias declaration must match the category of the referenced component type.	
4.2 (06) Naming Rule N(14)	An alias for a package can introduce a name that is the same as that of another package in the with clause with a single identifier. Currently the naming rule states that aliases have to be unique in the local namespace.	Add a statement to rule N(14) " If an alias_declaration defines an alias for a package then the alias name must not conflict with any package name listed in an import_declaration or that of the package containing the alias_declaration."	

4.4 (01)	Currently component implementation extends statements can bind prototypes of component implementations and types. Component implementations cannot bind	Change shown in italics:
		
		`component_implementation ::=`
		`  component_category implementation`
		`defining_component_implementation_na`
			We allow prototype binding for subcomponent declarations that refer to component types or component

	prototypes.	me *[prototype_bindings]*	implementations. We should allow binding of component type prototypes in component implementation declarations.
4.4 (02)	The standard is silent as to whether component implementations can refine prototypes declared in component types.	Prototype refinement in component implementations cannot refine prototypes declared for component types.	
4.5 (01), Syntax	It is currently not possible to bind prototypes when initializing a subcomponent array with component implementations.	Change the rule to array_element_implementation_list ::= (unique_component_implementation_reference *[prototype_bindings]* { , unique_component_implementation_reference *[prototype_bindings]* }*)	To be consistent about the ability to bind prototypes with actual we should allow the component implementation to be used for an array element to be declared with the prototype bound. Otherwise the user has to explicitly declare a classifier extension to bind them.
4.5 (03) legality rules	There are no constraints on which component implementations can occur in an array element implementation list.	Add: An array element implementation list is valid only if (a) the subcomponent classifier is a component type and (b) all component implementations in the list are implementations of the specified type.	The intent is to allow variation of the same component, but not different components. Otherwise, the array acts as a set.
4.7 (01) Syntax	The syntax for feature_group_type_prototype_actual does not allow for prototype bindings when specifying a unique_feature_group_type_reference.	Replace rule with feature_group_type_prototype_actual ::= (**feature group** unique_feature_group_type_reference *[prototype_bindings]*)	To be consistent about the ability to bind prototypes with actual we should allow binding of prototypes for FTG

			(**feature group** *feature_group_type_prototype*_identifier)	
4.7 (03)	Grammar issue: Missing parentheses in rule feature_prototype_actual, such that one cannot provide a classifier if the actual is a port.	Replace rule with feature_prototype_actual ::= (((in | out | in out) (event | data | event data) port [unique_component_classifier_reference]) | ((requires | provides) (bus | data | subprogram group | subprogram) access [unique_component_classifier_reference]) | ([in | out] feature feature_prototype_identifier)	
4.7 (04) Syntax	In feature_prototype_actual, when a feature actual is supplied it allows a unique classifier, but not a prototype of a classifier.	Replace rule with feature_prototype_actual ::= ((in | out | in out) (event | data | event data) port [unique_component_classifier_reference]) | ((requires | provides) (bus | data | subprogram group | subprogram) access [unique_component_classifier_reference]) | *([in | out] feature*	Meta Model supports the reference to the classifier, but not the prototype yet.

		feature_prototype_identifier)	
4.7 (05)	For component prototype declarations we need a legality rule about the referenced classifier category matching the category in the prototype declaration.	(L2) The component category of the optional component classifier reference in the prototype declaration must match the category in the prototype declaration.	

5.2 (01) Syntax	A subprogram call can refer to a provides subprogram access in a data type or in a subprogram group type, but not in an abstract type.	Add the following option to the syntax definition of "called_subprogram":
		
		`abstract_unique_component_type_reference .`
		`provides_subprogram_access_identifier`	

5.2 (02) syntax	Replace "*prototype*_identifier" with "*component_prototype*_identifier".	We currently allow any prototype to be referenced.	Meta Model supports reference to prototype and needs to be restricted to Component Prototype for enforcement by the Meta Model (in addition to enforcement by the compiler).
5.2 (04) Syntax	The syntax allows a subprogram call to refer to a requires subprogram access in the classifier, but not one in a feature group.	Add the following option to the syntax definition of "called_subprogram": *feature_group*_identifier . *requires_subprogram_access*_identifier	
5.2 (05)	The syntax allows a subprogram call to refer to a subprogram subcomponent, but subprogram implementations are not allowed to contain subprogram subcomponents.	Add subprogram subcomponent	
5.3 (01)	Subprogram groups cannot contain other	Allow for nested subprogram groups and	To better support modeling of

Legality Rules Table	subprogram group, only subprograms, i.e., no notion of nested subprogram libraries. Also subprogram groups are limited to contain subprograms, but not static data (data subcomponents).	allow data subcomponents inside them. The subprogram group type provides a way of limiting the visibility (access to) its elements. Add: data and subprogram group to Subcomponents in implementation. Add: provides subprogram group access to Features in type to allow access to an inner library. Update legality rules L1, L2, and L3.	static architectures following a HOOD type of paradigm the use of package name nesting was considered insufficient. In HOOD you can have non-terminal passive objects, i.e., objects that contain data instances and subprogram instances. (see [1] for further details on the topic and a white paper).
5.4 (01) Legality	A thread currently can provide access to a data subcomponent. This allows other threads to access that data concurrently without involving the containing thread.	Remove provided data access from the legal thread features.	However, the containing thread should manage concurrent access. If a thread wants to support protected access to a data object then you make accessible the subprogram feature of the data component as service function (provides subprogram access).
5.4 (03)	The standard is silent on what it means when the compute_execution_time is set to zero. Any dispatch of the thread would require the execution of a return instruction which uses up processor cycles. It is desirable to have it set to zero and mean that the thread is not dispatched when modeling a system with multiple threads each possibly being idle and its	Add: Dispatch_Able property. The Dispatch_Able property specifies whether a thread should be dispatched. Threads can be activated for dispatch in given modes, which is specified as part of the subcomponent declaration of the component using the thread. In some cases the thread itself may have modes and that	

	idle mode determined by the evaluation of a mission goal plan. If we were to express this through modes in the enclosing component, we would need a multiplicity of modes to activate and deactivate all desirable combination of idle threads.	mode determines whether the thread is active or idle. For example, various combinations of low level control threads may be active or idle at various points in time. Expressing this through modes in the enclosing component would lead to possibly having to model many mode combinations of subcomponents. Specification of zero compute_execution_time for a thread indicates that thread is dispatched and its application code decides there is nothing to do.	
6.1 (01) L4	Processor implementation only allows bus access connections.	Change L4 to: A processor implementation can contain bus access, *subprogram access, subprogram group access, port, feature, and feature group* connections. MM: remove parameter connection.	Port, subprogram (group) access, and feature (group) connections are needed in a processor to be able to connect to a feature of a virtual processor subcomponent of the processor.
6.1 (02) Properties	It is desirable to indicate whether a processor supports enforcement of protected address spaces. We also need to explain that the enforcement works whether we have virtual processors or not. In other words, enforcement is not a property of a virtual processor. However, a virtual processor may require address space enforcement if it is used to model a	Change: Runtime_Protection to also apply to virtual processor. Add: Runtime_Protection_Support to apply to virtual processor and processor.	

	system address space for device drivers.		
6.2 (01) Legality	Virtual processor implementation only allows bus access connections.	Change to: "Connections *yes*" Add L5: A virtual processor implementation can contain subprogram access, subprogram group access, port, feature, and feature group connections. MM: remove parameter connection.	Port, subprogram (group) access, and feature (group) connections are needed in a virtual processor to be able to connect to a feature of a virtual processor subcomponent.
6.3 (01) Legality	A memory component can contain a bus component, but not provide external access to the bus.	Change legality table to include *Provides bus access* as features. Change L4 to: A memory implementation can contain bus access connection declarations. *Bus access connections can connect a memory subcomponent to a bus subcomponent or a requires bus access feature, as well as connect a provides bus access feature to a bus subcomponent.*	Processors do allow access to their internal buses.
6.6 (01) Legality	Devices currently cannot be represented to maintain state.	Change legality table to allow: *Data* as subcomponent. Do not introduce provides data access. Add semantic description: A device component can contain a data subcomponent to represent persistent state. This data subcomponent cannot be made accessible via data access. Device behavior can be specified via Behavior Annex	Devices can have state that is relevant from the external perspective. The Behavior Annex has a need for that.

		subclauses, which can refer to the data subcomponent.	

8 (01) Syntax	The standard does not allow contained property associations for features, even if the feature is a feature group.	Change syntax rule
		`feature ::=`
		`...`
		`[ {` ***contained*** `property_association }+ } ] ;`
			Contained property associations allow properties to be associated with elements of a feature group.

8.1 (01) Syntax	Incorrect syntax rules. Does not allow identifier to be defined of abstract features.	Change rules to
		`abstract_feature ::=`
		defining_abstract_feature`_identifier : ( ( [ in | out ] feature [ unique_component_classifier_reference | component_prototype_identifier ] ) | ( [ in | out ] feature feature_prototype_identifier ) )`
		`abstract_feature_refinement ::=`
		defining_abstract_feature`_identifier : refined to ( ( [ in | out ] feature [ unique_component_classifier_reference | component_prototype_identifier ] ) | ( [ in | out ] feature`
			editorial

		`feature_prototype_identifier ) )`	
8.1 (03)	An abstract feature can refer to a feature prototype to allow parameterized refinement into a concrete feature. It also allows the specification of a component classifier or reference to a component prototype. In the latter case the abstract feature is not parameterizable into a concrete feature via prototype - only by refinement of the feature declaration. Note that refinement of a feature requires an extension of the enclosing component – we have introduced prototypes to avoid creating these component extensions and as a result refinement of all the places they are used.	Change syntax to: abstract_feature_spec ::= *defining_abstract_feature*_identifier : [**in** | **out**] **feature** [~~*component_classifier_reference* |~~ ~~*component_prototype_reference* |~~ *feature_prototype*_identifier]	All the feature parameterizations can be accomplished through the feature prototype.
8.2 (01) Syntax	Prototype refinement is not allowed in feature group type extensions	Change shown in italics: `[ prototypes ( { prototype | *prototype_refinement* }+ | none_statement ) ]`	Prototypes can be refined, so it should be allowed to refine a prototype in a feature group type extension.

8.2 (02) Syntax	The standard does not allow contained property associations in the properties of feature group types or feature groups.	Change shown it italics:
		
		`feature_group_type ::=`
		`  feature group defining_identifier`
		`…`
		`  [ properties ( { feature_group_property_association }+ | none_statement ) ]`
		`  { annex_subclause }*`
			Contained property associations allows us to attach property values to elements of a feature group.

		Same with feature_group_type_extension feature_group(_spec) feature_group_refinement	
8.2 (03) Syntax (ed)	The syntax definition for feature_group_refinement has mismatched parentheses.	Replace `(unique_feature_group_type_reference)	prototype_identifier)]` with `(unique_feature_group_type_reference | prototype_identifier)]`	
8.4 (03) Legality	There is no legality rule about not changing the provides or requires in refinement.	Add a new legality rule: "A provides subprogram access cannot be refined to a requires subprogram access and a requires subprogram access cannot be refined to a provides subprogram access. Similarly, a provides subprogram group access cannot be refined to a requires subprogram group access and a requires subprogram group access cannot be refined to a provides subprogram group access."	
8.6 (03) Legality 8.7 (02) Legality	There is no legality rule about changing the provides or requires in refinement or refining an abstract feature with direction.	Add two new legality rules: "A provides data access cannot be refined to a requires data (bus) access and a requires data (bus) access cannot be refined to a provides data (bus) access." "An abstract feature can be refined into a data (bus) access. In this case, the abstract feature must not have a direction specified."	

9 (01)	Connections have optional identifiers.	Change grammar rule to make defining identifier non-optional. Connection ::= ~~[~~defining_connection_identifier ~~]~~	They are the only model element for which the identifier is optional.
9 (01)	Connection refinement syntax rules are not consistent across the different kinds of connections. In the case of port connection we allow refinement of direction. This is necessary because of a legality rule (L1) that requires the ports to be inout if the connection is bidirectional. We do not allow this for other connection types, instead determine the direction from the feature.	Recommendation to change L1 to be consistent with the rules for other connections, i.e., in the case of a bi-directional connection the connection endpoints determine the connection direction.	In the case of bidirectional we propose to allow the direction of the flow to be inferred from the direction of the connection end, thus, allow refinement of abstract features to ports with in or out direction without requiring refinement of the connection as well.
9 (02) 9 (N1) and (N3)	The two rules conflict with each other. The issue is whether connection names must be unique for mode-specific connections, or whether the name can be used twice.	They must be unique. Change N1 to indicate even for mode-specific connections. Delete N3. S0: system s; S1: system s in modes (m1); S2: system s in modes (m2); connections Conn1: port S0.output -> S1.input in modes (m1); Conn2: port S0.output -> S2.input in modes (m2); Flows Ete1: end to end flow S0.f1 -> conn1 -> S1.f1 in modes (m1); Ete2: end to end flow S0.f1 -> conn2 -> S1.f1 in modes (m2); properties Latency => 1.5 ms applies to conn1; Latency => 2.5 ms applies to conn2;	

9.1 (01) Syntax	Feature connections don't require a keyword "feature".	Change syntax rule by adding the keyword:
		`feature_connection ::=`
		`  ***feature*** *source*_feature_reference`
		`connection_symbol`
		`  *destination*_feature_reference`
			All other connections use a keyword to indicate the connection type.

9.1 (02) Syntax	Feature connections cannot refer to a feature in a subprogram call.	Add the following option to feature_reference:
		
		`*subprogram_call*_identifier .`
		`*feature*_identifier`	

9.2 (04) (N4)	It is unclear what this means. Does this imply that any event source can only occur as the source of exactly one connection?	(N4) The event identifier of event source specifications (**self**.*event*_identifier) must not conflict with defining identifiers in the namespace of the component that contains the connection referencing the event source.	

9.3 (01) Syntax (ed)	The syntax definition of "parameter_reference" contains the following text:
	
	`component_type_parameter_identifier [ . parameter_identifier ]`
	
	The use of the word parameter is unclear.
		Replace the text with this:
		
		`component_type_parameter_identifier`
		`[ . data_subcomponent_identifier ]`	

9.3 (02)	The syntax for parameter_reference contains the following line: `*component_type_feature_group*_identifier [ . *element_port*_identifier`	Parameter connections should not refer to a feature group. Replace with this: `*component_type_feature_group*_identifier . *element_port*_identifier`	Would need a legality rule to ensure that the reference is to a data element (feature)

	] This allows a parameter connection to refer to a feature group.		
9.4 (04) Syntax	An access reference can be a feature group without an access identifier. It is not clear why the access identifier is optional.	Make the requires/provides_access_identifier non-optional. Along with changing the syntax, naming rule (N2) must be changed. The following text should be removed from (N2): "an incomplete feature group of the containing component type, ".	
9.4 (09) Syntax	Access connections cannot refer to a feature in a subprogram call.	Add the following option to rule access_reference: *subprogram_call*_identifier . *access*_identifier	Subprograms accessing shared data need to reference an incoming access.
9.5 (02) Syntax	Feature groups can have a direction, so uni-directional feature group connections should be allowed.	Replace "bidirectional_connection_symbol" with "connection_symbol" in syntax rule feature_group_connection.	
10.1 (01) Syntax	The syntax definitions for "flow_source_spec_refinement", "flow_sink_spec_refinement", and "flow_path_spec_refinement" require at least one property association.	The syntax should be rewritten to allow flow spec refinements to provide an in modes clause without requiring a property.	
10.1 (01) Syntax	The syntax for flow specifications and refinements only allows mode in the **in modes** clause.	Change to in_modes_and_transitions. Flow implementations and ETE flows are mode and transition specific.	
10.1	There is no statement in the standard to the	Proposed new wording with new text	

(02) Legality rules	fact that the flow implementation in modes must be consistent with the in modes of the referenced flow specs in the type if they are modal.	underlined: (L4) If the component implementation provides mode-specific flow implementations, as indicated by the **in modes** statement, then the set of modes and mode transitions in the **in modes** statement of all flow implementations for a given flow specification must include all the modes or mode transitions for which the flow specification is declared. (L5) In case of a mode-specific flow implementation, the connections and the subcomponents named in the flow implementation must be declared at least for the modes and mode transitions listed in the **in modes** statement of the flow implementation. (L7) Component type extensions may refine flow specifications and component implementation extensions may refine subcomponents and connections with **in modes** statements. A flow implementation that is inherited by the extension must be consistent with the modes and mode transition of the refined flow specifications, subcomponents, and connections if named in the flow implementation according to rules (L4) and (L5). Otherwise, the flow	

		implementation has to be defined again in the component implementation extension and satisfy rules (L4) and (L5).	
10.2 (03)	It is unclear what "Each flow implementation must be declared at most once" means. Does it imply that there can only be one flow implementation for each flow specification, or does it mean that any two flow implementations for the same flow specification must be different?	New text: (N1) The flow identifier of a flow implementation must name a flow specification in the component type. A flow implementation name may appear more than once in each component implementation, either as alternative flows under different modes or transitions (using **in modes**), or to represent multiple flows for the same flow specification such as replicated flows to support redundancy.	Instantiation needs to handle this.
10.2 (04) And 10.3 (01)	we allow multiple flow implementations for the same flow spec. A flow implementation can refer to a subcomponent without naming one of its flow specifications	1. It represents a separate flow implementation for each of the flow specifications of the subcomponent - if it has any 2. It represents an unnamed flow specification if no flow specification is declared for the subcomponent. Note that in this case you cannot attach or utilize flow related properties for the flow of that	Instantiation needs to handle this.

		subcomponent.	
10.2-05 Syntax	Flow implementation syntax allow properties to be attached. Table C.3 does not allow properties on flow implementations. See also 10.2-01 for comments on properties on flow implementations. Furthermore, 10.2 (8) states that if a flow specification needs to have a property value that is component implementation specific, this can be accomplished by a property association in the implementation that names the flow spec in the applies to.	Add sentence explaining that the property is associated with the flow specification as implementation specific property.	Most analyses operate on the instance model. In the instance model only flow specifications and end-to-end flows exist.
11.3 (06)	A modal property value should apply to one contained element only, otherwise it may get confusing, because we cannot resolve the mode name to a unique element. Alternatively we can allow many if all refer to the "same" (including extends) classifier.	Recommendation: apply to only one path when modal property values.	
11.3 (08) Paragraph 12	According to the lookup rules, features cannot inherit properties from a containing feature group type.	Allow features to inherit properties from their containing feature group type if the property is declared as inherit.	
11.3 (10) Semantics (17)	Lookup for record fields is too complex if overwriting individual fields is allowed.	Change 'overwritten' to 'assigned' and add 'the remaining fields retain their default value.'	
11.3 (11)	The description of the append operator should be clarified for nested lists.	Add the following: "When the operator +=> is used with nested lists, the local value is	

		added to the outermost list of the inherited value."	
9.2.3 (01) 11.3 (12)	Nested property value lists are needed for connection pattern values. Also require the correct set of parameters. Otherwise it adds to the ambiguity with parentheses (see 11.3(03)).	Allow **list of** (**list of**)*	

11.3 (03)	Parentheses are used around mode specific property values
	foo = > 20;
	foo => ((21) in modes (mode1)); -- list of one value in mode1
	foo => ((22) in modes (mode1), (23) in modes (m2));
	
	foo => (((first=> (21);)) in modes (mode1)); -- list of one record with one field having a single valued list in mode1
	foo => (((first=> (21);)) in modes (mode1),(first=> (22);) in modes (mode2));
	
	foo => (22 in modes (mode1), 24); -- last value is for all other modes
	foo => (const1); -- was acceptable in grammar as a mode specific value, it could also have been interpreted as a list of one value, or even a single Boolean value
	They are used in Boolean expressions:
	Clear => true; -- single value
		1) Eliminate parentheses around mode specific property values.
		2) Limit Boolean expressions to true/false, property constant reference and property reference. Full logical expressions will be handled by the constraint annex and currently can be delegated to the compute(function) expression.
		3) Use [] around record values
		
		The above examples:
		foo => 20; -- value applies to all modes
		foo => 21 in modes (mode1);
		foo => (21) in modes (mode1); -- list of one value in mode1
		foo => (22) in modes (mode1), (23) in modes (m2);
		
		foo => ([first=> (21);]) in modes (mode1); -- list of one record with one field having a single valued list in mode1
		foo => ([first=> (21);]) in modes (mode1),([first=> (22);]) in modes (mode2);	

	Clear => (true); -- AMBIG: parens around a single Boolean value or list of Boolean value Clear => (true and boolconst); -- expression They are used in lists and list of one value was allowed to leave off the parenstheses: Filenames => "file1"; -- list of one value Filenames => ("file1"); Filenames => ("file1","file2"); And they are used in grouping record values: Person => (name => "peter";); -- record with one field assignment Persons => ((name => "peter";)); -- list of one record with one field Persons => ((name => "peter"; children => (("Miriam", "rudi","Daniel"));)); -- one of the fields has a list of values We then introduced some restrictions in an attempt to disambiguate: 1) Boolean expressions cannot have parentheses at the outermost level. 2) Lists of one value need parentheses 3) Mode specific values need to have at least one in modes Given all that the parser still has to use backtracking to disambiguate.	foo => (const1); -- list of one value foo => (bconst); -- List of single Boolean value Clear => true; -- single value Clear => (true); -- list of a single Boolean value We could still allow lists of one value to not have brackets. However, since we have nested lists it gets complicated when the property expects a list of list of. Does the rule apply only for the outer list, or also inner lists? Lolo => "singlevalue"; -- leave off both parens Lolo => ("singlevalue"); -- leave off one parens Lolo => (("singlevalue")); -- have both there Lolo => ("val1", "val2"); -- outer list of two elements containing a list of one value (inner brackets left out), or list of one element containing a list of two values (outer brackets left out). **Decision: we do not allow leaving off brackets for single element lists**.	

11.4 (05) Semantics (12)	The lookup of field values is too complex.	State that a field without a value in a record expression has the default value, if any.	
12 (03) Syntax	Mode transition property association is currently not optional	Make it optional	
12 (04) Syntax	The production "unique_port_identifier" only allows for "port_identifier" and "subcomponent_identifier.port_identifier". It should also allow for "feature_group_identifier.port_identifier" and "subprogram_call_identifier.port_identifier".		
12(05)	The in modes statement allows **all** for in modes applied to components, but not otherwise. It indicates that the property value applies to all modes and it means that the mode identifier should match if the mode of the subcomponent is derived.	Remove **all**. Change naming rule N5 and legality rule L4 as well as Semantic (12) in Subcomponent.	To indicate that all modes apply is the same as not specifying an **in modes** with the model element. Name mappings are specified for each mode. If a subset of all modes apply and the names are identical, the rule for subcomponents allows the mapping “=> <subcomponent_mode_identifier> ” to be optional.