Modeling of System Families

Peter Feiler

July 2007

TECHNICAL NOTE
CMU/SEI-2007-TN-047

Performance-Critical Systems Initiative

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Tables

Abstract

Over their lifetime, systems exist in many forms, such as instances of a system deployed in different contexts or a system evolving over time. Variability may also occur in terms of functionality reflected in the domain architecture, nonfunctional properties (such as performance, reliability, and safety-criticality) that are realized in the runtime architecture, interfaces to the deployment environment with which the system interfaces, and mapping to computing platforms.

The Society of Automotive Engineers (SAE) Architecture Analysis & Design Language (AADL) is an industry-standard, architecture-modeling notation specifically designed to support a component-based approach to modeling embedded systems. This technical note discusses how AADL can be used to model system families and configurations of system and component variants. It shows that AADL supports system families by providing component types that are used to specify component interfaces and multiple implementations for each component type. This report also shows that AADL uses properties to represent multiple dimensions of system variability ranging from variation through conditional compilation to variation through different sets of calibration parameters.

1 Introduction

Over their lifetime, systems exist in many variations, such as instances of a system deployed in different contexts or a system evolving over time. Variability may also occur in terms of

- functionality that is reflected in the domain architecture of the system
- nonfunctional properties such as performance, reliability, and safety-criticality that are realized in the runtime architecture of the system
- interfaces to the deployment environment with which the system interfaces
- mapping to computing platforms

These variations can be represented as different application software, conditionally compiled application software, calibration parameters to the application software, and different configurations of the computing platform and deployment context (where the deployment context is the actual deployment environment or a simulated deployment environment—in the simplest case in the form of realistic test data sets). A system can be configured in terms of these variations before it becomes operational and may support switching between multiple configurations during operation.

The Society of Automotive Engineers (SAE) Architecture Analysis & Design Language (AADL) international industry standard is an architecture modeling notation specifically designed to support a component-based approach to the modeling of embedded systems. Components provide an abstraction that separates the interface with other components from component implementations, allows multiple implementations to be specified as component variants, provides multiple interface views for different users of a component, and supports the representation of a system hierarchy through composition of components. The application component view is a logical architecture view that organizes the domain functionality into manageable units; it primarily represents the static aspects of a system.

AADL explicitly supports modeling of the runtime architecture through processes, threads, and interaction models between them—providing a concurrency view in terms of a task and communication architecture that models the active components of a system and their interaction. AADL provides the concepts of mode and mode-specific properties to capture dynamic aspects of a system architecture. AADL provides component concepts of processor, memory, and bus to model the computing platform and the concept of device to model the operational environment. Also, the AADL provides properties to specify mappings of application components onto the execution platform, a capability that provides a deployment view of the system.

In this report, we examine how system families (i.e., different configurations of systems) can be modeled with AADL.[1] In particular, we discuss

- the AADL language concepts for component-based system modeling (Section 2)

[1] Except where noted, we discuss the initial version of the AADL language standard.

- modeling system variations without changes to the system structure and interaction topology (Section 3)
- modeling variation in the system hierarchy and component interaction topology (Section 4)
- modeling of variation in component interfaces (Section 5)
- modeling of variation in the deployed system (Sections 6 and 7)
- modeling of a reference architecture (Section 8)
- how capabilities of AADL V2, namely abstract components and parameterization of component templates, improve modeling of configurable system architectures (Section 9)

2 Component-Based Modeling

An AADL model is composed of component types, implementations, and instances. A component type defines the interface of the component to the other components. A component implementation defines the internal structure of the components. Both the component types and implementations represent classes of systems. The component instance represents a sample system (instantiated from a specific implementation) amenable to analysis. AADL supports component-based modeling through the following categories of components: system, process, thread group, thread, data, subprogram, processor, memory, bus, and device.

2.1 COMPONENT TYPES

A component type specifies the features (such as data, event, and message output) the component provides to other components through output ports, access provided to (data, bus) components contained within the component, and provided subprogram services. A component type also specifies the features (such as data, events, and message) the component requires from other components through its input ports, access the component requires to other components (data, bus), and subprogram services it requires. In addition, a component type provides a flow specification from the component's input ports to its output ports. A component type may have a set of property values that apply to all implementations and instances of the component unless explicitly overwritten in the component implementation or subcomponent (component instance) declaration.

Table 1 illustrates the specification of a **`thread`** type that has properties indicating that it is a periodic thread with a certain period. The **`features`** specify the interaction points with other components. The thread has **`data`** ports that communicate the latest value in a data stream, an **`event`** port to reset its computation, and data **`access`** declarations to indicate shared access to data components.

Table 1: Component Type Specification

```
thread controller
features
  -- provided features
  desiredValue: out data port BaseTypes::UInt16
{ Typing::DomainType => data appTypes::angleValue ;
  Typing::AngleRange => 0 degree .. 45 degree; };
  calibrationParameter: provides data access ;
  resetState: in event port;
  -- required features
  currentValue: in data port AppTypes::angleValue;
  setPoint: requires data access AppTypes::angleValue
{ Typing::BaseType => data BaseTypes::UInt32; };
flows
  signalFlow: flow path currentValue -> desiredValue;
properties
  Dispatch_Protocol => Periodic;
  Period => 20 ms;
end controller;

property set Typing is
  BaseType: classifier (data) applies to ( data, port );
  DomainType: classifier (data) applies to ( data, port );
  AngleRange: aadlinteger units ( degree ) applies to ( data, port );
end Typing;

 Source_Data_Size => 16 bits;
  end UInt16;
  data UInt32
  properties package BaseTypes
public
  data UInt16
  properties
Source_Data_Size => 32 bits;
  end UInt32;
end BaseTypes;

package AppTypes
public
  -- a data type with base types as property values
  data angleValue
  properties
    Typing::BaseType => data BaseTypes::UInt16;
  end angleValue;
end AppTypes;
```

In Table 1, we illustrate the use of data types on ports and data access to model the base type and domain type of data being interchanged. For that purpose, we introduce a **`package`** `BaseTypes` that contains a collection of base type definitions and a **`package`** `AppTypes` that contains application data type definitions. Furthermore, we introduce two **`properties`** that allow us to associate a base type or an application type with a port or access feature. The **`out data port`** `desiredValue` in the **`thread`** `controller` example in Table 1 has a data **`classifier`** that identifies the base type, while the domain type is recorded as `Typing::DomainType` property. This instance would be a direct reflection of component source code that does not use domain typing (e.g., Simulink models); in this case, domain types and other domain constraints,

such as a limit on the data value and its unit, are documented to support consistency checking in the AADL model.

The **`requires data access`** feature called `setPoint` in the `controller` **`thread`** example specifies the **`classifier`** in terms of a domain type. In this case, the `BaseType` property is used to record the base data type with the feature. (Note that we have defined a default base type representation with the data component type declaration.) This base type is then overridden by the `BaseType` property on the data access feature to indicate that this component makes a base type assumption different from the default value. Consistency between the base type specifications of both ends of a port or access connection can be checked on the AADL model, if not already checked by the application language. If the default base type is not overridden in the feature declaration, **`classifier`** matching of the endpoints of a connection already ensures that the base types match as well.

2.2 COMPONENT IMPLEMENTATIONS

A component implementation declaration specifies implementation-specific property values. It may specify subcomponent declarations (i.e., specification of component instances that are contained in the component) and how subcomponents interact with each other and with external components through the features of the component as expressed by connection declarations.

Properties in a component implementation may identify the source text that makes up the application logic. This source text may be written in any application language (i.e., programming languages such as C, Ada, or Java, or modeling languages such a Matlab/Simulink). Other properties specify characteristics of the application component that are relevant to architecture analysis (e.g., timing properties or reliability properties).

Subcomponent and connection declarations act as a blueprint for a component. Subcomponent declarations specify the parts of a component and their connections. The classifier of a subcomponent declaration identifies the type of component to be used as subcomponent. A classifier may refer to a component type—where the interface of the subcomponent is known and its connection to other subcomponents can be specified. A classifier may also refer to a component implementation—where the interface and the content of the subcomponent are known (i.e., we have specified more than one level of the system hierarchy). In other words, we can specify partially complete component blueprints by identifying only a component type or a more fully specified system hierarchy by identifying component types and implementations. We use the component **`extends`** concept to refine and extend a component type or implementation, as illustrated for the hybrid power train **`system implementation`** in Table 2.

Table 2: A Car System Architecture

```
system car
end car;
system implementation car.singleengine
subcomponents
  PowerTrain: system power_train.singleengine;
  ExhaustSystem: system exhaust_system;
end car.singleengine;

system power_train
features
  exhaustoutput: requires bus access Manifold;
end power_train;
system implementation power_train.singleengine
subcomponents
  ETC: system ThrottleController;
  ABS: system AntilockBrakingSystem;
  CruiseControl: system CruiseControl;
  Transmission: system Transmission;
  PowerPlant: device Engine;
end power_train.singleengine;
system implementation power_train.hybrid
    extends power_train.singleengine
subcomponents
  PowerPlant: refined to device Engine.Gasoline;
  AlternatePowerPlant: device Engine.Electric;
end power_train.hybrid;

system ThrottleController
features
  actualAngle: in data port;
  desiredAngle: out data port;
end ThrottleController;

system AntilockBrakingSystem
features
  slip: in data port;
  brakeActive: out data port;
end AntilockBrakingSystem;

system CruiseControl
system Engine
end Engine;
system implementation Engine.Gasoline
end Engine.Gasoline;
system implementation Engine.Diesel
end Engine.Diesel;
system implementation Engine.Electric
end Engine.Electric;
```

3 System Configuration by Properties

3.1 TYPES OF PROPERTY-BASED CONFIGURATION

In this section, we discuss how the following five forms of system variation can be modeled through property values:

1. variation through alternative source code files
2. variation through conditional compilation in source code
3. variation of execution platform binding
4. variation through application parameters that remain constant during the operation of a system, such as calibration parameters
5. variation of data sets used in the execution of system components

3.1.1 Alternative Source Code Files

The AADL standard defines a number of predeclared properties that specify mappings to application component source code. One such property is `Source_Text`. This property specifies the file name(s) that contains the source code for a component specified in AADL. This source code can be written in one of a number of source languages, which can be specified through the `Source_Language` property.

`Source_Text` property values (i.e., application source files) are typically associated with processes, threads, or subprograms called by threads. When a source file is specified, we assume that the application source code it contains complies with the AADL specification of the application component. We can check that compliance with tools that process both the source code and the component specification in AADL.

We can initially specify a system architecture without source file names, or we can preconfigure it with a default set of source files. We can then specify source file selections for a system architecture by declaring contained property associations in the properties section of the top-level system implementation in the same manner as we have specified the configuration selection of subcomponent implementations. An example of the use of contained property associations to specify `Source_Text` files is shown in Table 3.

Table 3: Two Configured Systems

```
system implementation car.diesel_automatic_java
  extends car.diesel_automatic
properties
  Source_Text => “Transmission_automatic.java”
      applies to PowerTrain.Transmission;
  -- other source files as configuration selections
  Source_Text => “Engine_Diesel.java”
      applies to PowerTrain.PowerPlant;
end car.diesel_automatic_java;
```

3.1.2 Conditionally Compiled Source Files

A single application source file can contain conditionally compiled code. In other words, portions of the source code are tagged, and this code is only included in the compilation if the appropriate tag value is set as one of the compilation parameters.

Typically, these conditional compilation tags act as system-wide configuration parameters (i.e., different source files use the same set of tags if they contain code fragments that address a particular system characteristic). For example, a conditional compilation tag may specify that the power train includes a turbo, and as a result some of the source code files will contain code fragments that perform special processing.

In the example in Table 4, we define an enumeration property type that introduces the tag values for a particular tag as literals, and we define a property whose enumeration literal value represents the tag value to be used for conditional compilation. In Table 4, the tag values are defined by `TurboType`, while the `Turbo` property represents the conditional compilation flag. If the values are Boolean, integer, or real, then we can use the corresponding built-in property types through the **`inherit`** concept. By doing so, we declare the property value of a tag property with the top-level component in the system hierarchy, and the property value applies to all components in the system hierarchy.

The conditional compilation property values can be utilized by a tool that generates build scripts from AADL models, populating a build script with the appropriate parameter values for the source code compiler.

Table 4: Conditional Compilation through Tag Values

```
property set CondComp is
  TurboType : type enumeration ( NoTurbo, Bosch, Eaton );
  Turbo : inherit TurboType applies to (all);
  Cylinders : inherit aadlinteger applies to (all);
end CondComp;


system implementation car.diesel_automatic_GTD
  extends car.diesel_automatic
properties
  CondComp::Turbo => Eaton;
  CondComp::Cylinders => 6;
end car.diesel_automatic_GTD;
```

In some source language systems, the presence of the conditional compilation tag itself indicates whether a code fragment should be included. In this case, we can use a simplified approach for representing the conditional compilation tags that make up a particular compilation configuration parameter set. In Table 5, we define an enumeration property type, whose enumeration literals are the names of the conditional compilation tags. We then define a property that accepts a list of those literals. This property is defined as **`inherit`**, and its values are declared in the top-level system implementation (i.e., the property value list applies to all components). Table 5 shows an example.

Table 5: Conditional Compilation through Tags

```
property set CondComp is
  CondCompTags : type enumeration ( Turbo, ABS, SixCylinder );
  CondCompParameter : inherit list of CondCompTags applies to (all);
end CondComp;

system implementation car.diesel_automatic_GTD
  extends car.diesel_automatic
properties
  CondComp::CondCompParameter => ( turbo, ABS );
end car.diesel_automatic_GTD;
```

3.1.3 Deployment Configuration: Software/Hardware Mapping

The AADL standard defines properties that specify the binding of application components to processor, memory, and bus components. These properties define actual binding through `Actual_<bindingtype>_Binding,` where `<bindingtype>` is processor, memory, or connection.

The binding is specified through contained property associations that are declared in the system implementation, the common root of the application system and the execution platform. The value of the binding refers to an execution platform component, and the **`applies to`** clause identifies the component the property value is associated with.

In Table 6, we use the binding properties in conjunction with inheritance (**`extends`** construct) to separate the logical composition from the hardware binding. This separation allows us to define an implementation that chooses the desired implementation of the application system and the desired implementation of the hardware platform. This system instance configuration is then tailored by specifying a software/hardware mapping through binding property values in the implementation extension.

Table 6: Alternative Deployment Configurations

```
system carSystem
end carSystem;
system implementation carSystem.DualProcessor
subcomponents
    carApp: system car.diesel_automatic;
    carECU: system ECU.DualProcessor;
end carSystem.DualProcessor;

system implementation carSystem.DualProcessorConfig1
    extends carSystem.DualProcessor
properties
  Actual_Processor_Binding =>
           reference carECU.ProcLeft applies to carApp.PowerTrain.ETC;
  Actual_Processor_Binding =>
           reference carECU.ProcRight applies to carApp.PowerTrain.ABS;
end carSystem.DualProcessorConfig1;

system implementation carSystem.DualProcessorConfig2
    extends carSystem.DualProcessor
properties
  Actual_Processor_Binding =>
           reference carECU.ProcLeft applies to carApp.PowerTrain.ABS;
  Actual_Processor_Binding =>
           reference carECU.ProcRight applies to carApp.PowerTrain.ETC;
end carSystem.DualProcessorConfig2;
```

3.1.4 Calibration Parameters and Other Constant System Parameters

We can also configure application systems by setting certain application parameters during system build, load, or start-up. In cases such as calibration parameters, these values may even be set through special tools while operation is suspended. We can model constant parameters such as those in AADL through property constants and property values.

Each application parameter can be defined as a property constant as shown in Table 7. Parameter values can be Boolean, string, integer and real with and without measurement unit, range of values, user-defined values in the form of enumeration literals, and lists of any of these values.

Property constants are not associated with specific components in a system architecture; instead, they provide global values. Property constants are defined in property sets. As a result, we must specify alternative configurations (in terms of different sets of values) as variants of the same property set. We can accomplish that by managing property sets through a version control system used on AADL models. In other words, we can select different parameter configurations by including the appropriate version of the property set in the workspace of an AADL modeling environment. Examples of system parameters as property constants are shown in Table 7.

Table 7: System Parameters as Property Constants

```
property set SystemParameters is
  Turbo: constant aadlboolean => false;
  ThrottleSetting: inherit aadlreal => 0.576 applies to (all);
  Speed: type units (kph);
  TopSpeed: constant aadlinteger SystemParameters::speed => 250 kph;

  TireType: type enumeration ( Touring, Winter, Offroad, Highspeed );
  Tires: constant SystemParameters::TireType => Touring;

end SystemParameters;
```

The parameters can also be modeled by properties. In this case, we define the properties in a property set and specify different configurations of parameter values through property associations in different implementations of the top-level system component. (For more information, see Section 3.2.1.) This process mirrors the approach taken for modeling conditional compilation tags or source file selection.

Properties are associated with components (and other AADL model elements such as connections and ports) of the system architecture. Consequently, property values are accessible only in the context of constructs such as system components, connections, and ports. We can define properties to be globally applicable by defining a property as **`inherit`** and associating it with the top-level system component, as shown in Table 8. When retrieving a property value for one of the system components, we follow the system hierarchy up to find a component that has the desired property value. Notice that the use of property sets for system parameters allows different property values to be used for different system components (i.e., different subsystems).

Table 8: System Parameters as Component Property Values

```
property set SystemParameters is
  Turbo: inherit aadlboolean applies to (all);
  ThrottleSetting: inherit aadlreal applies to (all);
  TopSpeed: inherit aadlinteger units ( kph ) applies to (all);

  TireType: type enumeration ( Touring, Winter, Offroad, Highspeed );
  Tires: inherit SystemParameters::TireType applies to (all);
end SystemParameters;

system car
end car;

system implementation car.hybrid
subcomponents
-- the car parts
connections
-- the connectivity between parts
properties
  SystemParameters::Turbo => false;
  SystemParameters::ThrottleSetting => 0.576;
  SystemParameters::TopSpeed => 195 kph;
  SystemParameters::Tires => Touring;
end car.hybrid;
```

Application parameters such as calibration parameters are part of the application data that gets loaded into the execution platform and consumes memory resources. A memory requirement may simply be specified for the application component through the `Source_Data_Size` property or `Source_Code_Size` property, depending on how programming language compilers include those constants in the code binaries. Therefore, it may not be necessary to explicitly model these parameters as data components of the application system architecture.

If desirable, we can model this calibration data by declaring a data component in a process or system, as illustrated in Table 9. We define the size of the calibration data area as a property of the data component type and apply it to all implementations. We can use the data implementation declaration to define a particular set of calibration data values. In the top-level system, the declaration of the calibration data component instance chooses the desired data set by naming the appropriate data component implementation.

Table 9: System Parameters as Data Components

```
system car
end car;

system implementation car.hybrid
subcomponents
  Calibration: data CalibrationData.config_1;
-- the car parts
connections
-- the connectivity between parts
end car.hybrid;

data CalibrationData
properties
  Source_Data_Size => 40 B;
end CalibrationData;

data implementation CalibrationData.Config_1
properties
  SystemParameters::Turbo => false;
  SystemParameters::ThrottleSetting => 0.576;
  SystemParameters::TopSpeed => 195 kph;
  SystemParameters::Tires => Touring;
end CalibrationData.Config_1;
```

3.1.5 Modeling of Data Sets

Data sets are used in simulation environments in two ways: (1) they represent data that is part of the application system, or (2) they represent a data stream from the embedded system environment such as sensors that are replayed for simulation runs. If the data is part of the application software, we can model the data set as a data component with the data set values as its values, as was done in Table 9.

If the data set represents a sensor data stream for simulation runs, we can represent it through a property associated with the device port through which the data stream is communicated. When associated with an **`out`** port, the data stream represents data to be sent to other components. When associated with an **`in`** port, it represents data that is expected to be received from other compo-

nents. This property may refer to a file that contains the data set or contain the data stream values, both of which are illustrated in Table 10.

In Table 10, we show the respective property being declared in a device implementation. In this representation, we can select different data sets by choosing different implementations for an instance of the device. Alternatively, we can use contained property associations that are declared at the top-level system component and apply to the appropriate port of the desired device instance. In the latter case, we select alternative data set configurations by choosing the appropriate top-level system component with the desired configuration parameters. The benefits of each approach are discussed in the next section.

Table 10: Simulated Device Data Set

```
device WheelSensor
features
  Signal: out data port Common::Real;
end WheelSensor;

device implementation WheelSensor.dataset1
properties
  Simulation::DataSetFileName => “dataset1.csv” applies to Signal;
end WheelSensor.dataset1;

device implementation WheelSensor.dataset2
properties
  Simulation::RealDataStream => ( 0.111, 0.211, 0.321, 0.432)
    applies to Signal;
end WheelSensor.dataset2;
```

3.2 MANAGING CONFIGURATION PROPERTIES IN ARCHITECTURE MODELS

In the examples of the previous sections, we have seen that properties representing the configuration of a system apply to specific components of the system or to the system as a whole. We can declare component-specific properties as part of a component type declaration or a component implementation declaration. When we declare them in a component type declaration, all instances of the component have the same property values. Similarly, when we declare them in a component implementation, all instances of that implementation have the same property values.

In Table 11, we show an example in which a throttle control variant is defined as an explicit component implementation by specifying a specific file as `Source_Text` property value. We can select alternative property values by choosing the appropriate component implementation in the subcomponent declaration of the parent component implementation declaration. In the next section, we discuss how to model component implementation selection throughout the system hierarchy.

Table 11: Properties Represent Component Implementation Variants

```
system ThrottleController
features
  actualAngle: in data port;
  desiredAngle: out data port;
end ThrottleController;

system implementation ThrottleController.Regular
properties
  Source_Text => “ETCLowOctane.c”;

end ThrottleController.Regular;
```

Different instances of the same component might require different property values. For example, different instances of a simulated device might require different seed values. We can also declare component-specific properties through a contained property association placed with the top-level system component and declared to apply to a specific instance in the system hierarchy, as in the following: `Simulation::SeedValue => 0.31415` **`applies to`** `sys1.subsys1.process2.device3`.

In this section, we examine how these properties can be managed as part of the declarative AADL model through a collection of component implementations of the top-level, system component, XML-based system configuration files that are associated with XML-based instance model files.

3.2.1 Properties as System Configuration Parameters

Some properties can be viewed as parameters of a system configuration. In this case, we specify all property values that act as parameters of a system configuration in a single location (i.e., the properties section of the top-level component implementation). For example, the properties specifying the binding of application components to the execution platform represent a particular runtime configuration of the operational application system, as illustrated in Table 6 on page 10.

Other properties apply to specific component instances in a system model. We can declare those properties in the top-level system implementation by indicating that they belong to a subcomponent (or features, connection, or other model element) through the **`applies to`** clause.

Table 12 illustrates the use of the **`extends`** construct on the top-level component implementation in defining a full system configuration in multiple steps. For this example, we declare a system implementation that chooses the application system implementation and the hardware platform implementation. Then extensions of this implementation specify the conditional compilation parameters. This implementation is the basis of two component implementation extensions that add alternative deployment configuration (processor binding) information.

Table 12: Partial and Full System Configurations

```
system carSystem
end carSystem;
system implementation carSystem.DualProcessor
subcomponents
    carApp: system car.diesel_automatic;
    carECU: system ECU.DualProcessor;
end carSystem.DualProcessor;

system implementation carSystem.DualProcessorCondCompConfig1
  extends carSystem.DualProcessor
properties
  CondComp::CondCompParameter => ( turbo, ABS );
end carSystem.DualProcessorCondCompConfig1;

system implementation carSystem.DualProcessorDeploymentConfig1
    exends carSystem. DualProcessorCondCompConfig1
properties
  Actual_Processor_Binding =>
           reference carECU.ProcLeft applies to carApp.PowerTrain.ETC;
  Actual_Processor_Binding =>
           reference carECU.ProcRight applies to carApp.PowerTrain.ABS;
end carSystem. DualProcessorDeploymentConfig1;

system implementation carSystem. DualProcessorDeploymentConfig2
    exends carSystem. DualProcessorCondCompConfig1
properties
  Actual_Processor_Binding =>
           reference carECU.ProcLeft applies to carApp.PowerTrain.ABS;
  Actual_Processor_Binding =>
           reference carECU.ProcRight applies to carApp.PowerTrain.ETC;
end carSystem.DualProcessorDeploymentConfig2;
```

The AADL standard does not prescribe how AADL models are stored as files in a file system. One possible mapping is to store each AADL package in a separate file. If it is desirable to store information about different configurations in separate files, each component implementation contains a set of configuration parameters as property associations in a separate package. Table 13 shows this circumstance, with the assumption that the `DualProcessor` system implementation is declared in a **`package`** `car::Baseline`.

Table 13: System Configuration in Separate AADL Package

```
package car::configuration2

system carSystem extends car::Baseline::carSystem

end carSystem;
system implementation carSystem. DualProcessorDeploymentConfig2
    exends car::Baseline::carSystem. DualProcessor
properties
  CondComp::CondCompParameter => ( turbo, ABS );
  Actual_Processor_Binding =>
           reference carECU.ProcLeft applies to carApp.PowerTrain.ABS;
  Actual_Processor_Binding =>
           reference carECU.ProcRight applies to carApp.PowerTrain.ETC;

end carSystem.DualProcessorDeploymentConfig2;

end car::configuration2;
```

3.2.2 Instance Models and System Configurations

AADL supports an XML-based persistent storage of AADL models. The AADL Meta Model and XML/XMI Interchange Representation Annex [SAE-AS5506/1 2006] defines an AADL-specific XML representation for declarative AADL models (i.e., those that correspond to the textual AADL representation), for AADL instance models (i.e., models that are instantiated from a specified system implementation as the root of an embedded system), and for system configurations (i.e., sets of property values that can be associated with AADL instance models).

The AADL XML representation allows AADL instance models to be stored in separate files. It also allows system configuration information in the form of property values that are associated with a specific AADL model instance to be stored in files separate from the AADL instance model file. These sets of property values can be specified to apply to specific modes of system operation.

4 Configuration of System Structure and Connection Topology

We might need to configure a system through component implementation selection for two reasons: (1) component variants are represented by different property values recorded in different component implementations, and (2) component variants represent variation in the system structure and connection topology (i.e., they differ in the set of subcomponents and connections, which are declared in component implementations).

In this section, we use **`extends`** declarations to express the classifier selection in subcomponents and to add connections. In Section 9, we explore the use of parameterized component classifier declarations, a feature of Version 2 of the AADL language, to model a reference architecture and its variants and instantiations.

4.1 IMPLEMENTATION SELECTION THROUGH COMPONENT EXTENSION

In this approach, we model the component blueprint by specifying component implementations that identify the classifier of a subcomponent by its component type. This identification allows us to specify connections between subcomponents and with the features of the component itself. However, the implementations of the subcomponent have not been chosen yet. In other words, we have not yet identified the substructure of the subcomponents, as illustrated in Table 14.

Table 14: A Common System Architecture

```
system car
end car;
system implementation car.common
subcomponents
  PowerTrain: system power_train;
  ExhaustSystem: system exhaust_system;
connections
  bus access ExhaustSystem.exhaustManifold -> PowerTrain.exhaustManifold;
end car.common;

system power_train
features
  exhaustManifold: requires bus access Manifold;
end power_train;

system implementation power_train.singleengine
subcomponents
  ETC: system ThrottleController;
  ABS: system AntilockBrakingSystem;
  CruiseControl: system CruiseControl;
  Transmission: system Transmission;
  PowerPlant: system Engine;
end power_train.singleengine;
system implementation power_train.twoengine
extends power_train.singleengine
```

Table 14: A Common System Architecture (cont.)

```
subcomponents
  AlternatePowerPlant: system Engine;
end power_train.twoengine;

system ThrottleController
features
  actualAngle: in data port;
  desiredAngle: out data port;
end ThrottleController;
system implementation ThrottleController.bosch
end ThrottleController.bosch;

system AntilockBrakingSystem
features
  slip: in data port;
  brakeActive: out data port;
end AntilockBrakingSystem;
system implementation AntilockBrakingSystem.bosch
end AntilockBrakingSystem.bosch;

system CruiseControl
end CruiseControl;
system implementation CruiseControl.delphi
end CruiseControl.delphi;

system Transmission
end Transmission;
system implementation Transmission.automatic
end Transmission.automatic;

system Engine
end Engine;
system implementation Engine.Gasoline
end Engine.Gasoline;
system implementation Engine.Diesel
end Engine.Diesel;
system implementation Engine.Electric
end Engine.Electric;

system exhaust_system
features
  exhaustManifold: provides bus access Manifold;
end exhaust_system;
system implementation exhaust_system.sporty
end exhaust_system.sporty;

bus Manifold
end Manifold;
```

These component implementations will be refined through a component implementation extension. The **`refined to`** construct in AADL allows the addition of detail in the component classifiers (i.e., the component type in the classifier reference may be refined to identify a specific implementation of the given component type), as shown in Table 15.

Table 15: A Specific System Configuration

```
system implementation power_train.diesel
    extends power_train.singleengine
subcomponents
  ETC: refined to system ThrottleController.bosch;
  ABS: refined to system AntilockBrakingSystem.bosch;
  CruiseControl: refined to system CruiseControl.delphi;
  Transmission: refined to system Transmission.automatic;
  PowerPlant: refined to device Engine.Diesel;
end power_train.diesel;

system implementation car.diesel
    extends car.common
subcomponents
  PowerTrain: refined to system power_train.diesel;
  ExhaustSystem: refined to system exhaust_system.sporty;
end car.diesel;
```

If we want to specify a different car configuration, one with a gasoline engine, we select a different implementation of the engine for the engine subcomponent in the power train implementation. This is illustrated in Table 16.

Notice in Table 16 that we had to introduce a new power train implementation and a new car implementation to reflect this configuration. We document implementation selections as configuration choices, as part of a component implementation. We had to document, for instance, a change to a selection at the leaf node of the system hierarchy in the enclosing component as a new implementation. Also, we had to record this new implementation in its enclosing component. This degree of documentation results in a proliferation of a new set of component implementations up the system hierarchy. In the next section, we will discuss how to address this issue by utilizing a property that specifies the desired component implementation as its classifier value and associates this value with the appropriate subcomponent in the system hierarchy.

Table 16: Configuration Change in one Component

```
system implementation power_train.gasoline
    extends power_train.singleengine
subcomponents
  ETC: refined to system ThrottleController.bosch;
  ABS: refined to system AntilockBrakingSystem.bosch;
  CruiseControl: refined to system CruiseControl.delphi;
  Transmission: refined to system Transmission.automatic;
  PowerPlant: refined to system Engine.Gasoline;
end power_train.gasoline;

system implementation car.gasoline
    extends car.common
subcomponents
  PowerTrain: refined to system power_train.gasoline;
  ExhaustSystem: refined to system exhaust_system.sporty;
end car.gasoline;
```

Notice also in Table 16 that we had to repeat the implementation choices of the other components. We can address this issue by specifying partially bound configurations of components. In other words, we can define a power train implementation with the implementation of all subcomponents specified, except for the engine. This component implementation can then be refined to a completely configured power train, as illustrated in Table 17.

Table 17: A Partially Bound System Configurations

```
system implementation power_train.selectEngine
    extends power_train.singleengine
subcomponents
  ETC: refined to system ThrottleController.bosch;
  ABS: refined to system AntilockBrakingSystem.bosch;
  CruiseControl: refined to system CruiseControl.delphi;
  Transmission: refined to system Transmission.automatic;
end power_train.selectEngine;

system implementation power_train.gasoline
    extends power_train.selectEngine
subcomponents
  PowerPlant: refined to system Engine.Gasoline;
end power_train.gasoline;
```

In the scenarios in this section, the modeler chooses an implementation of the top-level system component (in our example the car) as the root of a system instance. The implementation name acts as the desired configuration of the system. At each level in the system hierarchy, configurations are explicitly named. Including a new variant introduced to one of the leaf components in the system hierarchy in a configuration results in a new component implementation declaration (acting as an explicitly named configuration) for each enclosing component in the system hierarchy.

In summary, AADL supports partially configured systems by refining a subset of the classifiers. The resulting component implementation is then further refined. The fact that a component type or implementation is partially configured is inferred from incomplete subcomponent declarations. Implementation selection through component extension is achieved as follows:

1. Subcomponents are declared with component types as classifiers.
2. They are then refined to the desired implementation selection in **`extends`** declarations of the enclosing component implementation and **`refined to`** constructs of the subcomponent. If default implementations are specified, the current AADL standard does not allow their refinement in **`extends`** declarations.

In this approach, every configuration at each level of the system hierarchy is explicitly modeled by a named component implementation.

5 Modeling of Component Interface Variation

Component types can be declared as extensions of other component types, allowing interfaces to be refined and extended. Some features may be incompletely specified (i.e., may not name a component type or implementation in a **`classifier`** reference).

5.1 COMPONENT INTERFACE TEMPLATES

Component type declarations can represent a pattern of component interfaces. For example, the component type declaration shown in Table 18 specifies a set of interaction points through feature declarations without classifier references. In this case, the intended type of component interaction is reflected in the feature category, but specifics of the information being communicated are not committed yet. This component type declaration can be referenced in subcomponent declarations, allowing interactions with this component to be specified through connection declarations.

Table 18: Controller Component Type Template

```
thread controllerTemplate
features
  -- provided features
  desiredValue: out data port;
  -- required features
  currentValue: in data port;
  setPoint: requires data access;
properties
  Dispatch_Protocol => Periodic;
  Period => 20 ms;
end controllerTemplate;
```

If the specific set of interaction points varies between different variants of a component family, we can use a port group feature to specify a collection of interaction points as a single feature. By using a port group feature, we can specify that this component will interact with other components through a port group connection without specifying the number and types of interactions yet. This flexibility is useful when specifying patterns of architectural structures and interactions.

Table 19 shows an example of a generic component interface with a port group for each of the inputs and outputs through collections of ports.

Table 19: Component Type with Port Group Interaction Points

```
thread genericComponentTemplate
features
  providedValues: port group;
  requiredValues: port group;
end genericComponentTemplate;
```

5.2 COMPONENT INTERFACE REFINEMENT

These templates can be refined into more completely specified component interfaces through the use of the component type extension mechanism. This mechanism allows the modeler to refine an existing component type declaration and to extend its set of features.

We first look at the refinement of component types into more completely specified interfaces. This form of component type extension acts as a specialization of the component type. Table 20 refines the previously declared `controllerTemplate` into a throttle controller by refining the previously declared **`features`** to have specific domain data types.

Table 20: Component Type Refinement

```
thread ThrottleController extends controllerTemplate
features
  -- provided features
  desiredValue: refined to out data port appTypes::voltageValue;
  -- required features
  currentValue: refined to in data port appTypes::angleValue;
  setPoint: refined to requires data access appTypes::angleValue;
end ThrottleController;
```

In a second step, we refine this component interface to make use of specific base types for the data. We do this through a component type extension to the throttle controller type (see `ThrottleControllerUInt16` in Table 21). We introduce an additional component type as an extension for each desired base type combination for the features to allow base type matching on connections to be performed without having selected an implementation for a subcomponent.

Table 21: Type Selection for Ports

```
-- base type selection via type extension
thread ThrottleControllerUInt16 extends ThrottleController
features
  -- provided features
  desiredValue: refined to out data port appTypes::voltageValue.UInt16;
  -- required features
  currentValue: refined to in data port appTypes::angleValue.UInt16;
  setPoint: refined to requires data access appTypes::angleValue.UInt16;
end ThrottleControllerUInt16;
```

5.3 COMPONENT INTERFACE EXTENSION

An incompletely specified component type can also be viewed as a common interface as the basis for multiple component interface variants. Variants can be defined as component type extensions with additional features. This form of component type extension acts as a subclass of the component type being extended. Table 22 shows a variant of the throttle controller interface that permits calibration. The calibration parameter does not have its implementation specified; this allows the details of the calibration parameters to be supplied through refinement of the classifier to a data component implementation.

Table 22: Extended Component Interface

```
thread ThrottleControllerWithCalibration extends ThrottleController
features
  -- A component variation that permit calibration
  calibrationParameter: provides data access ThrottleCalibration;
end ThrottleController;
```

5.4 CONFIGURING COMPONENT INTERFACE VARIANTS

Component types are named in classifier references of features and subcomponent declarations. They may be named with an implementation name or by themselves. When making use of component interface specifications that are incomplete, the modeler should be aware that a component type reference cannot be refined into a different component type in the AADL standard, even if that component type is an extension of the original component type. Table 23 illustrates such an example. A generic control process is defined to consist of a controller, whose classifier is the generic controller template. Since this component type has ports specified, the component implementation can include connection declarations.

Table 23: Use of Generic Component Type

```
process ControlProcess
features
  Signalin: in data port;
  Signalout:  out data port;
end ControlProcess;
process implementation ControlProcess.impl
subcomponents
  -- A component variation that permits calibration
  controller: thread controllerTemplate;
connections
  data port SignalIn -> controller.currentValue;
  data port controller.desiredValue -> Signalout;
end ControlProcess.impl;
```

6 Variation in Deployment Configurations

AADL supports modeling of execution platform components through the component types of processor, memory, bus, and device. An AADL model of a system consists of an instance of an application system, an instance of an execution platform, and a specification of bindings of application components to execution platform components.

A processor component is an abstraction for an execution platform component that schedules and executes application threads bound to it. A processor type can be defined to represent processor hardware (as an abstraction of the hardware details) or processor hardware together with an operating system. For example, a dual core processor may be represented by a processor type that hides the fact that two CPUs are contained if the application has no awareness of and ability to bind to the individual CPU. A processor can have multiple implementations, where each implementation represents a different variant in a processor family. The models of the variants may only differ in the property values associated with each processor implementation.

In a similar fashion, the memory component can represent various types of storage media, and the bus component can be used to model entities that connect execution platform components (e.g., buses, networks, and wired and wireless connections). These physical connections may include connection protocols as part of their abstraction. The device component models a component of the physical target environment of an embedded system (e.g., sensors and actuators or a physical plant being controlled) where the device ports represent sensor and actuator inputs or outputs.

The system component configures individual execution platform components into systems. A system may have one or more than one system interface specification (system type declaration). Multiple system implementations can be defined for each system interface.

Modelers can define a family of embedded system configurations by specifying a top-level system that contains an application system subcomponent and an execution platform component. Different top-level system implementations may choose different execution platforms for the same application system.

First, we introduce the AADL package concept; then, we define hardware platforms and variations embedded systems configurations.

6.1 USE OF AADL PACKAGES

AADL packages[2] allow users to group collections of component type and implementation declarations into related groupings. AADL packages are named, and nested package names are supported. For example, a package may be named *SEI::Avionics::Classes.* Name nesting allows name uniqueness to be relative to the enclosing package name. AADL package declarations are not nested inside other AADL packages. However, tools may present a view that allows users to navigate a collection of packages in a user's workspace according to the package hierarchy.

[2] AADL packages are similar to Java packages.

There is no restriction imposed by AADL on the visibility of packages. However, AADL packages do restrict the visibility of component classifier declarations by providing public and private sections of the package. This allows classifiers in a given package to be referenced by any other package, as long as they are declared in the public section as shown in Table 24.

Table 24: Visibility of Packages

```
package BasicTypes
public
  data integer
  end integer;
  data implementation integer.u16
  properties
    Source_Data_Size => 16 bits;
  end integer.u16;
  data implementation integer.u32
  properties
    Source_Data_Size => 32 bits;
  end integer.u32;

  data real
  end real;
  data string
  end string;
end BasicTypes;

package CarDomain::Engine
public
  data Engine
  features
    getRPM: subprogram getTheRPM;
  end Engine;
  subprogram getTheRPM
  features
    rpm: out parameter integer.u16;
  end getTheRPM;
end CarDomain::Engine;
```

6.2 THE HARDWARE PLATFORM

Execution platform components—such as processors, memory, and network components—as well as components that represent the physical environment of an embedded system are represented by processor, memory, bus, and device component type and implementation declarations. These hardware component declarations represent a library of components; modelers can organize them into different packages, each representing a different supplier, and through nested package names into different product families. Table 25 illustrates the nested-package-name concept.

Table 25: Nested Package Names

```
package IBM::PowerPC
public
  processor PowerPC7XX
  end PowerPC7XX;
  processor PowerPC970
  end PowerPC970;
  processor implementation PowerPC970.Basic
  end PowerPC970.Basic;
end IBM::PowerPC;
```

If a component product family is large, we can organize it through a combination of component classifier extensions and multiple packages. In Table 26, we use the preceding example to define members of the FX family in a separate package.

Table 26: Defining Family Members in a Separate Package

```
package IBM::PowerPC::FX
public
  -- make the processor type name locally visible
  processor PowerPC970 extends IBM::PowerPC::PowerPC970
  end PowerPC970;
  -- introduce the FX variant as an extension of the basic one
  processor implementation PowerPC970.FX
      extends IBM::PowerPC::PowerPC970.Basic
  -- add any FX specific properties
  end PowerPC970.FX;
end IBM::PowerPC::FX;
```

These hardware components make up standardized modular motherboards, which themselves are configured into computing platforms for different embedded system platforms. In other words, we now define system types and implementations that represent different boards. These system components require or provide access to buses to model the fact that the board may be plugged into a PCI bus or connected to a 1553 bus, a CAN bus, or high-speed Etherswitch.

We can keep the board system declarations in one set of packages and the airframe-specific hardware architectures in other packages. The hardware architectures themselves can be defined as a family that is configurable by using the classifier extensions, explicit parameterization of component classifier declarations (an AADL Version 2 capability, see Section 9), and properties—as illustrated in Section 3.

In Table 27, we show packages that define motherboards with slots and components of certain types and that contain specific configurations of the populated motherboard. In this instance, we do not take advantage of the explicit parameterization of component classifier declarations. A model with explicit parameterization would require a separate prototype declaration for each sub-component classifier, which increases the size of the model.[3]

Table 27: Package Families

```
package IBM::Motherboard
public
  -- make the processor type name locally visible
  system Motherboard
  features
    pci: requires bus access PCI;
    Bus1553: requires bus access BUS1553;
  end Motherboard;
  -- introduce the FX variant as an extension of the basic one
  system implementation Motherboard.PPC
  subcomponents
    pr
oc : processor;
    mem1: memory;
    mem2: memory;
    mem3: memory;
    dma1: bus DMA;
  connections
    m1: bus access dma1 -> mem1.dma;
    m2: bus access dma1 -> mem2.dma;
    m3: bus access dma1 -> mem3.dma;
    p1: bus access dma1 -> proc.dma;
  end Motherboard.PPC;
end IBM::Motherboard;

package IBM::Motherboard::Populated
public
  system implementation Motherboard.PPC970
    extends Motherboard.PPC
  subcomponents
    proc : refined to processor IBM::PowerPC::PowerPC970;
    mem1: refined to memory DRAM;
    mem2: refined to memory DRAM;
    mem3: refined to memory DRAM;
  end Motherboard.PPC970;
end IBM::Motherboard::Populated;

package IBM::Motherboard::Configured
public
  system implementation Motherboard.PPC970HighPerformance
    extends IBM::Motherboard::Populated::Motherboard.PPC970
  properties
    Hardware::ProcessorSpeed => 1.4 GHZ applies to proc;
    Hardware::MemoryCapacity => 512 MB applies to mem1;
    Hardware::MemoryCapacity => 512 MB applies to mem2;
    Hardware::MemoryCapacity => 512 MB applies to mem3;
  end Motherboard.PPC970HighPerformance;
end IBM::Motherboard::Configured;
```

[3] In AADL, subcomponents can be independently refined with a classifier. Although the language allows for a mix of explicit parameterization and incomplete subcomponent declarations, it may be desirable to establish a modeling rule requiring prototypes for parameterization.

In Table 28, we illustrate descriptions of target platform-specific computer hardware configurations.

Table 28: Example Target Platform-Specific Hardware Configurations

```
package Lockheed::ComputePlatforms
public
  system ComputePlatform
  end ComputePlatform;
  system implementation ComputePlatform.F16
  subcomponents
    proc1 : system IBM::Motherboard::Motherboard;
    proc2 : system IBM::Motherboard::Motherboard;
    pci1: bus PCI;
    can1: bus BUS1553;
    can2: bus BUS1553;
  connections
    p1: bus access pci1 -> proc1.pci;
    p2: bus access pci1 -> proc2.pci;
    b1: bus access can1 -> proc1.Bus1553;
    b2: bus access can2 -> proc2.Bus1553;
  end ComputePlatform.F16;
end Lockheed::ComputePlatforms;

package Lockheed::ComputePlatformConfigurations
public
  system implementation ComputePlatform.F16HiPerf
    extends Lockheed::ComputePlatforms::ComputePlatform.F16;
  subcomponents
    proc1 : refined to
      system IBM::Motherboard::Motherboard.PPC970HighPerformance;
    proc2 : refined to
      system IBM::Motherboard::Motherboard.PPC970HighPerformance;
    pci1: refined to bus PCI.HighPerformance;
    can1: refined to bus BUS1553.V2;
    can2: refined to bus BUS1553.V2;
  end ComputePlatform.F16HiPerf;
end Lockheed::ComputePlatformConfigurations;
```

6.3 EMBEDDED SYSTEM CONFIGURATIONS

We can now define configurations of the complete embedded system (i.e., the embedded application software and the execution platform). The simplest form of such a configuration is a system implementation declaration that combines an instance of the application system and an instance of the computing platform (see Table 29). Again, we can define a generic configuration that is then refined into specific configuration variants.

Table 29: Configuration of an Embedded System

```
package Lockheed::F16EmbeddedSystem
public
  System implementation F16.GenericConfiguration
  subcomponents
    App:  system Lockheed::Avionics::F16Avionics;
    ComputePlatform: system Lockheed::ComputePlatformConfigurations::
      ComputePlatform;
  end F16.GenericConfiguration;

  System implementation F16.BaseConfiguration
    Extends F16.GenericConfiguration
  subcomponents
    App:  refined to system Lockheed::Avionics::F16Avionics.Baseline;
    ComputePlatform: system Lockheed::ComputePlatformConfigurations::
      ComputePlatform.Baseline;
  end F16.BaseConfiguration;

end Lockheed::F16EmbeddedSystem;
```

Given a specific system configuration, we can now define system variations that differ in how the application software is to be bound onto the computing platform hardware. AADL offers a set of predeclared properties to specify binding constraints as well as properties that record the actual binding decisions regarding binding of threads and processes to processors and memory, and connections to buses. The binding constraints are expressed through `Allowed_<type>_Binding` and `Allowed_<type>_Binding_Class` properties, where <type> is processor, memory, or connection. The former identifies specific instances of processors, memory, and buses to which an application component can be bound. The latter identifies the types of processors, memory, and buses that are acceptable targets of binding decisions.

It is expected that modelers take these constraints into consideration when making actual binding decisions. The actual binding decisions are then recorded (by a tool or a person) in the `Actual_<type>_Binding` properties. The actual binding properties can be recorded as additional system implementation declarations—as shown in the following example (see Table 30). Or the property values can be associated with the AADL instance model and stored as part of a modified instance model XMI file or as an XMI-based configuration file separate from the instance model. (See the instance model description of the AADL Meta Model and XML/XMI Interchange Format Annex AS-5506/1 [SAE-AS5506/1 2006].) Some toolsets may choose to limit users to specify binding constraints only and let tools store the actual bindings in the instance model. In this case, users may specify binding constraints to limit the choice to a single component.

Table 30: Binding Properties Recorded as Additional System Implementation Declarations

```
package Lockheed::F16EmbeddedSystem::Configurations
public
  System implementation F16.BaselineConfiguration extends
    Lockheed::F16EmbeddedSystem::F16.BaseConfiguration
  properties
    Allowed_Memory_Binding_Class => classifier nonvolatileDRAM
       applies to App.FlightManager.CriticalData;
    Allowed_Processor_Binding => reference ComputePlatform.proc1
       applies to App.FlightDirector.FDProcess;
  end F16.BaselineConfiguration;
end Lockheed::F16EmbeddedSystem::Configurations;

package Lockheed::F16EmbeddedSystem::BoundConfigurations
public
  System implementation F16.Baseline1 extends
Lockheed::F16EmbeddedSystem::Configurations::F16.BaselineConfiguration
  properties
    Actual_Memory_Binding => reference ComputePlatform.mem1
       applies to App.FlightManager.CriticalData;
    Actual_Processor_Binding => reference ComputePlatform.proc1
       applies to App.FlightDirector.FDProcess;
  end F16.Baseline1;
end Lockheed::F16EmbeddedSystem::BoundConfigurations;
```

The example in Table 30 shows the composition of embedded application software with the hardware at the top-level of the system hierarchy. AADL does not prescribe or limit modelers to compose systems that way. A system defined to consist of software and hardware components can itself become a component of a larger system. In other words, systems with hardware and software components may have an external interface that is documented in the root level system type. An instance of such a system can be integrated with another system by declaring both of them as subcomponents of a system of systems. They may be integrated by connecting them directly in terms of the application logic (port/connections) and the hardware platform (bus access). This integration may require additional integration infrastructure components, such as a network that ties the systems together.

7 Runtime Variation of System Configurations

The mode concept in AADL allows modelers to represent dynamic characteristics of embedded system architectures. We can define any component to have multiple modes. Transitions that are triggered by port events can be defined for modes.

Modelers can associate mode-specific property values with those components. For example, a thread may represent an algorithm that can calculate a trajectory at several levels of precision. A different execution time value can be associated with this thread for each of the levels of precision that the thread can operate in; this example is illustrated in Table 31. A second property may specify the level of precision that is achieved in each of the modes. When an instance of a component is connected to other components, the system instance model can be analyzed for possible inconsistencies in the levels of precision provided by one component and expected by another component under various mode combinations, if both components have modes.

Table 31: Mode-Specific Variation of System Components

```
thread implementation GPS.Deluxe
modes
  Normal : initial mode ;
  HiDef : mode ;
  properties
    Compute_Execution_Time => 10 ms in modes (Normal);
    Compute_Execution_Time => 15 ms in modes (HiDef);
end GPS.Deluxe;
```

Modelers can also declare that the processor binding of application components can have mode-specific property values—as illustrated in Table 32. In this example, we have modeled the instance where an application component may execute on one processor and later, during operation, on a different processor. This modeling technique can be used to represent the migration of application components between processors to balance the workload or to adapt to hardware failures.

Table 32: Mode-Specific Variation of Application Binding

```
properties
  Actual_Processor_Binding =>
           reference carECU.ProcLeft applies to carApp.PowerTrain.ETC in
modes nominal;
  Actual_Processor_Binding =>
           reference carECU.ProcRight applies to carApp.PowerTrain.ETC in
modes backup;
```

Finally, modelers can specify that subcomponents and connections are active only in certain modes. In other words, they can specify different application configurations of active treads and communication connections. Mode transitions represent runtime changes from one such runtime configuration to another. This allows us to represent different operational modes in which different subsystems may be active and in which they communicate in different ways. The example in Table 33 illustrates two runtime configurations of a hybrid car. In one operational mode, it runs on the diesel engine only, in the other operational mode it runs both engines as a hybrid.

Table 33: Mode-Specific Subcomponent Configuration

```
system implementation carSystem.DualProcessor
subcomponents
    carApp: system car.diesel_automatic in modes DieselOnly;
    carApp: system car.Hybrid_automatic in modes Hybrid;
    carECU: system ECU.DualProcessor;
modes
    DieselOnly: initial mode;
    Hybrid: mode;
end carSystem.DualProcessor;
```

8 A Car System Reference Architecture

In this section, we illustrate how the current version of the AADL standard can be used to represent an abstracted system description. In Section 9, we revisit this example to illustrate how the classifier parameterization feature in the next version of AADL (i.e., AADL Version 2) can improve the conceptual and runtime representation and the instantiation of this reference architecture.

8.1 THE CONCEPTUAL ARCHITECTURE

In Table 34, we define an abstract model of the system by using the system component to represent a general component concept.

Table 34: A Conceptual Model

```
system car
end car;

system implementation car.generic
subcomponents
  PowerTrain: system powertrain;
  ExhaustSystem: system exhaustsystem;
end car.generic;

system powertrain
features
  exhaustoutput: requires bus access Manifold;
end powertrain;
system exhaustsystem
features
  exhaustManifold: provides bus access Manifold;
end exhaustsystem;
```

8.2 REFINEMENT INTO A RUNTIME ARCHITECTURE

To turn the abstract model into runtime architecture, we make decisions about which components require space partitioning (refinement into process) and which components are active components that can execute concurrently with other active components.

In the example shown in Table 35, we simply refine the classifier by replacing the **`system`** keyword with **`process`** as appropriate. The two subsystems, `PowerTrain` and `ExhaustSystem`, are redefined as **`process`** classifiers. Within `powertrainProcess`, we choose **`thread group`** as the component category. The **`thread group`** allows those components to consist of multiple threads, while the different components within `powertrainProcess` share an address space.

Table 35: A Runtime Architecture

```
system carRT
end carRT;
-- prototypes bound to actuals that are of the process category
system implementation carRT.impl
subcomponents
  PowerTrain: process powertrainProcess;
  ExhaustSystem: process exhaustsystemProcess;
end carRT.impl;

process powertrainProcess
end powertrainProcess ;

process implementation powertrainProcess.dualengine
subcomponents
    TheEngine: device Engine;
    TheAlternateEngine: device Engine;
    TheTransmission: thread group Transmission;
    Throttle_Controller: thread group ThrottleController;
    Antilock_Braking_System: thread group AntilockBrakingSystem;
    Cruise_Control: thread group CruiseControl;
end powertrainProcess.dualengine;

process exhaustsystemProcess
end exhaustsystemProcess;
```

8.3 SPECIFIC CARS BASED ON RUNTIME ARCHITECTURE

We can now define specific car types. We can do so with respect to the conceptual architecture or the runtime architecture. In Table 36, we refine the reference runtime architecture for cars into a specific brand. We also refine the `power_train` reference architecture into a specific instance, namely a hybrid gas/electric configuration. It is used to refine the reference implementation of the car runtime architecture into the specific brand Toyota Prius. The table does not show the refinement of the exhaust system into its sporty implementation.

Table 36: Defining Specific Car Types

```
system Toyota extends carRT
-- bind the component category to be system
end Toyota;

system implementation Toyota.Prius
extends carRT.impl
subcomponents
   power_train: refined to process powertrainProcess.Toyota_hybrid;
     exhaust_system: refined to process exhaustsystemProcess.sporty;
end Toyota.Prius;

process implementation powertrainProcess.Toyota_hybrid
 extends powertrainProcess.dualengine
subcomponents
    TheEngine: refined to device Engine.gasoline;
    TheAlternateEngine: refined to device Engine.Electric;
    TheTransmission: refined to thread group Transmission.Automatic;
    Throttle_Controller: refined to thread group ThrottleController.Bosch;
    Antilock_Braking_System: refined to thread group
AntilockBrakingSystem.Bosch;
    Cruise_Control: refined to thread group CruiseControl.Delphi;
end powertrainProcess.Toyota_hybrid;
```

9 Modeling with AADL Version 2

The SAE AADL standards committee is making improvements to the AADL notation based on user feedback. These improvements are in review and will go into ballot in 2007 and be published as SAE AADL Version 2 in early 2008. In this section, we present two of these improvements intended to improve the way reference architectures and their instantiations can be modeled.

The first of two new capabilities is the ability to declare component types and implementations without choosing a specific concrete component category. In other words, we can declare components with the keyword `abstract` and later refine it into any of thread, thread group, process, system, data, subprogram, processor, memory, bus, and device.

The second capability is the ability to explicitly specify parameters for component type and implementation declarations that must be supplied to complete the classifier specification. We specify the classifier parameter as a prototype.

In Table 37, we explicitly specify the parameterization of the component type through **`prototypes`** declarations for two **`data`** types to be supplied. One prototype is used in two **`features`** declarations. This ensures that both **`features`** have the same **`data`** type.

Table 37: A Parameterized Component Type

```
thread controllerTemplate
prototypes
  dt : data;
  sd : data;
features
  -- provided features
  desiredValue: out data port prototype dt;
  -- required features
  currentValue: in data port prototype dt;
  setPoint: requires data access prototype sd;
end controllerTemplate;
```

In Section 9.1, we illustrate the use of classifier parameterization to configure system architectures by supplying the desired classifiers for subcomponents throughout the hierarchy of the system architecture. Then, in Section 9.2, we revisit the exercise of modeling a conceptual architecture, the runtime architecture, and their instantiation into actual system architectures.

9.1 IMPLEMENTATION SELECTION BY CLASSIFIER PARAMETERIZATION

The **prototypes** declaration acts as an explicit parameter specification for a component type and component implementation declaration. This declaration may specify

- a component category, indicating that a component classifier of the specified category must be supplied
- a component type, indicating that a component implementation of the specified type must be supplied
- an abstract component type, indicating that component classifiers of any category that can be instances of this prototype (i.e., whose features, flows, and properties match those of the template) can be supplied
- an abstract component implementation, indicating that component classifiers of any category that can be instances of the template implementation (i.e., whose features, flows, and properties match those of the generic component type and whose subcomponents and connections match those of the generic component implementation) can be supplied

We illustrate the use of prototypes in two ways. In the first approach, we define the complete system hierarchy with selectable component implementations to be used as subcomponent classifiers specified as **prototypes**. These **prototypes** of lower level components are recursively declared as **prototypes** in the enclosing component. In other words, all component implementation selections of a system configuration are prototypes of the top-level system implementation. In the second approach, we define subcomponent classifiers as **prototypes** of the directly enclosing component implementation only. In this case, we select classifiers at one level of the system hierarchy at a time. The second approach is similar to the use of component extensions discussed in Section 4.1.

9.1.1 A Configurable Car System Architecture

Table 38 illustrates the use of prototypes to specify the car example as a configurable architecture. In this example, we make all implementation selections of the system accessible as prototypes of the top-level system component. A number of the top-level prototypes are passed on as actual **prototypes** values for the classifier of a subcomponent, while one prototype is used as a classifier of the subcomponent itself.

Table 38: Parameterization via AADL Prototypes

```
system car
end car;
system implementation car.single
prototypes
  tc: system ThrottleController;
  abs: system AntilockBrakingSystem;
  cc: system CruiseControl;
  tm: system Transmission;
  en: device Engine;
  exhaust_system: system exhaustsystem;
subcomponents
  PowerTrain: system powertrain.singleengine(
                Throttle_Controller => prototype tc,
                Antilock_Braking_System => prototype abs,
                Cruise_Control => prototype cc,
                TheTransmission => prototype tm,
                TheEngine => prototype en);
  ExhaustSystem: prototype exhaust_system;
end car.single;
system implementation car.dual extends car.single
prototypes
  se: device Engine;
subcomponents
  PowerTrain: system powertrain.dualengine(
          Throttle_Controller => prototype tc,
          Antilock_Braking_System => prototype abs,
          Cruise_Control => prototype cc,
          TheTransmission => prototype tm,
          TheEngine => prototype en,
          TheAlternateEngine => prototype se);
  ExhaustSystem: prototype exhaust_system;
end car.dual;

system powertrain
features
  exhaustoutput: requires bus access Manifold;
end powertrain;

system implementation powertrain.singleengine
prototypes
  Throttle_Controller: system ThrottleController;
  Antilock_Braking_System: system AntilockBrakingSystem;
  Cruise_Control: system CruiseControl;
  TheTransmission: system Transmission;
  TheEngine: device Engine;
subcomponents
  ETC: prototype Throttle_Controller;
  ABS: prototype Antilock_Braking_System;
  CruiseControl: prototype Cruise_Control;
  Transmission: prototype TheTransmission;
  PowerPlant: prototype TheEngine;
end powertrain.singleengine;

system implementation powertrain.dualengine
    extends power_train.singleengine
prototypes
  TheAlternateEngine: device Engine;
subcomponents
  AlternatePowerPlant: prototype TheAlternateEngine;
end powertrain.dualengine;
-- throttle controller etc. as before
```

Table 39 illustrates a configured car system architecture by supplying classifiers as actuals for all prototypes.

Table 39: Configuration Selection via AADL Prototype Actuals

```
system implementation Toyota.Prius
extends car.dualengine (
   en => device Engine.gasoline,
   se => device Engine.Electric,
   tm => system Transmission.Automatic,
   tc => system ThrottleController.Bosch,
   abs => system AntilockBrakingSystem.Bosch,
   cc => system CruiseControl.Delphi,
  exhaust_system => system exhaustsystem.sporty );

end Toyota.Prius;
```

With this approach, we can also limit the selection to preconfigured subsystem configurations by not exposing their subcomponent classifiers as prototypes. In our example, we could have preconfigured the power train with two fully configured component implementation descriptions. In that case, the top-level system implementation would have a prototype for the power train, replacing the prototypes for the classifiers to be supplied to the power train declaration within the car implementation. The `powertrain.Hybrid` declaration of Table 40 defines the power train as a named system implementation, which is a refinement of the `powertrain.dualengine` template. This system implementation is then referenced in the `Toyota.Prius` declaration, which is refined from the `car.generic` template.

Table 40: Preconfigured Subsystems as Prototype Actuals

```
system car
end car;

system implementation car.generic
prototypes
  power_train: system powertrain;
  exhaust_system: system exhaustsystem;
subcomponents
  PowerTrain: prototype power_train;
  ExhaustSystem: prototype exhaust_system;
end car.generic;

system implementation powertrain.Hybrid
extends powertrain.dualengine (
   PowerPlant => device Engine.gasoline,
   AlternatePowerPlant => device Engine.Electric,
   TheTransmission => system Transmission.Automatic,
   Throttle_Controller => system ThrottleController.Bosch,
   Antilock_Braking_System => system AntilockBrakingSystem.Bosch,
   Cruise_Control => system CruiseControl.Delphi);
end powertrain.Hybrid;

system implementation Toyota.Prius
extends car.generic (
   PowerTrain => system powertrain.Hybrid,
  ExhaustSystem => system exhaustsystem.sporty );
end Toyota.Prius;
```

9.1.2 Configuration by Nested Prototype Actual Specifications

In this scenario, we leverage the `car.generic` declaration of Table 40. However, we do not assume to have preconfigured power train configurations. Instead, we define the power train configuration at the time it is supplied with specific component classifiers for the prototypes. This approach results in nested prototype specifications shown in Table 41.

Table 41: Configuration Nested Prototype Actuals

```
system Toyota extends car
end Toyota;
system implementation Toyota.Prius
    extends car.generic (
          power_train => system powertrain.dualengine (
                  TheEngine => device Engine.gasoline,
                  TheAlternateEngine => device Engine.Electric,
            TheTransmission => system Transmission.Automatic,
            Throttle_Controller => system ThrottleController.Bosch,
            Antilock_Braking_System => system AntilockBrakingSystem.Bosch,
            Cruise_Control => system CruiseControl.Delphi ),
     exhaust_system => system exhaustsystem.sporty );
end Toyota.Prius;
```

We can also work with predeclared configurations of subsystems. If we do that for the power train, the result is the example shown in Table 40 on page 40.

9.2 A CAR SYSTEM REFERENCE ARCHITECTURE REVISITED

In this section, we illustrate how the abstract component category as well as classifier parameterization can be used to represent a system reference architecture and its instantiation.

9.2.1 The Conceptual Architecture

In Table 42, we define a conceptual model of the system through the use of the **`abstract`** category.

Table 42: A Conceptual Model

```
component car
end car;component implementation car.generic

prototypes
  power_train: abstract powertrain;
  exhaust_system: abstract exhaustsystem;
subcomponents
  PowerTrain: prototype power_train;
  ExhaustSystem: prototype exhaust_system;
end car.generic; component powertrain

features
  exhaustoutput: requires bus access Manifold;
end powertrain; component exhaust_system

features
  exhaustManifold: provides bus access Manifold;
end exhaust_system;component implementation exhaust_system.sporty
end exhaust_system.sporty;
```

9.2.2 Refinement into a Runtime Architecture

When we want to turn the abstract model into a runtime architecture, we can make decisions about which components require space partitioning (refinement into process) and which are active components that can execute concurrently with other active components.

In the example shown in Table 43, we simply refine the classifier by referencing an abstract component classifier and qualifying it with the concrete category **`process`**.

Table 43: Runtime Architecture

```
system carRT extends car
end carRT;
-- prototypes bound to actuals that are of the process category
system implementation carRT.impl
   extends car.generic (
       power_train => process powertrain;
       exhaust_system => process exhaust_system; );
end carRT.impl;
-- prototypes restricted to the process category
system implementation carRT.impl
   extends car.generic
   prototypes
       power_train : refined to process powertrain;
       exhaust_system : process exhaust_system;
end carRT.impl;
```

In the example shown in Table 44, we go through an explicit step of defining classifiers of the concrete category as named classifiers that then are referenced in the refinement of the whole car architecture.

Table 44: Defining Classifiers as Named Classifiers

```
process powertrainProcess extends powertrain
end powertrainProcess ;

process exhaust_systemProcess extends exhaust_system
end exhaust_systemProcess;
system carRT extends car
end carRT;
system implementation carRT.impl
   extends car.generic (
       power_train => process powertrainProcess;
       exhaust_system => process exhaust_systemProcess; );

end carRT.impl
```

9.2.3 Specific Cars Based on Runtime Architecture

We can now define specific car types. We can do so with respect to the conceptual architecture or the runtime architecture.

In Table 45, we assume that the **process** powertrain has been defined to consist of several components, and each component classifier is specified as prototype. In the example, we supply a concrete component category (**thread group**). **Thread group** allows those components to consist of multiple threads, while the different components within the powertrain share an address space.

Table 45: Defining Specific Car Types

```
system Toyota extends car
-- bind the component category to be system
end Toyota;

process Toyota_powertrain extends powertrain
end Toyota_powertrain;

process implementation Toyota_powertrain.hybrid
 extends powertrain.dualengine (
    TheEngine => device Engine.gasoline,
    TheAlternateEngine => device Engine.Electric,
    TheTransmission => thread group Transmission.Automatic,
    Throttle_Controller => thread group ThrottleController.Bosch,
    Antilock_Braking_System => thread group AntilockBrakingSystem.Bosch,
    Cruise_Control => thread group CruiseControl.Delphi );
end Toyota_powertrain.hybrid;
```

In Table 46 we define the specific car implementation (Toyota.Prius) in terms of the runtime architecture (carRT.impl). In this case, the decision that parts of the power_train share an address space by being threads inside a **process** has been made for us. When we specify Toyota.Prius, we supply two processes (Toyota_powertrain.hybrid and exhaustsystem.sporty) for the prototypes of the runtime architecture carRT.impl.

Table 46: Defining the Car Type in Terms of Runtime Architecture

```
system implementation Toyota.Prius
extends carRT.impl (
   power_train => process Toyota_powertrain.hybrid,
     exhaust_system => process exhaustsystem.sporty );
end Toyota.Prius;
```

9.2.4 Specific Cars in Terms of Conceptual Architecture

In Table 47, we define a specific car based on the conceptual architecture.

Table 47: Defining a Specific Car

```
system implementation Toyota.Prius
extends car.generic (
   power_train => process Toyota_powertrain.hybrid,
     exhaust_system => process exhaustsystem.sporty );
end Toyota.Prius;
```

We can supply a power train that is a system of processes to indicate that we want address space protection between ABS, cruise control, and so on within the power train. In this case, we utilize the nested actual declaration notation to illustrate the example, as Table 48 shows.

Table 48: Defining a Power Train for the Specific Car

```
system implementation Toyota.Gasoline
extends car.generic (
  power_train => system powertrain.singleengine (
            TheEngine => device Engine.gasoline,
            TheTransmission => process Transmission.Automatic,
            Throttle_Controller => process ThrottleController.Bosch,
            Antilock_Braking_System => process AntilockBrakingSystem.Bosch,
            Cruise_Control => process CruiseControl.Delphi ),
  exhaust_system => process exhaustsystem.sporty );
end Toyota.Gasoline;
```

10 Summary

In this technical note, we discussed how AADL can be used to model system families and configurations of system and component variants. We have shown that AADL supports system families by providing component types to specify component interfaces and multiple implementations for each component type. We have shown how multiple dimensions of system variability ranging from variation through conditional compilation to variation through different sets of calibration parameters can be represented by properties.

We illustrated several approaches for organizing these properties. We have shown how the ability to define component types and implementations as extensions of previously defined component types and implementations is used to manage variation along these dimensions. We discussed the ability of AADL to support modeling of system configurations that change at runtime during system operation. We illustrated how a system reference architecture and its instantiation can be expressed in AADL. Finally, we discussed proposed revisions to AADL that support component implementation configuration through classifier parameterization.

References

[SAE-AS5506/1 2006]
Society of Automotive Engineers. *SAE Standards: AS5506/1, Architecture Analysis & Design Language (AADL) Annex Volume 1*, June 2006. http://www.sae.org/technical/standards/AS5506/1

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	July 2007	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Modeling of System Families	FA8721-05-C-0003

6. AUTHOR(S)
Peter Feiler

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2007-TN-047

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Over their lifetime, systems exist in many forms, such as instances of a system deployed in different contexts or a system evolving over time. Variability may also occur in terms of functionality reflected in the domain architecture, nonfunctional properties (such as performance, reliability, and safety-criticality) that are realized in the runtime architecture, interfaces to the deployment environment with which the system interfaces, and mapping to computing platforms.

The Society of Automotive Engineers (SAE) Architecture Analysis & Design Language (AADL) is an industry-standard, architecture-modeling notation specifically designed to support a component-based approach to modeling embedded systems. This technical note discusses how AADL can be used to model system families and configurations of system and component variants. It shows that AADL supports system families by providing component types that are used to specify component interfaces and multiple implementations for each component type. This report also shows that AADL uses properties to represent multiple dimensions of system variability ranging from variation through conditional compilation to variation through different sets of calibration parameters.

14. SUBJECT TERMS	15. NUMBER OF PAGES
AADL, architecture analysis and design language, system families, model-based engineering	54

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL