



# Using the AADL for mission critical software development

paper presented at the ERTS conference,  
Toulouse, 21 January 2004

**Pierre Dissaux,**  
pierre.dissaux@tni-world.com

*TNI-Europe Limited  
Mountbatten Court, Worrall Street  
Congleton CW12 1DT  
UK  
+44 (0) 1260 291449  
www.tni-world.com*

## 1 Introduction

The Avionics Architecture Description Language (AADL) is an emerging standard, prepared by the Society of Automotive Engineers (SAE), Architecture Description Language Subcommittee, Embedded Computing Systems Committee, Aerospace Avionics Systems Division (AS-2C). The AADL standard is based on MetaH, an avionics architecture description language and toolset developed at Honeywell Laboratories under the sponsorship of the US Defense Advanced Research Projects Agency (DARPA) and US Army Aviation and Missile Command (AMCOM).

The Avionics Architecture Description Language (AADL) is a computer language used to describe the software and hardware components of an avionics system and the interfaces between those components. Extension to other mission critical application domains is of course also considered. The language is used to describe the structure of a real-time system as an assembly of software and hardware components. The language can describe functional interfaces to components (such as dataflows and control flows) and non-functional aspects of components (such as timing properties). The AADL describes how components are combined, in terms of subcomponents composition, interfaces connection and software to hardware allocation. The AADL was developed to meet the special needs of embedded real-time safety critical systems.

This paper presents how the AADL can be used for the software modeling phases of the development lifecycle, in association with other applicable standards for embedded real-time systems, or more generally for mission critical systems. The awaited benefit of using the AADL in such a context is to widely improve the formal definition of the real-time software architecture, and to enforce a better interaction between system engineering and software engineering activities.

The compatibility with other model based software engineering approaches, such as the very practical and well proven Hierarchical Object Oriented Design (HOOD) method is established, as well as some tips for using new graphical notations as defined by the emerging UML 2.0 standard.

This new approach combining the AADL rigorous semantical definition, the HOOD modeling process and some UML 2.0 graphical notation, provides a comprehensive, ready to use and up to date solution that complies with the requirements of industrial standards like DO-178B for avionics, ECSS-E40 for space systems, and EN-50128 for railways equipments. This approach is fully supported by an off the shelf tool, which already offers various code and documentation generators, as well as reverse engineering features. Finally, this approach is closely connected to the COTRE project, sponsored by the French department of research (RNTL), and which should provide enhanced real-time software verification techniques for this solution.

## 2 A summary of the AADL

The following definitions have been extracted from the current draft of the Avionics Architecture Description Language definition (AADL v 0.95). The reader is invited to refer to the final definition of the standard, when it will be released.

The AADL standard provides semantical and syntactical definitions to formally describe a real-time architecture in terms of interacting specialized components. An AADL specification of a system consists of a set of packages declaring a list of abstract components (components types and components implementations) and a static structure of component instanciations to describe the executable application and its allocation to the execution platform.

The abstract declaration of an AADL component in a package consists of one component type, and one or several component implementations. A component type describe the visible functional interface of the component, including a provided interface (list of declarations of public ports and subprograms), and a required interface (list of references to remote component provided interfaces). A component implementation contains all the additional details to fully define the architectural structure of the component, including a list of subcomponents and connections between these subcomponents and modes to describe the various operating states of the system. Several implementations may be defined for a same component type. All the implementations must strictly comply with the corresponding type declaration.

In order to provide an advanced support for real-time modeling, the AADL standard offers a set of predefined components categories which semantics is formally specified. These categories have been grouped into three sets:

- Data, threads and processes are the software components categories.
- Processors, memories, bus and devices are the execution platform categories.
- Systems represent composite sets of software and execution platform components.

Each component category can contain a controled set of features. Features represent connectors in the provided interface of the component, through which control flows and dataflows will be propagated. The AADL specifies three kinds of connectors:

- Ports are point to point connectors for individual data or events.
- Subprograms are composed sets of a control flow and dataflows (subprogram parameters).
- Subcomponent access represent access to remote data or bus components.

Finally, components and features descriptions are completed by a set of predefined specialized properties, providing all the necessary lower level additional information. Properties can be used to specify various kinds of data, such as links to source files or real time attributes. Properties are grouped into property sets, which provide a powerful extension mecanism for the language.

Due to these rigorous and well focused definitions, the AADL is the appropriate language to describe real time or other mission critical system and software architectures. However, in order to bring a maximal benefit to the projects, the use of this language must be considered within the overall development lifecycle. In particular, the issues of specifying a precise software design process, providing standardized graphical notations and advanced verification techniques, must also be addressed. Our proposed solution to meet these requirements, is to use the proven HOOD design process and the new UML2.0 graphical notations, to support the AADL modeling activities. The issue of verifying AADL model is being directly addressed by the COTRE project which is also presented during this conference. Please refer to the corresponding paper to know more about the COTRE project.

### 3 The HOOD method

The HOOD method first appeared in 1987 to meet the requirements of the European Space Agency (ESA). The early versions already included a set of notations and precise design rules to support advanced software engineering concepts and Ada code generation rules. In 1992, the version 3.1 of the HOOD Reference Manual (HRM) was published. In 1995, two concurrent major releases of the method were issued : HOOD4, conducted by the French Space Agency (CNES) improved widely the support of true Object Orientation to enable Ada95 and C++ code generation. At the same time, a new ESA project produced a Hard Real Time version of the method, called HRT-HOOD, dedicated to embedded real-time systems, and providing a direct support for schedulability analysis. After several years of operational use of these different variants of HOOD by major European projects, it appeared that HOOD 3.1, HOOD4 and HRT-HOOD were fully compatible and reflected the implementation of the same basic principles for various application profiles.

In HOOD, each Module (including the application itself), has an interface and a body. The interface consists of the `Provided_Interface` that lists the declaration of all the software elements that are implemented by the Module and made visible to be used by other Modules, and the `Required_Interface` that lists the references to all the remote software elements that are required to implement the Module. The body is called the `Internals` of the Module and contains either child Modules (for `Non_Terminal Module`) or a list of declaration of private software elements and all the implementation (for `Terminal Module`).

Additionally, HOOD only distinguishes between the Modules "type" and Modules "instances" in two particular cases only. The general case being that a Module is handled as an Object, that is, the unique instance of an anonymous "type". The first particular case is when the Module "type" is described by a Class (like in UML), but then, instances of this Class (which are in effect only instances of the main data Type provided by the Class) becomes Data, Constants, etc... embedded somewhere inside another Module. The other particular case is when the Module "type" is described by a Generic, then, instances of this Generic (which are in effect only instances of the formal parameters) becomes other Modules called `Instance_Of`. HOOD also supports Generic Classes (similar to C++ templates) that need to be instantiated twice. The only extension mechanism offered by HOOD is the Class Inheritance.

The main contribution of the HOOD method, however, comes from its well defined software modeling process. Based on a recursive top down hierarchical breakdown, this process provides rigorous design rules to enforce "good practices", like incremental documentation and coding, and improve the level of quality and productivity of a project. Moreover, its support by a tool offers automatized features such as multi languages code and documentation generators.

The use of the HOOD method fully complies with the main software engineering standard recommended for mission critical development of avionics, space, defense and some ground transportation projects, such as:

- DO-178B for airborne applications (Software Design Process)
- ISO/IEC-12207: for information technology (Software Architectural Design and Software Detailed Design)
- ECSS-E40: for space applications (Software top-level Architectural Design and Design of Software Items)
- EN-50128: for railway applications (Software Architecture and Software Design and Development)

The COTRE project, conducted by Airbus, and with the partnership of several French laboratories (LAAS, IRIT, CERT, ENSTB) and TNI, was an opportunity to investigate the connection between HOOD and the AADL. The result is that most HOOD concepts can be represented by an equivalent AADL paradigm without any major semantical discrepancy.

Although HOOD offers an appropriate set of graphical notations, the main issue that was encountered by many projects using this design method during the recent past years, was probably the unsuitability of the UML 1.x concepts and notations to represent real-time software architectures. This need is about to be satisfied with the new features that are offered by UML 2.0 structure diagrams.

## 4 Using UML 2.0 structure diagrams

UML 2.0 introduces structure diagrams to model component based architectures. These new UML concepts and notations are being studied in order to consider their use as a support for both AADL and HOOD paradigms.

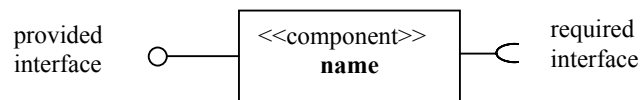
Following descriptions of structure diagrams refer to the "UML 2.0 Final accepted draft" dated August 2003. In the following paragraphs, the texts in italics are quotations from this document. As for the AADL, the reader is invited to refer to the final definition of the standard, when released.

### 4.1 UML 2.0 Components

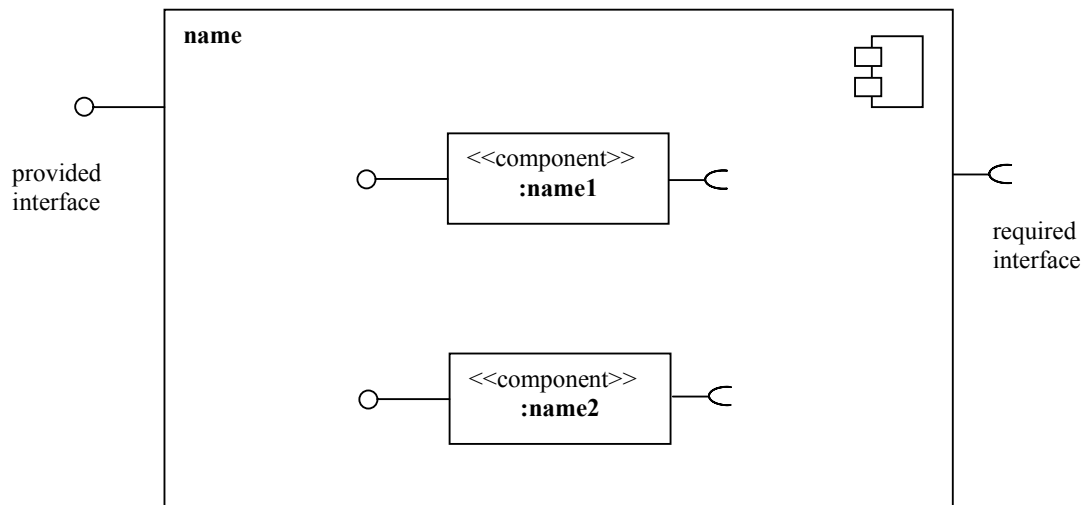
*"A component is a self contained unit that encapsulates the state and behavior of a number of classifiers. A component specifies a formal contract of the services that it provides to its clients and those that it requires from other components or services in the system in terms of its provided and required interfaces."*

*"A component is a substitutable unit that can be replaced at design time or run-time by a component that offers that offers equivalent functionality based on compatibility of its interfaces"*

The graphical notation for a UML 2.0 Component is either a "black box" view to only show the interfaces, or a "white box" view to show the internal assembly of subcomponents if any (parts or classifiers). The component logo can be associated or replace the <<component>> keyword.



*black box view of a UML 2.0 component*



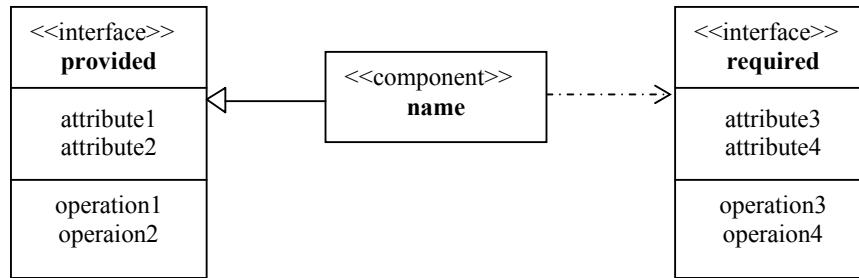
*white box view of a UML 2.0 component*

### 4.2 UML 2.0 Interfaces

*"An interface declares a set of public features and obligations that constitute a coherent service offered by a classifier. Interfaces provide a way to partition and characterize groups of properties that realizing classifier instances must possess. An interface does not specify how it is to be implemented, but merely what needs to be supported by realizing instances. That is, such instances must provide a public facade (attributes, operations, externally observable behavior) that conforms to the interface."*

"The set of interfaces realized by a classifier are its provided interfaces, which represent the obligations that instances of that classifier have to their clients. They describe the services that the instances of that classifier offer to their clients. Interfaces may also be used to specify required interfaces, which are specified by a usage dependency between the classifier and the corresponding interfaces. Required interfaces specify services that a classifier needs in order to perform its function and fulfill its own obligations to its clients."

Interfaces can be represented with the compact "ball-socket" graphical notation, or expanded as classes, with operations and attributes:



expanded UML 2.0 interfaces

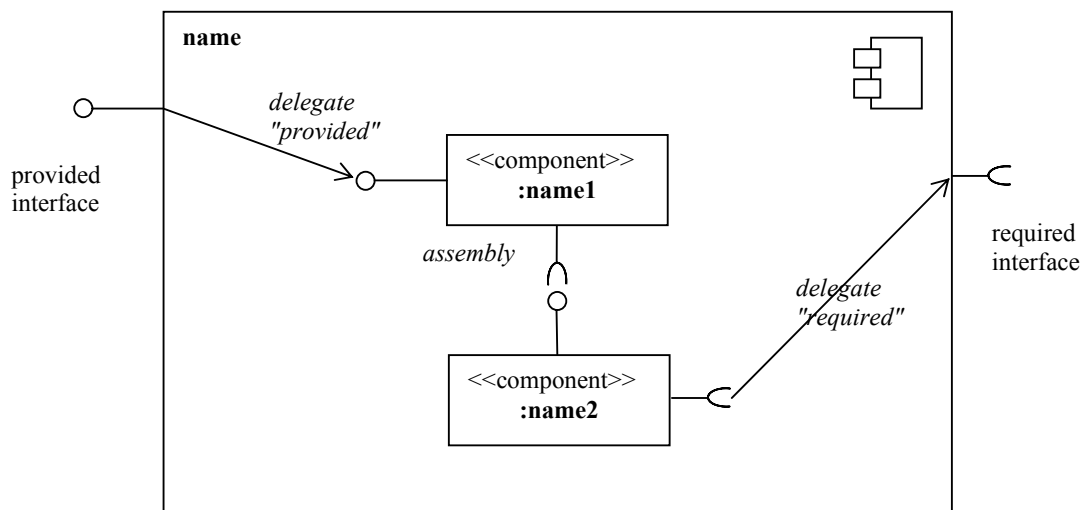
### 4.3 UML 2.0 Connectors

Like the other compositional notations, UML 2.0 specifies two main kinds of connectors between components: the links between components that are at the same level, and the links between a container component and its parts:

"A delegation connector is a declaration that behavior that is available on a component instance is not actually realized by that component itself, but by another instance that has "compatible" capabilities. This may be another Component or a (simple) Class."

"An assembly connector is a connector between two components that defines that one component provides the services that another component requires. An assembly connector is a connector that is defined from a required interface or port to a provided interface or port."

To comply with the HOOD method, the assembly connector can be mapped to the Use relationship between sibling components, the (required) delegation connector can be used to describe the Use relationship between a component and its environment outside its immediate parent, and the (provided) delegation connector can be used to represent the Implemented\_By relationship. However, unlike with HOOD or the AADL, the delegation connectors only link two interfaces, and doesn't provide the details of the individual links between each feature.



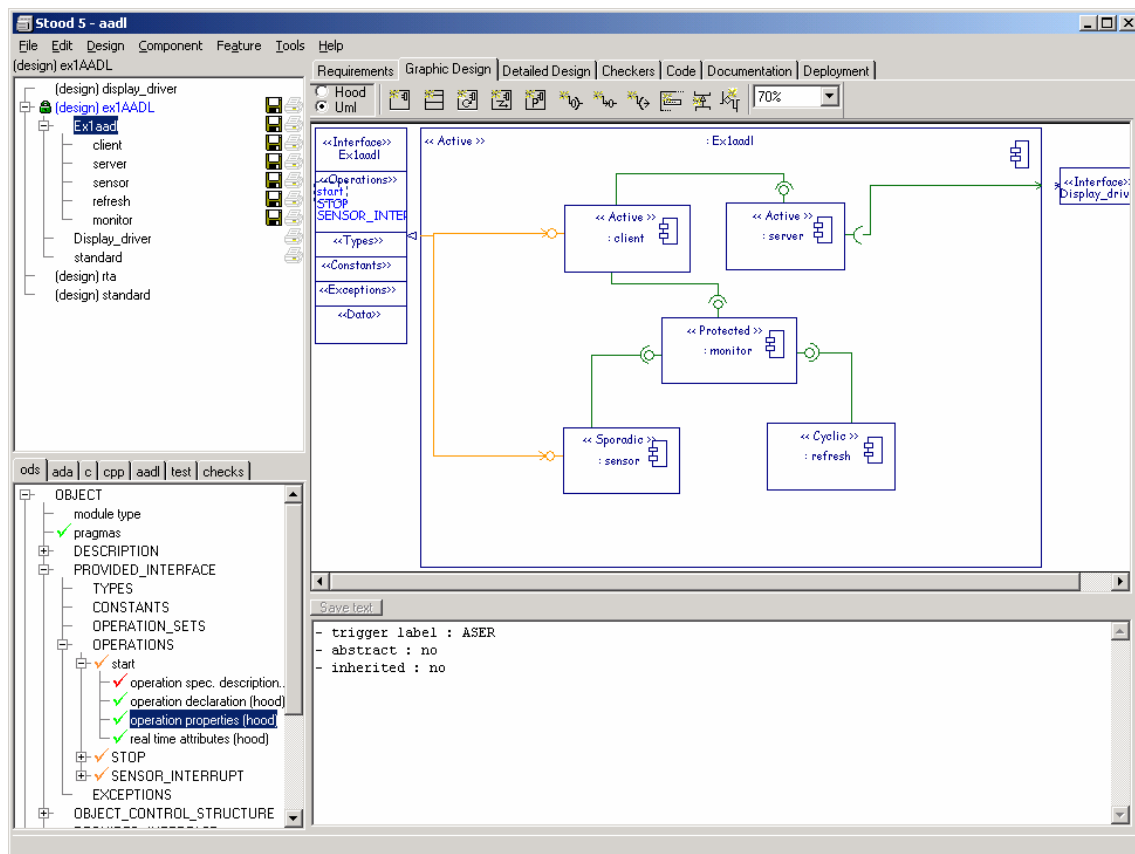
connectors in a UML 2.0 component

## 5 All together: Stood v5

Stood is the HOOD tool that is in use for many mission critical projects such as the Airbus A380 embedded software. The last release of the product now includes an AADL interchange feature and a UML 2.0 structure diagrams editor, together with the HOOD notation. All the existing processing features such as design rules checkers, code generators and reverse engineering, and documentation generators remain available.

This was made possible by the various connections that was defined between these three modeling techniques. Each of them brings its specific benefits: AADL offers a powerful and precise semantics for real-time architectures, HOOD provides a rigorous and validated software design process, and UML 2.0 now brings a standard notation to describe these systems. A summary of the corresponding mappings and a preliminary snapshot of the Stood v5 tool are shown below:

AADL	UML 2.0	HOOD
component	component	(parent) module
subcomponent	part	(child) module
provides features	provided interface	provided interface
required subcomponents	required interface	required interface
(server) containment connection	delegate (provided)	implemented by
(client) containment connection	delegate (required)	use (uncle)
components connection	assembly	use (sibling)



By integrating these new technologies into an already deployed software development environment, the aim is to ease the dissemination of model based solutions to new projects, and to preserve the investment of those who already chose advanced modeling techniques during the past ten years.

## References

1. RTCA, *Software Considerations in Airborne Systems and Equipment Certification (DO-178B)*, 1992.
2. ISO/IEC, *Information technology, Software life cycle process (ISO/IEC 12207)*, 1995
3. ECSS, *Space Engineering: Software (ECSS-E40B)*, ESA Publication, 2000.
4. HOOD User Group, *HOOD Reference Manual release 3.1*, Masson & Prentice-Hall, 1993.
5. HOOD User Group, *HOOD Reference Manual release 4.0*, HUG, 1995.
6. A. Burns, A. Wellings, *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems*, Elsevier, 1995
7. J.P. Rosen, *An Industrial Approach for Software Design*, HUG, 1997.
8. P. Dissaux, *HOOD4 and Ada95*, Proceedings DASIA conference in Lisbon, 1999.
9. T. Vardanega, *Development of On-Board Embedded Real-Time Systems: An Engineering Approach*, ESA Technical Report STR-260, 1999.
10. P. Dissaux, *Real-Time C Code Generation from a HOOD Design*, Proceedings DASIA conference in Montreal, 2000.
11. P. Dissaux, *HOOD Patterns*, Proceedings DASIA conference in Nice, 2001.
12. P. Dissaux, *UML & HOOD for aerospace software development*, LTRE conference in Toulouse, 2002
13. P. Farail, P. Dissaux, *COTRE, A new Approach for Modelling Real-Time Software for Avionics*, Proceedings DASIA conference in Dublin, 2002.
14. P. Dissaux, *HOOD and AADL*, Proceedings DASIA conference in Prague, 2003.
15. SAE, *Draft Avionics Architecture Description Language (AADL) Version 0.95*, AS-2C, 2003
16. P. Farail & P. Gauffillet, *The COTRE Project: How to model and verify Real Time Architecture?*, Proceedings ERTS conference in Toulouse, 2004.