VHDL Modeling Terminology and Taxonomy

Version 3.
July 29, 1998

http://rassp.scra.org

RASSP Taxonomy Working Group (RTWG)

Contents

The current taxonomy working group is comprised of the following members:

Carl Hein
Lockheed Martin
Advanced Technology Laboratories
Camden, NJ 08102
chein@atl.lmco.com

Todd Carpenter
Honeywell Technology Center
MPLS, MN 55418-1006

Vijay Madisetti
School of Electrical & Computer Eng.
Georgia Institute of Technology
Atlanta, GA 30332-0250

Allan Anderson
MIT Lincoln Laboratory
Lexington, MA 01273-9108

Arnold Bard
US Army CECOM
Fort Monmouth, NJ 07703

J.P. Letellier
Navel Research Laboratory
Washington, DC 20375-5336

Robert Klenke
Dept. of Elect. Eng.
Charlottesville, VA 22903-2442

Capt. Greg Peterson
Wright Labs
WPAFB, OH 45433-7319

Mark Pettigrew
Sanders - A Lockheed Martin Company
Hudson, N.H. 03051

Perry Alexander
U.Cincinnati, Elect. & Comp Dept.
Cincinnati, OH 45221-0030

Geoffrey Frank
RTI Center for Systems Eng.
Research Triangle Pk, NC 27709-2194

The following have been past contributors to the working group:

Anthony Gadient
SCRA
North Charleston, SC 29418

Randy Harr
Advanced Research Projects Agency
Electronic Systems Technology Office
Arlington, VA 22203-1714

Paul Kalutkiewicz
Lockheed Sanders
Advanced Engineering & Technology
Nashua, NH 03061-0868

Summary

The increasing complexity, time to market pressures, and life-cycle-costs of digital systems motivate development of new design and prototyping methods. In particular, hierarchical VHDL model-based design techniques were developed and demonstrated as part of the ARPA/Tri-Services sponsored Rapid-prototyping of Application Specific Signal Processors (RASSP) program. Meeting the rapid prototyping challenges requires unambiguous transfer of design information and communication about modeling modes between developers. To address the need for conventions in modeling and terminology, the participating organizations of the RASSP program formed the RASSP Terminology Working Group (RTWG). Based upon examination and comparison of previously published modeling taxonomies, the working group designed a multi-axis taxonomy to describe the information content of computer model types and abstraction levels and to facilitate selection and construction of interoperable models. The taxonomy is then used to concisely refine the definitions of modeling terms that are especially important in RASSP.

1 Modeling Taxonomy

The modeling taxonomy provides a means to categorize models according to a set of attributes. The attributes are useful in distinguishing models intended for distinctly different purposes. The taxonomy is used to establish formal definitions which are concise and unambiguous for the various model types. The taxonomy was developed by the RASSP Terminology Working Group (RTWG) to address increasing terminology and model interoperability challenges facing designers. Appendix A provides background information about the RTWG and the needs addressed by the taxonomy. The resulting taxonomy and glossary was partially based on prior efforts as described in Appendix B: Prior Taxonomies. Descriptions and definitions are provided in Section 8 for many of the terms used in this document.

1.1 Taxonomy Axes

The taxonomy represents model attributes that are relevant to designers and model users. It is based on common terminology that is readily understood and used by designers. The taxonomy consists of a set of attributes or axes that characterize a model's relative resolution of details for important model aspects. The taxonomy axes, shown in figure 1, identify five distinct model characteristics:

1. Temporal detail
2. Data Value detail
3. Functional detail
4. Structural detail
5. Programming level

The first four attributes do not address the hardware/software codesign aspect of a model, because they do not describe how a hardware model appears to software. The fifth axis represents the level of software programmability of a hardware model or, conversely, the abstraction level of a software component in terms of the complementary hardware model that will interpret it.

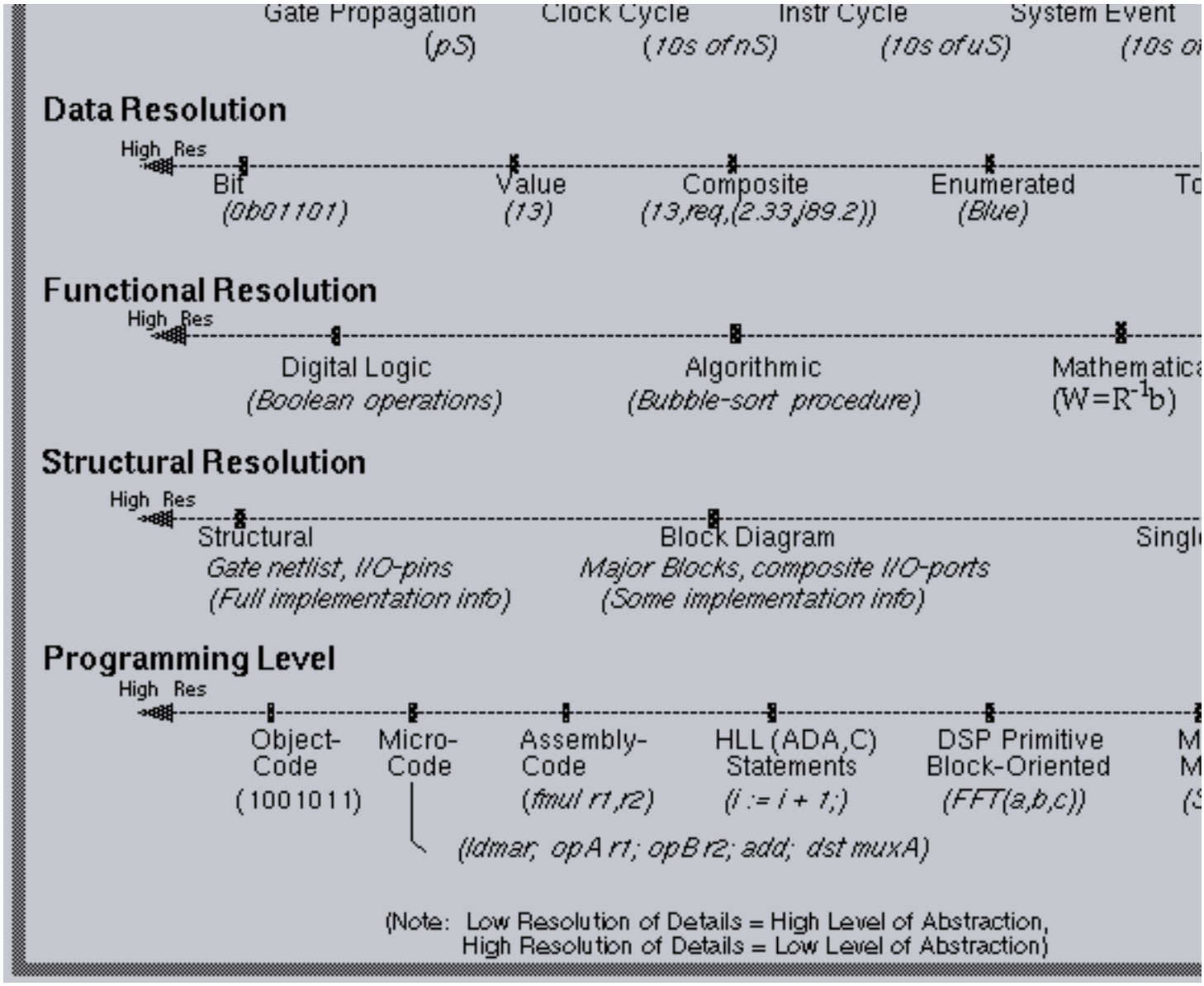
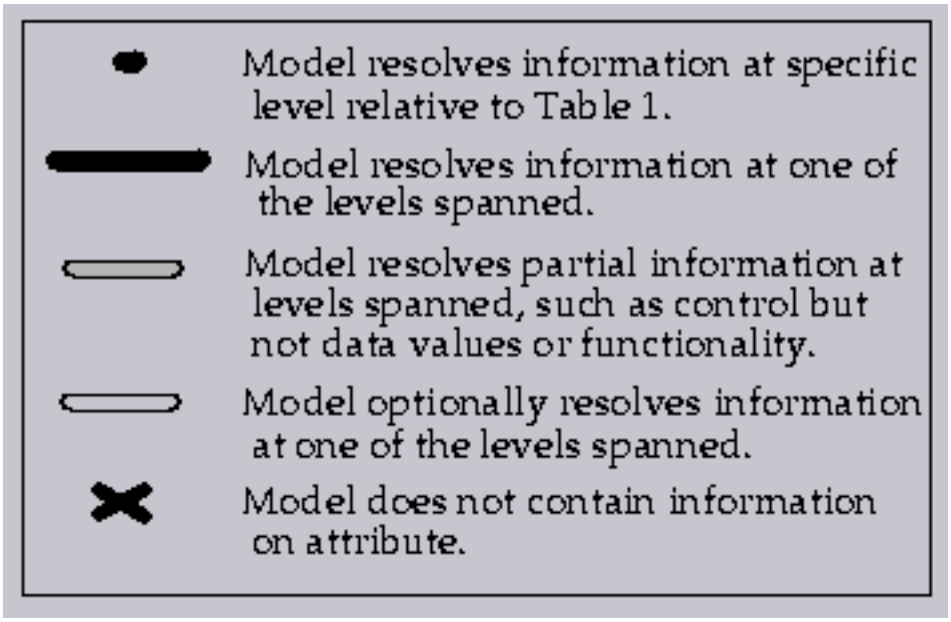


Figure 1 - Taxonomy Axes

Sections 3-7 define vocabulary terms graphically relative to the taxonomy axes to show each term's applicable coverage. The key in figure 2 should be used to interpret the graphic. Although some terms may span a range of abstraction levels, a given model instance describes information at one specific level within the span.

Figure 2 - Symbol Key:



Another convenient method for specifying a particular model's information content is to use the textutal notation of (temporal, value, function, structure, program).

1.2 Temporal Resolution Axis

The Temporal Resolution axis represents the resolution with which events are represented on a time scale. A particular event could be described at these levels of time resolution, from high to low:

- measured from some reference point, such as 1.5 ns after a clock transition
- someplace within a defined time span, such as within a specific clock period
- during a specific instruction period
- the begin or end of a major event
- no time information except causal sequence

1.3 Data Resolution Axis

The Data Resolution axis represents the resolution with which the format of values are specified in a model. The contents of a register could be described at these levels of resolution, from high to low:

- Binary: b101
- Signed Integer: -9
- Enumerated: blue
- No data values

Note that resolution is analogous to precision as distinguished from accuracy. Each representation is equally accurate; however, the first case resolves the value closer to the form actually contained in the target device. The more abstract the representation of a value is, the less implementation details are resolved.

1.4 Functional Resolution Axis

The Functional Resolution axis represents the level of detail with which a model describes the functionality of a component or system. A digital filter component could be represented by these levels of resolution, from high to low:

- boolean operations
- algorithmic processes

- mathematical relationships
- no functionality

1.5 Structural Resolution Axis

The Structural Resolution axis represents the level of detail with which a model describes how a component is constructed from its constituent parts. An integrated circuit could be represented at these levels of resolution, from high to low:

- connection of simple functions, such as logic gates
- connection of composite functions, such as flip flops
- connection of more complex functions, such as registers and multipliers
- connection of large blocks, such as ALU and register files
- connection of computer networks
- no structural information

1.6 Software Programming Resolution Axis

The Software Programming Resolution axis represents the granularity of software instructions that the model of a hardware component interprets in executing target software. A programmable device could be represented at these levels of resolution, from high to low:

- microcode instructions
- assembly code instructions
- major functions, such as FFT

For example, a token-based performance model interprets instructions which represent major tasks or subroutines such as matrix invert, vector multiply, or Fourier transform. Though the primitives may represent hundreds of lines of source code, they are interpreted as a single instruction in terms of a time-delay by a network performance model. An Instruction-Set-Architecture (ISA) model interprets individual assembly instructions. In this sense, the ISA model is programmable at a much finer granularity, or higher resolution, than the network performance model.

At the lower extreme, a model of a microcode programmable processor is programmable at an even lower level of granularity than the ISA model because it allows control of individual-register and multiplexor structures within the device during execution of an assembly-level instruction. Software design components or non-programmable models are at the opposite extreme because neither interprets programmable instructions.

---

2 Internal / External Concepts

Previous terminologies often mixed attributes as viewed from inside a model, with similar attributes, as viewed from the model's interface boundary. These two views are referred to as the *internal* and external views, respectively. Distinguishing between the internal and external views is important in selecting, using, and building models because it enables clarity and precision [5].

The RASSP Taxonomy specifies each of the first four axes from both an internal and external viewpoint. Consequently, the taxonomy effectively contains eight attributes:

- Internal: temporal, value, structure, and function resolutions,

and

- External: temporal, value, structure, and function resolutions.

The programming level represents the interaction of hardware to software and presents no distinction between internal and external views.

Internal resolution references how a model describes the timing of events, functions, values, and structures of the elements that are contained within the boundaries of the modeled device.

External resolution describes how a model describes the interface of the modeled device to other devices. The external aspects refer to the input/output or I/O details at the boundary of the modeled device. The external details relate to how the model describes a device's interaction with devices to which it connects. External details may include timing and functional aspects, commonly refered to as *protocol* , as well as port structure and signal values. All of which may be abstracted to various levels in a model.

A convenient method for specifying a particular model's information content is to use the Internal / External (temporal, value, function, structure) notation. For example, the content of a particular RTL model could be specified as:

```
Internal(time=uS,    value=boolean, function=full, structure=register),
External(time=clock, value=boolean, function=full, structure=pin),
SW-Program(assembly)
```

For contrast, an example of a particular algorithm model could be specified as:

```
Internal(time=none,  data=composite, function=full, structure=none),
External(time=none,  data=composite, function=none, structure=none),
SW-Program(none)
```

The internal/external distinction is also known as the white-box/black-box distinction. To better understand the *internal/external* concept, consider viewing an integrated circuit chip (IC-chip) from outside its package as shown in figure 3 below.

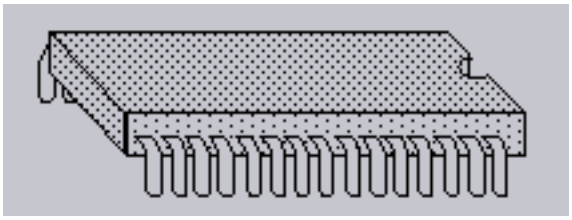


Figure 3 - External View.

When viewed from outside, or *externally* , we can observe only the structure and behavior of the **ports**, (for example: how many pins there are, and what values they have when driven with various stimuli), but we cannot observe any details about how the IC-chip is implemented inside the package, or *internally* .

In contrast, we can think of seeing an *internal* view if we were to pop the lid off the IC package as shown in figure 4 below.

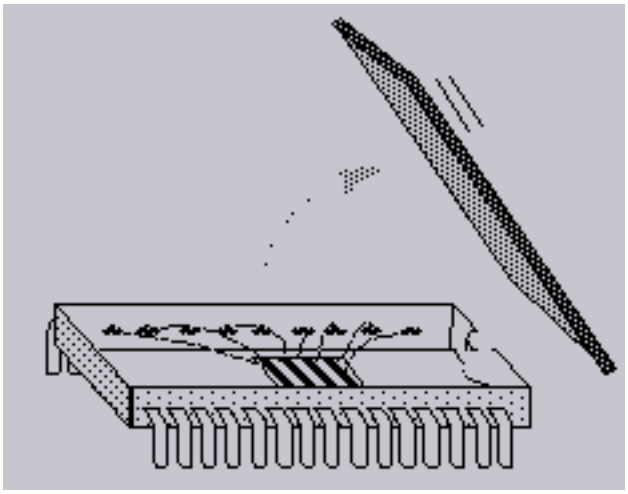


Figure 4 - Internal View.

Notice that now we can see some detail about how the IC-chip's insides, or *internals* , are implemented. This view contains internal implementation detail.

External structure is the structure of the externally viewable features, which is the structure of the externally viewable ports, or interface. Figure 5 below depicts an abstract model of the external ports (interface) of the chip. The external implementation detail is resolved as two signals which are of abstract type, integer. They are abstract because they do not reveal the bit-level implementation detail.

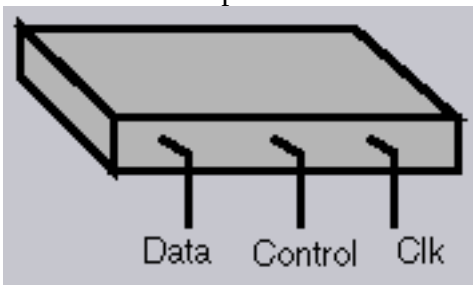


Figure 5 - Abstract model of external features.

A less abstract model of the external interface of the component could show the actual bit-level implementation detail of the signal ports as shown in figure 6 below.

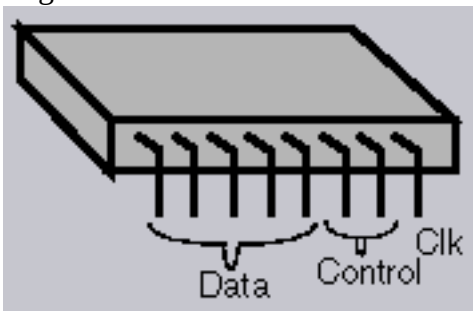


Figure 6 - Less abstract model of external features.

This more detailed view shows the hand-shaking lines and the data port resolved as individual pins at the bit level.

Similarly, the internal details can be depicted at various levels of abstraction.

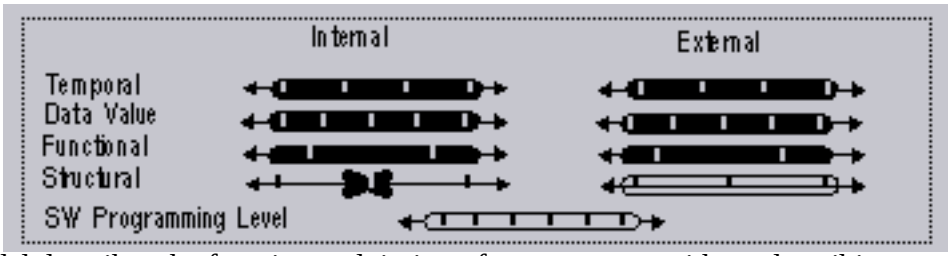
3 General Modeling Concepts

The following section contains definitions of concepts that are pervasive across many types and levels of models. The modeling concepts are divided into two groups: Primary and Specialized model Classes.

3.1 Primary Model Classes

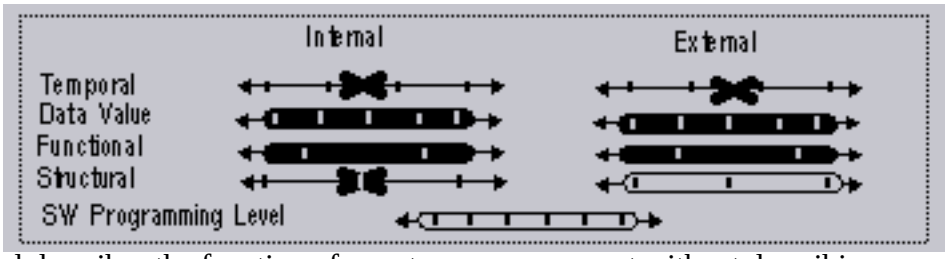
All models are described in terms of one or more of the following three primary classes.

3.1.1 Behavioral - (Behavior = Function with Timing) (Synonym: Interpreted Model)



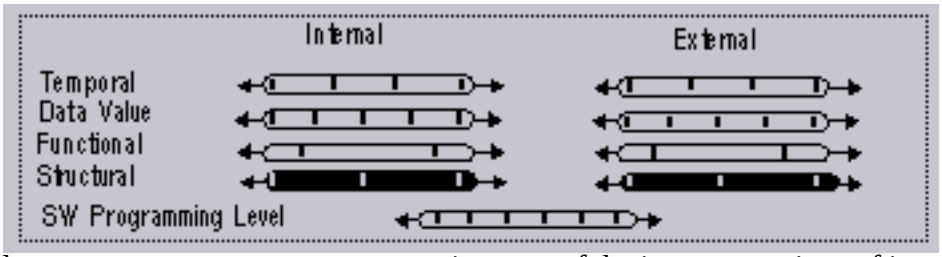
A behavioral model describes the function and timing of a component without describing a specific implementation. A behavioral model can exist at any level of abstraction. Abstraction depends on the resolution of implementation details. For example, a behavior model can be a model that describes the bulk time and functionality of a processor that executes an abstract algorithm, or it can be a model of the processor at the less abstract instruction-set level. The resolution of internal and external data values depends on the model's abstraction level.

3.1.2 Functional - (Function = Behavior without Timing)



A functional model describes the function of a system or component without describing a specific implementation. A functional model can exist at any level of abstraction. Abstraction depends on the resolution of implementation details. For example, a functional model can be a model that abstractly describes the function of a signal processing algorithm, or it can be a less abstract model that describes the function of an ALU for accomplishing the algorithm. The resolution of internal and external data values depends on the model's abstraction level.

3.1.3 Structural -

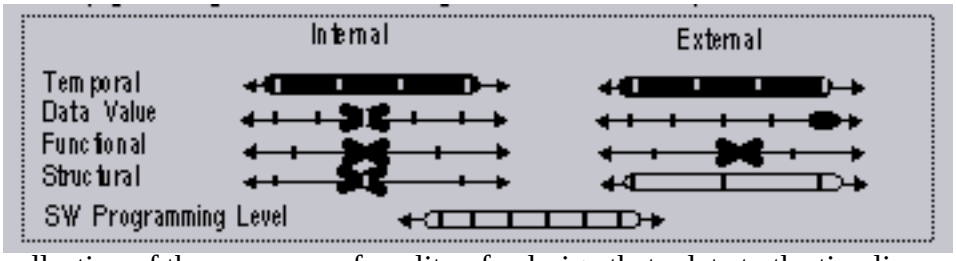


A structural model represents a component or system in terms of the interconnections of its constituent components. A structural model follows the physical hierarchy of the system. The hierarchy reflects the physical organization of a specific implementation. A structural model describes the physical structure of a specific implementation by specifying the components and their topological interconnections. These components can be described structurally, functionally, or behaviorally. Simulation of a structural model requires all the models in the lowest (leaf) branches of the hierarchy to be behavioral or functional models. Therefore, the effective temporal, data value, and functional resolutions depend on the leaf models. A structural model can exist at any level of abstraction. Structural resolution depends on the granularity of the structural blocks.

3.2 Specialized Model Classes

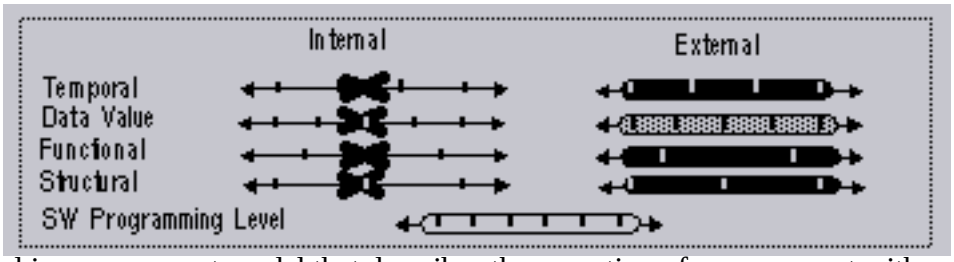
The following model classes describe models intended for specific purposes that are not unique to a particular level of abstraction.

3.2.1 Performance Model - (Synonym: Uninterpreted Model)



Performance is a collection of the measures of quality of a design that relate to the timeliness of the system in reacting to stimuli. Measures associated with performance include response time, throughput, and utilization. A performance model may be written at any level of abstraction. In general, a performance model may describe the time required to perform simple tasks such as memory access of a single CPU. However in the context of RASSP, the typical abstraction level for performance models is most often at the multiprocessor network level. For clarity, such a model is called a Token Based Performance Model (See 5.2).

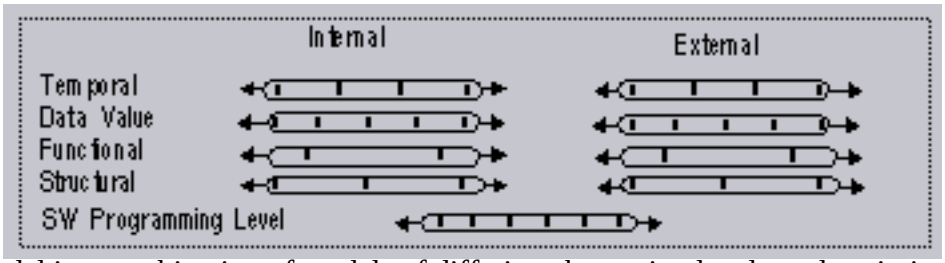
3.2.2 Interface Model -



An interface model is a component model that describes the operation of a component with respect to its surrounding environment. The external port-structure, functional, and timing details of the interface are provided to show how the component exchanges information with its environment. An interface model contains no details about the internal structure, function, data values, or timing other than that necessary to accurately model the external interface behavior. External data values are usually not modeled unless they represent control information. An interface model may describe a component's interface details at any level of abstraction. The terms bus functional, and interface behavioral have also been used to refer to an interface

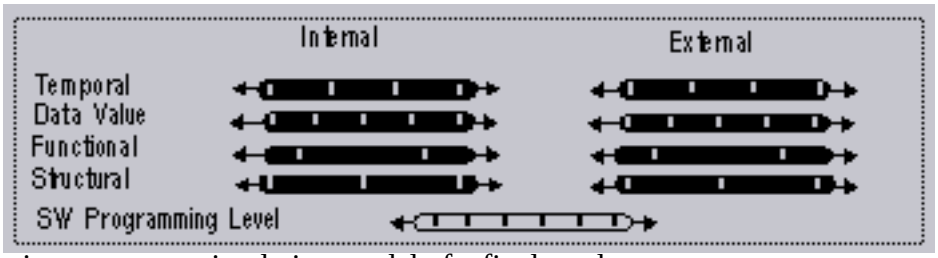
model and are considered synonyms. The more general interface model name is preferred to the anachronistic bus functional term.

3.2.3 Mixed-Level Model - (Synonym: Hybrid Model)



A mixed-level model is a combination of models of differing abstraction levels or descriptive paradigms. Such a model is sometimes called a hybrid model. However, the RTWG prefers the term mixed-level over the term hybrid since the former is more specific.

3.2.4 Virtual Prototype -



A virtual prototype is a computer simulation model of a final product, component, or system. Unlike the other modeling terms that distinguish models based on their characteristics, the term virtual-prototype does not refer to any particular model characteristic but rather it refers to the role of the model within a design process; specifically for the role of:

- exploring design alternatives,
- demonstrating design concepts
- testing for requirements satisfaction/correctness

See prototype, physical-prototype.

Virtual prototypes can be constructed at any level of abstraction and may include a mixture of levels. Several virtual prototypes of a system under design may exist as long as each fulfills the role of a prototype. To be useful in a larger system design, a virtual-prototype model should define the interfaces of the component or system under design.

In contrast to a physical prototype, which requires detailed hardware and software design, a virtual prototype can be configured more quickly and cost-effectively, can be more abstract, and can be invoked earlier in the design process. A distinction is that a virtual prototype, being a computer simulation, provides greater non-invasive observability of internal states than is normally practical from physical prototypes.

3.3 Distinguishing Structure, Behavior, and Interface Models

The following example models distinguish:

- Interface model (no internal details),
- Behavioral model (internal details described behaviorally),
- Structural model (internal structure described).

Figures 7 depicts an interface model. Notice that it contains details about the interface, or external ports, but

contains no information about the internal implementation. Some level of external structure and data values can be observed, as well as some level of function and timing response to interface activities.

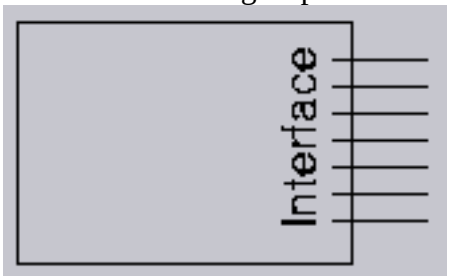


Figure 7 - Interface Model.

In contrast, figure 8 below depicts a behavioral model of the same component as above. Notice that this model contains information about the internal data values, functions, states, and timing aspects of the component, but no information about how the internal structure is implemented. Therefore, the internal view is said to be represented behaviorally.

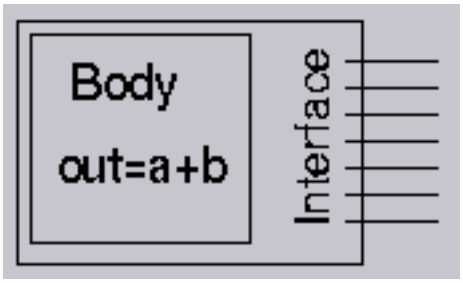


Figure 8 - Behavioral Model.

In contrast to the above model, the figure 9 below depicts a model that describes the internal structure of the component to some level of detail. Remember that *structure* is inter-connection information. Notice that this diagram shows a decomposition into internal blocks. Because it shows how the blocks are connected to each other, at some level of abstraction, it is called a structural model, or internal structure.

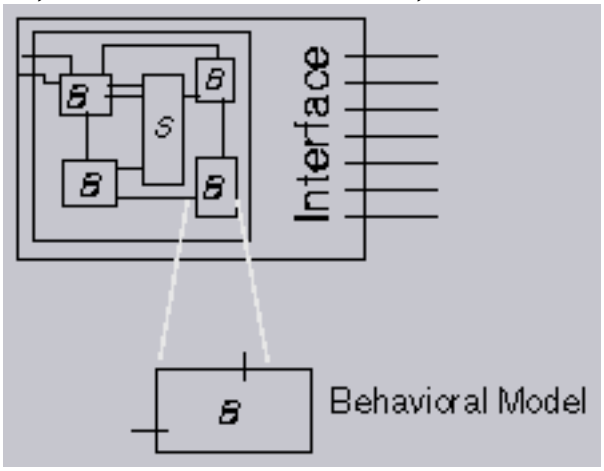


Figure 9, - Structural Model.

The internal blocks of a structural model can be described either behaviorally, or can themselves be further decomposed structurally, etc.. If behaviorally described blocks exist at the bottom (leaf-level) of a structural hierarchy, then the model can be simulated. Note that the behavior of such a composite model is provided by the underlying behavioral blocks: not by the structure. The structural descriptions merely provide means for combining separate behavioral pieces. A different behavior would be exhibited if the underlying behaviors were changed.

If behavioral blocks do not exist at the bottom (leaf-level) of a structural hierarchy, then the model is a *purely*

structural model . No behavior can be inferred if the behaviors of the underlying blocks of a structural model are unknown.

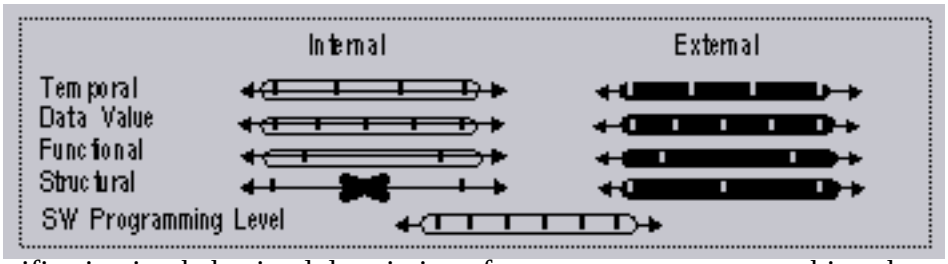
The level of abstraction of the internal view depends on the level of implementation details. The structure could be described abstractly as the interconnection of high-level blocks, or concretely as the interconnection of logic gates. Independently, the timing and functional abstraction can be described for the high-level blocks abstractly as coarse events or concretely as specific times, or for the gate-level model abstractly as the stable per-clock boolean values, or all switching rise and falls resolved to pS (intra-clock events) and signal levels and strengths.

---

4. System Models

The following section defines terms used to abstractly describe models of digital electronic systems, such as digital signal processing (DSP) systems or control systems. The abstraction does not include any information about any hardware or software structure for implementing the system.

4.1 Executable Specification -

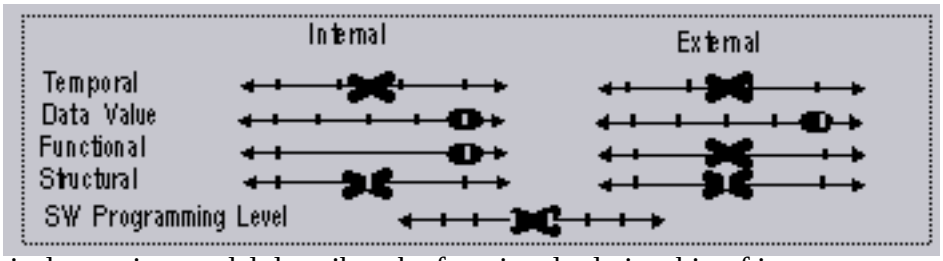


An executable specification is a behavioral description of a component or system object that reflects the particular function and timing of the intended design as seen from the object's interface when executed in a computer simulation. The executable specification may describe the electrical, behavioral, or physical aspects including the power, cost, size, fit, and weight of the intended design. Denotational items such as power are normally considered factual (derived) items to be checked but not executed. Executable specifications describe the behavior or function of an object at the highest level of abstraction that still provides the proper data transformations (correct data in yields correct data out; DEFINED bad data in has the SPECIFIED output results). Executable specifications may describe an object at an arbitrary abstraction level such as the multi-processor system, architecture, or hardware or software component level. In contrast to a virtual prototype, the executable specification does not describe a system's internal structure.

The primary purposes of an executable specification are:

- testing that the specified behavior of a design entity satisfies the system requirements when integrated with other components of the system and,
- testing whether an implementation of the entity is consistent with the specified behavior.

4.2 Mathematical-Equation Model -



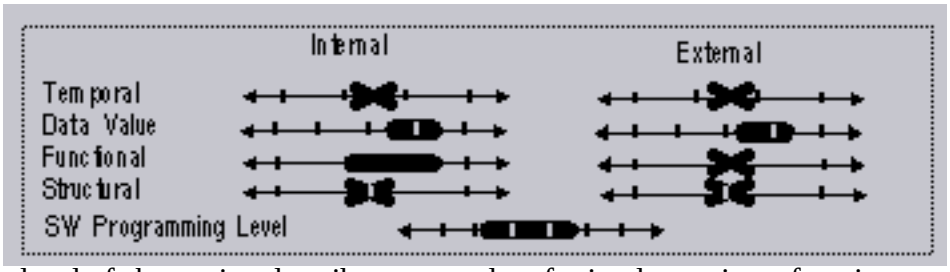
The mathematical-equation model describes the functional relationship of input to output data values. A mathematical-equation model is a purely algebraic expression of the function the target system is to provide. The mathematical model is differentiated from an algorithm description in that a mathematical model does not imply a specific sequence of operations to implement the function. Examples of mathematical descriptions for system functions are:

```
y = sqrt( x )   or   y = sin( x )
```

These functions represent well-defined mathematical relationships but do not indicate methods for their computation, for which there are many, such as lookup table, Newton's method, or Taylor-series expansion.

The primary purpose of a mathematical-equation model is to test that the mathematical equations and parameters developed to solve a design challenge do satisfy a system's numerical performance requirements.

4.3 Algorithm Model -



The algorithm level of abstraction describes a procedure for implementing a function as a specific sequence of arithmetic operations, conditions, and loops. An algorithmic description is less abstract than a purely mathematical description because it provides more detailed information for implementing the function(s). An algorithm model transforms actual data. Representative eXAMPLES OF algorithms are quick-sort, Givens triangularization, Cholesky matrix decomposition, bisection method, Cooley-Tukey FFT, and Winograd FFT [8].

The primary purpose of an algorithm model is to test how well an algorithm designed to implement a mathematical task satisfies the system numerical performance requirements. Algorithm models are also used for determining the numerical effects of finite precision and the parameters of floating or fixed formats.

Return

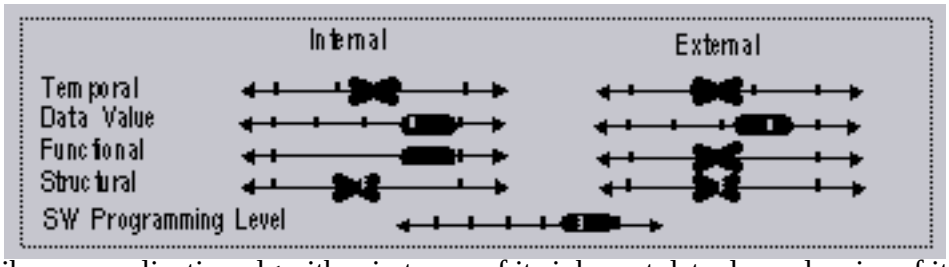
---

5. Architecture Models

The following section defines terms used to describe abstract models of a system's hardware and software architecture. As such, these models describe only the basic structure of the application and

the hardware to which it can be mapped. Details that are not relevant to the architecture design process are relegated to the detailed hardware and software models.

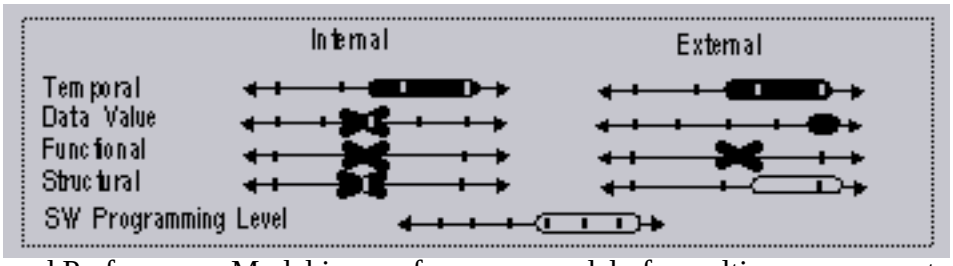
5.1 Data Flow Graph (DFG) -



A DFG describes an application algorithm in terms of its inherent data dependencies of its mathematical operations. The DFG is a directed graph containing nodes that represent mathematical transformations and arcs that span between nodes and represent their data dependencies and queues. It conveys the potential concurrencies within an algorithm, which facilitates parallelization and mapping to arbitrary architectures. The DFG is an architecture independent description of the algorithm. It does not presume or preclude potential concurrency or parallelization strategies. The DFG can be a formal notation that supports analytical methods for decomposition, aggregation, analysis, and transformation. The DFG nodes usually correspond to DSP primitives such as FFT, vector multiply, convolve, or correlate. The DFG graph can be executed by itself in a data-value-true mode without being mapped to a specific architecture, though it can not resolve temporal details without co-simulation with an architecture performance model.

The primary purposes of a data flow graph are to express algorithms in a form that allows convenient parallelization and to study and select optimal parallelization or execution strategies through various methods involving the aggregation, decomposition, mapping, and scheduling of tasks onto processor elements.

5.2 Token-Based Performance Model -



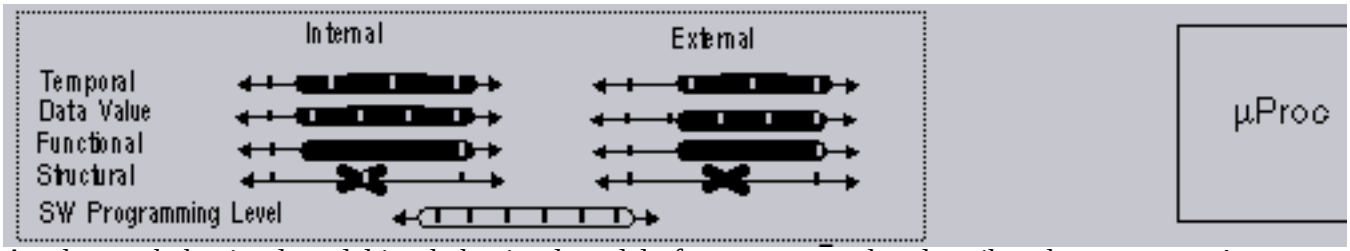
The Token-based Performance Model is a performance model of a multi-processor system's architecture. See performance model 3.2.1. Measures associated with performance include response time and throughput of the system and the utilization of its resources, Typically, the token-based performance model resolves the time for a multiprocessor networked system to perform major system functions. Data values are not modeled, except for control information. The structure of the system network is described down to the network node level. The internal structure of network switches, processor elements, shared memories, and I/O units is not described in a Token Based Performance Model.

The primary purposes of a token-based performance model are to determine sufficiency of the following selections in meeting the system processing throughput and latency requirements: the number and type of processor elements, the size of memories and buffers, network topology (bus, ring, mesh, cube, tree, or custom configuration), network bandwidths and protocols, application partitioning, mapping, and scheduling of tasks onto processor elements, and flow control scheme.

6. Hardware Models

The following section defines terms used to describe the various types of hardware models. These models are used to describe the hardware at specific levels of abstraction.

6.1 Abstract Behavioral Model -

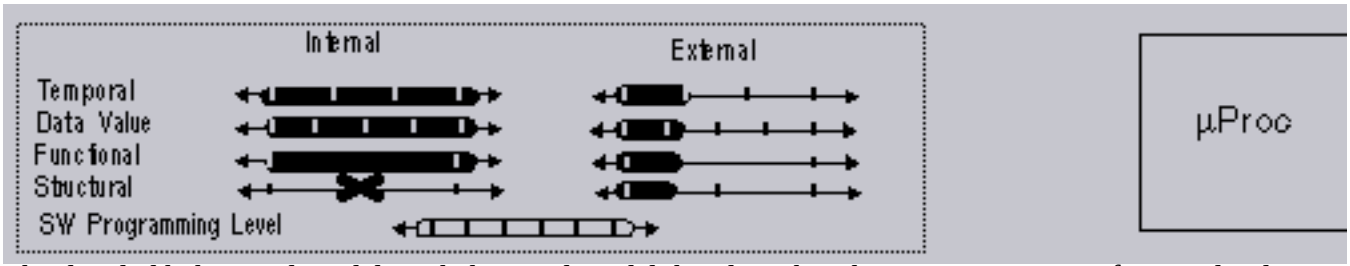


An abstract behavioral model is a behavioral model of a component that describes the component's interface abstractly. It does not resolve the interface ports to their pin structure. For instance, a microprocessor's memory interface may be described as a single port having a complex data type, as opposed to specifying the constituent control lines, and address and data buses and their bit-widths. See 3.1.1 behavioral model.

The primary purposes of an abstract behavioral model are:

- To establish and verify the joint functional and timing requirements for the components to ensure that collectively they are consistent with the overall system requirements.
- To verify the numerical correctness of the hardware/software mapping as modeled by the performance model.
- To facilitate reuse of the system design when implementation of the components or interfaces are changed.
- To produce test-vectors for use in the detailed design of the components and to aid in system integration, diagnosis, and testing.
- To help visualize the operation of complex systems for understanding its characteristics for optimizing the design, especially at the board level prior to making detailed decisions about the exact nature of component interfaces, to accomplish top-down design.
- To document the intended operation of the system implementation.

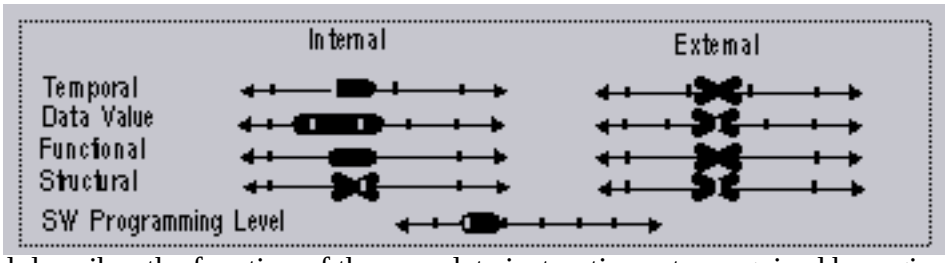
6.2 Detailed-Behavioral Model -



The detailed behavioral model is a behavioral model that describes the component's interface explicitly at the pin level. It exhibits all the documented timing and functionality of the modeled component, without specifying internal implementation structure. This type of model has traditionally been called a full- functional model and is therefore a synonym. However, the newer term is preferred for its better accuracy and consistency to the definitions of the related models.

The primary purpose of a detailed behavioral model is to develop and comprehensively test the structure, timing, and function of component interfaces, especially of a board level design. Also to examine the detailed interactions between hardware and software (drivers), and to provide timing values that are used to replace initial estimates in the higher level models to increase their accuracy.

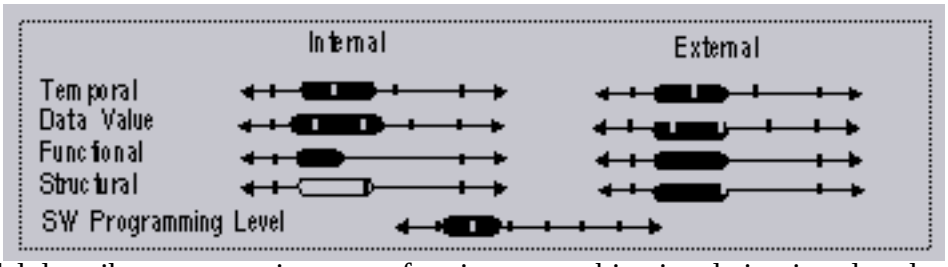
6.3 Instruction Set Architecture (ISA) Model -



An ISA model describes the function of the complete instruction set recognized by a given programmable processor, along with (and as operating on) the processor's externally known register set and memory/input-output (I/O) space. An ISA model of a processor will execute any machine program for that processor and give the same results as the physical machine, as long as the initial states (and simulated I/O) are the same on the ISA model simulation as they are on the real processor. Such a processor model with no external ports is classed as an ISA model. If the processor model has external I/O ports, then it would be classified as a behavioral model. Data transformations of ISA models are bit- true, in terms of word length and bit values as observable in the internal registers and memory states. Port buffer registers, if modeled, are also bit-true. The temporal resolution of an ISA model is at the instruction cycle. Instruction cycles may span multiple clock cycles. An ISA model contains no internal structural implementation information.

The primary purposes of ISA models are for efficient development of uni-processor resident software prior to hardware realization, optimization and design of application specific instruction sets and register architectures, documenting the functionality of processor's instruction set, and for measuring software routine runtimes, to increase the accuracy of the more abstract models.

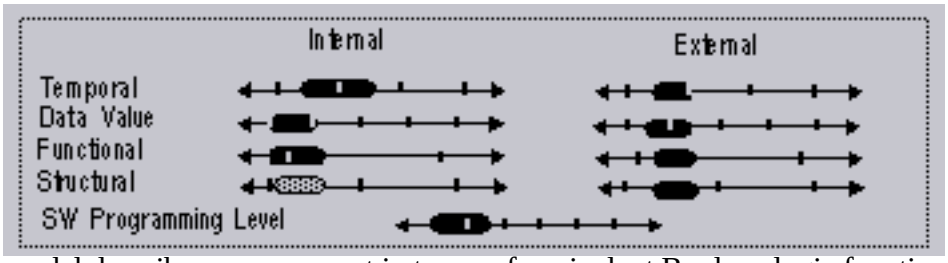
6.4 Register Transfer Level (RTL) Model -



An RTL model describes a system in terms of registers, combinational circuitry, low-level buses, and control circuits, usually implemented as finite state machines. Some internal structural implementation information is implied by the register transformations, but this information is not explicitly described.

The primary purpose of RTL models is for developing and testing the internal architecture and control logic within an IC component so that the design satisfies the required functionality and timing constraints of the IC. The RTL model is also used for specifying the design in a process neutral format that is retargettable to specific technologies or process lines through automatic synthesis. It is often used for the generating detailed test vectors, gathering timing measurements to increase the accuracy of more abstract models, investigating interactions with closely connected components, and it unambiguously documents the design solution.

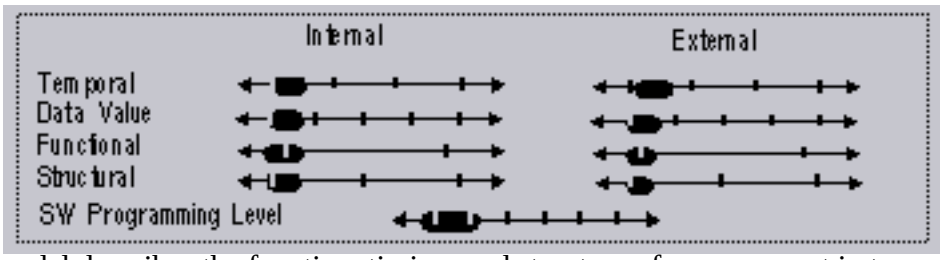
6.5 Logic-Level Model -



A logic-level model describes a component in terms of equivalent Boolean logic functions and simple memory devices such as flip-flops. The logic-level model does not describe the exact implementation in logic gates. The logic expressions can be transformed or reduced into functionally equivalent forms prior to target implementation in logic blocks.

The primary purpose of logic models is to develop logical expressions for reduction into logic gates, and to test that these expressions implement the required functionality, usually for a portion of an IC component. Also to support re-use of a detailed design by documenting the logic in a fairly process neutral format that can be retargettable to specific technologies or process lines through automatic synthesis.

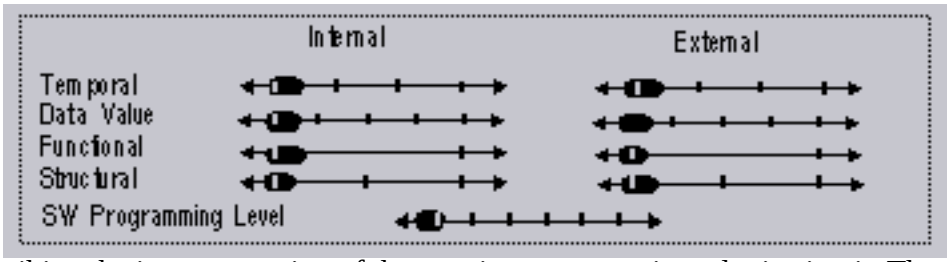
6.6 Gate-Level Model -



A gate-level model describes the function, timing, and structure of a component in terms of the structural interconnection of Boolean logic blocks. The Boolean logic behavior blocks implement simple boolean functions such as NAND, NOR, NOT, AND, OR, and XOR. A gate level model describes the actual structure and versions of logic gates that are assembled to implementing the target component.

The primary purpose of a gate level model is to document the particular implementation of an IC component in terms of the interconnection of elements from a specific logic family library, for fault-grading and the production of operational test-vectors, to determine the precise timing response of the circuit to stimuli to increase the accuracy of more abstract models, to test that the design meets all timing and functionality requirements, and to optimize the gate-level implementation of the logical design.

6.7 Switch Level Model -

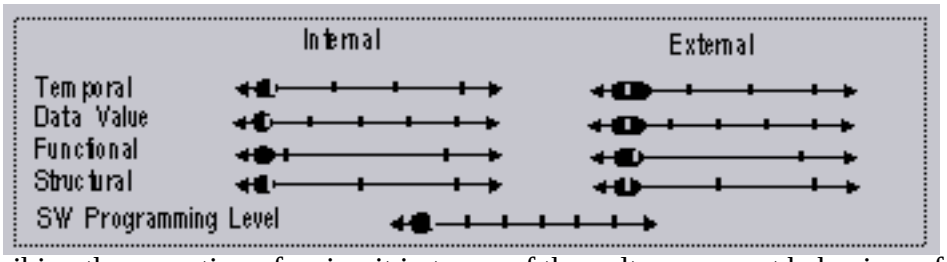


A model describing the interconnection of the transistors composing a logic circuit. The transistors are modeled as simple level-controlled on-off switches.

The primary purpose of a switch-level model is to efficiently determine the response of a portion of an IC, usually a gate or set of gates, to increase the accuracy of more abstract models. This determination

is more efficient though coarser than circuit level models.

6.8 Circuit Level Model -



A model describing the operation of a circuit in terms of the voltage-current behaviors of resistor, capacitor, inductor, and semi-conductor circuit components and their interconnection.

The primary purpose of a circuit model is to determine the response of a portion of an IC, usually a gate or set of gates, to optimize its design according to its requirements, and in particular to most accurately determine its minimum and maximum propagation, switching times and loading and driving capabilities in terms of transistor and conductor properties and configurations, to increase the accuracy of more abstract switch and gate-level models.

Return

---

7. Software Hierarchy

The following section defines the forms of software and the levels of abstraction in the software hierarchy.

7.1 Data Flow Graph (DFG) Task Primitive -

Software at the task-primitive level expresses the software application in terms of its building-block subroutines and functions. Usually expressed in a graphical form.

7.2 Pseudo-code -

Pseudo-code is a simplified or abstracted code form that is used as an intermediate step toward preparation of standard high-level-language code.

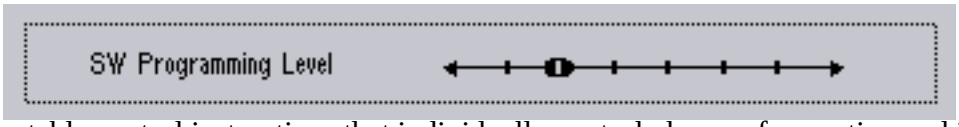
7.3 High Level Language (HLL) -

High-level-language software is a machine independent (retargetable) form of software conforming to a standard language grammar and syntax. Characterized by text-based arbitrary symbolic variable names and control constructs. Has algebraic expression statements.

7.4 Assembly Code -

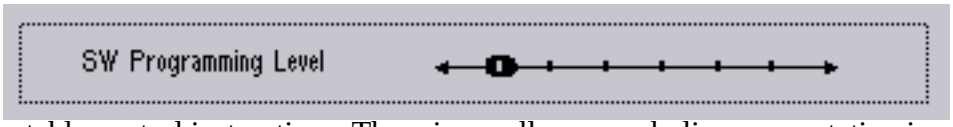
Assembly code is mnemonicized object code. See object code. Assembly code tends to be text-based, but with limited expression syntax, usually restricted to a set of explicit operators and register names. Operators tend to be simple airthmetic and logic operations. Each line typically specifies one operation per instruction cycle. Data variables are usually related to specific memory addresses.

7.5 Micro-Code -



Machine executable control instructions that individually control phases of execution and internal processor structures. Much like assembly code, but instead of specifying one operation per instruction cycle as in assemble code, micro code specifies the settings for several units, such as buses, multiplexors, and function units for each clock cycle to accomplish the type of operation typically specified by a single assembly instruction. Several micro-code cycles typically compose a traditional assembly instruction in a micro-code based architecture. Each assembly-level instruction is typically decomposed into a micro-code routine which is a set of micro-code instructions.

7.6 Object code -



Machine executable control instructions. There is usually no symbolic representation in object code. The code is expressed directly in the form acceptable by the digital logic on the processor, usualy as numeric ones's and zero's.

---

8. Supporting Terms

The following section lists a collection of terms and their definitions which support the definitions in the previous sections. These terms tend to be more general than the model-specific terms above, yet clear interpretations of these terms are not often found within our context. The definitions above rely on an unambiguous understanding of their meanings. 8.1. Abstraction Level Terms

8.1 Abstraction Level and Hierarchy

8.1.1 Abstraction Level -

The abstraction level is an indication of the degree of detail specified about how a function is to be implemented. Abstraction is inversely related to the resolution of detail. If there is much detail, or high resolution, the abstraction is said to be low. More implementation details become constrained as the abstraction level is lowered.

The abstraction levels form a hierarchy. A design at a given abstraction level is described in terms of a

set of constituent items and their inter-relationships, which in turn can be decomposed into their constituent parts at a lower level of abstraction.

For example, the function:

```
c = a convolved with b
```

Is more abstract than:

```
c := idft( dft(a) * dft(b) );
```

which shows more information about how to compute (or implement) the function. The function could have been implemented in other ways (such as c = $\sum a_{i+k} * b_{n-k}$), so the second equation provides details that are not contained in first equation.

An even lower level of abstraction for the example would be an implementation of the equation in the form of a multiplier/accumulator with a finite state machine controller. This description provides more constraining information about the implementation, since the second equation could have been implemented and described in other ways, such as a software programmed computer.

A still lower abstraction level would be a logic-gate netlist for the finite state machine and/ or multiplier/accumulator. It would resolve more details about, for example, how to implement the adder function, and therefore it constrains the implementation further.

An even lower abstraction level would be the polygon layout for the logic, since more details would be resolved and constrained.

It should be evident that many intermediate levels of abstraction were skipped in the above example.

It should be noted that abstraction level does not indicate accuracy. Abstraction and accuracy should be distinguished in the same sense that precision is different from accuracy. For example, an abstract behavioral model of a deterministic processing element could describe the execution of a given function as consuming 100.288-uS, while a much less abstract RTL model would concur with exactly the time in terms of 10288 10-ns clock cycles. Thus, both can be equally accurate but differ in abstraction.

8.1.2 Hierarchy -

A multi-level classification system that supports aggregation and decomposition. A node at a given level of the hierarchy can be represented by the set of its descendant nodes (and their inter-relationships) on the next lower level.

In RASSP, several hierarchies are important, namely: the functional or logical hierarchy and the structural or physical hierarchy. Often there is correspondence between these hierarchies, but not always.

The functional hierarchy decomposes a system according to functions. For instance: receiver, product detector, convolver, multiplier/accumulator, register/mux, logic gate.

A structural hierarchy often decomposes a system according to the physical structure, such as racks, frames, chassis, boards, modules, chips, cells, gates, transistors. Structure may also refer to logical

relationships, such as data-structures.

The functional to structural mapping tends to shift with model year as integration levels increase.

8.2 Design Object Classes

8.2.1 System -

Anything consisting of multiple parts that performs a function or set of functions. The boundaries of a system usually follow the structural implementation, but may also cross physical boundaries. For instance, "The memory system xyz shares boards p,q,r,s with other systems.". Systems are typically hierarchical in that a system is often composed of multiple sub- systems. Each sub-system is in turn a system which may be composed of sub-systems or components.

Examples of systems are:

1). **Carrier battle group defense system, composed of sub-systems:**
   a.) command, control, and communications systems,
   b.) navigation systems such as GPS and inertial gyro,
   c.) weapons systems,
   d.) counter measures systems,
   e.) sensor systems such as search-radar and sonar,

2). **Weapons system, composed of sub-systems:**
   a.) tracking radar system,
   b.) guidance system,
   c.) telemetry system,
   d.) propulsion system,
   e.) fusing system.

3). **Tracking radar system, composed of sub-systems:**
   a.) antenna system, and mechanical control system
   b.) transmitter and receiver system,
   c.) analog preprocessor,
   d.) digital signal processing system
   e.) data processing system (track interpretation & navigation fusion)
   f.) operator control and display systems

4). **Digital signal processing system, composed of sub-systems:**
   a.) I/O system,
   b.) network communication system,
   c.) local and distributed operating systems,
   d.) runtime command processing and control systems,
   e.) processor clusters, or multi-node board systems
   f.) power supply system

5). **Multi-node processor board system, composed of sub-systems:**
   a.) processor element subsystems,
   b.) shared memory system,
   c.) scan chain control system

6). **Processing element system (CPU system) composed of sub-systems:**
   a.) local memory system
   b.) local node operating system
   c.) processor unit subsystem
   d.) inter-PE communication system
   e.) runtime node control system

Notice that the above examples follow a system hierarchy from 1c to 2, 2a to 3, 3d to 4, 4e to 5, 5a, and 5a to 6. Note that RASSP is concerned particularly with the type of systems below 3d; namely digital signal processing systems and all their subsystems.

To disambiguate the term system, it is recommended that the appropriate qualifier be stated before its

use , such as "CPU memory system", "radar system", "DSP system", etc..

Most systems are a component of a larger system. The terms "system" and "component" can therefore be used to refer to the same item, but from different view-points: the former when speaking of the item and its constituent parts, and the later when speaking of the item as a constituent of a larger system.

8.2.2 Component, Module -

A component or module is any part of a design that may be instantiated one or more times and combined with other components to form a system or encompassing component or module. A functional component of a system implements a specified function. A hardware component of a system may be a (populated) chassis, board, IC, macro-cell, connector, etc.. A software component or module may be a collection of routines, such as an operating system, or library, as well as a single routine.

Often the word component has been used to specifically designate an IC chip, as in a board component, which indicates a specific structural partitioning. However, since terms such as "IC-chip" adequately describe such devices, the RTWG prefers to not relegate the usage of the word component to that single structural level.

Similarly, the word module has often been used to designate a multi-chip substrate that is a board component such as the Multi-Chip Module (MCM). However, the word module has also been used to designate larger systems such as the Lunar Module and other non-hardware items such as software code modules. Therefore, in RASSP the synonyms component and module are used in a recursive/hierarchical terminology that may correspond to either the functional or structural implementation.

Every complex component is itself a system. The terms system and component (or module) can therefore be used to refer to the same item, but from different view-points: the former when speaking of the item and its constituent parts, and the later when speaking of the item as a constituent of the larger system.

8.2.3 Architecture -

1. In general, the structure of anything.
2. In the context of Software Architecture, is the configuration of all software routines and services for meeting a system's objective.
3. In the context of Hardware Architecture, is the configuration of all physical elements for meeting a system's objective.
4. In the context of System Architecture, is the collection and relationship of the system's constituent hardware and software components. For example, a multi-processor system's architecture would include the hardware network architecture and the software architecture in the form of distributed and local operating systems, and application and control routines.

8.2.4 Hardware -

Hardware is a physical component or system implementation of function(s). Especially the intrinsic functional aspects of physical systems which are not electronically modifiable. (Software configurable or programmable hardware is simply a software selection of intrinsic hardware functionality.) Modification of HW intrinsic functionality usually does not occur after construction and cannot be changed without physical alteration, since it requires mechanical alteration of circuits, In other words, the intrinsic hardware functionality is set "Hard" and fast, unless physically altered. Not intended to change over the unit's life- cycle without physical alteration.

8.2.5 Software -

Software is the electronically modifiable aspect(s) of a system's behavior. Especially such aspects that are intended to be changeable (multiple times) over a unit's life-cycle. Software is often the set of electronically modifiable instruction sequences which are interpreted by hardware and thereby control the operation of the hardware.

8.2.6 Firmware -

Firmware is a set of electronically modifiable aspect(s) of a system's behavior which are not intended to be altered often, if at all, during a unit's life cycle.

8.3 Information Classes

It is often useful to distinguish between two major types of information that are present in a system: *application-data* and *control-data* . Several other terms depend on distinct definitions for the terms that follow.

8.3.1 Application Data -

Information that is the object of computation or communication, and information that does not affect, determine, or change the sequence of subsequent operations.

Because decisions as to what to include in a model often pertain to portions of the design defined as control or data, an orthogonal classification must be made to distinguish between control versus data. The challenge in making this classification is that data is control to some and control is data to others; it is "view" specific. For example, all the "signals" used to control a protocol to transfer data over a bus may be viewed as control by a bus designer. But another designer may view any clocked input as data for a synchronous design.

8.3.2 Control Data -

Information that affects, determines, or changes subsequent events or operations.

8.4 Design Process Terms

8.4.1 Synthesis -

Design synthesis is the process of creating a representation of a system at a lower level of design abstraction from a higher level (more abstract) representation. The synthesized representation should have the same function as the higher level representation. Synthesis literally means: the combining of constituent entities to form a whole unit. In system design, synthesis refers to the process of finding a set of elements and a way of combining them, such that when so combined to form a system, the system meets its requirements. Synthesis may be automated or done manually, but the term is usually used in reference to an automatic process.

8.4.2 Simulation -

Simulation is the process of applying stimuli to a model and producing the corresponding responses from the model (when those responses would occur).

8.4.3 Top-Down Design -

The term top-down design refers to the flow of design-driving requirements from the abstract function (high-level or top) to the specific implementation (low-level details or bottom). It is the process whereby requirements are developed for the components of a given level of the design abstraction hierarchy that are used to drive the design and selection of components in the lower levels.

In contrast, bottom-up design would refer to the process of pre-selecting certain components, and then partitioning the remaining requirements accordingly.

An example of a top-down design process would be one that consists of the following steps:

1. a behavioral level model is partitioned into submodules,
2. interfaces between submodules are defined,
3. resources and requirements for each component module are defined,
4. verification that the partitioned form is equivalent in timing and function to the unpartitioned behavioral model,
5. steps 1-4 are repeated recursively for each submodule, until sufficient detail is resolved for physical construction, loading, and operating the system.

During a recursion of steps 1-4, if the verification fails due to an unobtainable requirement, the critical issue is passed upward for reallocation of the requirements. Then the process continues recursing.

Top-down design has in the past been interpreted by some designers literally as the time order of abstraction focus, such that the abstract design would be completed prior to the detailed design in sequence. The RTWG recommends avoiding that interpretation because it does not apply to realistic design situations in which top-levels could specify unattainable requirements for the lower sub-modules. The RTWG instead prefers the interpretation where multiple levels of the design process are active concurrently, but the flow of requirements is from top to bottom, with feedback on how well the requirements can be met flowing from bottom to top.

8.4.4 Prototype -

A prototype is a preliminary working example or model of a product, component, or system. It is often abstract or lacking in some details from the final version. Two classes of prototypes are used in design processes: physical prototypes and virtual prototypes.

The purpose of a prototype is for testing, exploration, demonstration, validation, and as a design aid. It is used for testing design concepts and exploring design alternatives. Prototypes are also used to demonstrate design solutions or validate design features.

8.4.5 Physical Prototype -

A physical prototype is a physical model of a product, component, or system. The traditional prototype is a physical prototype, as opposed to a virtual-prototype. See prototype, virtual-prototype.

Examples of physical prototypes are: bread-boards, mock-ups, and brass-boards. Physical prototypes are characterized by fabrication times that typically require weeks-to-months and that typically require days-or-weeks modify. Construction usually involves detailed design, lay out, board or integrated-circuit fabrication, ordering, and mounting via solder or wire-wrap. Additionally, programmable systems or parts require detailed target-software design of drivers and operating system, or programming PLAs, FPGAs, PROMS.

8.4.6 Virtual Prototyping -

Virtual prototyping is the activity of configuring (constructing) and using (simulating) a computer software-based model of a product, system, or component to explore, test, demonstrate, and/or validate the design, its concept, and/or design features, alternatives, or choices. Specifically, the act of using the virtual-prototype model as if it where an example of the final (physical) product. See prototype, virtual prototype.

8.4.7 Verification -

The process of determining whether a developed *model* is constructed correctly and fulfills it's specified requirements. Usually conducted through examination of model outputs in response to

specific input stimuli or through analytical/formal proof methods. Verification has been equated to asking, "Did we build the *thing-right* ?".

8.4.8 Validation -

The process of determining whether a developed *component* or *system* satisfies its overall goals and requirements. Also the process of evaluating a model to ensure accuracy to the component or system being modeled. Validation is usually considered more comprehensive than verification. Validation has been equated to asking, "Did we build the *right-thing* ?".

8.5 Design Tool Terms

8.5.1 Model -

A model is the description of a function, system, or component that when executed, usually upon a simulator, simulates the operation of the intended function on applied stimuli.

8.5.2 Emulator -

Hardware device that mimics the electrical behavior of a component for in-circuit operation.

8.5.3 Simulator -

A software utility for executing models within a computer. In the case of VHDL, the simulator manages the passage of simulated time, and the illusion of concurrent model and process execution. The simulator also provides user interactive or batch development capabilities such as execution control, which includes break-points, stepping, running, stopping, and continuing; tracing and examining model states, and setting model states.

8.6 Test Related Terms

8.6.1 Test Bench -

A test bench is a model or collection of models and/or data files that applies stimuli to a module under test (MUT), compares the MUT's response with an expected response, and reports any differences observed during simulation.

8.6.2 Test Vector -

A test vector is a set of values for all the external input ports (stimuli) and expected values for the output ports of a module under test.

8.6.3 Functional Test -

A test for the required function of a unit. Functional tests are independent of the implementation of the unit under test. Functional tests do not require implementation knowledge, but test for design errors/correctness. As such, functional tests do not check for physical hardware faults in the manufactured system. For instance the functional test of a multiplier unit could be 4 * 7 = 28. Such tests check that the unit would perform multiplication and handle corner conditions such as four quadrant signage.

8.6.4 Operational Test -

A test for the proper operation of a unit. This test is implementation dependent, since it checks for hardware faults such as stuck-at, open, and short. Tests for physical faults in manufactured system.

8.6.5 Boundary Scan -

Boundary scan is a structured test technique for testing digital circuits. It consists of embedding shift registers at each and every pin (I/O) of a component so as to control and observe each and every pin independent of the internal logic of the component. Though designers have previously built scan cells in their own ways, IEEE has standardized a test architecture for boundary scan. The standard is IEEE 1149.1: Test Access Port and Boundary Scan Architectures.

8.6.6 Signature Analysis -

Signature analysis is the testing of digital circuits by applying stimuli (a set of Inputs) and measuring the response of the circuit (called the test result). The result is compared against an expected pattern (called the signature) and fault analysis based on the stimuli (also called a test vector) and the response is called signature Analysis.

8.7 Requirements and Specifications

8.7.1 Specification -

Any written document or executable program that explicitly states the quanities and functionalities either needed or provided by a system or component. The former class are called *requirements-specifications* while the latter are called *design-specifications* . Many other types of specifications exist as well such as manufacturing-specifications, maintenance-specifications, and test-specifications.

8.7.2 Executable-Specification (E-Spec)

Traditionally, specifications are a collection of statements written as a human-readable documents. The automated form of such statements implemented as computer-analyzable or *Executable* progamming models are known as E-Specs. Executable versions of requirements and design specifications, called ER-Specs and ED-Specs respectively, interact to test a design relative to its requirements.

8.7.3 Requirement-Specification (Req-Spec) -

A Requirement-Specification states the necessary and sufficient qualities, quantities, and functions that a system or component must exhibit. Requirements may be expressed as functions, specific values, allowable ranges, or inequalities such as maximums, minimums. Includes electrical behavior of function and timing as seen from the interface, as well as physical constraints of power, cost, size, fit, and weight.

8.7.4 Executable-Requirement-Specification (ER-Spec) -

Computer analyzable or *Executable* versions of Req-Specs are called ER-Specs. They test for requirement compliance by applying tests to candidate systems. In this role, an ER-Spec forms a test-bench.

8.7.5 Design-Specification (Design-Spec) -

A Design-Specification is the statement of a design solution. The design-spec states the requirements for each the system's constituent components and how to configure them as well as the resultant performance, functionality, and other pertinent quantities that characterize the system as designed. The components may be architectural blocks, hardware elements, software elements, or combinations.

8.7.6 Executable-Design-Specification (ED-Spec) -

Computer analyzable or *Executable* versions of Design-Specs are called, ED-Specs. They are used to interact with ER-Specs for automatic requirements testing. An ER-Spec model represents the component or system, while the ER-Spec forms a test-bench.

8.8 Reusability and Interoperability

8.8.1 Reusability -

The degree to which a module, component, or system may be used again in other instances for which it may or may not have been specifically intended. Reuse occurs across several dimensions, such as the life-cycle phases, at the packaging levels, and across model-years. Reuse occurs at various distinct levels. For example: 1. Reuse of components (hardware parts) or modules (software object-code), also called direct implementation, 2. Reuse of hardware logic or software source-code recast in new technology or integrated with other logic or code. 3. Reuse of architecture through re-implementation of functional block concept with new partitioning, integration, or technologies.

8.8.2 Interoperability -

Model Interoperability designates the degree to which one model may be connected to other models, and have them function properly, with a modicum of effort. Model interoperability requires agreement in interface structure, data format, timing, protocol, and the information content/semantics of exchanged signals.

---

9. Closing

The current listing of the complete definitions is maintained on the World-Wide-Web page: http://rassp.scra.org. Definitions for many problematic and supporting terms important throughout the RASSP program were added to the list.

The RTWG attempted to use the refined nomenclature to define how the various types of models are implemented in VHDL and how interoperability between models of the same and differing types is obtained, as well as to identify what design risks and benefits they address.

Feedback and comments are encouraged to maintain a comprehensive taxonomy that supports electronic industry needs. Please direct comments to the RTWG in care of Carl Hein, at chein@atl.lmco.com, or by phone at 609-338-3924.

Appendix A: Background

The Rapid-prototyping of Application Specific Signal Processor (RASSP) program reduced system prototyping time and cost by promoting the use of interoperable models throughout all stages of the design process. Interoperable models increase re-use and accelerate the design cycle by enabling library-based design techniques. Inter-related aspects of the RASSP environment, including design methodologies, libraries, and tools, are being developed by the many participating RASSP organizations which include aerospace contractors, university developers, educators, and CAD

vendors.

Differing terminology created confusion among the RASSP organizations. Some organizations used common modeling terms with divergent meanings, while others used different words to describe the same type of models. Clearly communicating ideas about modeling techniques and model types among organizations was essential in achieving the goals of RASSP. Without a common language, the RASSP community could effectively communicate, and the simulation model compatibility would be hindered.

The Terminology Working Group (RTWG) was formed at the January 10, 1995 RASSP Principal Investigators meeting in Atlanta, GA., to address the modeling and terminology challenge. The core working group consists of members representing the two prime contractors, a technology base developer, the educator facilitator, and the government.

The RTWG's mission was to develop a systematic basis for defining VHDL model types and to use this basis for concisely and unambiguously defining a terminology that describes the models that are used within a RASSP design process. One crucial requirement for the basic taxonomy is that it must be useful for selecting, using, and building appropriate interoperable models for specific roles in a design process. Models are used for several purposes, which include specifying or documenting design solutions and for testing and simulating proposed designs. The terminology is based on the commonly documented and applied vocabulary in the digital electronic design and modeling industry, and it draws heavily from related previous and ongoing efforts by the EIA [1], ESA [2], Army [3], Navy [4], and from the annals of related literature from Design Automation Conference (DAC), VHDL International User's Forum [5], and text books[6].

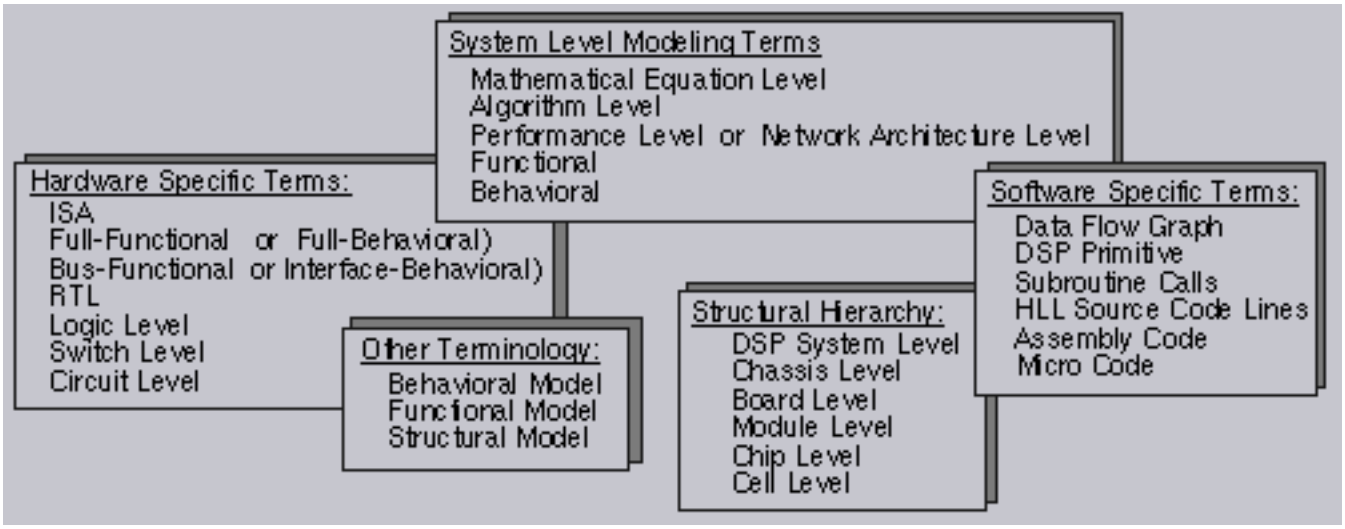
Previous efforts focused on narrower domains than RASSP. RASSP spans many domains, including: parallel processing; multi-board and multi-chassis systems; software; digital signal processing; digitical circuit design, and application functions, with strong interaction with other domains such as analog, mechanical, physical, and RF [7].

Though the RASSP program focused primarily on digital signal processor (DSP) systems design, much of the results are applicable to the design of any complex digital system, such as control systems or multiprocessor systems.

For a given domain, there are many possible vocabularies. The coverage and overlap depend on the domain to which it applies. The pre-existing vocabularies were the result of many disconnected vocabulary generation processes. They contained many ambiguities.

In Figure A1, the working group's initial modeling terms were categorized according to major area, such as: abstraction levels, system levels, hardware levels, software levels, structural hierarchy levels, and general modeling-type terms.

Figure A1 - Commonly Used Modeling Terms



Vocabulary words represent concepts and allow us to share and communicate ideas. Unambiguous communication requires not only that we have a common mapping of words to concepts, but that we have the right set of words to accurately describe the concepts we are dealing with. Developing, agreeing to, and using a concise common terminology are therefore vital to advancing the state of the art in design methods.

In defining the vocabulary terms, attempts were made to defer to the general English meaning of words as defined by the Webster's New Collegiate Dictionary so that outsiders and new-comers may be more likely to rapidly understand and adopt the terminology.

Some terms may have multiple meanings (due to historic overloading) which can be differentiated by context. We try to recognize and define each of them.

To avoid the problem of vague or circular definitions, a heavy emphasis was placed on providing examples to accompany the definitions. These examples should provide a level of understanding and concreteness to any discussions regarding the terms. The examples also tend to associate the terms to their intended uses and domains. To reduce the tendency of examples to limit or over-constrain the definitions, a range of typical and extreme cases are given and identified wherever possible.

To further avoid ambiguous definitions, attempts were made to eliminate definitions based purely on relative terms, such as "high", or "abstract", since their interpretation would be subject to one's point-of-view or experience. Instead, definitions should be based on absolute, concise, and testable statements, with special emphasis on differentiating related terms (i.e. hardware, software, firmware).

Reaching the consensus on terminology required compromise from everyone. The RTWG made a conscious attempt to borrow evenly from many sources.

The development of the more efficient vocabulary, where a minimal set of words were assigned to span the appropriate concepts, was an orthogonalization process. The process developed a set of terms that represent all of the concepts to be distinguished. Separate words were selected for distinct concepts. Words for classes of concepts were selected to represent useful generalizations.

The RTWG modified and augmented the previously defined terminology sets, broadened parochial definitions, distinguished overlapping definitions, equated close synonyms, removed non-applicable terms, added needed or missing terms, clarified poorly defined or misunderstood terms, and suggested new terms as replacements or synonyms to outdated terms. When appropriate existing definitions were not available for significant terms used within the RASSP community, the RTWG attempted to create them.

The refined nomenclature identifies the models developed within the RASSP process that document design information, their purpose in the design process, and the practices for developing interoperable versions of those models. The RASSP program developed broad consensus for a modeling vocabulary that is readily adopted by engineers and students. The vocabulary promotes the requirements of rapid design processes for interoperable models.

Appendix B: Prior Taxonomies

The working group initially compared three existing model definition approaches (shown in Table I). The group considered the features of the existing approaches as a foundation for a VHDL model taxonomy[5,8].

The Ecker and Madisetti spaces share two axes, while their remaining axes do not directly correspond. Both have an axis for Time resolution and a second axis representing the resolution of data Values in a model.

Ecker calls the second axis Value, while Madisetti calls it Format. The Y-chart's Functional-Representation axis expresses some information that is similar to the value-format axes. However, the Y- chart's Functional-Representation axis does not exactly correspond to the value-format axes because it contains information about functionality as well.

The third axis of the Ecker cube is similar to the Structural-Representation axis of the Y-chart but has no corresponding axis in the Madisetti case. (The latter situation arises intentionally.)

None of the remaining axes of the taxonomies directly correspond. The Y-chart seems limited to only the logic level. None of the taxonomies appear to directly address the hardware/software codesign aspect.

Table I - Comparison of prior and proposed taxonomy concepts

Source - Taxonomy	Axes						
Gajski and Kuhn: Y-chart			Struct. Rep.	Funct. Rep.	Geom. Rep.		
Ecker: Ecker cube	Timing	Value	View				
Madisetti Taxonomy	Timing	Format				Value	Sta
RASSP RTWG Taxonomy	Timing Res.	Data Value	Struct. Res.	Funct. Res.			Inte Ext

---

Appendix C: Additional Attributes

The RTWG considered the inclusion of additional attributes. For example, a model accuracy aspect could be specified as a percent tolerance and whether a model describes actual, minimum, typical, or maximum values. A completeness aspect could specify the portion of functionality or functions that a model describes or excludes.

Other potentional attributes included model maturity or validation level, simulation efficiency portability, complexity/ size/lines-of-code, maintainability, flexibility, modifiability, expandability, and licensing cost.

Although the additional attributes appear to be useful in selecting, building, or using models for appropriate purposes within the design process, some are less easily quantified than others. For instance, simulation efficiency could be specified in events, cycles, or instructions per simulation-host CPU cycle, with memory requirements for the model's program code and data. Comparably objective units for portability or maintainability are less obvious.

A balance must be struck between the taxonomy's expresiveness and its ability to be comprehended widely. Additional attributes should be considered in future efforts, but were beyond the scope of the current version.

Acknowledgments

The RTWG gratefully acknowledges the contributions of Randy Harr, John Hines, Paul Kalutkiewicz, Maya Rubeiz, Gerry Caracciolo, Marimuthu Sivakumar, and the many others who have provided valuable assistance in defining the taxonomy and terms and supporting the effort.

References

[1] IEEE Std EIA 567 (August 1994) and IEEE 100

[2] Preliminary VHDL Modeling Guidelines, European Space Agency / ESTEC

[3] Army Handbook - The Documentation of Digital Electronic Systems With VHDL

[4] TIREP (Technology Independent Representation of Electronics Products) NSWC

[5] Madisetti, V., " System-Level Synthesis and Simulation VHDL: A Taxonomy and Proposal Towards Standardization", VIUF Spring, 1995 Proceedings.

[6] "Modeling and Simulation", Texas Instruments Semiconductor Group, 1990.

[7] Armstrong, J., "High Level Generation of VHDL Testbenches", Spring 1995 VIUF Proceedings.

[8] Ecker, W., Hofmeister, M., "The Design Cube - A Model for VHDL Designflow Representation", Proceedings of the EURO-VHDL, 1992, pp. 752-757.

[9] Famorzadeh, S., et.al., "Rapid Prototyping of Digital Systems with COTS/ASIC Components", Proceedings of RASSP Annual Conference, August, 1994.

[10] Blahut, R., Fast Algorithms for Digital Signal Processing, Addison Wesley, New York, 1985.

(Questions, Comments, & Suggestions: chein@atl.lmco.com)