Domain-Specific Software Architectures for Guidance, Navigation and Control

Pam Binns, Matt Englehart, Mike Jackson, Steve Vestal*
Honeywell Technology Center
Minneapolis, Minnesota
January 1994

Abstract

We describe two integrated languages and associated tools for capturing and analyzing two different views of the architecture of an embedded system. One language is tailored to address guidance, navigation and control issues, while the other is tailored to address real-time, fault-tolerance, secure partitioning and multi-processor system issues. Both languages have tools that perform analyses appropriate for the issues each addresses, and tools to automatically configure the application software from a sufficiently detailed specification. The integrated languages and tools are intended to support an architecture reuse development process, in which the development of a new product in a family of similar products starts from a generic or reusable architecture specification for that product family.

1 Introduction

The concept of architecture that we wish to build upon is the intuitive one: it is the high-level design, the annotated drawings of modules and the relations between them that engineers sketch out for each other. What we are seeking to do is add rigor to this intuitive concept by creating more precise descriptions of architectures, descriptions that simultaneously present the important high-level features of a particular design while implicitly capturing or constraining the many details that are essential for a smooth transition to implementation. Moreover, we want to provide good estimates for various characteristics of a design or architecture as early in the development process as possible, coupled with a development process that provides assurance that the final implementation will in fact exhibit the predicted characteristics. Our approach to architecture-oriented software development exhibits three basic themes or principles:

- extensive use of formal models as a basis for specification languages and architecture semantics
- integrated support for multiple disciplines and multiple views
- formal architecture specification languages with accompanying analysis and code assembly tools

In playing out these basic themes, we have constrained the nature of the software we have concerned ourselves with. We have focused our attention on embedded software in general and GN&C software in particular. Within these constraints, there exist several formal models for important aspects of system behavior (e.g. linear systems theory, rate monotonic analysis). The team performing our work is a multi-disciplinary one, consisting of individuals with varied experience in real-time software, embedded computer system architecture, fault-tolerance, and guidance, navigation and control. The various formal models together with the details of suitable computation and communication paradigms provide the semantics for two integrated languages and toolsets we have developed to specify two complementary views of the architecture of an embedded

*This work has been supported by Honeywell and by ARPA and ONR under contract N00014-91-C-0195.

GN&C software system. The ControlH language and its toolset support specification, analysis, and code assembly for GN&C system architectures; and the MetaH language and its toolset support specification, analysis, and code assembly for embedded software architectures.

Figure 1 provides an overview of the integrated toolsets for the ControlH and MetaH architecture specification languages. Both toolsets operate from a high-level specification language, ControlH for GN&C engineering and MetaH for embedded software engineering. Both the ControlH and MetaH languages exist in both textual and graphical form, where mixed-mode display and editing is supported (e.g. a ControlH operator can be displayed as either a graphical block diagram or a series of text formulas, where a text editor can be invoked within a block from the graphical editor). Both the ControlH and the MetaH toolsets consist of a series of analysis tools that are based on appropriate formal models, together with tools to automatically assemble application software from a high-level specification.

These two languages and their associated tools are designed to be used by different specialists to address different issues and requirements. ControlH allows GN&C engineers to specify dynamical system models and GN&C algorithms, to perform analyses that are appropriate to the field of GN&C engineering (e.g. dynamical system simulations, linearizations and equilibria determinations, linear analyses), and to automatically generate code that implements the functionality of a specified algorithm. MetaH allows computer systems engineers to specify how functional source modules (ControlH-generated and/or obtained from other sources) are combined to form a load image that can be executed in real-time on a specified multi-processor target system. The analyses performed by the MetaH tools are appropriate to the field of embedded software engineering (e.g. real-time schedulability, and eventually reliability and secure partitioning).

We will first give an overview of the architecture-oriented development process these languages and associated tools were developed to support. We will then briefly outline in somewhat greater detail the languages ControlH and MetaH, and discuss how the concepts common to both languages apply in a multi-disciplinary development process.

2 Architecture-Oriented Development

The ControlH and MetaH languages and tools have been developed to support an architecture reuse development process. The explicit specification and reuse of software architectures promises significant cost, schedule and quality improvements when many similar software systems are to be developed for a family of products. (This situation is not a prerequisite for obtaining significant benefits from the MetaH and ControlH toolsets, but support for this paradigm has been a major design goal.)

Architecture-oriented development is based on the reuse of product family requirements and software architecture, with source module reuse occurring within the context of reuse at these higher levels. The development of each new member of a product family is to be started with the following resources available.

- a generic, reusable requirements outline that captures the common categories of services, behaviors and characteristics of all members of that product family
- a generic, reusable software architecture that captures the common overall software structure, functional decomposition, and computation and communication paradigms of all members of that product family
- a library of generic, reusable component specifications and source modules that obey the computation and communication paradigms of, match the interface conventions of, and implement variants of the functional components in, the reusable software architecture
- cross-references or relationships between the elements of the reusable generic requirements, reusable generic software architecture, and library of reusable generic component specifications and source modules

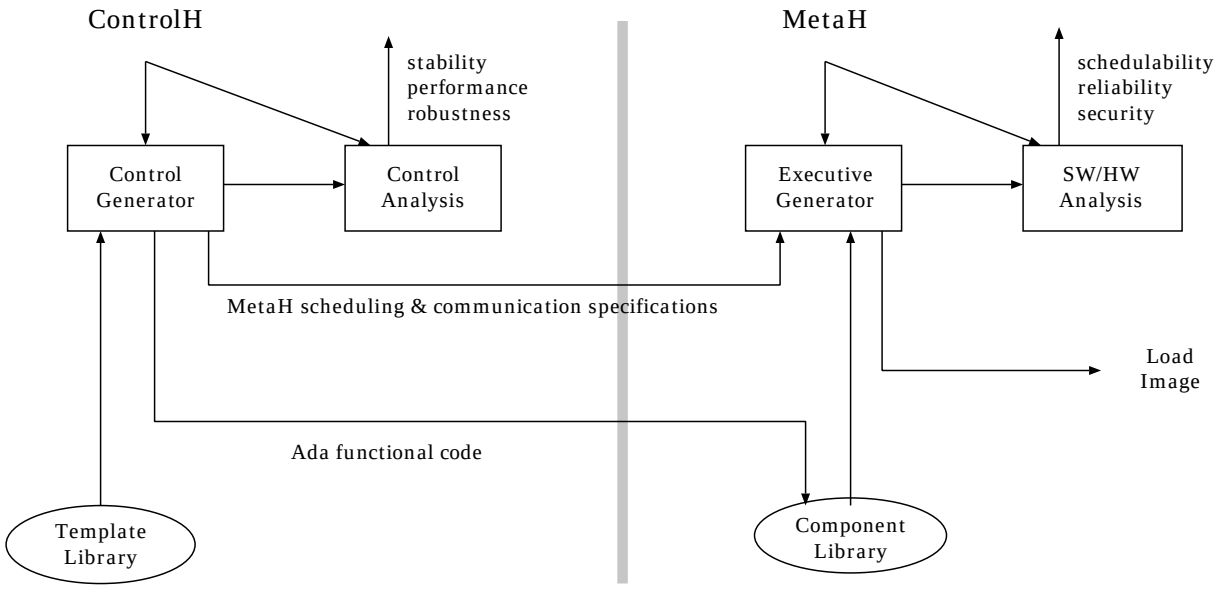

Figure 1: Integrated ControlH/MetaH Toolset

An important aspect of all these resources is that they not only identify what is common among all members of a product family, but also identify and are designed to facilitate variations between products and evolution of the family as a whole.

The development of a particular product proceeds by iteratively refining the generic requirements, architecture, and source components until a complete and detailed specification of the requirements, architecture, and source code for that particular product are obtained. Documentation and V&V are also assumed to fall within this paradigm: there are reusable generic document outlines, V&V requirements, system integration testing architectures, and test specifications and drivers.

These artifacts constitute development assets that are consciously accumulated to increase quality and production efficiency within a given market or business domain. The manner in which these assets are accumulated; the amount of investment in these assets; the quality and efficiency returns obtained from the investment; the "physical" form of these assets (the human and machine readable representations for reusable generic requirements, architectures, components, and relationships); and the tools and processes used to carry out product development; are all technical and business decisions that require analysis on a case-by-case basis.

Within this context, MetaH is a language used to capture software architectures for products whose generic requirements may include real-time, fault-tolerance, secure partitioning, and high-performance computing criteria. The accompanying analysis tools support cross-referencing from a product architecture back to the real-time, reliability, secure partitioning and performance requirements for that product. ControlH is a language used to capture plant models and controller and filter architectures for products whose generic requirements include guidance, navigation and/or control functions. The accompanying analysis tools support cross-referencing from a controller specification back to the performance and robustness requirements for that product. The accompanying application assembly tools support cross-referencing from an architecture to the source components of a product, and automate the production of the final product software.

3 ControlH

The ControlH language is used to capture high-level specifications for real-time guidance, navigation and control systems [6]. The predefined data types and operations, the syntax, and the semantics of the language have been tailored for that specific domain. The analysis tools provided with the language support the kinds of analyses, both mathematical and simulation, that are used to assess performance and robustness criteria of concern to the GN&C engineer. The associated code generator translates ControlH specifications into Ada or C code in a way that addresses software engineering concerns with execution efficiency and verification and validation. We will first summarize the basic features of the ControlH language, then discuss some of the available and planned tools to operate on GN&C system specifications.

3.1 ControlH Language

The look-and-feel of the ControlH language was developed to mirror traditional notations in the fields of guidance, navigation and control. In one sentence, the language supports specification of hierarchical block diagrams constructed from primitive mathematical operations. The language is primarily declarative or functional rather than procedural in nature. The language has both a textual notation that allows equations and formulas to be entered in a traditional form, and a graphical notation that allows specifications to be viewed and edited in a form similar to the block diagram notation commonly used in the field. The details of the language are designed to allow the specification of information necessary for automated analysis, and for management of certain characteristics of the automatically generated software.

In ControlH, variables and states are used to represent time-varying signals. The historical roots of dynamical system modeling, analysis and control lie in the continuous mathematics of differential equations. To a first conceptual approximation the operators of ControlH provide a mapping from continuous time-varying input signals to continuous time-varying output signals, where feed-back loops from outputs to inputs almost invariably appear. Computer simulations or implementations of such specifications are performed using a sampled data computation paradigm in which the operators are evaluated periodically at some rate, mapping discrete-time samples of input signals to discrete-time outputs. This model of behavior and computation underlies the basic constructs of the ControlH language: variables and named constants, states, operators, processes, global blocks of variables and named constants, and conditionals.

Data Type	Operation
booleans	and, or, xor, not
switches	increment, decrement
scalars	+, -, *, /, exponentiation trigonometric, square root
vectors	augmentation, selection
matrices	augmentation, selection inverse
state spaces	linear simulation inverse
data tables	interpolated table lookup

Table 1: Example ControlH Primitives

3.1.1 Data Types and Operators

The language provides a set of primitive or predeclared data (signal) types and operators that are appropriate for GN&C applications. The data types defined in ControlH currently include booleans; switches (essentially enumerated types); integer scalars, vectors and matrices; real scalars, vectors and matrices; discrete-time and continuous-time state spaces; and one-, two-, three-, and four-dimensional data tables. A partial listing of the primitive operations on these data types appears in Table 1.

The language allows users to connect instances of operators (either primitive or declared earlier by the user) to define new operators. In the graphical notation this appears as a collection of operators whose inputs and outputs are connected together by lines. In the textual version, the syntax for operators resembles that of a function in a traditional programming language, e.g.

`x = f(y)`. However, there is usually no concept of order of evaluation. The variable `x` in this example may typically only appear on the right-hand-side of a single statement. The output of function `f(y)` is connected to the input of every function in which `x` appears as an input parameter, regardless of whether those statements appears before or after `x = f(y)` in the textual specification of the new, containing operator. Although the textual appearance of an operator declaration (operator name followed by a sequence of input and output parameter declarations, state and variable declarations, and formulas) appears similar to that of a subprogram declaration in traditional procedural computer languages, the notion of sequential execution is absent. An example ControlH operator specification in both textual and graphical form is shown in Figure 2.

3.1.2 State

Within the context of ControlH and GN&C engineering, the term "state" has a rich and precise meaning. In ControlH, a state provides retention of values between successive executions of an operator, encapsulation of data within an operator, initialization, and assurance of semantic correctness with respect to the sampled data computation model.

Syntactically, a state consists of the declaration of two specially designated variables; the first represents the value of the state during the current processing period, while the second represents the value the state will assume during the next processing period. The language enforces the restriction that the current state variable cannot be explicitly set, it can only be implicitly set by an assignment to the next state. ControlH allows a state to be of any data type.

Figure 3 gives an example of the use of state variables to provide retention of values between successive executions or samples in a ControlH operator. This example defines the operator `delay`, called within Figure 2. The statement `x <- x_next` is shorthand for "`x_next` is the variable which serves as the placeholder for the state `x` during the next computational period." In our example, we implement a delay operation by stating that the operator's output is the current value of the state, while the value of the state during the next computational period will be the present value of the input. In a sampled data computation model, the output of this operator at a given sample time will be its input at the previous sample time.

```
operator delay is
    inputs: real_scalar: u;
    outputs: real_scalar : y;
begin
    state: real_scalar : x <- x_next;
    x_next = u;
    y = x;
end delay;
```

Figure 3: A ControlH Specification of the Delay Operation for Scalars

GN&C algorithm designers view operators as objects in a block diagram whose internal states are encapsulated within that block. To support this view, ControlH provides data encapsulation of the states declared within an operator. When an operator is called or invoked, that call corresponds logically to a unique instance of that operator and possesses a unique set of state variables. To put this another way, each time the name of an operator with state appears in the specification of another operator, a new state is created for that particular invocation or instance of the operator. Two instances of the `delay` operator appear in Figure 2, for example, and each must have its own internal state variables. In the graphical representation, if there are two blocks labeled `delay` then each has its own internal state variables.

Initialization is another important part of the concept of state. The initial value of a state must be set to some reasonable value, where reasonable values are determined by the user from the particular application at hand. In ControlH, we generally require the programmer to specify the initialization of states, although certain data types have a default initial value.

Finally, the ControlH state mechanism pro-

```
operator sum_and_delay is
   inputs: real_scalar: u1, u2;
   outputs: real_scalar: y1, y2;
begin
   r  = u1 + u2;
   y1 = delay(filter(r));
   y2 = delay(r);
end sum_and_delay;
```

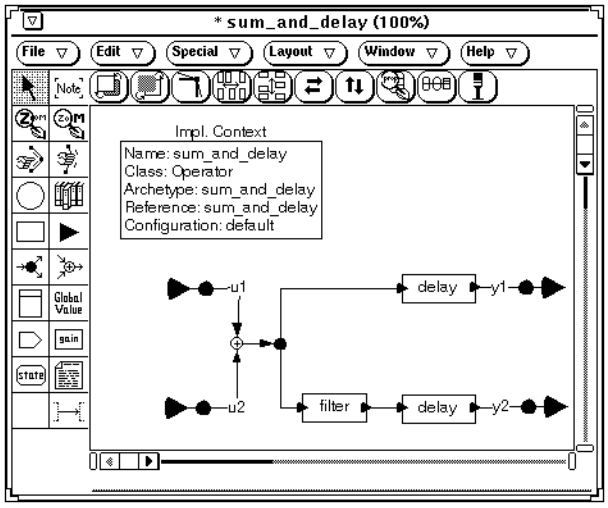

Figure 2: An Example ControlH Specification

vides semantic correctness for a sampled data periodic computation model. It is not uncommon to find in hand-written control code that a state variable is updated in a particular execution and then the value of that variable is used later in that same execution (same sample period). However, strict correctness (by which we mean correspondence to the underlying mathematics of discrete time sampled data systems) requires that the state variable only assume the next value during the next execution. To put this another way, state variable updates only occur between successive executions of an operator, never during the execution of an operator. The ControlH state construct makes it possible to assure semantic correctness in this respect.

3.1.3 Time and Space

Many GN&C applications are designed to be run at multiple rates when implemented in software. Different operators in a GN&C algorithm often sample their inputs and produce outputs at different rates, which is to say they are periodically executed with differing periodicities. For these reasons, ControlH supports a multi-process programming environment by incorporating a process object. The specification of a process currently consists only of the declaration of its name and its period.

The power of the process object is associated with its semantics in relation to operator instances. ControlH allows the user to specify in which process a called operator should be executed. If an explicit specification is not given, the default is to execute an operator instance in the same process as its parent or calling operator (what we sometimes refer to as process inheritance by a child operator from its parent or structurally containing operator). As an illustration of syntax, suppose that in the specification of one operator we wanted to explicitly execute a contained `delay` operator in a process named `Hz10` (presumably a process with a period of 0.1 seconds). This would be expressed as `y2 = delay.Hz10(r)` in textual ControlH, or by annotating the block with the process name `Hz10` in the graphical representation. This explicitly specifies that a delay of 0.1 seconds is desired in this example.

The ability to specify the rate structure of an application provides a fundamental mechanism for the GN&C engineer to manage processor utilization. In this case, the trade-off is not the traditional software one of time versus space, but

a trade-off of sampling rates with GN&C performance and robustness.

ControlH provides a powerful mechanism for performing time-space tradeoffs in the generated code that we call parametric instantiation of operators. By parametric instantiation of an operator, we mean that invocations of that operator can be translated into code that incorporates input values that are statically known at translation time (constants or computed only from constants). The semantic effects are as if the static values of those inputs are embedded in the definition of the operator, and an alternate version of the operator created for that particular invocation. This is a form of user-controlled ControlH operator inlining combined with inter-operator constant propagation.

As with traditional compilers, constant propagation and folding within the ControlH translator can greatly magnify the benefit of instantiating an operator invocation with its static inputs. The consequences of parametric instantiation are a potential significant decrease in execution time at the expense of a potential significant increase in the size of the generated code. We will note that the same effect cannot be achieved by relying on inlining and optimization of the underlying traditional programming language (e.g. Ada). The design decisions involved must be made by the GN&C engineer at the ControlH specification level. Moreover, ControlH has specialized semantics, and the ControlH translator can perform optimizations during parametric instantiation based on knowledge of these semantic constraints that cannot safely be made later, since much of this specialized semantic information cannot be recovered by the underlying language compiler.

As we will see shortly, the ControlH translator maintains a systematic mapping from the ControlH specification structure to the structure of the generated code. This mapping makes it possible to transfer traditional pragmas relating to time and space from the ControlH specification to the generated code in a reasonable way. As a consequence, the full range of time-space tradeoffs through Ada pragmas is available at the ControlH level.

3.1.4 Other Features

ControlH provides mechanisms to specify generic or polymorphic operators, which is to say operators whose exact meaning depends on the types of the input values used in a particular invocation or call to that operator. This finds its greatest use in writing generic operators that can manipulate scalars, vectors or matrices interchangeably without built-in knowledge of dimensionality, but the mechanism is a fairly general one that can also be applied in other ways.

The language supports conditional selection of input and output signals through constructs that resemble if-then-else and case statements (typically called switches in GN&C block diagrams).

There are algorithms which defy a declarative description in pure ControlH. ControlH recognizes this fact by including certain escape mechanisms in the language. In particular, true procedural (sequential) execution of conditionals, loops and assignments is provided in a way that allows users to directly write procedural algorithms for an operator where this is deemed essential.

ControlH allows the specification of global blocks of constants and variables. Global constants make it convenient to define parameterized models and controllers. Global variables make it possible for different operators to access common values in a synchronized manner, including operators executing at different rates in different processes.

3.2 ControlH Tools

Figure 4 shows the toolset used to support ControlH specification development. An important aspect of our approach is that much detailed analysis and V&V activity by the GN&C engineer is based on the generated code that will be the actual implementation. The traditional GN&C engineering task of verifying the system specification is thus also simultaneously testing and verifying the behavior of the implementation software.

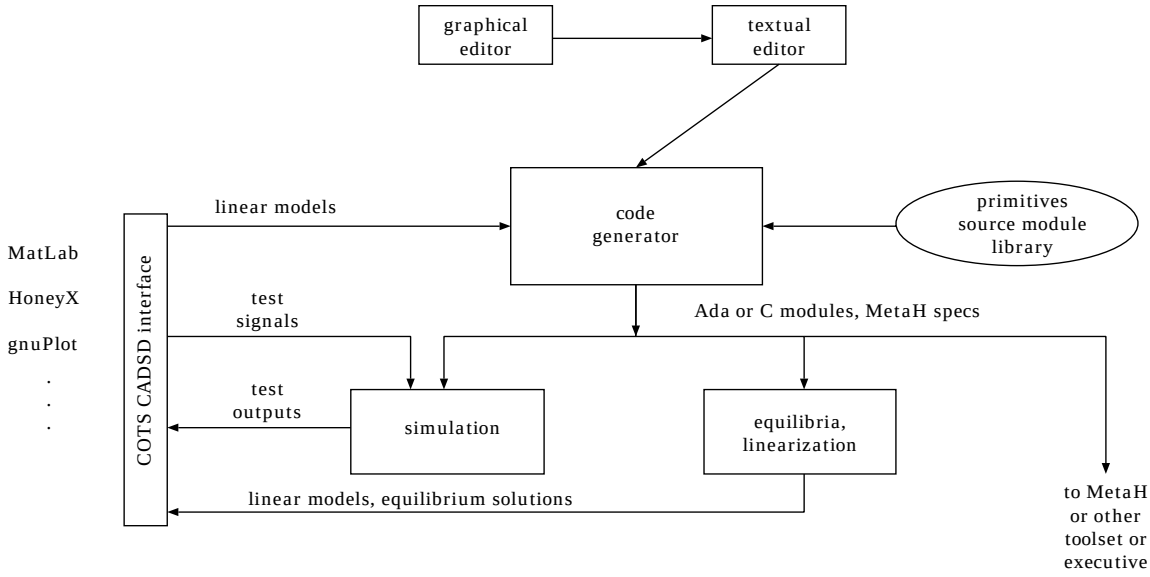

Figure 4: ControlH Tools

3.3 Code Generation

In the previous section, we described the flavor and power of the language in the sense of providing a natural grammar for expressing GN&C applications. However, we are not simply concerned with the creation of an attractive GN&C specification and analysis environment. The quality of the code that is produced by the translator is also of great importance to us. We have designed the language and translator to produce high quality GN&C software that is optimized for use in embedded real-time systems. Our goal is to improve the quality of the code through automatic generation, rather than gain the convenience of automatic generation only at the expense of code quality.

ControlH	Ada, C
local variable	local variable
operator	procedure
states	global variables
process	procedure & packages
global constants block	global package
global variables block	global package

Table 2: Mappings from ControlH Objects to Ada Constructs

Our ControlH code translator currently generates either Ada or C software. An important aspect of our approach is the generation of well-structured source code. Among other things, there is a reasonably intuitive mapping from ControlH objects to the Ada or C constructs generated from them. This mapping is shown in Table 2. It is important to note that the mapping between ControlH operators and source language procedures preserves the modularity of the algorithm. The resulting ease of traceability between specification and implementation is extremely useful during multi-disciplinary trade-off studies and during verification and validation.

A key aspect of our code generation approach is that the set of primitives is very loosely coupled with the code generation algorithms themselves. In particular, the set of primitive operators and the library of source code templates used during translation can be fairly easily modified. We use a rule-based approach for code generation in which the translator does not directly select and insert a particular block of code for a particular primitive operator. Instead, the translator invokes a rule that is provided with any user-specified pragmas and the results of the translator's type inferencing, data flow and constant propagation analysis. This operator-dependent rule then selects

and tailors the final code generated for a particular primitive operator. Although it is transparent to the typical GN&C engineer using the ControlH toolset, in fact the ControlH translator is more like a module assembly tool than a traditional compiler whose coding idioms are largely immutable.

While translating a ControlH specification, the code translator attempts to statically evaluate operations. In general, when it is possible to determine the values of the inputs during translation, and it is appropriate to evaluate that operation at that time, the expression will be evaluated during translation. Static evaluation of operations is not limited to a small subset of common primitives, such as addition, subtraction, and multiplication. Indeed, every hierarchical operator and nearly every primitive one is a candidate for static evaluation during translation. As mentioned earlier, the constant propagation and folding capabilities of the translator can be used in a powerful and user-controlled way by parametrically instantiating operators.

3.4 Simulation

Functional simulation is performed by combining the generated source modules with with a simulation executive, which is an idealized (non-real-time) scheduler and state update function. The resulting simulation does not provide any feedback about computer resource utilization or timing, but does provide the GN&C engineer with the ability to perform dynamical system simulations and assess the mathematical functionality of the specified GN&C algorithms. We will note again that this assessment is made using the generated source modules that will ultimately be the actual implementation of the GN&C functionality, since these same modules will be passed on to the MetaH toolset for inclusion in the embedded software.

3.5 Linear Analysis

Many toolsets exist to perform standard mathematical analysis of linear systems (e.g. determine poles and zeros, generate frequency response curves). The code generation and simulation capabilities of our toolset allow us to easily integrate with existing linear analysis toolsets.

By substituting a linearization executive for the simulation executive, the user can perform linearizations about points of interest. The executive numerically computes an equilibrium for the nonlinear discrete-time system implemented by the software, and the Jacobian for the system about that equilibrium. The equilibrium search is configurable in terms of the value of the state and input perturbations, initial estimates for the trim condition, exit criterion from the equilibrium search, and specification as to which inputs and states are fixed within the equilibrium search. Linearizations can then be passed to existing linear analysis toolsets.

The ControlH translator also allows the user to specify that a particular operator is in fact a transfer function imported from a linear analysis toolset. Design and analysis of linear systems or subsystems can be performed in another toolset, with the results incorporated into a ControlH specification and, ultimately, into the generated code and the final embedded implementation.

4 MetaH

The MetaH language and tools support the development of real-time, fault-tolerant, securely partitioned, multi-processor software[1, 15]. Like ControlH, the MetaH language is largely structural or declarative rather than procedural; and, like ControlH, it has both a textual and a graphical representation. However, the MetaH user is typically a computer systems engineer who uses the toolset to combine ControlH-generated sub-architectures and source components with pieces obtained elsewhere (e.g. device drivers, real-time data base managers, display management code) to produce a load image for an embedded computer system architecture. In many cases the specification of the hardware architecture is also part of the development process. The criteria of interest to the computer systems engineer include real-time schedulability, reliability, and se-

cure partitioning. We will summarize the basic features of the MetaH language and then discuss some of the available and planned tools that operate on computer system (software+hardware) architecture specifications.

4.1 MetaH Language

At first glance, the MetaH language is similar to many existing structured software design or object-oriented design notations. The graphical representation of MetaH allows entities to be placed on a screen and connected together, where each entity may be defined as another collection of connected entities in a hierarchical manner, as in Figure 5. However, MetaH has a rich set of entity types and detailed semantics, extensive and precise enough to perform a detailed real-time schedulability analysis and to automatically assemble a load image. For example, MetaH supports the specification of real-time periodic tasks. It also provides a real-time connection, where data movement is synchronized with periodic computation in such a way as to support distributed state update semantics in multiprocessor systems (where "state update" has the GN&C meaning).

MetaH allows a specification of system components and connections, and attributes of those components and connections that are relevant to real-time, fault-tolerant, secure partitioning, and multi-processor aspects of an application. The kinds of entities in a MetaH specification can be divided into lower-level entities that describe source code modules (e.g. subprograms, packages) and hardware elements (e.g. memories, processors); and higher-level entities that specify how previously declared entities may be combined to form new entities (modes, macros, systems, applications).

The language has both a textual and a graphical syntax defined, and tools exist that allow a MetaH specification to be viewed and edited interchangeably in either format. Figure 6 illustrates the general structure of MetaH specifications using the textual syntax. Figure 5 shows the graphic representation for the mode M interface (on the left) and the implementation M.EXAMPLE (on the right).

Entities have separate interface and implementation specifications, where multiple implementations can be specified for a given interface. An interface specification (e.g. mode M in Figure 6 and the left screen in Figure 5) may contain declarations of shareable monitors and packages; and/or source entities called ports and events (e.g. mode M has one port in its interface). An entity implementation may contain declarations for component entities and connections between those components (e.g. P1 and P2 are components within mode implementation M.EXAMPLE). In the graphical representation, the interface corresponds to the icon to which it is connected (e.g. mode M on the left of Figure 5), while an implementation corresponds to a diagram that shows one possible set of internal components and connections (e.g. the right screen of Figure 5 is an implementation for the mode M interface shown on the left).

A process is the fundamental unit of scheduling, allocation to hardware processors, and fault containment. Process implementations identify source code modules and can specify a variety of attributes relating to real-time scheduling, secure partitioning, fault behavior, and multi-processor allocation. Both periodic and aperiodic processes are supported. While a periodic process is active it is automatically and repeatedly dispatched with some periodicity that is specified as an attribute of that process. Aperiodic processes are dispatched in response to an event, where events may be raised by hardware (i.e. an interrupt) or by software. A variety of user-settable attributes allow fine control over process scheduling (e.g. process criticality, response of an aperiodic to an event arrival while still servicing a previous event).

A process interface may identify input and output ports, which correspond to input and output buffer variables within the source code. Process ports are connected together to indicate communication that is to occur between processes at run-time (e.g. between P1 and P2 in Figures 5 and 6). Ports are strongly typed using the type system of the underlying source language, and port-to-port connections are type checked. Port-

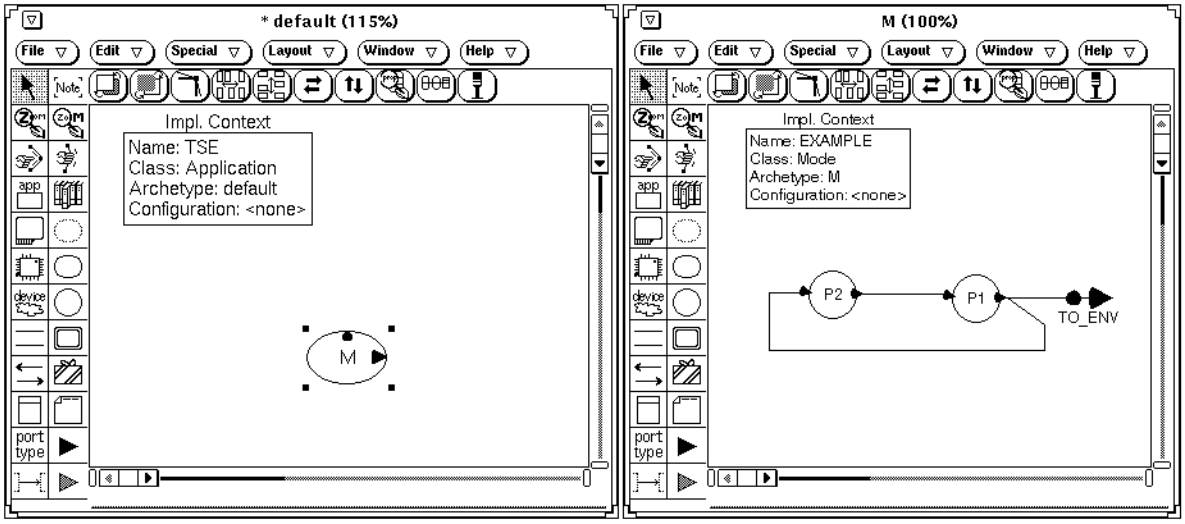

Figure 5: Example Graphical MetaH Specification

to-port connections cause periodic assignments to occur between the buffer variables associated with the ports declared in the process interfaces. These communications are executed in real-time and synchronized with process execution in such a way as to provide the afore-mentioned distributed state update semantics.

Process interfaces may also identify output events, which may be connected to aperiodic processes or to modes (explained in the next paragraph); and shareable monitors or packages, which may be equivalenced to shareable monitors or packages in the interfaces of other processes to indicate that the same data is to be shared by those processes. A service call allows a process to raise an event, and real-time semaphores are available to synchronize access to shared monitors.

Modes and macros specify collections of processes. A mode or macro interface may identify shareable monitors and packages, input and output events, and input and output ports. A mode or macro implementation may specify a collection of processes, together with connections between those processes and between the entities declared in the mode or macro interface (ports, events and shared entities in a mode or macro interface must ultimately be connected to components in some implementation of that mode or macro). In addition to the port-to-port connections mentioned above, event-to-aperiodic process and event-to-mode connections; and equivalence connections between shareable monitors and packages; may also appear in mode and macro specifications.

Not all processes in a system need be active at the same time. In particular, the language allows a specification of multiple modes of operation, where each mode may identify different sets of processes and/or different port and event connections to be in effect during that mode of operation[14]. One or more out events from processes (or from hardware events, i.e. interrupts) may be connected to a mode. When that event occurs at run-time, a change is made from the current mode to the connected mode. When this change occurs, all active processes that are not in the new mode (including possibly the one that raised the mode change event) are made inactive. Such processes are aborted cleanly w.r.t. port-to-port communication and monitor locking. Periodic processes cease being periodically dispatched, and aperiodic processes will no longer be dispatched in response to an event connected to them. When the mode change occurs, all inactive processes that are in the new mode are made active. Process-specific initialization code is executed, after which process dispatching as described above is performed. The port and event connections in effect become those specified for

```
with port type TARGET_T;
process P1 is
  TO_P2 : out port TARGET_T.INTEGER_T;
  FROM_P2 : in port TARGET_T.INTEGER_T;
end P1;

periodic process implementation P1.IMP is
  sub1, sub2: subprogram;
attributes
  self'Period := 25 ms;
  self'SourceTime := 1 ms;
  self'SourceFile := "p1_root.a";
  sub1'SourceTime := 2 ms;
  sub1'SourceFile := "sub1.a", "sub1_b.a";
  sub2'SourceTime := 8 ms;
  sub2'SourceFile := "sub2.a", "sub2_b.a";
end P1.IMP;

with port type TARGET_T;
process P2 is
  TO_P1 : out port TARGET_T.INTEGER_T;
  FROM_P1 : in port TARGET_T.INTEGER_T;
end P2;

periodic process implementation P2.IMP is
  foo: subprogram;
attributes
  self'Period := 50 ms;
  self'SourceTime := 1 ms;
  self'SourceFile := "p2_root.a";
  foo'SourceTime := 10 ms;
  foo'SourceFile := "foo.a", "foo_b.a";
end P2.IMP;

with port type TARGET_T;
mode M is
  TO_ENV : out port TARGET_T.INTEGER_T;
end M;

mode implementation M.EXAMPLE is
  P1: periodic process P1.IMP;
  P2: periodic process P2.IMP;
connections
  P1.FROM_P2 <- P2.TO_P1;
  P2.FROM_P1 <- P1.TO_P2;
  TO_ENV <- P1.TO_P2;
end M.EXAMPLE;

application TSE is
  mode M.EXAMPLE
  on processor I80960MC.CVME;
connections
  I80960MC.DISCRETE_WRITE <- M.TO_ENV;
end TSE;
```

Figure 6: Example Textual MetaH Specification

the new mode. Mode declarations may be nested to create submodes, which allows some processes to continue unaffected during changes between submodes.

The MetaH language also supports hardware architecture specification. Memories, devices, channels (processor-to-processor communication hardware), and processors may be connected together to specify multi-processor hardware architectures. Hardware events, ports and monitors are supported and provide a mechanism for the application software to interface to target hardware in a flexible way. Attributes of the hardware entities support both automated code assembly (e.g. identify processor-specific modules that interface between the generated application and the underlying system) and analysis (e.g. processor overhead times). A simple automatic software-to-hardware binder has been implemented, where the user can optionally specify binding constraints or explicit bindings for particular software entities. The language and toolset can thus be used to support rapid analysis of alternative multi-processor architectures and alternative software-to-hardware bindings.

4.2 MetaH Tools

Figure 7 shows some existing and planned tools to support MetaH specification development, where shaded tools have not been developed or integrated yet (the workspace does not yet support persistent storage and so is considered only partially complete). The eventual plan is to have a suite of tools that can be invoked independently with a degree of concurrent access by multiple analyzers. Additional useful tools have been identified, but the figure shows core tools that emphasize the theme: automatic code assembly and formal analysis driven from the same high-level architectural specification.

4.2.1 Code Assembly

Application source modules identified in the MetaH specification are assembled into processes, together with an automatically generated KERNEL service call package that provides a target-

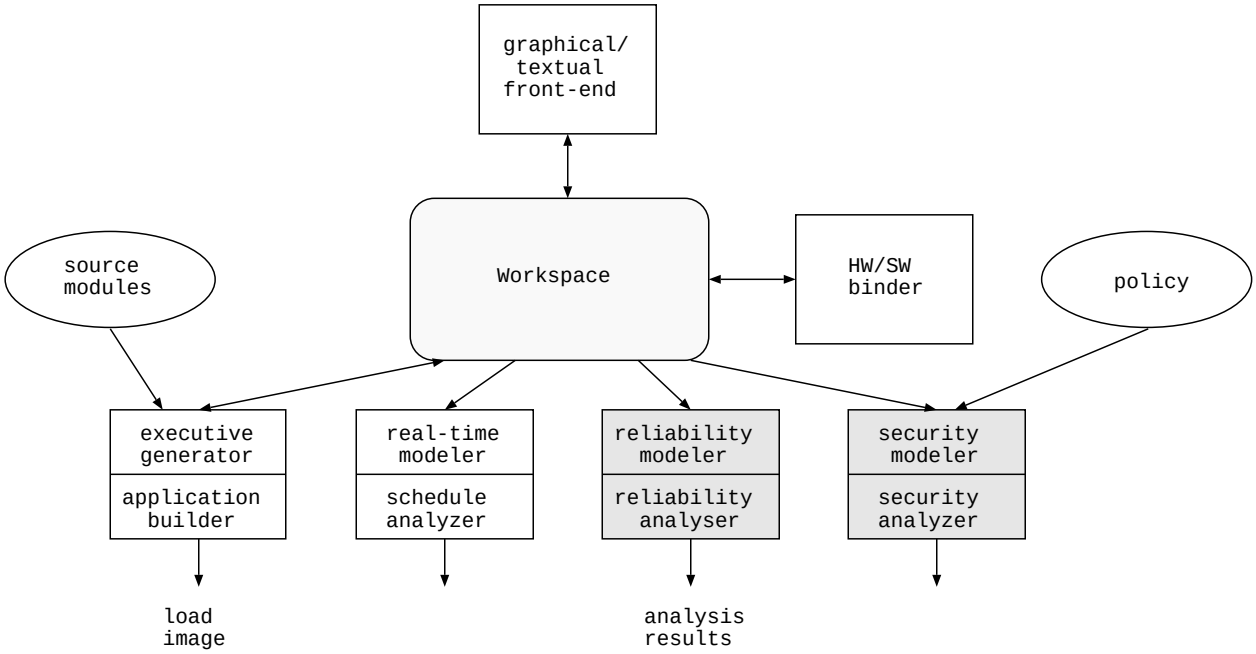

Figure 7: MetaH Tools, Existing and Planned

independent interface to kernel services. Each application process becomes a program that is compiled and prelinked into its own protected virtual address space. The current tool accommodates simple versioning techniques ("make" facilities) so that repeated MetaH compiles do not force source recompiles unless an application source module has been changed or added (a few hundred lines of MetaH can easily specify applications having many dozens of source modules and many tens of thousands of lines of source code).

The code assembly tool essentially generates an application-specific executive that performs the process dispatching, message passing, access synchronization, capabilities checking, and event vectoring specified in the MetaH specification[2]. Process scheduling is based on rate monotonic scheduling theory, where the tool automatically assigns priorities, transforms periods as necessary, and generates code to manage process dispatching and continuation[8]. Inter-process and inter-processor communication code is statically scheduled and executed as part of the generated dispatcher. Event vectoring, and the starting and stopping of processes at mode changes, are also controlled by generated code. All this must be done in a way that is mode-dependent, since scheduling and communication varies from mode to mode.

The generated executive makes use of a standard interface to certain low-level scheduling primitives typically provided by a microkernel. Our current implementation uses the Tartan Ada multi-application run-time as the underlying microkernel. An application builder has been developed for the Tartan Ada toolset targeted to an i80960MC processor. This build tool automatically performs the compilations and links needed to produce a final load image for this type of processor.

4.2.2 Schedulability Analysis

Figure 8 shows the timing report produced from the specification in Figure 6. Summary figures for each process are shown on a line (e.g. TSE_kernel, P1, P2), where the first process is the overhead associated with kernel dispatching. Indented beneath each process appears a list of that process' components, where _self is a component that represents source associated with the process entity itself and _kernel is the scheduler overhead at-

tributable to that process (e.g. context swaps of that process).

The Budget times are those specified or estimated for leaf source components and must be given in the MetaH specification (e.g. estimated by the ControlH translator). The Critical times show how much the compute times of all processes and their components can simultaneously be raised while preserving feasibility. The Maximum times show how much the compute time of an individual process or component can be raised while preserving feasibility, providing all other processes and components stay within their budgeted compute times. The % Margin is the safety margin between the budgeted and the maximum compute time, the % Util is the processor utilization devoted to that process or component, and Max Pri is the name of the highest priority process of which a component is a member (changing the compute time of a component will affect this and all lower priority processes).

We feel that allowing such a decomposition of processes into components, and the provision of sensitivity analysis information, is useful in practice[12, 13]. This supports an assessment of the risks associated with uncertainties in actual source module compute times and allows easy identification of bottlenecks or source modules on which to focus optimization efforts. For example, sub2 in Figure 8, with a relatively high utilization and low margin, represents a risk to the extent its compute time is uncertain, or a good candidate for further optimization. In conjunction with the intuitive traceability between ControlH operators and generated source modules mentioned earlier, the results of the timing analysis can be easily related to the GN&C engineer in a meaningful way. This supports complex, multidisciplinary time/space/performance/robustness design trade-offs.

4.2.3 Reliability Analysis

Traditional reliability analysis is used to determine the probabilities that a system will enter particular states within a particular time in response to randomly arriving hardware fault events, where the states of interest are degraded modes of operation, fail-safe, catastrophic failure, etc. [11]. A number of tools exist that, when given a model that describes the various fault events that may occur and a system's response to those events, will solve mathematically for the desired probabilities.

Several basic mechanisms to specify events and event propagation paths already exist in MetaH, and the addition of fault arrival rate attributes for hardware entities is trivial. The existing toolset should eventually be extended by the addition of a tool to build a reliability model from the description of fault event propagations contained in a MetaH specification, then solve this model using an existing Markov analysis tools.

4.2.4 Secure Partitioning Analysis

The current system provides three secure partitioning mechanisms at run-time. First, data security is provided by allocating each application process its own protected virtual address space. Processes can access common memory only if an equivalence connection "authorizes" this in the MetaH specification. Second, control security is provided using capability lists generated for each process. A process can only change to a specific mode, dispatch a specific aperiodic process, lock a specific monitor semaphore, or invoke a specific kernel service if this appears in the MetaH specification. Third, timing security is provided in the form of enforced execution time limits on process initialization, process computation, and monitor locking times. A command-line option causes enforced times to be used for timing analysis, where the results of this analysis are guaranteed to be enforced at run-time. Process criticalities can also be assigned, where the scheduling of a higher criticality process cannot be affected in any way by the behavior of a lower criticality process.

However, mechanism is not policy. A tool that allows a MetaH specification to be checked against a specified secure partitioning policy (e.g. red/black compartmentalization, integrated modular avionics partitioning[5]) would be of great assistance in developing and verifying applications that have such requirements. It should be noted that security is provided against undesirable be-

```
    Processor timing, application TSE, mode TSE, processor I80960MC

Module             Period    Budget  Critical   Maximum  % Margin  % Util  Max Pri

TSE_kernel          25000      1000      1397      8099      87.7     4.0
 _kernel                       1000      1397      8099      87.7     4.0  TSE_kernel

P1                  25000     11200     15642     18300      38.8    44.8
 _self                         1000      1397      8099      87.7     4.0  P1
 sub1                          2000      2793      9099      78.0     8.0  P1
 sub2                          8000     11173     15099      47.0    32.0  P1
 _kernel                        200       280      6400      96.9     0.8  P1

P2                  50000     11200     15642     25399      55.9    22.4
 _self                         1000      1397     15199      93.4     2.0  P2
 foo                          10000     13966     24199      58.7    20.0  P2
 _kernel                        200       280      7300      97.3     0.4  P2

Total utilization =  71.2%
Breakdown utilization =  99.4%
Critical scaling factor =  1.40
Processor schedule is feasible

Times are reported in microseconds
Nominal compute times were used in the compute paths.
```

Figure 8: Example Timing Analysis Report

havior of the source modules included in an application. The assurance of correctness of the MetaH specification (which includes the security specifications) and the use of the toolset for analysis and load image generation must be performed by a trusted party.

5 Integrating The Views

There are four concepts that appear in both the ControlH and MetaH languages. Each language deals with somewhat different semantic aspects of each concept.

Both languages include a concept of time. ControlH implicitly deals with time-varying signals and explicitly declares process periods. MetaH accepts scheduling requirements that must be met in the assembled application and can perform a schedulability analysis to determine whether these requirements will in fact be met in all cases.

A process entity appears in both languages. ControlH can be used to specify a name and a period for a process, and to specify how operators are partitioned among processes. MetaH allows a specification of a number of other properties, such as the processor on which that process is to execute, secure partitioning rights and capabilities, stack sizes, etc. (there are currently 17 MetaH process attributes that can be set by the user).

Many ControlH operators translate to a subprogram (very low-level or optimized operators may translate into in-line code, e.g. a scalar gain translates to a simple multiplication; and certain ControlH operators must be translated into a pair of subprograms to deal with feed-back loops). The subprograms are itemized in the MetaH description so that the MetaH user has some limited visibility into the composition of each process. This is useful during discussions between the GN&C and software engineers about possible time and space trade-offs. We noted earlier that structured translation and the resulting traceability aided multi-disciplinary design trade-off studies. The ease with which schedulability analysis can be related back to the original ControlH spec-

ification greatly facilitates the study of complex time/space/performance/robustness trade-offs.

State variables appear throughout a typical ControlH specification, though usually only explicitly at the lowest level of detail, inside predeclared operators (e.g. integrators). A ControlH state variable translates to a statically allocated variable in the generated code, which is identified in the MetaH specification of that source code as a MetaH port. MetaH ports are also used for input and output signals, both those between operators assigned to different processes and inputs and outputs to the overall GN&C algorithm. A MetaH port can be connected to other ports in other processes, and such connections are used to communicate signal samples and state updates. That is, rather than generating a central vector of all state and signal variables, the ControlH toolset produces distributed variable declarations that are connected in the generated MetaH specifications. The MetaH toolset generates the code to move values between these signal and state variables. Advantages of this approach are that the MetaH user can move software processes between various hardware processors without any changes to the GN&C specification or generated code; and visibility is provided to state and signal values during testing.

6 Summary

This article has focused on two integrated languages for capturing GN&C and computer system architectures, together with associated tools to analyze architectures and automate the assembly of the application software. Major elements of our approach have been to base our languages on suitable formal models, and to automate mutually-consistent formal analysis and automatic code assembly in such a way that each analysis tool accurately captures with high assurance the relevant characteristics of the final implementation. Our goal has been to support architectures that are easily parameterized and scalable, that can be analyzed very early in the development process, and for which it is possible to maintain assurance that the actual implementation characteristics are accurately predicted by analysis during system development.

We feel another interesting aspect of our work is its inherently multi-disciplinary nature, which for us manifested itself in two integrated languages and toolsets. We attempted in our languages and toolsets to allocate responsibility and authority for different design decisions to specialists in two different fields, where each has a language and toolset suited to the issues being addressed. Our goal was to do this in a way that allowed the GN&C engineer and the software engineer to solve the detailed problems of their respective disciplines in a largely independent way while supporting broad, multi-disciplinary design trade-offs. We feel we have been reasonably successful in this, primarily by codifying certain details of semantic and implementation consistency in the languages and tools themselves, and by maintaining a reasonably intuitive mapping between important common concepts and architectural elements in the two views.

A major issue we have not addressed in this article is the development of a reusable architecture for a specific product family. The architecture reuse approach is based on the development of standard designs or software architectures for particular product families (e.g. flight management systems, building security and climate control systems, petrochemical process control systems), where for us these architectures are captured as partial and reusable ControlH and MetaH specifications. This introduces such issues as architecture language features that facilitate generic specifications, techniques for systematically migrating such a partial specification into a final product description and implementation, the technologies that go into a particular product family, and the creative processes that go into the development and evolution of a good reusable generic architecture for a particular product family.

Acknowledgements

We gratefully acknowledge the contributions of Todd Beckering, Rashmi Bhatt, Jeff Clark, Eric Engstrom, Al Goldberger, Debra Hutchins,

Rakesh Jha, Jon Krueger, Carl Lippitt, Steve Paesano, Tom Peterson, Todd Sorensen, Jon Ward, and Kent Younkin to the development and implementation of the MetaH and ControlH toolsets; the helpful guidance and suggestions of Gunter Stein, Jim Krause, and other employees of the Honeywell Technology Center; and the conceptual contributions of the ARPA DSSA community.

References

[1] Pam Binns and Steve Vestal, "Formal Real-Time Architecture Specification and Analysis," *Tenth IEEE Workshop on Real-Time Operating Systems and Software*, New York NY, May 1993.

[2] Pam Binns and Steve Vestal, "Communication and Scheduling in MetaH," *Proceedings of the Real-Time Systems Symposium*, December 1993.

[3] Cellier, F. E., and H. Elmqvist (1993), "Automated Formula Manipulation Supports Object-Oriented Continuous-System Modelling," *IEEE Control Systems*, April, 1993.

[4] Matt Englehart and Mike Jackson, "ControlH: A Fourth Generation Language for Real-Time GN&C Applications", to have appeared *Proceedings of the CACSD*, Tucson, Az, March, 1994.

[5] *Design Guidance for Integrated Modular Avionics*, AEEC/ARINC 651, Airlines Electronic Engineering Committee/ Aeronautical Radio Inc., 1991.

[6] Mike Jackson and Jim Krause, *ControlH Programmer's Manual*, Honeywell Technology Center, Minneapolis, MN.

[7] Korn, G. A. (1989), *Interactive Dynamic-System Simulation*, McGraw-Hill, New York.

[8] Lui Sha and John B. Goodenough, "Real-Time Scheduling Theory and Ada," *IEEE Computer*, April 1990.

[9] Erik Mettala and Marc H. Graham, "The Domain Specific Software Architectures Program," *Proceedings of the DARPA Software Technology Conference 1992*, Los Angeles CA, April 1992.

[10] Mitchell, E. E. L. and J. S. Gauthier (1986), *ACSL: Advanced Continuous Simulation Language - User Guide and Reference Manual*, Mitchell & Gauthier Assoc., Concord, Mass.

[11] Andrew
L. Reibman and Malathi Veeraraghavan, "Reliability Modeling: An Overview for Systems Engineers," *IEEE Computer*, April 1991.

[12] Steve Vestal, "On the Accuracy of Predicting Rate Monotonic Scheduling Performance," *Tri-Ada '90*, December 1990.

[13] Steve Vestal, "Fixed Priority Sensitivity Analysis for Linear Compute Time Models," *IEEE Transactions on Software Engineering*, April 1994.

[14] Steve Vestal, "Mode Changes in a Real-Time Architecture Description Language," to have appeared *Second International Workshop on Configurable Distributed Systems*, March 1994.

[15] Steve Vestal, *Software Programmer's Manual for the Honeywell Aerospace Compiled Kernel (MetaH Language Reference Manual)*, Honeywell Technology Center, Minneapolis, MN.

[16] Steve Vestal, "A Cursory Overview and Comparison of Four Architecture Description Languages," Honeywell Technology Center, Minneapolis, MN.