Hardware/Software Co-design:
VHDL and Ada 95 Code Migration and Integrated Analysis

Mike Mills and Greg Peterson
Air Force Research Lab
Information Technology Division
Wright-Patterson AFB OH
(937)-255-6653 x3591

1. ABSTRACT

Optimizing the design of complex weapons systems constitutes an important task in efficiently developing and deploying effective complex weapons systems. Often architectural tradeoffs between hardware and software implementation must be performed early in the design cycle, resulting in potentially inefficient systems or subsystems. As technologies and costs in hardware and software implementation change over time, the best partitioning of system functionality into hardware and software components will also change. Currently, recasting a component from hardware to software (or vice-versa) is a difficult and error-prone activity. This paper explores a new approach to ease the hardware software co-design and repartitioning activities by providing a mechanism to exchange software written in Ada 95 with behavioral VHDL.

1-58113-033-3/98/0011...$5.00

2. INTRODUCTION

The electronics design industry enjoys dramatic improvements in circuit capabilities, with the number of transistors on a chip doubling every eighteen months as noted by Moore's Law [1]. This explosive growth represents tremendous potential for military and commercial electronics systems and components, while at the same time presenting designers with the seemingly insurmountable task of exploiting this growth effectively and economically. As a corollary to Moore's Law, the electronic systems design process demands an exponential increase in design productivity to keep pace with improvements in device density and speed.

When developing electronic systems, the designer typically develops models of the hardware components by starting with abstract behavioral VHSIC Hardware Description Language (VHDL) models of functionality and refining these models into register transfer level and eventually gate level models. Software design using high level languages such as Ada often follows a similar development path in parallel to the hardware development. This top-down design methodology is appealing due to its effectiveness in handling complexity, but often results in sub-optimal implementations due to the need to partition functionality into hardware and software too early in the design cycle. In addition, the hardware and software domains use different representations (languages) which are challenging to make inter-operate. Bringing the hardware and software components into a consistent framework could help provide an effective means of co-executing hardware models with software code.

Design methodologies and tools must account for these issues to ensure correct, efficient implementations of designs. Simply exploiting these improvements in designing and manufacturing electronics hardware solves only

half the problem: we need efficient and reliable methods to develop and maintain both the hardware and software. With advances in reconfigurable computing, the delineation between these two domains continues to blur. Effective language support, tools, and methodologies addressing these issues will help enable the future deployment of sophisticated systems and the affordable, effective maintenance of existing weapons systems.

As a result of research projects funded by the Air Force Research Lab and DARPA and commercial tool development projects, an emerging, extensible intermediate format for VHDL promises high performance and space efficiency for analyzed (compiled) VHDL models. The Advanced Intermediate Representation with Extensibility (AIRE) structures VHDL models in abstract syntax trees with additional information provided as the models are analyzed, elaborated, and simulated. Standardization of AIRE for VHDL, VHDL-AMS, and Verilog support continues, and a number of AIRE-compliant implementations near completion.

Abstract syntax trees serve a common role as a standard representation of software code during compilation. In order to develop a common AST representation for multiple software languages such as Ada, JOVIAL, and C++, the Arcadia group developed IRIS. By using IRIS as a common foundation, a number of software engineering projects have been able to exploit a common front end and support multiple languages. One Air Force Research Lab software maintenance project exploits this approach to port older JOVIAL code into Ada 95 [2].

This paper addresses efforts to provide an interface between VHDL models and Ada software via a common abstract syntax tree by exploiting AIRE and IRIS. We discuss this interface, language construct limitations, and the potential for solidifying the hardware/software interface. A number of very interesting applications exist for such an interface between VHDL hardware models and software written in languages such as Ada. Models in hardware can be ported to software for implementation, and vice-versa. In addition, some of the Arcadia software engineering tools built upon IRIS provide analysis capabilities that could prove quite useful for VHDL. The potential for greatly improving the state of the art in hardware modeling via these tools is discussed.

By exploiting state of the art software engineering tools and formats, a bridge between hardware models and software code can be constructed. This bridge has great potential for enhancing the systems design effort by supporting relatively effortless migration between the hardware and software domains in addition to enabling an integrated analysis tool suite.

In response to a perceived need for additional support of abstract system design and specification, a system-level design language development effort sponsored by the Industry Council continues to investigate needs in this area in preparation for extensions to VHDL or the possible development a new system description language [3].

Significant related research is in progress which could be exploited in developing the system level design language. There is a wealth of work going on with VHDL and hardware design, synthesis, and test. The same can be said of work for software engineering with Ada and with process improvements. Complex systems include both hardware and software components. Significant recent attention focuses on how to specify, design, and verify the hardware and software together. Current standardization efforts related to codesign focus on co-simulation approaches in the short term [4] and general codesign/systems design support in the long term. The goal of this effort is to bring together research results from both communities to improve systems design capabilities.

Long-term software engineering experience proves Ada to be effective for cost-effectively managing large program development and maintenance efforts, particularly when compared to languages such as C [5,6]. Ada is syntactically similar to VHDL, which makes it attractive as a basis for the software domain. Despite the end of the DoD Ada mandate, Ada remains a powerful language often chosen for weapons systems development [7]. Ada provides strong typing, encapsulation via the package mechanism, and supports generics among other features. Ada 95's object oriented programming, real time, and programming in the large features makes Ada highly competitive. Yet, opposition to Ada has hindered general adoption by industry. VHDL enjoys significant use in the design automation industry.

3. OVERVIEW OF VHDL

VHDL provides the best current capability for supporting the development and ongoing maintenance of large, complex digital systems. Developed by the DoD in the 1980s to support the documentation of electronic systems, VHDL derives much of its syntax and semantics from Ada. A major difference in the languages comes from the notion of concurrent processes that permeates VHDL. Because the language is intended to model hardware components which are "always executing," VHDL is a highly concurrent language built upon a simulation cycle-based timing model. Interactions between VHDL processes occur over signals and include data and temporal aspects. VHDL is a Turing complete language and capable of representing very low level device models on up to system models by employing appropriate abstraction. Data encapsulation is part of VHDL. Although developed initially for documentation and simulation, synthesis tools now transform more abstract VHDL models into detailed gate-level implementations, resulting in a dramatic productivity improvement.

Several past and current research and standardization efforts focus on ways to extend the capabilities of VHDL to better support systems design and, in particular, to provide an integrated mechanism for modeling, simulation, and synthesis of software components. Although VHDL has some characteristics which differentiate it from software programming languages, system and hardware design tools and methodologies can potentially benefit by exploiting practices in software engineering. One effort to transition software engineering standard practices into hardware description languages is the work focused on creating object-oriented VHDL extensions [8,9]. In essence, the proposal extends VHDL with object orientation using similar mechanisms to those of Ada 95.

Other extensions and enhancements to VHDL include support for formal methods. VSPEC [10] is a Larch interface language for VHDL that allows a designer to specify non-functional performance constraints. VSPEC extends to VHDL to allow a designer to declaratively describe the data transformation a digital system should perform and performance constraints the system must meet. The designer axiomatically specifies the transformation by defining predicates over entity ports and system state describing input precondition and output postconditions. A constraints section allows the user to specify timing, power, heat, clock speed and layout area constraints.

The VHDL-AMS (Analog Mixed Signal) extensions to VHDL provide a capability for modeling analog and mixed signal designs. The VHDL-AMS standard provides a single language in which both digital and analog circuitry effects can be modeled, thus giving us an integrated tool for considering analog effects for advanced digital design.

4. OVERVIEW OF AIRE

The Advanced Intermediate Representation with Extensibility/Common Environment (AIRE/CE) is currently being developed by a number of electronic design automation tool vendors and users and is being standardized through the Electronic Industries Association and the IEC. Before the design and implementation of AIRE/CE, integration of tool components from distinct applications to form an entire system was severely limited by proprietary, inflexible, low-functionality and low-performance interfaces. The cost and time required to integrate best-of-breed tool components generally lead to pragmatic compromises intended to reduce cost rather than increase application capability. Rising interest in the exchange of hardware and system components, often referred to as design re-use or intellectual property commerce, emphasizes the value of being able to transport design components from one usage to another.

AIRE/CE solves both the tool component integration and information exchange problems. AIRE/CE supports information exchange both within a single address space and via the host computer file system. The in-memory communication protocol, the *Internal Intermediate Representation* (IIR) facilitates integration of the most suitable tool components within a single operating system process. The File Intermediate Representation (FIR) provides for the integration of self-contained, AIRE/CE-compliant tool components and the exchange of partially compiled designs (intellectual property).

Efforts to build previous generations of VHDL tools on a standard intermediate format did not converge or achieve the support of multiple vendors due to competing commercial concerns and technical problems. Earlier approaches no longer meet the requirements of today's advanced tool developers.

AIRE's in-memory intermediate representation, the IIR, is defined in terms of a class hierarchy.

An IIR class forms the top level. Classes derived from IIR include:

- IIR_DesignFile,
- IIR_Literal,
- IIR_Tuple,
- IIR_TypeDefinition,
- IIR_Declaration,
- IIR_Name,
- IIR_Expression,
- IIR_SequentialStatement,
- IIR_ConcurrentStatement,
- IIR_SimultaneousStatement, and
- IIR_List.

Information associated with classes is accessible via class methods. Basic data types, introduced in the previous section, appear within the type signature of these methods, however the IIR does not define the organization of data within an IIR object (enabling many cost/performance tradeoffs).

Design file classes represent the information contained in a single HDL source file. Such information includes a list of comments, a list of design units / modules present in the design file, and information required for tools to cross-reference from the IIR representation back to the designer's original files.

Literal classes represent design elements such as text literals (strings), identifiers, integer literals, floating point literals, bit vector literals, and enumeration vector literals.

Tuple classes represent collections of design information with a small, fixed number of elements. Examples include associations between formals and actuals, the value and delay of a waveform element, or the association of a choice and action within a case statement.

Type definition classes represent a range of data values and associated operators. Reflecting a broad view of language design, types include not only "traditional" types such as integers and arrays, but also signatures, attribute types, and group types.

Declaration classes generally represent named instances of types. Examples include VHDL's variable declarations, constants, and signal interface items.

Name classes generally refer to implicit or explicit declarations. Examples include VHDL's selected names and attributes.

One or more extension classes may be introduced just before any derived class in order for applications to add data or functionality to the predefined IIR specification. In order to preserve binary tool compatibility, data may only be added within the extension classes immediately preceding terminal classes from which objects may be created.

While the current version of the file format is believed to represent the full VHDL language, the class hierarchy is not intended to map one to one with a language reference manual. Source code analysis and transformations readily map arbitrary source code into a canonical subset of VHDL in order to facilitate optimization or other uses.

5. OVERVIEW OF ARCADIA/IRIS

IRIS [11] is a language-independent abstract syntax tree representation of software programs. This representation is produced when a software engineer uses an analyzer (compiler) to transform the input high order language source code into the IRIS abstract syntax tree representation. The IRIS AST can then be transformed into any of a number of other internal representations, including control flow graphs, dataflow graphs, reachability graphs, and dependence graphs.

The Arcadia language processing tools can construct, manipulate, store, and retrieve IRIS graphs. Analysis can be performed on the control flow graph and the rendezvous call graph to analyze concurrent programs. The language processing toolset is built upon the PLEIADES object management system and provides a number of application program interfaces to support analysis tasks.

Some of the Arcadia tools built upon IRIS include the control flow graph (CFG) toolset, the task interaction graph (TIG) toolset, the TIG-based Petri Net toolset, and the rendezvous call graph toolset.

Interfaces into the IRIS abstract syntax tree representations exist or are under development for Ada, JOVIAL, and C++. Employing the mapping into IRIS combined with a mapping back into Ada, the Advanced Avionics Verificaion and Validation program [2] provides a capability to transform legacy JOVIAL code into Ada.

6. MAPPING VHDL TO Ada

In order to show how VHDL could be mapped to Ada, examples of Ada source code follow selected VHDL B&F descriptions. Signals are key objects in the VHDL language and provide a challenge for Ada to model. Therefore, an example shows how signal declarations and signal assignments could be represented by Ada. In order to model VHDL signal attributes, an Ada record, called a signal description record, keeps track of signal status. Separate records could represent waveform elements making up each signal.

B&F representation of a VHDL signal declaration:

VHDL signal_declaration ::=
signal identifier_list subtype_indication [signal_kind] [:= expression]

subtype indication ::=
[***resolution_function***_name] type_mark [**constant**]

signal kind ::= **register** | **bus**

register means that the target signal is not assigned a changed value until the next clock signal (or simulated cycle time occures). Therefore, the time (or duration) corresponding to the signal value should be incremented.

bus means that the target signal **remains unchanged** if there are no driver signals present on the bus (or they all have '0' value).

Suggestions for transforming to Ada code:

resolution_function_name could correspond to a resolution function written in Ada-95 but would have to be hand coded for each resolution function.

type_mark can be directly translated to an equivalent Ada type except for floating, physical, or file types which have no corresponding Ada equivalents. Physical type could be handled by converting units such as 80 ms to 80 * ms. Femto-seconds would have to be converted to 0.001 * ns. Ada types common to VHDL are access, enumeration, integer, array, and record

signal_kind - is implemented by a function to set signal kind in the data structure representing the signal.

expression - is an initial Ada value

B&F representation of a VHDL signal assignment:

Signal_assignment ::=
[label:] target <= [delay_mechanism] waveform;

Suggestions for transforming to Ada code:

target - is the name of the Ada variable representing the signal.

delay_mechanism - is implemented with a function call that adds a delay value to the variable representing time when the signal is activated. This is represented as a signal attribute by changing the corresponding value in the signal description record implemented by Ada.

waveform - is implemented with a series of subroutine calls to waveform_element to set the activation time, period of persistance, and signal value (implemented by an Ada variable).

waveform_element (t1, value, period);

These values are stored in the signal data type representing the signal. When the model is executed, the signal description record triggers the signal behavior during the Ada model simulation.

More B&F description of a signal:

target ::= name | aggregate
aggregate ::= [**choices** =>] expression

delay_mechanism ::= **transport** | [**reject** ***time***_expression] **inertial**

delay_mechanism - is implemented by a function call which adds a delay to the total signal. This delay can be a transport delay which adds a clock delay to the signal activation time stored in the signal description record.

Transport means delay of the first waveform is a transport delay from hardware devices with infinite frequency response.

Inertial means there is a built in delay (from switching circuits).

Reject means that a pulse shorter than the switching time or whose duration is shorter than the rejection limit will not be transmitted.

B&F for two kinds of signal assignment:

concurrent_signal_assignment ::=
[label:] [**postponed**]
conditional_signal_assignment
| [label:] [**postponed**]
selected_signal_assignment

postponed - is implemented by a subroutine call which adds a delay triggered by an event.

conditional_signal_assignment ::=
target <= options {waveform **when** conition **else**} waveform [**when** condition];

Ada implementation can use an "if" or a "case" statement:

if condition_1 **then** waveform_1;
elseif condition_2
then waveform_2;
else waveform_3;
end if;

case condition **is**
when condition_1 =>
waveform_1;
when condition_2 =>
waveform_2;
when others =>
waveform_3;
end case;

Selected_signal_assignment ::=
with expression **select** target <= options
{waveform **when** choices,}
waveform **when** choices;

Ada implementation is similar to the above:

case choices
…
end case;

options ::= [**guarded**] [delay_mechanism]

Ada implementation could use a protected type with a guard, or just represent the guard as a signal attribute in the signal description record.

waveform ::= waveform_element
{, waveform_element} | **unaffected**

Each waveform_element represents a signal driver. A driver assigns a value to the target at a specified time. When a driver is turned off, it has no contribution to the target value.

unaffected is only for concurrent signals (or agregate of guarded signals)

Evaluation of waveform elements produces transactions that determine future behavior of drivers for a target. The result is a sequence of transactions in ascending order with respect to time.

waveform_element ::= value_expression [**after** time_expression] | **null** [**after** time_expression]

interface_signal_declaration ::=
[**signal**] identifier_list : [mode]
subtype_indication [**bus**] [:= static_expression];

interface_signal_declaration - is used in signal parameters of function or procedure calls or ports (or generics) of hardware structures called entities.

mode ::= **in** | **out** | **inout** | **buffer** | **linkage**

mode - has the same in, out, and in out as Ada. Modes. Buffer and linkage are handled in Ada by setting these attributes in the signal description record.

An Ada-95 representation of a waveform element could look like the following:

For digital signals (which encompasses all of VHDL signals), the only values a waveform at a given instant of time are either true or false (or bit '1' or '0'). Therefore, the Ada implementation for modeling a VHDL waveform could include a set of boolean drivers opearated on by logic **or**'s.

Ada_waveform ::= driver (expression_1, time_1) or driver (expression_2,time_2) or … or driver (expression_n, time_n);

function driver (expression, time) **return** value **is**
begin
…
end driver;

The above drivers correspond to functions that are active for given simulated time periods. The start and stop time (or time duration) that a driver is active could be passed as a parameter in a function.

An Ada 95 implementation could treat time as a global variable and increment it periodically at every tick.

```
Time, T : variable;
R := S when T >= Current_Time + Tick;  -- or something similar
```

Signals declared within a parameter list exist in the form of an interface signal declaration:

Representing VHDL Signal Attributes in Ada:

VHDL signals could be represented in Ada using record types that contain the signal value (in Boolean) at the represented duration of time and the signal attributes (which are mostly Boolean and could be implemented in bit, thus minimizing storage requirements).

Some Ada types to represent VHDL language defined types:

```
type Boolean is (False, True);
type Bit_vector is array (Integer range <>) of Boolean;

-- VHDL mode as an Ada enumeration type:
type mode is (in, out, inout, buffer, linkage);
```

Signal description block implemented as an Ada record to store VHDL signal attributes plus any other values to help model VHDL in Ada:

```
type signal is
    record
        value: Boolean;  -- for digital VHDL
        (integer for Analog VHDL)
        start_time: Time;
        end_time: Time;
        -- attributes:  (8 Boolean, 1 Bit, 2 Time
        types)
        delayed, stable, quiet, transaction,
        event, active, last_value, driving :
        Boolean;
        Last_event : Time;
        -- added for Ada modeling:
        io : mode;
        -- other variables added for model:
        …
    end record;
```

7. VHDL SIMULATION MODEL

Ada-95 contains the following data types from language defined packages:

```
package System is  -- (includes the following):
  pragma Preelaborate(System);
  Tick : constant := implementation-defined;
    -- (real time interval that Calendar.clock
remains constant.)

package Ada.Calendar is – (includes the
following):
  type Time is private;
  function Clock return time;
  function Seconds (Date : Time) return
Day_Duration;
```

VHDL model execution includes a kernel process that accomplishes the following:

1. Causes prapagation of signal values,
2. Updates values of implicit signals (S'stable(T)),
3. Detects events, and
4. Causes processes to execute in response to events.

The Kernel process contains a:

1. Variable representing current signal value and updates variable based on source values.
2. Variable representing current value of any implicitly declared guard signal (resulting from a guard expression of a block statement.
3. Driver and variable to represent the current value of any signal (S'Stable (T) and Time T, S'Quiet (T), and S'Transaction).

The VHDL Simulation Cycle:

Model execution:

Initialization phase: Tc = 0; -- current time

Compute **driving** and **effective** value (or **current** value of signal).

Execute process statements repetitively:
Simulation Cycle (each repetition)
compute all signal **values**
Result:
Event on a signal
Resume process statements that are **sensitive** to the **signal**.
Execute as part of the simulation cycle.

Initialize:
1. At Tc = 0 (Current time)

2. Compute **driving** value and **effective** value of signals.
(Set **current** value to effective value.)

3. Set value of implicit signals
S'Stable (T) = True;
S'Quiet (T) = True;
S'Delayed (T) = initial value of signal S.

4. Implicit guard signal = expression

5. Execute each **non**-postponed and postponed process until it suspends.

6. Tn = Time of next simulation cycle

Simulation Cycle Steps:

1. Tc = Tn (Simulation complete when Tn = TIME'HIGH and no active **drivers** or process resumption at Tn)

2. Update each explicit and implicit signal (events on signals may occur).

3. Each process P **resumes** if **event** occurs on signal that P is **sensitive** to.

4. Execute each non-postponed process (that has **resumed** in current cycle until it **suspends**).

5. Tn (time of next cycle) = earliest of (TIME'HIGH, time **driver** becomes active, or next time a **process** resumes).

6. Execute any postponed processes that resume.

VHDL Process is a concurrent statement made up of one or more sequential statements. A process is **passive** if it contains a signal assignment statement. A process is executed repetitively.

```
process ::=
  [label:] [postponed] process
[(sensitivity_list)] [is]
        declarations  -- such as variables
  begin
        statements
                -- wait on sensitivity list
  end block [label];
```

Signal assignment in a **process** statement defines a set of drivers for certain scalar signals.

A driver of a scalar signal is represented by a projected output waveform.
A driver consist of trasactions ordered with respect to time. "Transaction (value, time)"
A driver's initial value is the default value of a signal.
Past values of a driver get deleted as time progresses.

Previous efforts focused on transforming high level languages such as Ada and C into VHDL [12]. These efforts restricted the programming style of the software engineers to preclude constructs such as pointers and dynamic memory usage. The mapping from Ada to VHDL is not the primary focus of this effort, but will be revisited in more detail later.

8. ANALYZING VHDL

Several existing validation and verification techniques can be used to reduce the effort involved in maintaining Ada or VHDL. Automated dependence analysis [13] traces the interactions between parts of a program and indicates the impact that modifications of a section of code can have on the rest of a program. Dependence analysis is a static approximation of the control and data interactions throughout a program [14]. Dependence analysis can detect potential errors introduced by migrating software into hardware or vice-versa.

Coverage analysis automatically traces the parts of a program exercised by test cases during testing. Coverage analysis can indicate insufficient test sets and unexpected model behaviors, e.g., errors with control signals in state machines. Although several VHDL coverage tools exist on the market, this capability with IRIS/AIRE may enable coverage analysis of the integrated hardware and software *system.*

We are currently developing a prototype implementation of the AIRE/IRIS integrated analysis to provide insight into the potential for this approach. We plan to explore interfacing with the Arcadia analysis tools and VHDL analysis and synthesis tools

9. CONCLUSIONS

With the increasing complexity of electronic systems, better integration of the design and

analysis for hardware and software is needed. We discussed the potential for interfacing Ada and VHDL by employing the IRIS/Arcadia toolset with VHDL tools based on the AIRE intermediate representation. The productivity improvements from this integrated analysis promise dramatically reduced integration and test time and cost.

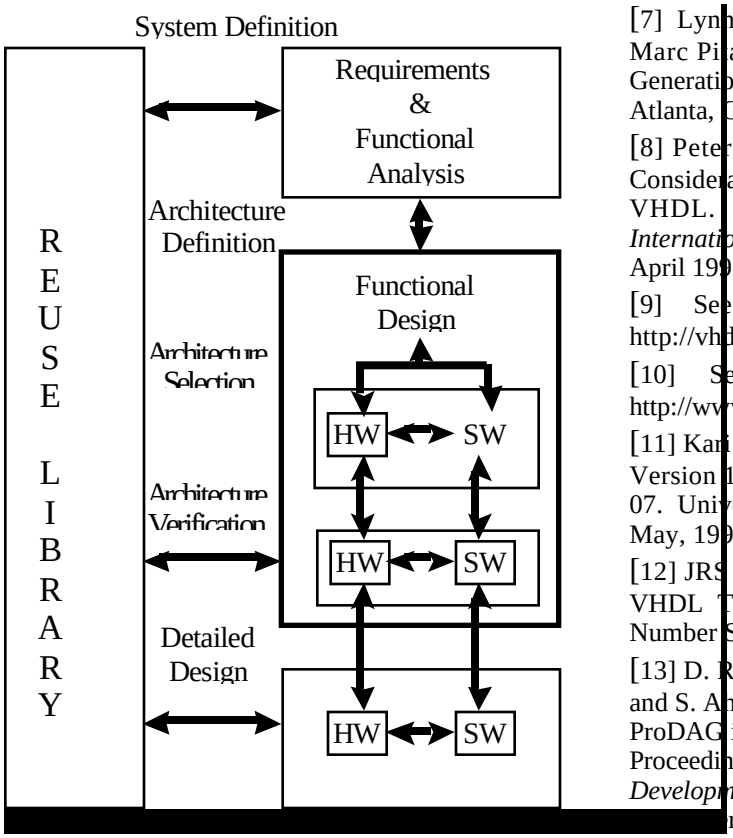

Figure 1: The RASSP design process

[1] Gordon Moore. VLSI: Some Fundamental Challenges. *IEEE Spectrum*, 16(4):30-37, April 1979.

[2] Joseph P. Loyall, Susan A. Mathisen, and Charles P. Satterthwaite. Impact Analysis and Change Management for Avionics Software. In Proceedings of the 1997 National Aerospace Electronics Conference, pages 740-747, July 1997.

[3] System Level Design Workshop, October, 1996.

[4] M. Mills and G. Peterson. Requirements and Concepts for Hardware/Software Codesign. In *Proceedings of the Spring VHDL International Users' Forum*, pages 129-138, April 1997.

[5] Stephen Zeigler. Comparing Development Costs of C and Ada. See http://sw-eng.falls-church.va.us/ AdaIC/docs/reports/cada/cada_art.html.

[6] P.K. Lawlis and T.W. Elam. Ada Outperforms Assembly: A Case Study. In *Proceedings of TRI-Ada*, 1992.

[7] Lynne Hamilton-Jones, Kenneth Littlejohn, Marc Pitarys. Software Technology for Next-Generation Strike Fighter Avionics. *DASC*. Atlanta, GA 27-31 October 1997.

[8] Peter J. Ashenden and Philip A. Wilsey, Considerations on Object-Oriented Extensions to VHDL. In *Proceedings of the Spring VHDL International Users' Forum*, pages 109-118, April 1997.

[9] See the OO-VHDL web page at http://vhdl.org/oo vhdl

[10] See the VSPEC web page at http://www.ececs.uc.edu/~kbse/vspec

[11] Kari Forester. IRIS-Ada Reference Manual Version 1.0. Arcadia Technical Report UM-90-07. University of Massachusetts, Amherst, MA. May, 1990.

[12] JRS Research Laboratories, Inc. Ada/C to VHDL Translator User's Manual. Document Number SUM.0187-01. October 1995.

[13] D. Richardson, T. O'Malley, C. Moore, and S. Aha. Developing and Integrating ProDAG into the Arcadia Environment. In Proceedings of *Fifth Symposium on Software Development Environments*, McLean VA, December 1992.

[14] A. Podgurski and L. Clarke. A Formal Model of Program Dependencies and Its Implication for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9), September 1990, pages 165-979.