AFRL-IF-WP-TR-1998-1527

STANDARD ANALYZER OF VHDL APPLICATIONS FOR NEXT GENERATION TECHNOLOGY (SAVANT)

HERBERT L. HIRSCH, PRAVEEN CHAWLA,
DALE E. MARTIN, PHILIP A. WILSEY,
MICHAEL W. SHELLHAUSE

MTL SYSTEMS, INC.
3481 DAYTON-XENIA ROAD
BEAVERCREEK, OH 45432-2796

JULY 1998

FINAL REPORT FOR 04/25/1995 - 06/30/1998

THIS IS A SMALL BUSINESS INNOVATION RESEARCH (SBIR) PHASE II REPORT

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7334

Al Scarpelli

AL SCARPELLI, Electronics Engineer
Embedded Information Sys Eng Branch
Information Technology Division

[illegible]

JAMES S. WILLIAMSON, Chief
Embedded Information Sys Eng Branch
Information Technology Division

Eugene C. Blackburn

EUGENE C. BLACKBURN, Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(leave blank)*	2. REPORT DATE July 1998	3. REPORT TYPE AND DATES COVERED Final Report, 04/25/1995 – 06/30/1998

4. TITLE AND SUBTITLE Standard Analyzer of VHDL Applications for Next Generation Technology (SAVANT)	5. FUNDING NUMBERS C F33615-95-C-1638 PE 65502 PR 3005 TA 06 WU 82
6. AUTHOR(S) Herbert L. Hirsch, Praveen Chawla, Dale E. Martin, Philip A. Wilsey, Michael W. Shellhause	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) MTL Systems, Inc. 3481 Dayton-Xenia Road Beavercreek, OH 45432-2796	8. PERFORMING ORGANIZATION REPORT NUMBER MFR-98-004/CSC349
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Information Directorate Air Force Research Laboratory Air Force Materiel Command Wright-Patterson AFB OH 45433-7334 POC: Al Scarpelli, AFRL/IFTA (937-255-7698, ext 3603	10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-WP-TR-1998-1527

11. SUPPLEMENTARY NOTES

THIS IS A SMALL BUSINESS INNOVATION RESEARCH (SBIR) PHASE II REPORT

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; Distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

The problem this Phase II SBIR effort addressed was the absence of an established, standard intermediate form (IF) for the exchange of VHDL-encoded electronic data among CAD systems. This absence has severely constrained basic resrearch environments, and has precipitated the current sub-optimal nature of CAD-in-VHDL tool development. Our Baseline Program developed this standard intermediate form (IF), as well as a VHDL design environment utilizing it, which is provided to the research community at no cost, and to commercial developers or users under a licensing fee. The SAVANT environment consists of an Analyzer (implementing the IF), a Code Generator, and a System Support Environment (SSE) containing debugger and visualization/analysis tools (called the VHDLyzer Toolkit or VTK). An option task implemented Object-Oriented Extensions to VHDL in the SAVANT environment, and assessed their utility and effectiveness.

The Phase II baseline objectives focused upon developing and validating the SAVANT system (analyzer, code, generator, debugger, visualization tools), and commercializing the resultant technology. The option task objectives addressed implementation of O-O VHDL costructs in SAVANT, and subsequent model selection, implementation and assessment of O-O VHDL performance in the SAVANT environment. All baseline and option task objectives were achieved, except for commercialization, which was partially achieved.

14. SUBJECT TERMS VHSIC Hardware Description Language (VHDL) Advanced Intermediate Form for Extensibility (AIFE) Electronic Design Automation (EDA) SAVANT Intermediate Form (IF) VHDL Analyzer VHDL Simulator Object-Oriented (O-O)	15. NUMBER OF PAGES 74
	16. PRICE CODE

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

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-102

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF FIGURES

LIST OF TABLES

1.0 EXECUTIVE SUMMARY

This Phase II SBIR effort addressed a **problem** defined as the **absence of an established, standard intermediate form for the exchange of VHDL-encoded electronic data among CAD systems**. The result of this absence was a severely-constrained basic research environment and sub-optimal nature of CAD-in-VHDL tool development. Our **Baseline Program** sought to **provide this standard intermediate form (IF), as well as a VHDL design environment utilizing it**, which could be provided to the research community at no cost, and to commercial developers or users under a licensing fee. The **SAVANT environment** consisted of an **Analyzer** (implementing the **IF**), a **Code Generator**, and a System Support Environment (SSE) containing a **debugger** and **visualization/analysis tools** (called the **VHDLyzer Toolkit**, or VTK). An **Option Task**, also reported herein, sought to **implement Object-Oriented (O-O) Extensions** to VHDL in the SAVANT environment, and to **assess their utility and effectiveness** for the designer.

The **Phase I program**, preceding the effort reported herein, **developed certain critical SAVANT concepts** to the point of **demonstrating their feasibility** to overcome the problem. In Phase I we **implemented and demonstrated a preliminary version of the analyzer and IF, and produced a preliminary design** to carry into Phase II.

Our Phase II **Baseline Objectives** focused upon **developing and validating the SAVANT system** (analyzer, code generator, debugger, visualization tools), and **commercializing** the resultant technology. The **Option Task Objectives** required **implementation of O-O VHDL constructs in SAVANT**, and subsequent **model selection, implementation and assessment of O-O VHDL performance** in the SAVANT environment. Our performance against these objectives, which this report elaborates in detail, is summarized as follows:

Develop Analyzer and (IF):	Achieved, software is being released on the WWW
Develop Code Generator:	Achieved, software is being released on the WWW
Validate the above:	Achieved, high conformance to accepted test suites
Develop the Debugger:	Achieved, not extensively tested
Develop Visualization Tools:	Achieved, not extensively tested
Commercialize SAVANT:	*Partially* achieved, diverges from MTL business model
Implement O-O Extensions:	Achieved, considered a prototype version

Implement O-O Models:	Achieved, results discussed
Assess O-O Performance:	Achieved, results discussed

Summarily, we regard the Phase II SAVANT Program as **successful in all aspects except commercialization**. The remainder of the report supports these assertions.

2.0 INTRODUCTION

In this introductory section, we provide certain background information which will be useful to the reader in perusing the remainder of the report. Our purpose here is to provide sufficient background information, in a concise and succinct manner, that the reader will have a good understanding of the background without having to review other documentation. We begin with a review of the original problem to be solved (Section 2.1), as stated in the Phase I proposal, and explain how that original problem relates to the problem we sought to solve in Phase II. Next, we describe our Phase I results (Section 2.2) in the context of how, by achieving certain Phase I Objectives, we made progress toward solving the problem addressed in Phase II. Then, we outline the Objectives (Section 2.3) we set for this Phase II program, and include the goals for an Option Task which was added while the Baseline Phase II program was in progress. We conclude with a discussion of our technical (Section 2.4) and management (Section 2.5) approaches to the Baseline and Option Task efforts.

2.1 The Problem

In this section, we review the original problem assertions from our Phase I program, and identify their specific relationships to the Phase II effort being reported herein.

The problem identified in the Phase I proposal was the absence of an established, standard intermediate form for the exchange of VHDL-encoded electronic data among CAD systems. The result of this absence was, and is, apparent within the presently-constrained basic research environment and sub-optimal nature of CAD-in-VHDL tool development.

Consider how VHDL is applied in such a tool development. VHDL presents a standard format for human comprehension or encoding of digital system designs. Analyzing and processing VHDL source code is quite difficult and requires considerable effort. Furthermore, a broad range of complex CAD tools are generally available to support the computer system design process, and each distinct CAD tool must input design data encoded in VHDL. Presently, most CAD tool vendors must execute a cumbersome process to realize a CAD-in-VHDL product. Typically, they:

1. Design an in-house intermediate form (IF).
2. Build a VHDL analyzer that validates the (static) correctness of the input VHDL, producing an IF representation of the input.
3. Input the IF to each in-house-developed CAD tool.

In other words, processing VHDL as the source input language places additional, unnecessary burden upon the construction of such CAD tools. The result is a tightly-coupled, non-standard analyzer and IF, within a particular, vendor (application)-specific environment.

A resulting problem is the current proliferation of several of these vendor-proprietary, non-standard IFs and analyzers. In rare instances, a vendor may sell (at high cost) the IF and analyzer to a third party. Unfortunately, no vendors are currently willing to standardize (and fully productize) their IF. Each vendor maintains an internal IF and markets both tools that use the IF and the VHDL analyzers to produce the IF. Consequently, either users are forced to use the CAD tools from one vendor or to purchase several VHDL analyzers, one from each CAD tool vendor whose design tools are being used.

This lack of a widely-available, standard IF for VHDL also inhibits basic research. Before embarking on a research investigation on CAD with VHDL, researchers must either design their own particular IF and then build an analyzer to translate VHDL to the IF, or they must purchase a vendor-supplied VHDL analyzer/intermediate form (A/IF). The former approach is expensive in time and effort, and generally results in an inferior VHDL analyzer/CAD system that operates only over a limited VHDL subset. The latter approach is subject to the nature of the chosen A/IF. It consequently suffers from high cost and the research project is vulnerable to changes in the IF produced by the vendor to support their internal tool development. Furthermore, because the IF is generally not a primary product for the vendor, accompanying documentation and support tools are generally of poor quality.

SAVANT was proposed to directly mitigate these problems. Its significance was expected to be that of a community-wide improvement in tool compatibility, as well as a significant enhancement to the overall effectiveness of basic CAD-in-VHDL research and development. We considered SAVANT to consist of two principal components: the IF and the analyzer; and a supporting component, the record/ playback tool for archiving the IF in file form. The standard IF would provide a common internal representation that vendors could follow and to which users could request adherence. Furthermore, the availability of a public domain analyzer and library subsystem would dramatically promote additional research and development in CAD and its integration with VHDL. Finally, source code availability would also enable and promote integration and cross-coupling between other design language efforts. For example, concurrently with the SAVANT program an MTL contract was underway, sponsored by AFRL Rome Site, to develop a standard Analog/Mixed-signal and mixed-technology hardware description language (VHDL-AMS) simulator. This VHDL-AMS simulator effort extended Scram (SAVANT's VHDL analyzer) to support the additional features of analog description and promotes a rapid integration of VHDL-AMS technology with VHDL technology among users.

The problem originally asserted for the Phase I program, the lack of an established, standard intermediate form for the exchange of VHDL-encoded electronic data, remained as the Phase II problem statement. However, in the Phase I program, we made significant progress by validating SAVANT's ability to solve the stated problem. In the next section, we describe the progress we made in Phase I.

2.2 The Phase I Results

In Phase I, generally speaking, we developed certain critical SAVANT concepts, as described in our Phase I proposal,[1] to the point of demonstrating their feasibility to overcome the problem cited in Section 2.1. More specifically, we satisfied particular Phase I Objectives and made progress toward solving the problem, as follows:

Phase I Objective 1: Establish the technical feasibility of the SAVANT technology as a standard A/IF exchange medium for CAD in VHDL. Here we produced quantified technical investigation results which confirmed that the innovations represented by SAVANT could be constructed in today's technology. We established the technical feasibility of the SAVANT technology by creating a preliminary definition of the IF and an analyzer prototype. The prototype was used to successfully parse over 1400 VHDL test files. In addition, the IF was defined to be object-oriented, thereby allowing easy integration with other CAD tools, and was planned to be well-documented, for easy comprehension.

To elaborate, we explored several aspects of this objective in Phase I. Most significantly, we reviewed different aspects of object-oriented representations and decided that an object-oriented design would be most suitable for the IF. This approach also had been followed by several others in their designs, however, our approach differed significantly in that our design allowed for an extensible class definition within the object-oriented representation. Thus, the CAD researcher could augment the class hierarchy with additional data and methods for problem-specific needs. Therefore, decoration of the IF with additional data/information/functionality would be well supported and furthermore, the CAD researcher would directly benefit from the fact that the IF is object-oriented (and self-defining). That is, the CAD researcher benefits from all aspects of an object-oriented representation such as inheritance, polymorphism, and encapsulation. Lastly, we also discovered an implementation technique for C++ that would fully support this design abstraction. Preliminary demonstration of this functionality was successfully accomplished during Phase I.

For the analyzer portion of this objective, we located and copied all of the public-domain parser generators announced in the monthly posting of the comp.compilers bulletin board. The parser generators were all examined for their suitability in a VHDL analyzer. In addition to reviewing the capabilities of each of the parser generators, an analysis of the available grammars for VHDL was conducted. We found that the Purdue Compiler Construction Tool Set (PCCTS) provided an excellent support environment for the SAVANT analyzer development effort. PCCTS supports an extended BNF (EBNF) notation and inputs LL(k) grammars. Inherited and synthesized attributes, parser exception handling, token classes, and lexical classes are all supported by PCCTS. The software is in the public domain and runs on a variety of platforms including SUN, DEC, SGI, VAX, HP, Linux, NetBSD, MSDOS, and OS/2. Furthermore, the VHDL grammar input to PCCTS was originally developed at UC and was the only available public-domain grammar that supported the VHDL 1993 standard (VHDL '93).

Phase I Objective 2: Establish community acceptance of the SAVANT technology and Define the Commercial Product. Here we focused upon what was required of SAVANT to ensure community endorsement of, and desire for, the SAVANT technology. We also defined what portion of the SAVANT technology could be effectively transitioned into a commercial product. We determined that the SAVANT Analyzer (Scram) and IF definition should be freely and easily made available (at no charge, through the World Wide Web (WWW)) to anyone who wanted it. In addition, we decided to provide a robust simulator based on SAVANT technology, also at no charge through the WWW. Users of the SAVANT Analyzer/IF/Simulator would then be allowed to create derivative products. In addition, we would allow distribution of derivative work for non-commercial purposes. However, for-profit distribution/support/rent/lease of SAVANT-based technology would be allowed only upon completion of a profit-sharing agreement between MTL and the distributor.

By using a liberal licensing and distribution scheme, such as the one described above, we expected to stimulate research in the VHDL community and to continually extend SAVANT's utility to the community. Stated simply, our strategy was to establish community acceptance by proliferating the SAVANT technology and by encouraging development of its extensions to realize a viable product. MTL would also profit by providing products that enhanced the utility of the basic SAVANT technology. Such products would include an interactive and fast simulation environment, a man-machine interface for SAVANT, and derivative products developed by third-party vendors. MTL would also be able to profit from a pay-per-use service based on SAVANT.

In addition, we began the process of creating SAVANT awareness. MTL representatives had discussed possible utilization of SAVANT with several EDA vendors, such as Exemplar, Synopsys, Intergraph, Mentor Graphics and Intermetrics, at the Fall 1996 VIUF conference. EDA vendors were receptive to our ideas and expressed interest in obtaining copies of the software and documentation when it became available.

Phase I Objective 3: Produce valid preliminary design concepts for the two principal elements of SAVANT: the IF and the Analyzer. Here we developed a solid foundation for Phase II development. Its achievement also supported the feasibility and commercialization aspects by showing the beginning of a clear path to development and subsequently to productization and proliferation of SAVANT within the community. Specifically, we produced preliminary analyzer and IF designs. As previously mentioned, we chose PCCTS for the construction of Scram primarily since it allowed the creation of an LL(k) parser. Then, we employed PCCTS to produce a preliminary analyzer which correctly parsed over 1400 test files. Although the Phase I-level analyzer performed no semantics testing, it did build parts of an initial version of the IF. In addition, the IF design was object-oriented, which made it easily extensible. This would allow easy integration of SAVANT with other CAD tools. Furthermore, we chose texinfo as the format for documentation of the IF. Texinfo can be translated into info format for on-line viewing, dvi format for hardcopy output, and HTML for WWW access. This easy conversion into various forms ensured easy comprehension of the SAVANT IF.

To elaborate, the basic elements of a VHDL '93 analyzer had been constructed and many of the initial class definitions for the IF had also been constructed. In particular, we had a full VHDL '93 grammar that input to PCCTS (see statement on Phase I Objective 1). The resulting parser correctly parsed over 1400 test files, however, it had no semantics testing. In addition, many classes for the object-oriented representation had been constructed. The classes were organized into three components, namely: base nodes, CAD tool nodes, and leaf nodes. The base nodes contained all the data and method definitions for the standard IF definition. The leaf nodes were dummy classes that were used by the parser to create IF objects. The leaf nodes were derived from the CAD nodes and allowed for user-added constructor/destructor invocations as well as enabling a search up the derivation tree for the correct implementation of virtual functions. The CAD tool nodes were organized into parts. The first part contained pure virtual functions that served as base classes from which the common base nodes for all of the standard IF nodes were derived (thus making the virtual functions visible). The second part was the actual implementations of the functions for each node in the standard IF. Thus, for example, a code generator CAD tool defined a pure virtual function *cgen* from which the base standard IF node was derived. The remaining standard IF nodes had a derived class containing an implementation for *cgen*. Lastly, the leaf classes were modified to derive from the classes for *cgen* and the desired extensibility was accomplished.

This functionality was completely demonstrated (but not fully implemented) in the Phase I effort. In addition, we actually showed the implementation of two functionalities. First, we implemented a *transmute* method that rewrote the IF nodes for concurrent statements into an IF node (and descendants) for the equivalent process statement. Second, we implemented a *publish_vhdl* method that outputs VHDL. Furthermore, we added a *publish_vhdl* method derived from the concurrent statement IF node that automatically caused an invocation of the *transmute* function. Thus, *publish_vhdl* need not be defined for the nodes derived from concurrent statement and an automatic translation to a process statement IF node was invoked. The *publish_vhdl* method could then operate only on a subset of VHDL but actually achieve the desired capability across the entirety of VHDL.

In summary, by achieving these Phase I Objectives, we made considerable progress toward solving the problems asserted in Section 2.1. Through accomplishing Objective 1, we established the feasibility of the SAVANT technology to fulfill the need for an established, standard intermediate form for the exchange of VHDL-encoded electronic data among CAD systems. By achieving Objective 2, we demonstrated how the SAVANT technology could be accepted and proliferated among the EDA community, thus ultimately abating the proliferation of vendor-proprietary, non-standard IFs and analyzers. Finally, by producing the preliminary designs demanded by Objective 3, we showed how our SAVANT concepts could be captured into a design which would support, rather than inhibit, basic research in VHDL/CAD tool technology. Our success in these Phase I endeavors then led us to define our Phase II Objectives, as we present in the next section. Additional details regarding the Phase I accomplishments and results may be found in the Phase I Final Report.[2]

2.3 The Phase II Baseline Program and Option Task Objectives

Our Phase II Objectives were designed to ensure the focus of the effort upon the problem described in Section 2.1. In the context of completing the problem solution begun under Phase I, the SBIR program demands from a Phase II effort the realization of a prototype system to serve as a proof-of-concept platform, and a specific plan for commercialization of the technology, either directly from the Phase II effort or as a result of subsequent Phase III actions. In consideration of these issues–the problem, the need for a demonstrable prototype, and the need for a resulting commercial product–we defined seven specific objectives for the Baseline Phase II program. In addition, an Option Task for integrating VHDL Object-Oriented Extensions into SAVANT was defined and subsequently activated.

2.3.1. Baseline Phase II Objectives - We first outline our Baseline Phase II Objectives, and then describe additional objectives defined for the Option Task. The objectives of the Baseline Phase II SAVANT project, and their relationship to the problem, were as follows:

Objective 1. Establish the Preliminary Standard Intermediate Form (IF). This activity was to act upon the preliminary IF design accomplished in Phase I, and to provide the basis for standardization and final refinement. Upon standardization, it would satisfy the need for a standard intermediate form for the exchange of VHDL-encoded electronic data among CAD tools. In addition, it would abate proliferation of proprietary, non-standard IFs and analyzers, as discussed in Section 2.1. The presence of a standard IF would allow the construction of CAD tools that could interface with a standard analyzer format to facilitate the insertion of research technology into the commercial sector.

Objective 2. Demonstrate the capability of object-oriented techniques for building an extensible intermediate form. This demonstration was to establish proof of extensibility, with benefits to both proliferation and ease of use. Extensibility of the intermediate form would allow easy integration of the SAVANT Analyzer/Intermediate Form (A/IF) with other CAD tools. In addition, it would promote reuse of already-existing software, since its object-orientedness would allow use of inheritance and polymorphism. Therefore, extensibility would stimulate further research in the design automation area due to the ease of CAD tool creation and integration.

Objective 3. Develop the software capability to support construction of the intermediate form, with the affiliated support routines needed to enable the SAVANT technology for CAD researchers. Here we were to produce the means for potential users to integrate SAVANT technology into their particular environments. This would further promote the proliferation of SAVANT in place of non-standard IFs. Since such software could be re-utilized by every CAD tool developer who used the SAVANT IF, free and easy access to such software would promote the use of SAVANT, thereby helping to establish it as a standard.

Objective 4. Develop robust public domain software that provides a simulation environment based on SAVANT technology. A simulator based on SAVANT, which would also be freely and easily accessible, would allow us to provide the users of SAVANT with an end-to-end electronic design simulation capability. We intended to provide such capability by interfacing SAVANT with the Quick Execution of Simulation, Synthesis, and Test (QUEST) simulator (from the University of Cincinnati's ongoing DARPA/QUEST project). A free and easily-available SAVANT-based simulator would further promote SAVANT use and again help establish it as a standard.

Objective 5. Promote Standardization of SAVANT. After we had created a clear definition of the SAVANT IF and produced robust software that allowed easy use of this IF, we were to encourage standardization of the IF through the IEEE Design Automation Standards Committee (DASC). We expected that the SAVANT IF and associated software would be available to the user community and, we hoped, in active use by several users, before any DASC subcommittee reactivation initiatives would be started. Existence of the SAVANT IF as an IEEE standard would establish it as a standard intermediate form for exchange of VHDL-encoded electronic data among CAD systems.

Objective 6. Produce the Prototype SAVANT System. Here we were to aggregate the elements of SAVANT into a deliverable package. Such a system would integrate the SAVANT IF definition and documentation with SAVANT-based software, namely the Analyzer, the Record/ Playback Tools, and the Simulator. The integrated SAVANT system would then be made available to anyone and everyone through the WWW. Easy access to a well-integrated system at a minimal cost would promote the use of SAVANT technology and research into its possible extensions. The prototype SAVANT software system was planned to include the following components:

1. *Scram:* Translating VHDL source programs into the intermediate form.
2. *Transmute:* Manipulating the intermediate form and rewriting nodes from one form to another. In particular, the rewriting of concurrent statements into their equivalent process statement definition.
3. *Publisher:* Output routines that generate VHDL (publish-vhdl) and C++ (publish-cpp) representations of the intermediate form.
4. *Archive:* Library manager functions that load and store the intermediate form. Initially these functions would rely on VHDL as the intermediate form and invoke Scram and publish-vhdl to read/write the library files.

The above-described SAVANT software would be made freely available to the user community through the WWW. The software would be free to all for non-commercial use and not under export control. Commercial use of the SAVANT software would require licensing through MTL Systems, Inc. In addition to the public-domain software, the following commercializable software was to be developed under the SAVANT Phase II program:

1. *Debugger:* Basic debugging utilities provided through a command-line interface. The debugger would be derived from the object-oriented QUEST simulation kernel and code generator.
2. *Interactive Simulator:* An easy-to-use, man-machine interface for interactive simulation and animation. The simulator was to be based upon the QUEST simulation kernel, the QUEST code generator, and the SAVANT debugger.

Objective 7. Commercialize the SAVANT System. Here we were to accomplish the proliferation of the SAVANT technology, as we have discussed in the preceding sections, and to establish the plan to realize the near- and long-term software enhancements which were to become the accompanying commercial products. Commercialization of the SAVANT system would essentially be done by selling these software enhancements to provide added functionality, performance and ease-of-use. In addition, profits/royalty from third-party tool developers providing extensions to SAVANT would make SAVANT immediately commercially viable, even without the MTL enhancements.

2.3.2. Option Task Overview and Objectives - For our baseline Phase II SAVANT Program, we asserted our problem as the lack of an established, standard intermediate form (IF) for the exchange of VHDL-encoded electronic data among CAD developers. Within this problem there was another, very important issue–Object-Oriented (O-O) design. As the practice and benefits of O-O continued to proliferate, we expected that more and more CAD tools would be aligned with this paradigm. Realistically, we could envision an almost exclusively object-oriented design world in the not-too-distant future. Hence it was not merely convenient, but imperative, that evolving CAD tool designs should support object-oriented design practices. The integration of VHDL-O extensions into SAVANT was deemed critical to its success, its acceptance within the design community it is intended to serve, and to the general improvement in effectiveness of VHDL-in-CAD tools we wished to achieve through SAVANT.

Our Option Task, whose objectives we describe herein, was designed to provide these needed O-O extensions, in a manner synergistic with the Phase II SAVANT development program. In general, the objectives of the proposed VHDL-O extensions Option Task related to technical accomplishment as well as to community acceptance. By performing this Option Task, we planned to achieve the following objectives:

Option Task Objective 1 - Demonstrate that O-O extensions to VHDL do not have to be expensive in VHDL analysis and simulation performance–that they can be both performance and cost-effective.

Option Task Objective 2 - Build a system to allow exploration of O-O VHDL for the 1998 VHDL standardization effort. This would allow the implementation of O-O constructs in the next standardization cycle.

Option Task Objective 3 - Provide validation of the utility of O-O extensions for hardware design and description. This would not only allow O-O benefits (reusability, modularity, etc.) in hardware design, but would also promote more cohesion between software and hardware designers as a whole.

Option Task Objective 4 - Leverage the public-domain nature of SAVANT to provide a VHDL-O test platform to the VHDL community for the early exploration of the proposed extensions for the 1998 standardization effort. This would provide the platform when the community is ready for it.

Option Task Objective 5 - Set the stage for early release of a 1998 O-O-conforming simulation environment, and possibly allow joint development of the 1998 language standard and IF definition. This would bring together the standardization and implementation elements as a cohesive package for the community.

These objectives, for both the Phase II Baseline Program and for the Option Task, were deemed achievable. Next, we describe the technical approaches and the program plan by which we sought to accomplish them.

2.4 The Technical Approaches

Here, we overview our Phase II Technical Approaches for both the Baseline Program and the Option Task. These discussions outline the application of the relevant technologies, leaving the programmatic aspects of tasks and management for the next section. The reader requiring additional detail regarding our technical approach may wish to review the Phase II Proposal[3] for this effort.

2.4.1 Baseline Program Technical Approach - Here, we explicitly detail the technical methods we planned to apply to produce the results, to achieve the objectives and to clearly show an advancement in research appropriate for this Phase II effort.

The key technical contribution of the SAVANT Phase II Program was seen to be the establishment of a standard intermediate form of digital systems for machine-processable CAD tools. In general, the intermediate form representation of a digital system design can be input from a variety of sources (textual languages, graphical languages, etc.); however, the primary source for this effort would be the DoD standard hardware description language, VHDL. Thus, in addition to designing and documenting the intermediate form, SAVANT would include a VHDL-to-intermediate form translator. The intermediate form would be an in-memory tree data structure. Consequently, SAVANT would also require some mechanism for off-line archival and retrieval of digital system designs represented in the intermediate form. Finally, the SAVANT project must address the problem of technology insertion; how would the industrial, governmental, and academic communities be encouraged to use the SAVANT technology? These issues are more fully addressed in the ensuing discussions.

The Standard Intermediate Form - The importance of and need for a standard intermediate form was discussed in Section 1. Briefly, a standard intermediate form is important and would serve the design automation community by providing a unifying, easy-to-process representation of electronic designs. That is, instead of analyzing, verifying correctness, and manipulating VHDL source code, CAD tools would be able to interface with a machine-generated intermediate form of the design which has already been analyzed for static semantic correctness.

The design of an intermediate needed to preserve as much semantic content from the original source input as possible. However, it was not necessary to retain an ability to exactly reproduce the source input. That is, for example, comments, new lines, and spaces are not language constructs with semantic content (while semantic constructs can be added as comments, such as "*cf VAL/VHDL*," the VHDL language does not formally relate semantic content with comments). Thus, some information from the original source input could be discarded.

While the intermediate form would not preserve all information from the original source input, it needed to allow for the augmentation of the design data by CAD tools. More precisely, a CAD tool may need to mark components of the intermediate form with additional information for later use (by the same or by other CAD tools). For example, a simulation code generator might need to decorate the intermediate form with code templates for later phases in the code generation process. Thus, the intermediate form needed to be *extensible* for the inclusion of additional CAD tool-synthesized information.

As a result of these needs, the intermediate form was to be designed as an object-oriented data structure with each node in the tree derived from a common base object. The intermediate form would be an extensible definition that is capable of adding additional data members and methods to each node in the intermediate form. This derivation structure, shown in Figure 1 for illustrative purposes and to help simplify the example for discussion purposes only, should be considered only a partial definition of the final derivation tree. The actual design has considerably more intermediate class definitions.

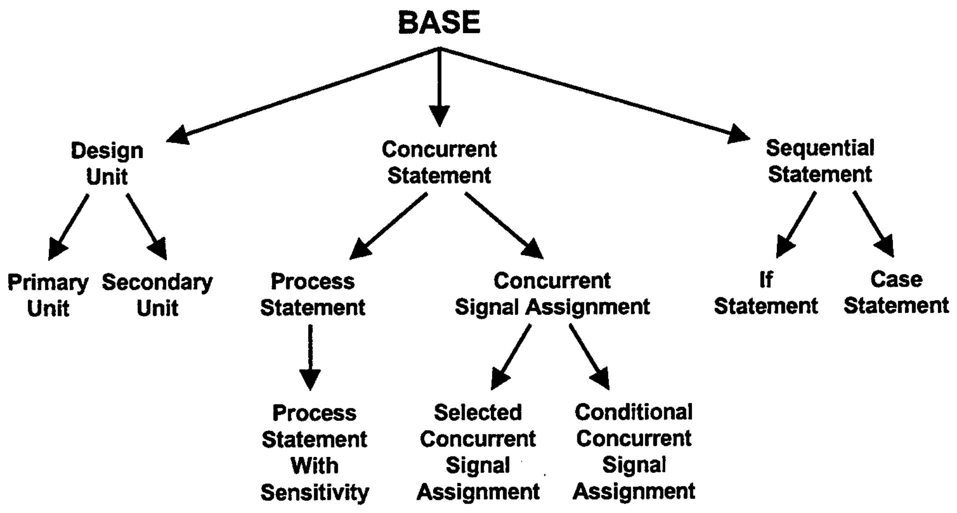

BASE
Design Unit
Concurrent Statement
Sequential Statement
Primary Unit
Secondary Unit
Process Statement
Concurrent Signal Assignment
If Statement
Case Statement
Process Statement With Sensitivity
Selected Concurrent Signal Assignment
Conditional Concurrent Signal Assignment

Figure 1. Illustrating the Derivation of Objects in the Intermediate Form.

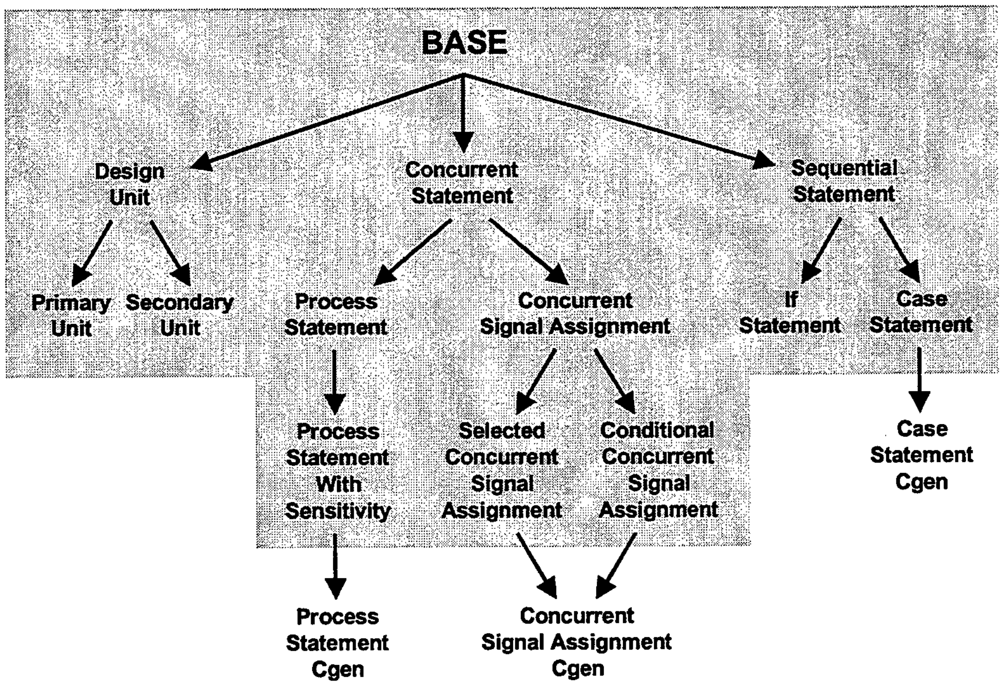

BASE
Design Unit
Concurrent Statement
Sequential Statement
Primary Unit
Secondary Unit
Process Statement
Concurrent Signal Assignment
If Statement
Case Statement
Process Statement With Sensitivity
Selected Concurrent Signal Assignment
Conditional Concurrent Signal Assignment
Case Statement Cgen
Process Statement Cgen
Concurrent Signal Assignment Cgen

Figure 2. Extending the Basic Intermediate Form.

An example of how the basic intermediate form was to be extensible is shown in Figure 2. In this figure, the nodes inside the shaded areas are the base intermediate form definition. The nodes outside the shaded areas illustrate what might be used for a simple code generator (cgen).

Figures 1 and 2 show the logical organization of the desired intermediate form. The software implementation accompanying SAVANT that builds the intermediate form would be written in C++ and would require some additional structure to achieve the desirable functionality. In particular, the implementation would follow a structure as shown in Figure 3.

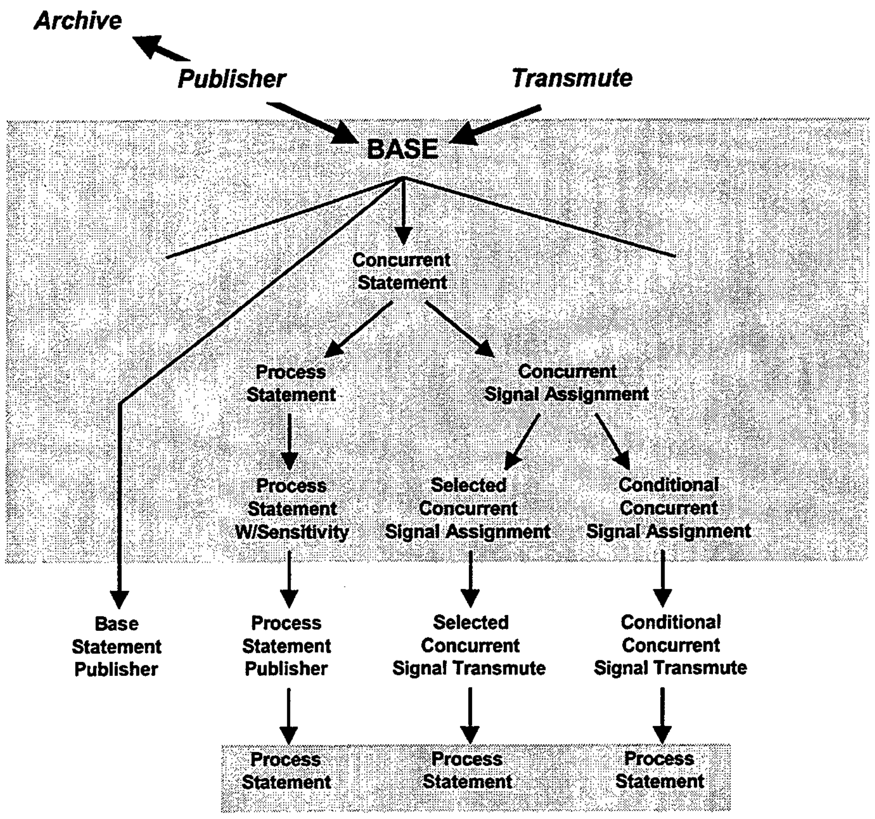

Archive
Publisher
Transmute
BASE
Concurrent Statement
Process Statement
Concurrent Signal Assignment
Process Statement W/Sensitivity
Selected Concurrent Signal Assignment
Conditional Concurrent Signal Assignment
Base Statement Publisher
Process Statement Publisher
Selected Concurrent Signal Transmute
Conditional Concurrent Signal Transmute
Process Statement
Process Statement
Process Statement

Figure 3. Implementation and Delivery Class Structure for the SAVANT Project.

In Figure 3, the basic intermediate form is captured by the nodes shown in the shaded areas. Other nodes such as those needed for research CAD tools are shown outside of the shaded areas. Four important observations need to be made about this figure.

1. The base node of the intermediate form class derivation tree is actually derived from base nodes for each of the research CAD tools.
2. In instantiating new nodes for the intermediate form, only those nodes shown in the shaded area at the bottom of the structure are to be created. This is enforced by having a single procedure called create_node defined in the base class that actually performs all node creation.
3. The leaf nodes of the intermediate form must be maintained as the research CAD classes and are added to the intermediate form. This is necessary so that constructors/destructors are invoked and methods/data of the intermediate classes become known.
4. Intermediate nodes in the intermediate form may also have classes derived by the research CAD tools. The reason for this is explained below (see the discussion of the publisher/transmute classes to be included with the initial SAVANT software release).

Planned Implementations and the Deliverable IF - As we previously described in Section 2, the initial SAVANT software release was planned to include the following components and capabilities:

Scram:	Translating VHDL source programs into the intermediate form.
Transmute:	Manipulating the intermediate form and rewriting nodes from one form to another. In particular, the rewriting of concurrent statements into their equivalent process statement definitions.
Publisher:	Output routines that generate VHDL publish_vhdl and C++ publish_cpp representations of the intermediate form.
Archive:	Library manager functions that load and store the intermediate form. Initially these functions would rely on VHDL as the intermediate form and invoke scram and publish_vhdl to read/write the library files.
Debugger:	Basic debugging utilities provided through a command line interface. The debugger would be derived from the object-oriented QUEST simulation kernel and code generator.
Interactive user I/F:	Easy-to-use, man-machine interface for interactive simulation and animation. The simulator would be based upon the QUEST simulation kernel, the QUEST code generator and the SAVANT debugger. (See Section 4 for details about the QUEST project).

The first four software objects would be publicly available and would be developed by our subcontractor, the University of Cincinnati, who retained the copyright. The last two software objects (for debugging and interactive simulation) would be commercial software, developed by and the property of MTL Systems, Inc. In the following discussions, we describe how these software objects which comprise the SAVANT system, were expected to perform their critical functions.

Scram: Translation from VHDL to the Intermediate Form - As previously mentioned, the construction of a VHDL analyzer is a complex problem. In fact, this problem is sufficiently complex that it prevents many research investigations from reaching full integration with VHDL. Even the problem of merely forming a machine-processable set of grammar productions for VHDL is quite difficult. Despite much interest and many queries, little progress had been made toward the construction of a public domain VHDL parser at the inception of this project. The chief problem is that the grammar given in the language reference manual is written primarily for human consumption and does not easily translate to a machine-processable form. Most attempts at building a VHDL parser fail because most available compiler-compiler toolsets produce parsers with only one token look-ahead and an LL(1) or LR(1) grammar for VHDL is difficult to construct.

In this project, we planned to use the PCCTS compiler construction toolkit to build the VHDL analyzer. PCCTS generates LL(k) parsers and was selected over other tools, such as YACC/LEX or Cocktail, because (i) it is LL(k), (ii) it supports predictive parsing, (iii) it readily supports attribute transmission, and (iv) it builds a parser compatible with C++. Because VHDL designs can easily grow quite large and because LL parsers tend to be slightly faster and more compact, we felt that PCCTS was an excellent choice for SAVANT's VHDL analyzer. Furthermore, the PCCTS developers were actively engaged in extending the tool suite and were incorporating extensive error recovery capabilities into the PCCTS. Finally, PCCTS generates ANSI C that is processable by g++. Consequently, we planned to build the parser actions in C++.

Initial development of a VHDL analyzer had been conducted under Phase I of this SBIR program. The analyzer was called Scram and the PCCTS grammar productions and token definitions for VHDL were already complete. The resulting parser was LL(2); it correctly parsed over 1400 test files. The version of Scram exiting at the Phase I conclusion performed no semantic testing but did build parts of an initial version of the intermediate form. A preliminary symbol table class had been developed that included both hash and scope class definitions.

Transmute: Rewriting the Intermediate Form - The transmute base class defines a single virtual method called transmogrify, that would be written for several of the objects in the intermediate form. The reason for having this method is that many VHDL objects have equivalent forms as VHDL process statements. Thus, instead of building tools that operate across the entire VHDL language, transmute would allow the research CAD tool developer to work with a smaller subset of VHDL. These methods would be used by the publisher class (next section) to reduce the complexity of the output generation task.

Publisher: Output Routines for the Intermediate Form **-** Output generation for the SAVANT project would be implemented by the publisher class definitions. In particular, two virtual methods (*publish_vhdl* and *publish_cpp*) would be defined in the publisher class. By default, this class would also automatically use the methods from *transmute* whenever a publisher method was not written for the intermediate form node. This ability is achieved by having publisher methods derived from the intermediate form base class definition that we call the transmogrify method. This was illustrated earlier in Figure 3 by the Base Statement Publisher.

The reader should note that *transmute()* does not run with *publish_vhdl()*. It is, however, used with *publish_cc()*. The problem we ran into was caused when *publish_vhdl()* used *transmute()*. The basic issue was that every time one invokes the analyzer, the VHDL is analyzed and then stored into a library. Since the library item was written using *publish_vhdl(),* the IIR tree for the model would be modified by the *transmute()* function. Thus, any later work (e.g., synthesis) over the IIR tree would see the modified tree instead of the original. Consequently, the post-library write phase would not see any concurrent statements other than process statements (since everything else was transmuted to the process statement). In some cases the backend tools needed to still see the original tree. Therefore, we modified the *publish_vhdl()* functions to work without the *transmute()* method (basically this meant implementing *publish_vhdl()* for all of the nodes in the IIR tree).

Archive: Archiving the Intermediate Form **-** The VHDL analyzer, Scram, would translate VHDL into the intermediate form. The intermediate form would be a memory resident data structure (tree) that must be archived into some library format for later use. That is, VHDL design units (design entities, packages, etc.) must be analyzable and storable into a design library for later use by other VHDL design units. Therefore, SAVANT would also include two additional functions called RECORD and PLAYBACK to archive the intermediate form. RECORD would save the intermediate form representation of a VHDL design unit into the design library and PLAYBACK would read a design unit from the design libraries into the intermediate form. For simplicity, initial implementations for RECORD/PLAYBACK would simply use VHDL as the library format using *publish_vhdl* to output the VHDL and Scram to input the VHDL.

Debugger: A Debugger for the Simulator - The debugger would provide basic debugging utilities through a command line interface. More specifically, at least the following functionality would be provided: (i) start or stop simulation at a specified simulation time, (ii) examine signal and variable values, and (iii) set breakpoints on specified statements. The debugger would be derived from the object-oriented QUEST simulation kernel and code generator.

Interactive User I/F: An Interactive User Interface to the Simulator - An easy-to-use, man-machine interface for interactive simulation and animation would be defined and implemented. Such interface would be based upon the QUEST simulation kernel, the QUEST code generator and the SAVANT debugger. The interface would be X-based and implemented using a cross-platform development tool suite such as XVT. However, we would evaluate other available tools during the initial phases of the program to determine a suitable development environment. Our objective was to develop an interface that would be portable across workstations and personal computer environments. The interface would provide all the functionality of the debugger through an easy-to-use man-machine interface. In addition, it would provide animation capabilities based upon process graph representations of VHDL.

In summary of our baseline technical approach, it was designed to produce the appropriate results and deliverables to achieve the program objectives. As described, these approaches were founded upon sound engineering principles and proven technologies from the Phase I program, and other associated projects (such as QUEST). Next, we describe our approach to the Option Task, designed to assess O-O extensions to VHDL.

2.4.2 Option Task Technical Approach - As VHDL continued to evolve, it was clear that many of the emerging concepts in programming languages would be considered for inclusion in the VHDL language standard. That is, as new constructs are incorporated into contemporary programming languages, it is only natural to explore the inclusion of the same (or similar) constructs into VHDL. Currently, one of the most rapidly emerging concepts in new programming languages is the notion of the language being object-oriented. While the specific constructs used to make a specific language object-oriented vary widely, the basic notion of designing and using object-oriented programming languages was gaining wide acceptance. In direct response to the widespread interest in object-oriented programming languages, several investigators were exploring object-oriented extensions for VHDL.

Despite the widespread interest in object-oriented technologies and programming languages, there remained considerable skepticism regarding the utility and cost of object-oriented computer languages. For example, in some cases object-oriented extensions to programming languages have resulted in serious compromises to compile-time and run-time performance (such as in Ada, C, and C++). Consequently, some argue that object-oriented extensions are a burden and should not be incorporated into (new or) existing languages. However, these costs can be mitigated by careful analysis of the cost/benefit ratio of a particular set of language extensions. Furthermore, the costs are also reduced as compiler technology advances (consider the early performance of Ada compilers from the mid 80's vs the performance of Ada compilers released today).

This option for the SAVANT program was to explore and to attempt to quantify the compile-time and run-time costs of object-oriented extensions to a VHDL analyzer and simulator. In particular, we planned to implement most (if not all) of the object-oriented extensions to VHDL that the IEEE DASC Object-Oriented Study Group was to submit in preparation for the 1998 language (re)standardization effort. The extensions were to be incorporated into the SAVANT software suite. Each extension would be carefully profiled in an attempt to quantify its analysis, elaboration, and run-time costs. Furthermore, the SAVANT IF would also be extended to accommodate the object-oriented extensions. The successful completion of this option would help support arguments in favor of object-oriented extensions whenever the costs are (or can be made) acceptable. Furthermore, the extended IF and public domain analyzer would allow the rapid integration of the 1998 language standard into the CAD community.

2.4.3 Technical Approach Summary - We have outlined our technical approaches to the Baseline SAVANT program, as well as the Option Task for O-O VHDL assessment. Next, we describe the approach we implemented to manage the SAVANT program.

2.5 The Project Organization and Management Approach

Our planned and actual organization of the project tasking included the Baseline Phase II program and the additional option which was exercised. The original program was focused upon developing, testing, and distributing the SAVANT system, while the option concentrated upon implementing and testing Object-Oriented (O-O) extensions to SAVANT. In the course of the program, additional subcontracting assistance was required, which MTL obtained from EDAptive Computing, Inc. In the remainder of this section, we describe the evolution from the planned to the actual execution of the program, as well as the roles each organization played in the activities.

2.5.1 The Original Program Project Plan - Figure 4 illustrates our original project plan for the SAVANT Baseline Program, as well as the manner in which we actually executed it. Although both MTL and UC participated to some degree in all tasks, those that were primarily MTL's or UC's responsibility are indicated in the Task numbers (Task x-UC or Task y-MTL). We originally planned that UC would develop the Analyzer and Code Generator (Tasks 1a, 2a). The Analyzer Development would include development of a standard Intermediate Form (IF), and the Code Generator Development would include the use or adaptation of an existing simulation kernel (TyVIS and WARPED). MTL would then add code improvements, toward the goal of producing a more "commercial-grade" level of code than was expected from the university. MTL would also provide some supporting tools in a System Support Environment (or "SSE"), and would test and distribute the entire package. All the development tasks within the plan were expected to incur several cyclic iterations of specification, design, implementation, and evaluation, in a more-or-less typical "spiral" development paradigm.

As it turned out, this organization of tasks, if executed as planned, would have introduced severe inefficiency into the whole process. Hence, we altered our approach.

First of all, we executed the code improvement tasks (1b and 2b) by providing recommendations from MTL back to UC throughout the course of the program. MTL would receive some interim version, would test it, would discover ways to improve it, and would then send these recommendations to the UC team. In this manner, the number of different versions at different organizations was minimized and the core development work remained at a single point–UC.

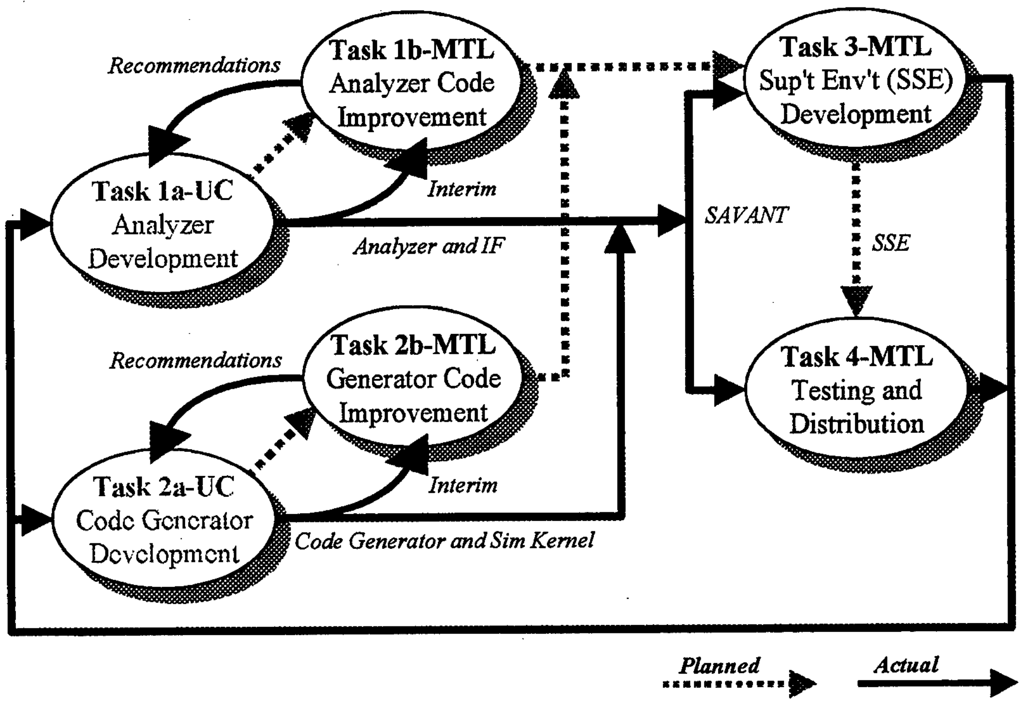

Task 1b-MTL Analyzer Code Improvement
Recommendations
Task 1a-UC Analyzer Development
Interim
Analyzer and IF
Task 3-MTL Sup't Env't (SSE) Development
SAVANT
SSE
Task 2b-MTL Generator Code Improvement
Recommendations
Task 2a-UC Code Generator Development
Interim
Code Generator and Sim Kernel
Task 4-MTL Testing and Distribution
Planned
Actual

Figure 4. The Original SAVANT Project Plan.

Second, MTL performed the Support Environment Development (Task 3), consisting of a VHDLyzer Toolkit and Debugger (which we describe in detail later in Section 3), concurrently with the Analyzer and the Code Generator development. MTL developed these components, with subcontract assistance from EDAptive, based upon interim releases of the Analyzer and Code Generator, and upgraded them as deemed appropriate throughout the project, to pace the changes in the Analyzer and the Code Generator. Finally, MTL and UC shared the testing and distribution activities (Task 4). UC tested interim versions as they developed, as part of the development process. MTL performed additional testing, and provided feedback to UC. MTL and UC released interim versions through coordinated WWW pages throughout the program.

2.5.2 Option Task Project Plan - Under the Option Task, MTL, UC, and EDAptive all performed research and development aimed toward assessing the value of object-oriented (O-O) extensions to VHDL. The SAVANT components provided the experimental platform for the assessment, as illustrated in Figure 5. Under this option, UC conducted a survey of the planned O-O extensions to VHDL (known as VHDL-O), both to understand the nature of the planned extensions and to obtain VHDL user community opinions regarding them (Task Z). Based upon the survey results, UC then modified certain components of SAVANT to produce a test-implemensurvey results, UC then modified certain components of SAVANT to produce a test-implementation of the extensions under the SAVANT framework. (Tasks A-D). EDAptive, in coordination with MTL, developed O-O models by which the performance of the extensions could be evaluated (Task E), and then executed these models within the extended SAVANT component framework to quantify the performance of the O-O extensions (Task F). The assessment of the O-O extensions concentrated more upon usability than upon performance issues.

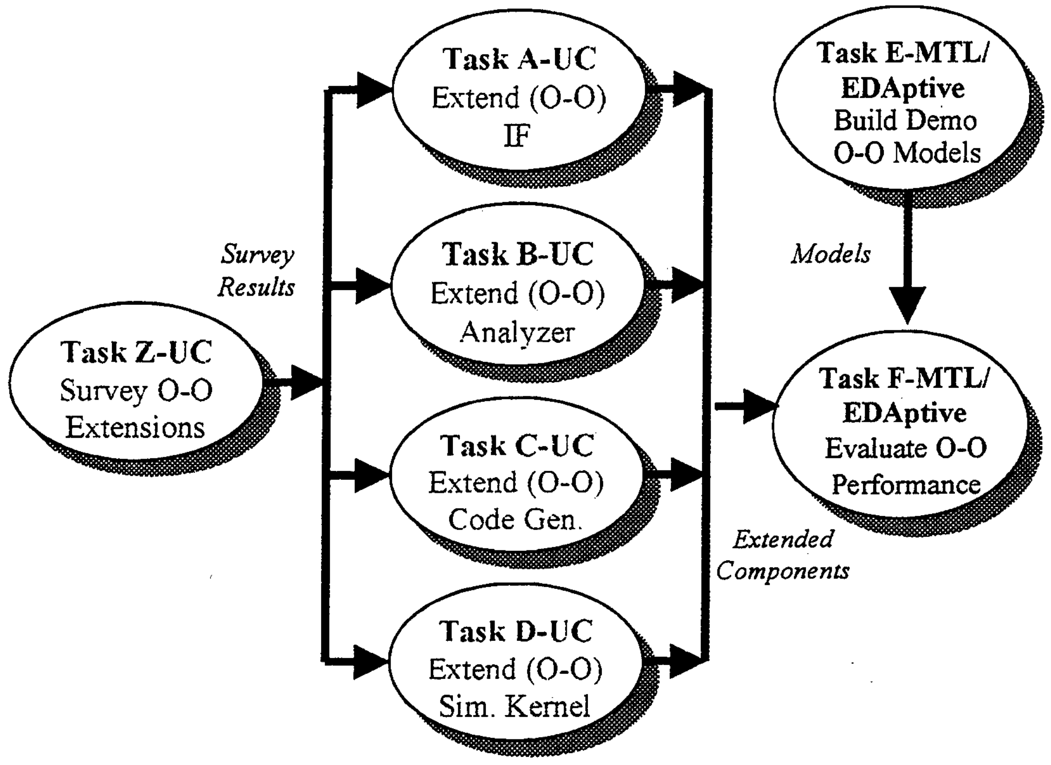


Figure 5. SAVANT Option Tasking.

2.5.3 Project Organization and Management Approach Summary - In summary, the project was executed slightly differently than was planned, in order to gain efficiency, and additional subcontracting support was obtained to provide necessary expertise. The results of the original task were the SAVANT components and their test results, and the results of the option tasking were the extended components, the O-O extension models, and the results of the O-O extension performance assessment.

In the next section, we describe all these results, and the activities leading to them, in detail.

3.0 RESULTS AND DISCUSSION

In this section, we describe the results of our development of the SAVANT system, and the assessment of the O-O extensions to VHDL in our Option Task. We also include any significant events, discoveries, or problems we encountered in the course of our work which we feel would benefit the reader. For the SAVANT portion of this section, we first discuss the components of SAVANT: the Analyzer (and IF), the Code Generator, and the Support Environment (consisting of the Debugger and the User's Toolkit). These components were developed for use on Unix systems. We did not achieve an NT-based version in this effort. We then address the testing and distribution aspects of SAVANT. For the O-O Extensions, we first outline a survey of the O-O extensions we conducted, then describe the O-O extensions we made to the components, the models we selected to assess them, and the results of the assessment. First, however, before delving into the SAVANT development and results aspects, we provide some background information regarding how other programs' technologies and products related to SAVANT.

3.1 The SAVANT System - An Overview of Related Technologies - The SAVANT project interacted with a number of important, related projects at UC, and it is helpful to understand how these contributed to the SAVANT development. Specifically, components from the DARPA RASSP, QUEST II, and HEPE programs were used by the SAVANT program. The interactions among the VHDL components from the SAVANT/RASSP (Rapid prototyping of Application Specitic Signal Processors)/QUEST (Quick Execution, Synthesis and Test Vector Generation)/HEPE (Heterogeneous Environment for Performance Evaluation) research programs are illustrated in Figure 6. Here, we also indicate use of an intermediate form (IF) standard, AIRE (Advanced Intermediate Form for Extensibility), being developed with support from this project.

The Advanced Intermediate Representation with Extensibility (AIRE/CE) is a specification describing an object oriented representation of VHDL parse trees. The AIRE specification includes support for VHDL '87, VHDL '93, and VHDL-AMS (although SAVANT supports only VHDL '93). The AIRE-CE specification was developed jointly by MTL Systems and the University of Cincinnati with contributions from many other organizations. Online documentation for AIRE-CE can be found on the web at http://www.mtlsystems.com/aire/.

AIRE/CE specifies two different representations of VHDL: (i) the Internal Intermediate Representation (IIR), an in-memory representation of the parse tree; and (ii) the File Intermediate

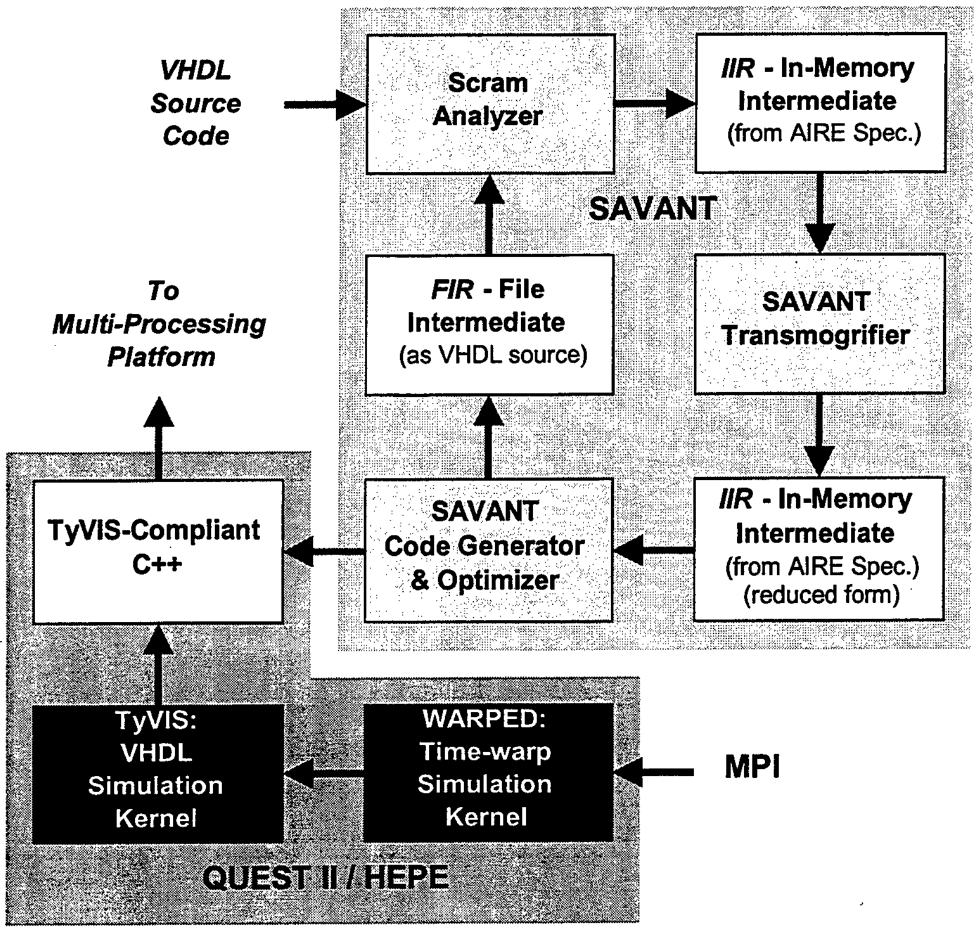

VHDL Source Code
Scram Analyzer
IIR - In-Memory Intermediate (from AIRE Spec.)
SAVANT
To Multi-Processing Platform
FIR - File Intermediate (as VHDL source)
SAVANT Transmogrifier
TyVIS-Compliant C++
SAVANT Code Generator & Optimizer
IIR - In-Memory Intermediate (from AIRE Spec.) (reduced form)
TyVIS: VHDL Simulation Kernel
WARPED: Time-warp Simulation Kernel
MPI
QUEST II / HEPE

Figure 6. UC Program Relationships to SAVANT.

Representation (FIR) used to store parse trees to files. This discussion will concentrate on the IIR. At present, SAVANT does not use FIR for the file format and instead uses a machine generated VHDL to build library files.

SAVANT's implementation of the IIR is written in C++, which uses the notion of a class in order to encapsulate objects. Each node specified by the IIR has been implemented as a class in the SAVANT implementation, and the various functions specified are methods of these classes. The resulting hierarchy is used to create an efficient representation of a VHDL program. Our implementation of AIRE/CE provides extensibility by using the inheritance feature built into C++.

The interactions among these projects provided a synergistic working relationship, while each project attended to its particular goals. The objective of the SAVANT program was to build a VHDL analyzer with a well-documented, extensible, object-oriented intermediate form. The RASSP program investigated the development of formal semantic models for VHDL and a set of affiliated theories for manipulating the models. The QUEST program investigated parallel algorithms and architectures for simulation, synthesis, and Automatic Test Pattern Generation (ATPG). The HEPE program attempted to simultaneously apply digital simulation and formal methods to network performance evaluation.

A single point of reference for the VHDL analysis and simulation aspects of these three projects is available on the WWW at http://www.ececs.uc.edu/~paw/lab. Individual sites for the QUEST and HEPE projects are:

QUEST Project:	(http://www.ececs.uc.edu/~hcarter/questII.html)
HEPE Project:	(http://www.ececs.uc.edu/~kbse/hepe)

We discuss the specifics regarding our use of these technologies from other projects in more detail, as we proceed through the various components of SAVANT within this section. We begin now with the Analyzer.

3.2 Task 1: The Analyzer

The SAVANT package includes a VHDL analyzer, called "Scram," that inputs and validates the correctness of VHDL models. For correct VHDL models, Scram then builds the corresponding Intermediate form Representation (IR) using the In-memory IR (IIR) format defined by the emerging AIRE/CE standard (discussed more fully in the next paragraph). Scram is built using the freely-available LL(k) parser/generator called PCCTS (from Purdue). The VHDL grammar for Scram is an LL(2) grammar, with some additional embedded lookahead predicates to aid the parsing process. The grammar also includes the affiliated action code to type-check and build the intermediate form (IIR).

As originally proposed, the SAVANT project was to develop its own intermediate form representation with accompanying online hyper-linked documentation. At the end of the first year, we decided to merge our intermediate form with FTL Systems' John Willis' work, and to strive toward an *international standard intermediate form*. Thus began the AIRE/CE intermediate form. AIRE/CE actually defines two intermediates:

1. The IIR, which is an In-memory Intermediate Representation form definition, and
2. The FIR, which is a File Intermediate Representation form.

At the beginning of the second year of the project, we began our conversion to this IIR. The version of the SAVANT intermediate form implemented at the close of the SAVANT project is fully compliant to the IIR, to the extent we have tested it.

The file format used by SAVANT is raw VHDL. More precisely, the SAVANT project has extended the IIR node definitions with several key extensions, namely: *publish_cc*, *publish_vhdl*, and *transmute*. The *publish_vhdl* methods reproduce VHDL code from the IIR intermediate. Thus, any model stored in the IIR can be rewritten as VHDL using the *publish_vhdl* methods. It should be noted that the VHDL written by *publish_vhdl* is not formatted exactly as the original input VHDL. More precisely, there may be extra or missing spaces, tabs, or new lines. Extensive testing has been performed to validate that the *publish_vhdl* methods faithfully reproduce the equivalent VHDL. This is discussed in Section 3.4.

After the conversion to the IIR intermediate, a good deal of time was spent attempting to implement the FIR format. Unfortunately, the FIR format definition was not as mature as the IIR format and we found too many gaps in the standard to continue at that time. Furthermore, the *publish_vhdl* functions were becoming crucial to our user community for examining the results of their manipulations of the IIR. More precisely, we have found that users of SAVANT will frequently decide to add routines to the IIR nodes that analyze and manipulate the IIR tree. They will then use the *publish_vhdl* functions to output the modified tree and finally they (frequently) use commercial tools against the rewritten VHDL. This is done by at least two different groups, one studying synthesis and one studying fault modeling. Furthermore, we found the *publish_vhdl* functions helpful to our debugging within the SAVANT project. Consequently, we abandoned our efforts to implement the FIR output routines and instead focused on the *publish_vhdl* methods. The work done in writing the FIR output routines has not been continued. However, the partial implementation is still present in the extension classes and the FIR output routine development could easily be continued.

The aforementioned *publish_cc* methods and *transmute* methods are important to our integration with the QUEST II and HEPE program codes to develop an operational VHDL parallel simulator. In particular, the *publish_cc* routines output C++ code that links with the TyVIS and WARPED simulation kernels (see Figure 6). The transmute methods implement the reduction algebra from the UC RASSP project in order to simplify the IIR nodes for which *publish_cc* methods would need to be written. More simply, the *transmute* nodes rewrite the VHDL concurrent statements (or, at least, their IIR nodes) into simpler forms. For example, except for the process statement, all concurrent statements are rewritten into their equivalent process statement form; process statements with sensitivity lists are rewritten as process statements with a trailing wait statement; and sequential signal assignment statements with a set of waveform elements are rewritten into a set of signal assignment statements, each with only one waveform element.

Originally the *transmute* functions were used before all of our extensions (*publish_vhdl* and *publish_cc*). However, since we were using the *publish_vhdl* functions to write our libraries, the IIR tree handed to backend users was actually the "transmuted" tree. Unfortunately, some backend CAD tools (such as behavioral synthesis) impose additional semantic information into concurrent statements (for instance, concurrent signal assignments are assigned additional semantic constraints over their equivalent process statement representation); consequently these tools were unable to perform optimally on the transmuted form. We discovered this error only at the beginning of 1998, but were able to quickly extend the *publish_vhdl* methods to the entire node set of IIR and are now able to hand the original IIR tree off to the backend CAD tools.

3.2.1 Task 1a: Analyzer Development. The analyzer (Scram) development built upon the '87 grammar originally developed by Dr. P. A. Wilsey in the early 90s. The grammar was originally written using only the PCCTS tools, however, it was discovered that the lack of a backtracking lexer tool in PCCTS complicated the implementation of a fully-compliant VHDL '93 analyzer. Consequently, at the beginning of the SAVANT project the grammar conversion to VHDL '93 (partially completed by Dr. Wilsey) was completed and the PCCTS lexer was replaced by a flex lexer (flex is also a freely-available tool).

Various versions of the analyzer have been released over the lifetime of the SAVANT project. The first releases occurred just after the first year of the program and have continued regularly throughout the program. In general, we have tried to release new versions just before significant VHDL related conferences were held or whenever significant advances were made to the capabilities of the system. Just before the Design Automation Conference (DAC '98), version 0.9.1 was released and a final version (1.0) is slated for release in October 1998. These versions are fairly complete, passing over 97% of the no error cases of the Billowitch test suite. We have also worked to build a system that produces reasonable error messages and that aborts gracefully (instead of giving segmentation violations).

3.2.2 Task 1b: Analyzer Code Improvement. In the general sense, MTL and UC interacted throughout the program to identify and implement coding changes deemed appropriate. While MTL sought to identify and suggest such changes, UC was the final authority on implementation for two reasons: (1) UC was doing the development, and, consequently, had the expertise to implement changes. MTL's expertise lay more within the ability to test and assess functionality. (2) UC was executing an iterative development program with their own version control implemented. For MTL to inject an additional version with changes was deemed potentially disruptive to the development process at UC. In this sense, several suggestions for improving output handling, entity identification, and other minor aspects were communicated to the UC development team.

The most significant improvement suggestion was one concerning compile time improvement (reduction). We discovered that although the initial compilation time was largely not improvable, techniques could be implemented to improve subsequent re-compilations. We developed a technique which would basically segment the compilation process, and then only recompile those portions affected by changes when re-compilation was invoked. This avoided the need to recompile the whole system every time, and seemed to have the effect of reducing the recompilation times by 30-50%, depending upon the degree of changes and the amount of code affected.

3.2.3 Summary of the Analyzer and IF Development - This development produced a working analyzer, as well as IIR and FIR specifications. Code improvements were made in the course of the development. Next, we describe the development of the Code Generator.

3.3 Task 2: The Code Generator

As noted above in Section 3.1, the IIR nodes have been extended for the SAVANT project to include three key methods, namely: *transmute*, *publish_cc*, and *publish_vhdl*. The *publish_vhdl* methods are used to write the library files (which are then parsed back in automatically by the parser as use clauses are encountered), and for debugging purposes. These output routines have been extensively tested; our tests show that anything that the parser can correctly recognize, we can correctly reproduce back into its VHDL form.

The *transmute* and *publish_cc methods* operate in conjunction to write C++ code for simulating the input VHDL. The resulting output links with TyVIS and WARPED, and uses runtime elaboration, to effect a parallel simulation of the input VHDL.

TyVIS is a VHDL simulation kernel that links the C++ output from SCRAM to the simulation support libraries of WARPED. VHDL constructs and statements are represented as C++ classes, methods, and members in the TyVis simulation kernel so they can be simulated using the WARPED distributed simulation kernel. The design of the TyVIS simulation kernel takes into consideration two sets of constraints. The first is imposed by the simulation semantics of VHDL as specified by the Language Reference Manual. The other constraints are imposed by the WARPED distributed (Time Warp synchronized) simulation kernel. Examples of these constraints are the ability of a process to rollback, save state, and so on. Careful consideration is required to make sure that the design adheres to all of these constraints.

The VHDL description must be elaborated before it can be simulated. According to the language reference manual, "The process by which a declaration achieves its effect is called the elaboration of the declaration." TyVIS uses dynamic elaboration to elaborate the design. In this technique, the "code" required to elaborate the design is generated when it is analyzed. This is then compiled and executed to elaborate the design. Once elaborated, the fanouts and the hierarchy information in the design are captured for use during simulation.

Another important construct of VHDL is the "wait" statement. This is typically used to synchronize processes. An execution of a VHDL wait statement is implemented in a process by sending a wait event to itself after the specified time interval. On a signal update the sensitivity list and the condition clause of the wait statement are inspected to check for its resumption.

Other major components in the TyVIS kernel are the type system, the marked queue for updating signals, the resolution tree, which is responsible for resolution and type conversion of signals, package standard, and package textio. Additional information, including the source code for the current release of TyVIS, is available on the web at http://www.ececs.uc.edu/~paw/tyvis.

The WARPED project defines a general purpose discrete event simulation interface and contains two simulation kernel implementations for that interface. One of the implementations is a sequential kernel and the other is a parallel kernel that uses the Time Warp synchronization mechanism. Consequently, any application that conforms to the WARPED API can be executed with either kernel. The following paragraphs provide a brief overview of WARPED, focusing on the parallel kernel implementation.

The parallel simulation kernel of WARPED is organized into a collection of communicating Logical Processes (LPs), with each LP containing numerous simulation objects. Each LP is a distinct simulator with a single, multi-threaded event list. Each simulation object in an LP maintains its own simulation clock and can rollback independently. The LPs schedule events for processing and exchange event messages to/from their constituent simulation objects to/from other LPs. Dynamic control for optimization occurs at the simulation object level and adjustment is triggered by a circulating token passed among the simulation objects.

The WARPED kernel provides the functionality to develop applications modeled as discrete event simulations. Considerable effort has been made to hide the details of Time Warp from the application interface. For example, sending events from one simulation object to another is done in the same way regardless of whether the objects are on a single processor or on different processors; all Time Warp specific activities, such as state saving and rollback, are performed automatically by the kernel without intervention from the application. A more detailed description of the internal structure and organization of the WARPED kernel is available on the www at http://www.ece.uc.edu/~paw/warped.

3.3.1 Task 2a: Code Generator Development - The code generator methods (*transmute*, *publish_vhdl*, and *publish_cc*) have been jointly developed with support from the QUEST II and HEPE programs. Investigators from all three programs (SAVANT included) have worked to complete the *publish_cc* methods and the TyVIS core. The WARPED core was developed separately from SAVANT. To the best of our testing, the *publish_vhdl* functions are complete and 100% correct; the *transmute* functions are also likewise 100% complete (again within the limits of our testing); and finally, the *publish_cc* routines are operational for about 90% of the Billowitch test suite and 80% of the Ashenden test suite (Section 3.5).

3.3.2 Task 2b: Code Generator Code Improvement - As in the case of the Analyzer Development (Section 3.1), MTL and UC interacted throughout the program to identify and implement coding changes deemed appropriate. While MTL sought to identify and suggest such changes, UC was the final authority on implementation, for the same development efficiency and configuration management reasons cited in Section 3.1.2. In this sense, several minor suggestions for improving code generator effectiveness were communicated to the UC development team.

The most significant improvement suggestion was one concerning output handling. We discovered that there was some redundancy in capturing data from the executable code, and outputting it to files. Elimination of the redundancies provided nearly a 50% reduction in execution time for a few small-scale models which were tested.

3.4 Task 3: SSE Development

Our support environment, developed under SAVANT, consisted of two components. The first, a debugger, was designed to provide all necessary debugging functionality from a command-line interface, and was set in the VHDL user's context. The second, which we named the VHDLyzer Toolkit, was an assortment of viewing tools designed to assist the user in working with the analyzer and IF. Here, we describe each of these components individually.

3.4.1 The Debugger - A command-line debugger is the primary system debugger. It is implemented as a driver for GNU GDB, translating user requests into one or more GDB commands, then collecting and reformatting GDB output into a form that a VHDL user will understand. The debugger consists of the primary debugging facilities needed for a usable debugger: start running, stop running, set breakpoints, clear breakpoints, and print values (signals and objects). Using GDB as the primary debugger with our own program as an interface allowed a number of benefits:

- The time and effort were compressed to be within the scope of the project. Even the simplest full debugger would have been too big and expensive to be done within the scope of the project.
- The debugging is done within a VHDL framework, unlike directly using GDB (or DBX), allowing a design engineer working in VHDL to understand the debugging. Using GDB directly would require intimate familiarity with system construction, class structures and naming conventions–all beyond what could be expected from any but a dedicated system hacker.
- There is no question of licensing problems.
- It is relatively feasible to convert to a different debugger such as DBX or totalview with only a modest effort at retargetting.
- By using GDB, the debugger will automatically port to most systems–as GDB will take care of the system interface issue.

The debugger has been implemented in a manner in which it only takes up memory and disk space when a user is actively debugging a program–and will only impact performance when in actual use. Since GDB was used for the system interface, the program will run in a separate address space and will be unaffected by the debugger unless a breakpoint is reached. In other words, if the user just watches the program run under the debugger and takes no action to affect the running, the program will run just as if the debugger were not present. This is about as much as can be reasonably expected from a debugger, as memory and disk resources must be used regardless of how it is designed.

3.4.2 The VHDLyzer Toolkit (VTK)

The VTK was developed to support users in working with the SAVANT technology and components, particularly the analyzer and IF. Here, we first overview the VTK, then describe its functionality, and finally describe its initial implementation and upgrade to conform to the final version of SAVANT developed under this effort.

VTK Overview - The VTK is a development and visualization tool which aids in the extension of the SAVANT VHDL Analyzer. The analyzer extension process is an iterative one and the VTK provides a convenient, user-friendly, graphical interface which provides tools and methods covering the complete process. These tools and methods include "making", analyzing, visualizing, and navigating the IIR results of analyzed VHDL code, and visualizing and navigating the object inheritance and aggregation hierarchy for specific IIR nodes.

The VTK-supported method, illustrated in Figure 7, allows the user to work iteratively and incrementally. Each iteration consists of four steps: compilation, analysis, visualization and extension. In the compilation step, the Analyzer source is partitioned into two parts, the IIR class hierarchy and the remaining Analyzer code (such as the lexical analyzer and the parser). Given the IIR class hierarchy in source form, the user compiles the IIR source and then compiles and links it with the remaining analyzer code. The result of this compilation step is an executable analyzer, shown in the figure as the VHDLyzer. In the analysis step, the user employs the VHDLyzer to analyze a VHDL description. To reduce the complexity of an iteration, a VHDL description that contains a small subset of the language is used, although VHDL descriptions analyzed during all iterations collectively contain all relevant language constructs that require extension. The analysis step produces an IIR which is saved in a file (shown as Node Log in the figure).

In the visualization step, the user views a visual representation of the IIR tree utilizing the file produced in the previous step. In addition, the user views visual representations of the class hierarchies, including inheritance and aggregation, for selected nodes in the tree. This step enables easy and intuitive identification of the classes in the IIR hierarchy where new methods should be inserted, or places in the class hierarchy where new classes should be inserted to extend the IIR functionality. In the extension step, add new methods or classes to accomplish the desired extension. The iteration is repeated with new VHDL descriptions until all the language constructs that require extension have been addressed, to completely customize the IIR class hierarchy.

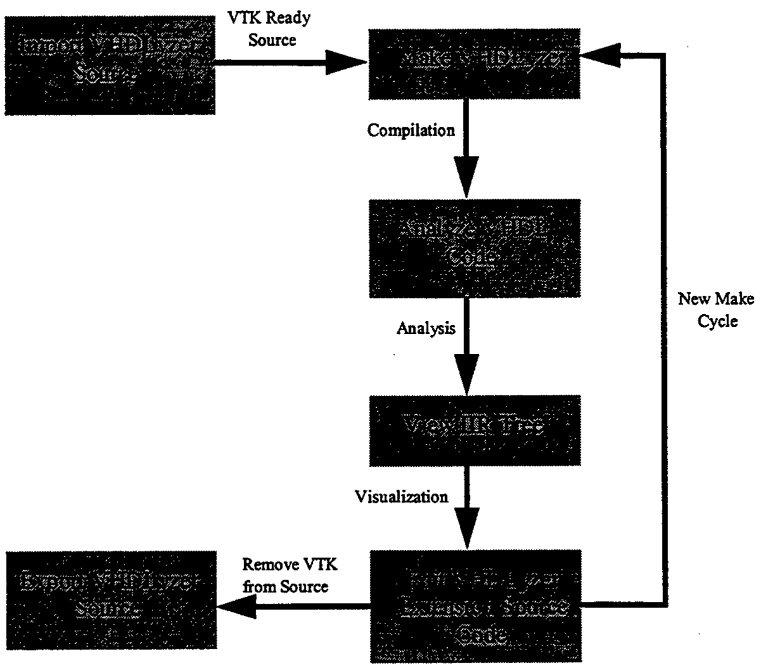


Figure 7: A method for Iterative and Incremental customization.

VTK Functionality - The following screenshots and accompanying discussions provide a flavor of the VTK functionality, through a tour of its primary user interface screens. We have organized our format to keep each screen shot and its description on the same page.

The VTK Main Window: Figure 8 illustrates the main VTK window. Its components and functions include:

- A Menu Bar (A)
- A Tool Bar which includes the following functions:
  (B) Exit Application
  (C) Perform Make Depend Process
  (D) Perform Analyze Process
  (E) Start IIR Tree Viewer
  (F) Start Default Text Editor
  (G) Start On-line Help
- A Progress Display Pane (H)
- A Status Bar (I)

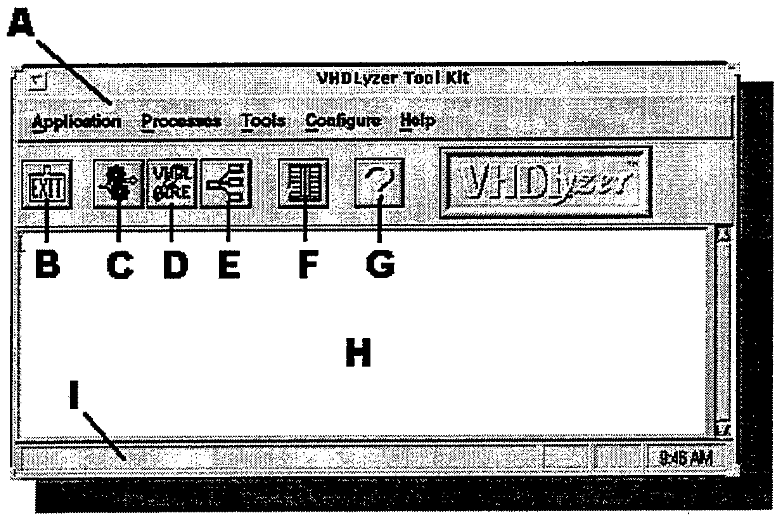


Figure 8. VTK Main Window.

The IIR Tree Viewer: The IIR Tree Viewer, shown in Figure 9, is one of the main components of the VTK. Its components and functions are as follows:

- A Tool Button Bar which includes the following functions:
  (A) Done Button, which closes the IIR Tree Viewer dialog.
  (B) View Class Hierarchy Button, which starts the IIR Class Viewer for the selected IIR Tree node.
  (C) Expand All Button, which completely expands the tree.
  (D) Find Root Button, which repositions the view to show the root node of the IIR Tree.
- A Node Browser (E), which permits closer inspection of node information.

- A IIR Tree Graph Pane (F), which provides the tree graphical view.
- A Status Bar (G), which indicates status of the tool or system.

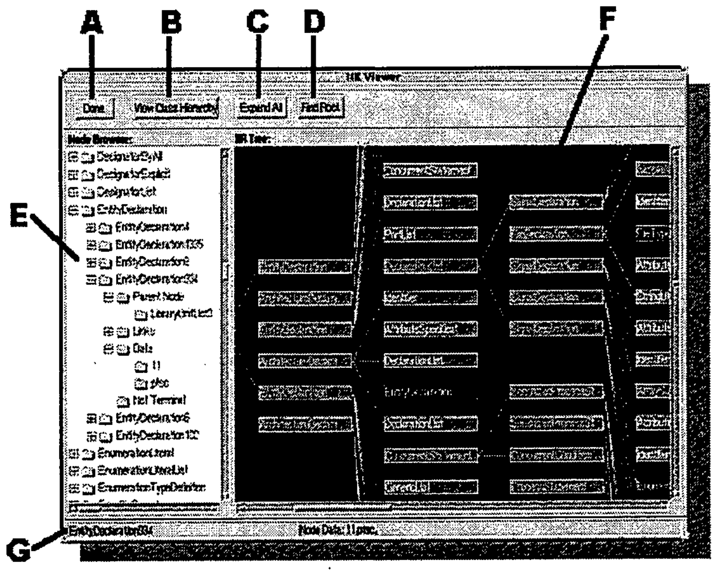


Figure 9. The IIR Tree Viewer.

The IIR Tree Viewer allows the user to graphically visualize the IIR produced when a VHDL description is processed by the analyzer. Navigation through the IIR tree can be accomplished by interacting with the Node Browser or with the IIR Tree Graph Pane itself, by using the mouse. The Node Browser allows the user to go directly to IIR nodes of interest in order to examine their contents. Any node selected with the mouse while using the Node Browser causes the IIR Tree Graph Pane to reposition itself to show the node's orientation within the IIR Tree. The IIR Tree Graph Pane responds to both left and right mouse button clicks. Left mouse button clicks performed on a displayed IIR Tree node either causes the tree to expand the next level of child nodes for display, or collapses all respective child nodes, removing them from view. Right mouse button clicks on a displayed IIR Tree node bring up the Class Viewer. Also, when the mouse pointer is positioned atop a displayed IIR Tree node, the unique name and any related node data are displayed in the Status Bar.

The IIR Class Viewer: The IIR Class Viewer, illustrated in Figure 10, is another major component of the VTK. Its views, controls and functions include the following:

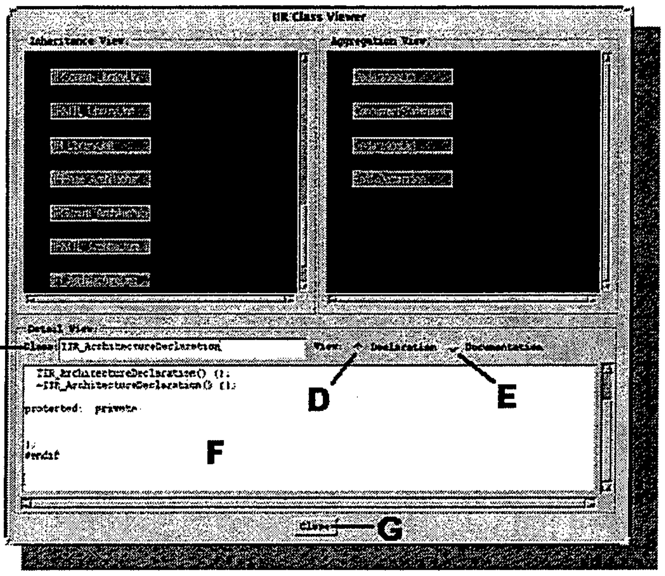
- An Inheritance View (A), which presents the object inheritance hierarchy.
- An Aggregation View (B), which presents a selected object's aggregations.
- A Detail View, which includes the following:

  (C) Class Display Box, which shows the full name of the selected object

  (D) View Declaration Button, which switches the Detail View's Information Box to show the declaration part of the selected object's header file.

  (E) View Documentation Button, which switches Detail View's Information Box to show the documentation part of the selected object's header file.

  (F) Information Display Box, which shows the Declaration or the Documentation depending upon which mode has been selected.
- A Close Button (G), which closes the viewer.



Class: IIR_ArchitectureDeclaration

```
  IIR_ArchitectureDeclaration() {};
  ~IIR_ArchitectureDeclaration() {};

protected:   private:


};
#endif
```

Figure 10. The IIR Class Viewer.

This dialog presents information about the object inheritance and aggregations which comprise a specific IIR tree node. It also presents either declaration information or documentation (Spy Class) information contained as part of an object's header file. This information is presented when the user selects a specific inheritance view class or an aggregation view class.

VTK Implementation. The VTK requires a SAVANT release to be imported, so that the VTK may work properly. It is this "Import" proces that requires updating if import problems occur. The "Import" process performs the following sequence of steps:

Step 1: Gathers IIR class information for use by the rest of the import code

Step 2: Generates the *nodes.dat* file

Step 3: Fills in the inheritance data for the IIR class information

Step 4: Generates the IIRMTL class files

Step 5: Changes the inheritance structure of the SAVANT code to include the IIRMTL classes

Step 5: Modifies the *main.cc* file

Step 6: Modifies the *makefile.in* file

The VTK Import process takes a SAVANT release, generates extension classes (IIRMTL) that publish the internal node information of the SAVANT IIR nodes, and modifies the inheritance hierarchy to include the generated IIRMTL classes. Other modifications are also made to SAVANT's *main.cc* and *makefile.in* files. The modification that is made to the *main.cc* file adds the processing for the "-publish-IIR" switch that starts the *publish IIR* process. The modification to the *makefile.in* file adds the appropriate lines to include the IIRMTL code directory in the make operation. Also, during the Import Process, a file called *nodes.dat* is generated that contains a list of all the IIR class names. This file is used by the VTK to look up the names of specific nodes by using an index into the list.

When Scram, the VTK-extended SAVANT analyzer, is used to analyze a vhdl file with the "-publish-IIR" switch provided, the result is a file called *IIRTree*. The *IIRTree* file contains the published information about the IIR created for the VHDL file analyzed. It is this *IIRTree* file that is generated by the IIRMTL class extensions to the SAVANT analyzer and that is needed by the VTK to graphically construct the IIR Tree Viewer.

Most of the complications occur when the VTK Import process parses the SAVANT source code to locate aggregate IIR data that needs to be published for the VTK.

VTK Issues: There are still SAVANT class files that have problems being imported by the VTK Import process. The following are discussions of areas where "hand modification" to the SAVANT/VTK generated code is required to allow 100% VTK operation. These import problems are not the result of an incomplete design of the VTK Import process.

1. *IIRMTL_ConfigurationItemList.cc* . Normally, SAVANT lists are imported using a "list walking" loop when the corresponding IIR publishing code is generated. In order to "walk" through a list, there needs to be an iterator of a specific type which is compatible with the list

elements for the list. The VTK import process tries to determine this type by looking at each header file for the "first" method's return type, starting with the list classes header file. If a type for the list elements cannot be found in the list classes header file, then the inheritance chain is followed until one is found, or until the base class header is reached. In this case, no type is found for the *ConfigurationItemList* elements, so it is assumed that the type is "*iir.*" Unfortunately, this causes a compilation error to occur, since this is not the correct type. The correct type is "*iir_ConfigurationItem.*" Changing the iterator type from "*iir*" to "*iir_ConfigurationItem*" and also adding an appropriate "*#include IIR_ConfigurationItem.hh*" directive cures this problem. To eliminate having to do this each time a SAVANT release is imported, a modification to the "*iirBase_ConfigurationItemList.hh*" file must be made to include the prototype for the correct "*iir_ConfigurationItem*" type for the "first" method of that class.

2. *IIRMTL_AssertionStatement.cc.* During the import process, a matching "get" function is not found for the IIR aggregate "expression." This occurs because there are two "get" member functions that have "expression" in their names, "*get_report_expression*" and "get_severity-_expression". Obviously, the import process cannot determine which is the correct one since the return types for both functions are the same. To alleviate this problem, *the IIRMTL_AssertionStatement.cc* file must be hand modified to include the correct publishing code for this aggregate data item. To eliminate having to do this each time a SAVANT release is imported, a modification to the IIRBase_AssertionStatement class must be made to change the name of the private aggregate data item from "expression" to "severity_expression." This was not a problem since this data item is private.

3. *IIRMTL_TextLiteral.cc.* During the import process, a matching "get" function is not found for the IIR aggregate "text." This occurs because there is a type inconsistency between the declared type of "text," "*iirScram_String **," and that of the "get_text" method, which is "*iir_-Char **." This will always cause an error when importing a SAVANT release because the types must match between the aggregate data item and the "get" method for that item. To alleviate this problem, the *IRMTL_TextLiteral.cc* file must be hand modified to include the correct publishing code for this aggregate data item. To eliminate having to do this each time a SAVANT release is imported, a modification to the *IIRBase_TextLiteral* class must be made to change the type of the "text" aggregate, or to change the return type of the "*get_text*" member function, whichever causes the least amount of impact.

4. *IIRMTL.cc.* During the import process, a matching "get" function is not found for the IIR aggregate "_my_design_file." This occurs because a "get" method for this aggregate data item does not exist. Because this "get" function does not exist, no access code is generated for this aggregate data item. This means that in order to publish data for this data item, custom code must be created to perform this function in this source file.

3.5 Task 4: Testing and Distribution

The purpose of our testing was to confirm the functionality of SAVANT, and to ensure that adequate performance levels were achieved prior to distribution. The testing included both *conformance testing* (analysis and simulation with expected results) and *performance testing* (simulation execution times). The distribution was done to provide users within the community access to working, interim versions, and to obtain any feedback or suggestions which would benefit the ongoing development. Here, we discuss these two aspects individually.

3.5.1 Task 4a: Conformance Testing - Our conformance testing sought to confirm three aspects of SAVANT functionality: (1) successful analysis (parsing and type-checking), (2) successful simulation (execution to completion producing expected results), and (3) portability (ability to function on multiple platforms). Here, we describe our testing and results in the context of these aspects.

Before getting into the testing specifics, we need to address the issues of configuration management. This activity was necessary in order to associate performance testing results with the version which produced them, and to subsequently track any changes we made to fix problems revealed by the testing. The SAVANT source code was all maintained using the CVS source code control software. This allowed us to have concurrent developers operating on the same code; CVS attempts to automatically merge differences of simultaneous users as commitments of changes are made into the archive. Change logs and histories were maintained by CVS to aid in recovery (if necessary).

Several benchmark suites were collected or constructed to support the SAVANT project. We used three primary benchmark suites: (a) the Billowitch benchmark suite, (b) the collected examples from Dr. Peter Ashenden's book on VHDL[4] (hereafter called the Ashenden benchmark suite), and (c) an informal collection of 22 VHDL models that we collected from the public domain (hereafter called the Public Domain benchmark suite). In addition to these suites, we organized a set of generic perl scripts that can run Scram against any of these three test suites. A policy of running the scripts every Sunday evening with results distributed Monday was established. A team of researchers were specifically assigned the task of running the scripts and of communicating the results to the project members. Finally, a BUGS file was maintained with assigned priorities and illustrative VHDL code (or pointers to VHDL code) that illustrated the problem.

The Billowitch test suite is divided into error cases and no-error cases. There are approximately 1600 no-error cases and 500 error cases. The no-error cases are further organized into simulatable and non-simulatable. The Billowitch test suite was originally written in VHDL '87 and had to be converted to VHDL '93. In the course of the three (3) years of the SAVANT project, we periodically located errors in the no-error cases (approximately 20) that had to be removed from the test suite. As of the 0.9.1 release of Scram, we are able to: identify errors in about 50% of the error cases; correctly parse and type check approximately 98% of the no-error cases; and correctly simulate just over 90% of the simulatable cases.

The Ashenden test suite contains 486 no-error test cases. These are all written in VHDL '93 and cover all aspects of the language. Version 0.9.1 of Scram can correctly parse and type check 84% of these models and correctly simulate 81%.

Finally, we have worked to ensure the portability of the code by compiling and developing on as many platforms as possible. In particular, we have active development being performed on Intel hardware running the GNU g++ compiler, on SUN hardware running the GNU g++ compiler, and on SUN hardware running the SUNPro CC compiler. In addition, periodic testing occurs on a DEC Alpha workstation running OSF. Before each release, we attempt to test the code on all four configurations to help ensure its portability.

3.5.2 Task 4a: Performance Testing - Our performance testing sought to assess SAVANT's execution performance. We accomplished this by executing several benchmarks upon a multiprocessor machine. Here, we first describe the benchmarking environment, and then present and discuss the performance analysis and results we obtained. These tests were configured and performed by Dr. Phil Wilsey and the UC project team, and are included in a UC report[5] which also describes performance results for two other parallel logic simulators; one from the University of Michigan and another from FTL Systems, Inc.

The Benchmarking Environment: A set of nine benchmarks were collated to serve as a test suite for comparing the performance of different parallel logic simulators. These test circuits, which included several International Symposium on Circuits And Systems (ISCAS) models, were selected such that the SAVANT simulator (as well as others) could parse and simulate the circuits. The benchmark set included the circuits listed in Table 1.

Table 1. Benchmark Set.

List of Benchmarks	Purpose	No. of Processes
Benchmark One	32-bit Parallel Multiplier	70
Benchmark Two	ISCAS'85 Model (c432)	160
Benchmark Three	ISCAS'89 Model (s641)	379
Benchmark Four	ISCAS'89 Model (s953)	395
Benchmark Five	ISCAS'89 Model (s5378)	2800
Benchmark Six	ISCAS'89 Model (s6669)	3080
Benchmark Seven	ISCAS'89 Model (s13207-1)	8000
Benchmark Eight	ISCAS'89 Model (s35932)	16065
Benchmark Nine	Parity Tree Model (from FTL Systems)	81000+

Benchmark One is a 32x32-bit behavioral-level model of a parallel multiplier, shown in Figure 11. In it, each 4-bit block of the multiplicand is multiplied by a separate 4-bit block of the multiplier in separate logical processes (LPs). These 64 partial products are then shifted appropriately and added together to arrive at the final result. This model, with support processes, contains a total of 70 processes. This model is not an optimal model for parallelization. While the actual multiply is performed with 64-way parallelism, all 64 partial products are shifted as needed and added together by a single process. This bottleneck restricts the overall speedup that can be achieved through parallel simulation but serves as a good test case to evaluate the performance of a parallel logic simulator

Benchmarks Two to Eight are combinatorial and synchronous sequential circuits that are part of the ISCAS'85 and ISCAS'89 benchmark suites. In Table 2, we list the characteristics of each ISCAS benchmark. These benchmarks were chosen because they use relatively simple VHDL constructs that we could suitably modify and are reasonably large. Although these circuits are large in size, the process granularity is relatively small. *Benchmark Nine* is a model of a parity tree supplied by FTL Systems. It is a combinatorial circuit similar to the ISCAS'85 benchmark circuits, the only difference being that this is a much larger circuit. It instantiated a total of 81,915 gates. This is an ideal benchmark to test the limits of a parallel logic simulation system.

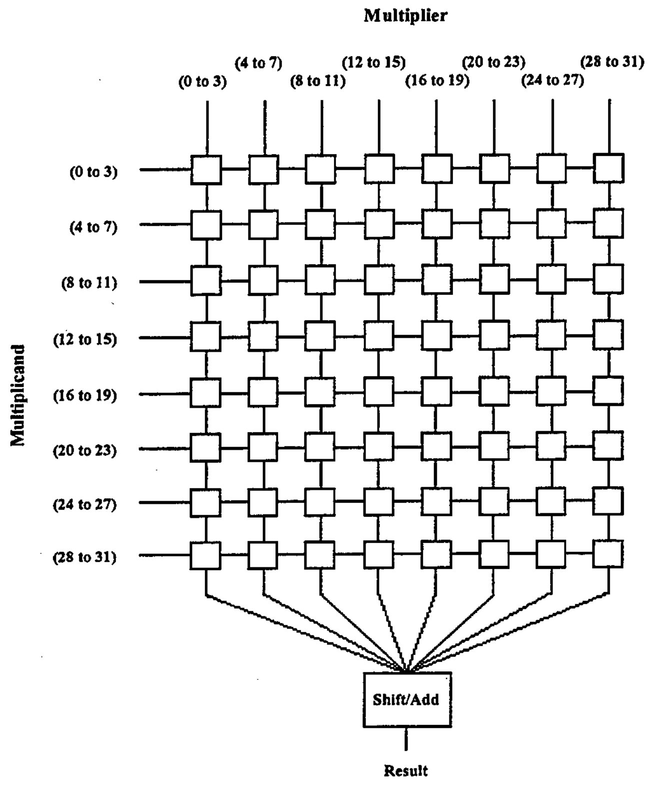


Figure 11. A 32x32 bit Parallel Multiplier

Table 2. ISCAS Benchmark Characteristics

ISCAS Benchmark	# of Model	# of gates	# of flipflops	# of input	# of output	# of signals
Benchmark Two	c432	160	0	36	7	153
Benchmark Three	s641	379	19	35	24	374
Benchmark Four	s953	395	29	16	23	401
Benchmark Five	s5378	2779	179	35	49	2909
Benchmark Six	s6669	3080	239	83	55	3264
Benchmark Seven	s13207-1	7951	638	62	152	8437
Benchmark Eight	s35932	16065	1728	35	320	17473

When we obtain a VHDL source file for benchmarking, it is prudent to perform some *validation* before proceeding to the performance testing. Hence, we performed the following steps on the circuits listed above:

1. To ascertain whether the test file uses legal VHDL, we analyzed the test file with both the SAVANT analyzer and a Synopsis analyzer. This allowed us to verify the errors generated by both analyzers.
2. Once we knew the VHDL was legal, we manually edited the source file to provide data (in the form of an output file) on internal signal values and results (testbench creation). This was then simulated using the Synopsys sequential simulator to generate an *out.good* file which was then used for correctness comparisons.
3. The test file was parsed and TyVis-compliant code was generated by using the *scram -publish-cc testfile.vhdl* command. The generated code was then compiled and executed on a parallel platform.
4. The results from the parallel execution were compared with the *out.good* file generated by the sequential simulator.

We used two platforms to conduct our experiments. The first platform is where we carried out most of our intermediate testing and development. This platform is a cluster of eight, Pentium-II, dual-processor machines running Linux (a Beowulf-like system). The cluster is interconnected by a 100Mbps Ethernet as well as a 1.28Gbps Myrinet. The characteristics of this platform are:

- Operating System: Red Hat Linux (version 2.1.126)
- 128 Mb per SMP node of main memory
- 2 Pentium II Processors per SMP node
- 16 300 MHz Pentium II Processors
- 32 KB on-chip unified data/instruction cache
- 512 KB off-chip unified data/instruction cache
- Interconnected by both 100Mbps Ethernet as well as 1.28Gbps Myrinet network.

The second platform used for experimentation was the Silicon Graphics Origin 2000 machine (remote access to this machine was given by the Ohio SuperComputer Center). All benchmarking results presented in this report were collected on the Origin 2000 machine. The Origin 2000 is a shared-memory multiprocessor system consisting of 24 250-MHz, IP27 processors and 3GB of main memory. Each processor has a MIPS R10000 CPU and a MIPS R10010 floating point unit. The current operating system is IRIX 6.5. These are the characteristics of this platform:

- Operating System: IRIX 6.5

- 3GB of main memory
- Twenty-four 250 MHz IP27 Processors
- MIPS R10000 CPU
- MIPS R10010 FPU
- 64KB on-chip data cache
- 4MB off-chip unified data/instruction cache
- has SGI's optimized implementation of the Message Passing Interface (MPI)

Performance Analysis and Results: In this subsection, we present the performance of the SAVANT simulator. The experiments detailed in this subsection were carried out on a 24 processor shared memory workstation (SGI Origin 2000). All the experiments were executed as batch jobs on the Origin. This was because batch jobs were allotted dedicated processors and usually resulted in the best performance. However, there is a limit on the number of processors that can be used in a batch job. The limit on the Origin we used, was a maximum of 8 processors per batch job. Each experiment was run five times and the average of the results was reported.

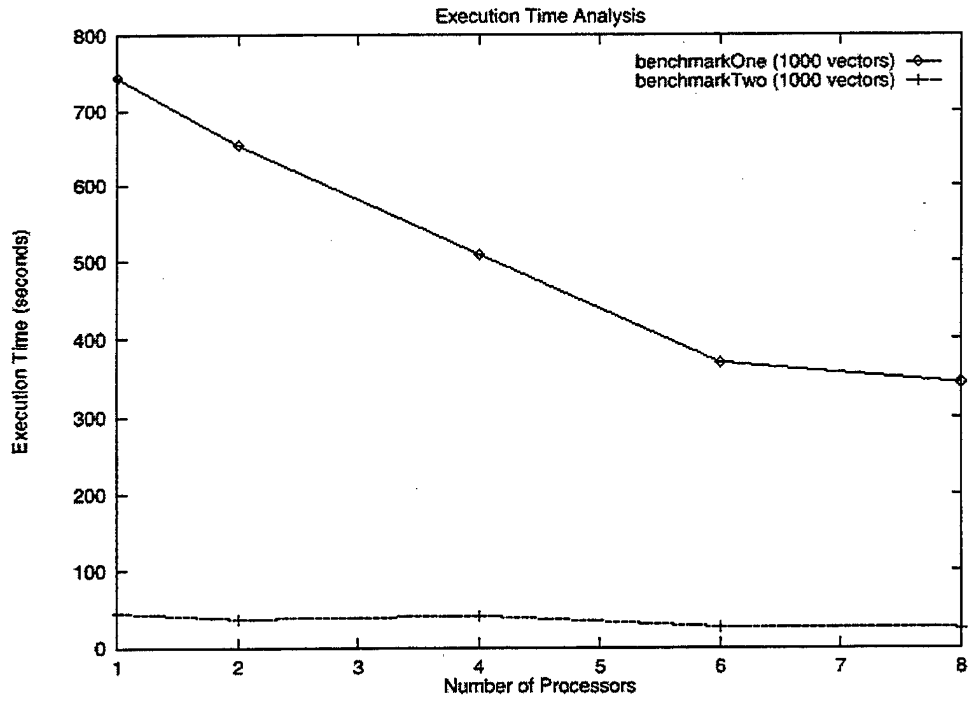


Figure 12. Performance of Benchmark One and Benchmark Two

Figure 12, above, illustrates the execution performances (in terms of execution time) of *Benchmark One and Benchmark Two.* Although Benchmark One has the least number of processes, each process has greater granularity than the processes in the other benchmarks. Due to this, there is almost a linear decrease in execution time as more processors are used to simulate the example. The reduction in execution time starts tapering off when more than 6 processors are used. *Benchmark Two*, on the other hand, is the smallest of the ISCAS benchmarks in the suite. In addition, the processes in Benchmark Two are typically of low-granularity. Hence, while there is an improvement in the execution performance when the number of processors is scaled up to 8, the decrease in execution time is not as large as in the case of Benchmark One.

The execution performances of *Benchmark Three* and *Benchmark Four* are illustrated in Figure 13. As can be seen from the figure, both these benchmarks are similar in nature and in size (similar number of processes). Their execution performance is also very similar. Due to their low granularity-processes, the performance of the benchmark is highly dependent on several factors. As the number of processors are scaled, we see a gradual decrease in the execution time. However, once the number of processors used exceeds four, we start to see the curve flatten out. If we continue to add processors, we actually see the curve go up, that is, the performance actually degrades. This can be attributed to several factors such as under-utilization of the processors, higher set-up and communication costs and poor partitioning.

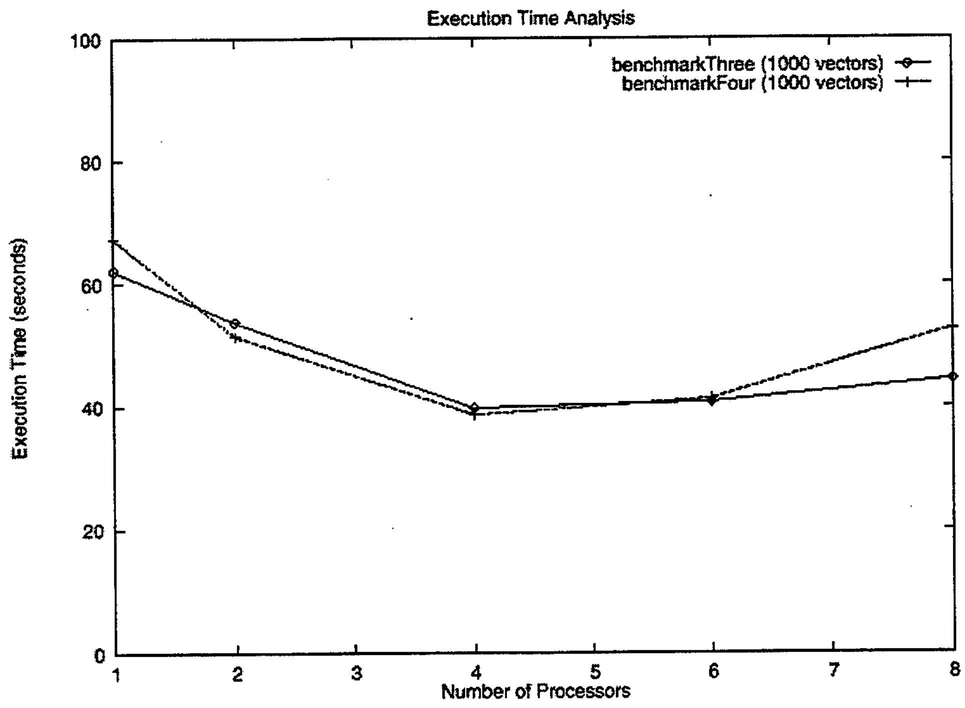


Figure 13. Performance of Benchmark Three and Benchmark Four

Figure 14 illustrates the execution performances of *Benchmark Five* and *Benchmark Six*. As can be seen from the figure, due to the larger number of processes in the simulation, the simulation takes a larger amount of time to complete. However, due to the larger number of processes in the simulation, the processors have more work to do. As a result, as the number of processors are scaled, the performance improves significantly. A performance improvement of 50% is obtained when 6 or more processors are used for the simulation.

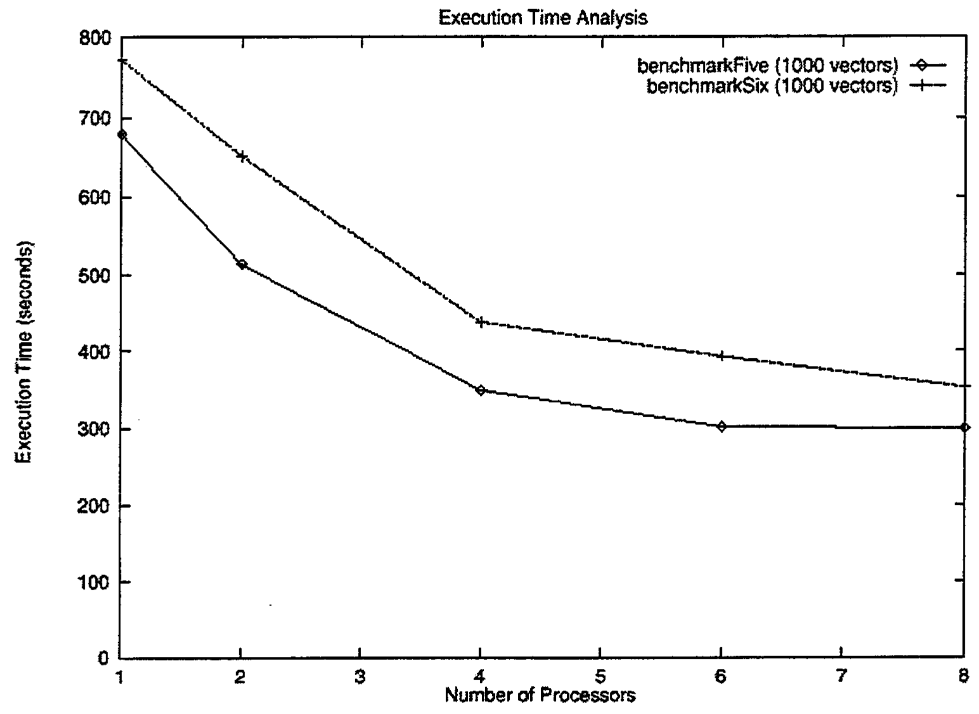


Figure 14. Performance of Benchmark Five and Benchmark Six

The execution performance of *Benchmark Seven* and *Benchmark Eight* is illustrated in Figure 15. These two benchmarks are two of the largest ISCAS benchmarks available. However, they have one peculiar trait. Some of the processes in the benchmarks do not process as many events as other processes and are significantly behind other processes in simulation time. Due to this, GVT-based garbage collection occurs very infrequently. Here, fossil collection of memory is severely handicapped by these processes. As a result, these examples require significantly larger quantities of memory and their performance is impacted by this impediment. As can be seen in Figure 15, the performance improvement is not as good as what was observed with the other benchmarks. The curves are flatter and the performance improvement is considerably lower than what was attained with the other benchmarks. Currently, efforts are on-going to compile and run *Benchmark Nine*, which is the largest of the nine benchmarks in the benchmark suite. Hence, we do not present the execution performance on *Benchmark Nine* in this report.

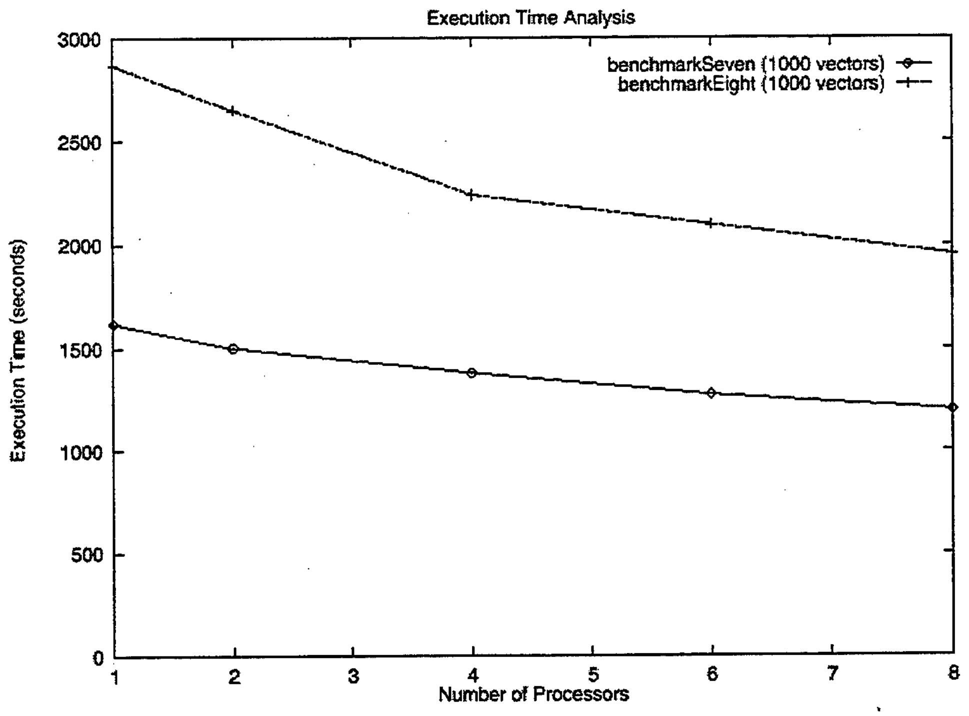


Figure 15. Performance of Benchmark Seven and Benchmark Eight

3.5.3 Task 4b: Distribution - We distributed interim versions of SAVANT via WWW page download, which was available at both MTL and UC sites. In the course of reviewing comments and suggestions from users, we identified and solved several potential problems in SAVANT. These included:

- Inability to compile or execute: These comments revealed several portability problems which we solved, and motivated our testing of releases on multiple platforms, as we described above.
- Difficulty with user interfaces: We were continually responding to users' comments by improving the system documentation being released with the software. This interactive fine-tuning, with user input, has enabled us to refine our user's manual to be what we consider a practical and useful document.
- Long compile times: Comments from users along this topic stimulated us to perform the compile-time improvement we discussed in Section 3.1.2.

In addition to the above issues, we received many other helpful suggestions as a result of our distribution activities. Altogether, we consider this activity successful in both providing the software to the community and in obtaining useful feedback for our development efforts.

3.6 Task Z: Survey of Proposed O-O Extensions to VHDL

Our survey was focused upon gaining an appreciation of current approaches to, and motivations for, O-O extensions to VHDL. The SAVANT project was fortunate to have Dr. Peter Ashenden from the University of Adelaide (Australia) join the project from January 1997 to January 1998. The project provided a minimal amount of support for Dr. Ashenden during his yearlong sabbatical at UC. The O-O Extensions have now been named SUAVE (SAVANT and University of Adelaide VHDL Extensions) [6] and a working draft of the extensions is serving to drive tool development.

The SUAVE project aims to introduce object-oriented extensions to data modeling, type genericity, and communication into VHDL in a way that does not disturb the existing language or its use. Designers regularly define abstract data types by using aspects of VHDL's type system, subprograms, and packages. The SUAVE approach builds on these basic mechanisms by strengthening the facilities for encapsulation and adding an inheritance mechanism. In addition to supporting object-orientation, these extended mechanisms improve the expressiveness of VHDL across the modeling spectrum, from high-level to gate-level. By choosing an incremental and evolutionary approach to extensions, SUAVE avoids major additions to the language that would complicate the choice of mechanisms for expressing a design.

In our review of work proposing extensions to VHDL to make it a more object-oriented language (VHDL is currently what is called an object-based language), we observed that two primary strategies were followed, namely; (a) the addition of C++ like classes, or (b) the addition of Ada-like extensions (specifically tagged types). While the former is quite popular, the latter has been explored by only a couple of investigators.

We reviewed the objectives for the extensions and reviewed some basic principles of computer language design. We were strongly influenced by the notions of "conceptual integrity" by Fred Brooks ("The Mythical Man-Month").[7] Specifically Dr. Brooks writes: "Conceptual integrity does require that a system reflect a single philosophy and that the specification as seen by the user flow from a few simple minds." To this end and in recognition that the VHDL language is based on Ada and *not* on C++, we studied closely the work of the Ada 9X community in extending Ada with items for object-orientation.

3.7 Task A: Extend the SAVANT IF

Only a few additional IIR nodes were required for SUAVE. They have all been implemented (complete with their publish_vhdl extensions) and are used by the SUAVE analyzer (see Section 3.8). The specific extensions are noted in a separate document that accompanies the SUAVE analyzer, therefore only a brief overview is presented here.

These IIR node definitions were added to the AIRE/CE class files in order to recognize SUAVE:

IIR_ChannelDeclaration
IIR_ChannelTypeDefinition
IIR_ClassAttribute
IIR_ClassWideTypeDefinition
IIR_ConcurrentProcessInstantiation
IIR_DerivedTypeDefinition
IIR_GenericPackageInstantiation
IIR_GenericSubprogramInstantiation
IIR_InterfaceChannelDeclaration
IIR_InterfaceDiscreteTypeDefinition
IIR_InterfaceIntegerTypeDefinition
IIR_InterfacePhysicalTypeDefinition
IIR_interfaceFloatingTypeDefinition
IIR_PrivateTypeDefinition
IIR_PrivateExtensionTypeDefinition
IIR_ProcessDeclaration
IIR_ReceiveStatement
IIR_RecordExtensionTypeDefinition
IIR_SelectStatement
IIR_SelectOr
IIR_SendStatement
IIR_SequentialProcessInstantiation
IIR_TagAttribute

In addition, the following IIR classes had additional data and method components added to their interface definitions.

IIR_PackageDeclaration
IIR_SubprogramDeclaration

These nodes have all been defined and added to the Scram parser. All of the C++ for the base AIRE/CE standard interface of these nodes is completed. The publish_vhdl methods have been implemented and tested, thus SUAVE libraries can be created. This task was completed without significant problems.

3.8 TASK B: Extend the SAVANT Analyzer

The SAVANT analyzer (Scram) was imported into a new archive and extended as per the SUAVE language specifications. The new analyzer parses and type checks all the test cases (about 50) that we developed. The SUAVE analyzer (version 0.9.2) is built on top of version 0.9.1 of the Scram analyzer. Since the CVS source code control system is used, new releases of Scram should easily integrate into future SUAVE releases. Version 0.9.2 of SUAVE was released at the beginning of July 1998 (http://www.ececs.uc.edu/~paw/suave).

3.9 TASK C: Extend the SAVANT Code Generator

The work to extend the SAVANT Code Generator is still ongoing. To date, we have been unable to simulate any models that are extensions of VHDL. Solution of this task boils down to designing how to write the C++ to link with TyVIS and writing the publish_cc methods to write the C++. A good start to the writing of the publish_cc methods has begun, but it is still too early to estimate a completion date. We hope to finish this task by the Spring of 1999, under other funding. For the purposes of this project, the completion of the Code Generator tasks, although desirable, was not significant to assessment of the applicability of VHDL-O constructs to designers' utilization, as we report in Section 3.11.

3.10 TASK D: Extend the QUEST Simulation Kernel

Extensions to the TyVIS simulation kernel have yet to begin. We anticipate that there will be very few changes needed to the TyVIS code, primarily in the channel queues and dynamic processes. Thus, the TyVIS kernel is essentially ready for the SUAVE extensions.

The WARPED kernel, however, does require additional capabilities before a full SUAVE system can be operational. Specifically, SUAVE requires the ability to dynamically create and destroy processes. While this generally seems an easy task to accomplish, it is somewhat complicated by the Time Warp synchronization mechanism. Specifically, because objects can execute out of order, it might be possible to request a process creation and actually receive a message for the new object before actual creation of that object. There are a number of issues to be addressed before we can begin coding that addresses this problem. We are experimenting with SUAVE and with other concurrent languages with dynamic process creation, in an attempt to ensure that WARPED is suitably modified to be sufficiently flexible for general support of dynamic process creation. We are testing three draft design solutions against several concurrent languages. These designs remain to be finalized and deselected to be operational in the WARPED kernel.

3.11 Task E: Build Demonstration VHDL-O Models

EDAptive Computing, Inc. performed this task for the SAVANT program. Our approach was first to review the SUAVE O-O extensions to VHDL, then to identify some candidate models, and finally to select one for implementation, based upon the desired goals for testing the O-O extensions within the O-O-extended SAVANT framework.

We reviewed the SUAVE extensions to VHDL as documented in Appendix A: Published Papers 1-11. SUAVE's primary goals are to aid the management of complexity in large designs and promote re-use. To achieve its goals, SUAVE employs the following extensions:

- Object-orientation
  - support definition of automatic data transfer (ADT) by extending the type system and package
  - type derivation and classes adopting "programming by extension" from Ada-95 inheritance using tagged record types
  - polymorphism using access types
- Genericity
  - subprograms and packages can have generic interface clause
  - generic interface clause includes formal types, formal subprograms and formal packages
- Abstract concurrency and communication
  - channel types, channel objects, dynamically allocated channels and message passing statements
  - process declarations and static and dynamic process instantiation

The reader is referred to Appendix A: Published Papers 1-11, for further details regarding SUAVE.

Further, we identified a set of possible test cases. Specifically, the three model candidates we considered were:

- SIMPLAN - A model of a local area network in VHDL. It was developed for the purpose of modeling the performance of a LAN, to assess the impacts of proposed LAN upgrades and additions upon network performance. Although more of an information technology application than an electronic design model, it represented a level of complexity and size representative of typical VHDL usage.
- ANTMOD - A model of a radar antenna pattern simulation, from an electronic

warfare (EW) modeling system. This was, once again, a higher-level abstraction rather than a circuit design application. It was smaller and significantly less complex than SIMPLAN, and was the same nature of model.

- GSM Network - A Global System for Mobile Telecommunications (GSM) Network, used for e-mail, fax, audio, and video applications.
- Move Machine: An ISP-level model of a simple processor.

From among these choices, we selected SIMPLAN, since it represented a real-world model of reasonable size and complexity. We did not feel that the fact that it was a higher-level performance model rather than a circuit model would deter from our objectives to assess the impacts of the O-O extensions.

Having selected our model, we set about implementing it in three steps:

1. First, we obtained a good understanding of the SIMPLAN model. In performing this activity, more time was required to comprehend the SIMPLAN model than we originally expected, due to a lack of documentation. We subsequently documented the model to enable easy comprehension and reuse.
2. Next, we studied and identified several areas of the model where the O-O SUAVE extensions could be applied for reduction of complexity and enabling reuse. These usability factors were our primary focus, rather than execution performance. The decision to focus in this manner was a decision we reached in concert with AFRL.
3. Finally, we modified SIMPLAN using the SUAVE O-O extensions to SAVANT.

This resulted in a model we could use for evaluation, as we report in the next section.

3.12 Task F: Evaluate performance impact of the object-oriented extensions

Under this task, we first established measures of effectiveness, or *metrics* by which the performance of the O-O extensions could be assessed, then implemented a test configuration to perform the assessment, and finally performed the assessment. Here, we detail the metrics, test configuration, and the assessment results.

3.12.1. Metrics Selection: We identified the following measures of effectiveness, or metrics, for our O-O extension evaluiations:

- Reuse
- Cohesion
- Coupling
- Interface Complexity

- Lines of Code

We elaborate on these metrics below. However, before we describe them, we explain the key concepts utilized in definition of these metrics. These key concepts include *modularity, invocation, control flow*, and *abstract nodes*,[8,9] as we discuss below.

Modularity(module): We believe that there are three levels of modularity in a HDL description. The highest level of modularity exists at the entity-architecture level. The entity-architecture modules, in turn, consist of concurrent process modules, which in turn consist of sequential procedure/function level modules.

Invocation: Invocation can be easily defined for a sequential module since we can easily deduce who invokes who by looking at the HDL design. Sequential modules are invoked either by another sequential module or by a concurrent process module. However, the concept of invocation cannot be defined in a conventional sense for concurrent modules. We propose that a concurrent process module, e.g. *A*, should be considered as invoked by another concurrent process module, e.g. *B*, if *A*'s sensitivity list includes a signal generated by *B*. Further, we propose that an entity-architecture module, e.g. *A*, is invoked by another, e.g. *B*, if *B* instantiates *A*.

Control Flow: Again, control flow is easy to identify in a sequential module since it is easy to identify flags in a procedure/function. We propose that we again use sensitivity lists to identify the controlling and controlled processes. In addition, if a signal/variable generated by one is used by another to make a decision, we will count that as a control flow. Further, we propose that we employ ports in an entity declaration that are directly or indirectly (through signal associations) used to make control decisions as control flow.

Abstract Nodes: We believe that either an entity-architecture pair or a process or a procedure/function can be considered as an Abstract Node.

The following summarizes the metrics, their significance, and how we utilized them in our assessment of the SUAVE O-O extensions.

Lines Of Code (LOC): This is the most straightforward of all metrics. We evaluated this metric in the VHDL and its equivalent SUAVE O-O models by comparing the lines of VHDL/SUAVE O-O code. LOC is easily measurable and is a good indicator of complexity. The higher LOC would imply higher complexity, whereas lower LOC would imply lower complexity.

Reuse: The Reuse metric reflects the number of times a module is accessed and is calculated based on the number of unique invocations for each module. Specifically, the reuse formula for software is as follows:

$$REU = \sum_{m}(s-1)$$

where

m = the total number of modules, and

s = the total number of module invocations.

A higher reuse number implies a design that is simpler, with higher reusability and lower costs.

We evaluated the reuse metric for procedures/functions, but not for processes and entity/architecture modules because: (i) processes are not reused in a VHDL design (SUAVE does allow such reuse, but we were unable to use the SUAVE extensions for the test case considered and therefore did not feel a need to measure this metric), and (ii) we kept the structure of our reimplemented test case the same by design. Therefore, there was no difference between VHDL and SUAVE-extended VHDL designs with respect to use of entity-architecture modules.

Interface Complexity: The Interface Complexity reflects the magnitude of control flows in relation to modules in a software design. Specifically, the Interface Complexity metric formula is as follows:

$$IC = (f - n + 2)$$

where,

f = the total number of control flows, and

n = the total number of modules.

Interface complexity is a measure of the complexity of design and focuses on interface aspects of the design. This measure would be applicable to all types of modules.

However, due to lack of automated tools, a large design would require substantial manual effort to measure this metric and is prone to errors. To simplify our task, we only measured the number of ports among various modules without regard to whether the port constituted a control flow. The number of ports, we believe, gave a preliminary indication of complexity. A lesser number of ports would imply less complexity.

Cohesion: The Cohesion metric reflects the goodness of functional partitioning of the design under consideration, and is calculated by examining collections of abstract nodes and their interaction with global data. Specifically, the Cohesion metric is calculated as follows:

$$InternalCohesion = \sum_{abstractnode}(InternalStrength)$$

where,

Internal strength per Abstract Node = the sum of global variables referenced by this node and not referenced by any other node.

$$ExternalCohesion = \sum_{abstractnode} ExternalStrength$$

where,

External strength per Abstract Node = the sum of global variables referenced by this node and referenced by any other node.

A high value for external cohesion indicates poor partitioning, whereas a low value for internal cohesion indicates poor partitioning. The comparison of these two values for cohesion determines the relative distribution of system functionality.

We did not measure this metric due to lack of automated measurement tools and, since the structure of the design was kept intact, the cohesion among the modules did not change significantly.

Coupling: The Data Coupling metric reflects the strength of the interconnection or dependence between modules. Specifically, the Data Coupling metric is calculated as follows:

$$CO = \sum_{invocations} C_M * I_{cc}$$

where,

C_M = the maximum coupling number per model determined based on number, type and scope of parameters [1], and

I_{CC} = the number of interconnections between pairs of data processes.

We did not measure this metric due to lack of automated measurement tools and, since the structure of the design was kept intact, the coupling among the modules did not change significantly.

We chose the suggested metrics for comparing the VHDL model with the equivalent SUAVE model. The metrics were selected due their ability to give a preliminary indication of complexity and reuse characteristics of the design. The choice was also influenced by the feasibility of what could be easily accomplished with the resources available.

3.12.2 Test System Implementation: The majority of this task was spent on understanding the SimPLAN model. A good understanding was required in order to develop a test system as proposed. We summarize our understanding of the model here. The reader is also referred to a VHDL International User's Forum (VIUF) paper,[10] which provides an overview of the model. Furthermore, we describe the test method below.

Model Understanding: The VHDL description models a LAN at performance-level for the purpose of capacity planning. This is a model of an actual LAN and therefore the physical structure of the LAN is reflected in the structure of the model. The model consists of three major components: (i) network stimulants, namely clients and servers, (ii) network consisting of physical components such as ethernet segments, Kalpana switches, Routers, and FDDI backbone, and (iii) simulation coordinator, that provides the interface between the stimulants and the network. The model is shown in Figure 16.

Test Method: The testing method was simple. After gaining a good understanding of the model and its components, we identified portions of the model that would qualify for reimplementation using SUAVE extensions. However, the structure of the model was kept intact, since our goal was to replicate the same functionality as the original VHDL model but use SUAVE extensions to do so wherever possible. And, since the structure of the model was important to capture the physical structure of the LAN, we kept it intact.

We then reimplemented the original model using SUAVE extensions and manually measured the metrics as defined earlier in Section 3.12.1. Lack of automated tools to measure the metrics hampered our ability to do a good comparison between models. Further, our desire to keep the same structure as the original model hampered our ability to fully exploit the strengths of SUAVE. We believe SUAVE extensions would enhance the reuse and reduce the complexity

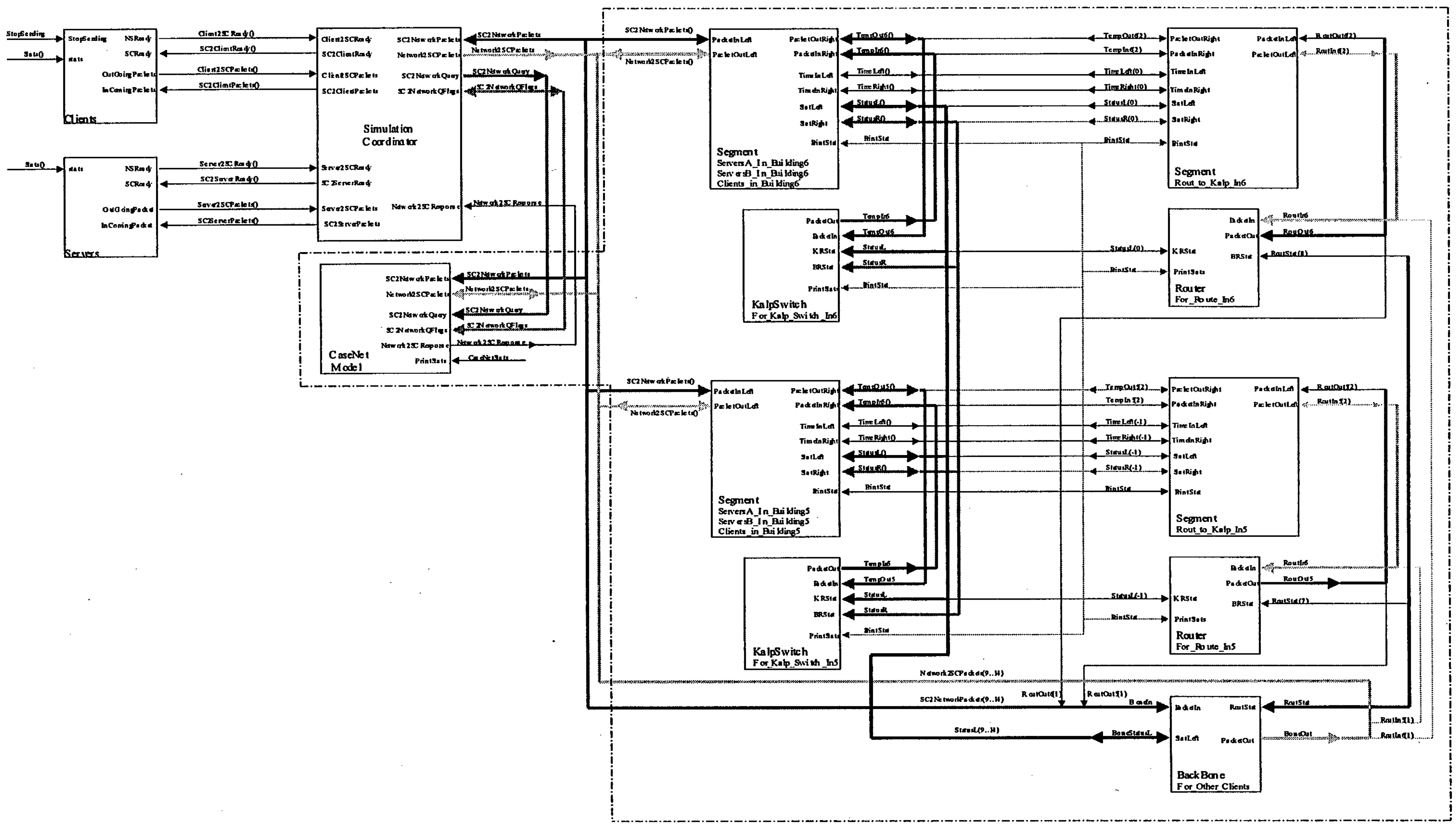

Clients
Servers
Simulation Coordinator
CaseNet Model
Segment ServersA_In_Building6 ServersB_In_Building6 Clients_in_Building6
KalpSwitch For_Kalp_Swith_In6
Segment ServersA_In_Building5 ServersB_In_Building5 Clients_in_Building5
KalpSwitch For_Kalp_Swith_In5
Segment Rout_to_Kalp_In6
Router For_Route_In6
Segment Rout_to_Kalp_In5
Router For_Route_In5
BackBone For Other Clients

Figure 16. SimPLAN Model

greatly if we could reimplement the model at a higher level of abstraction. Lack of resources did not permit us to do so. Figure 17 illustrates our test method.

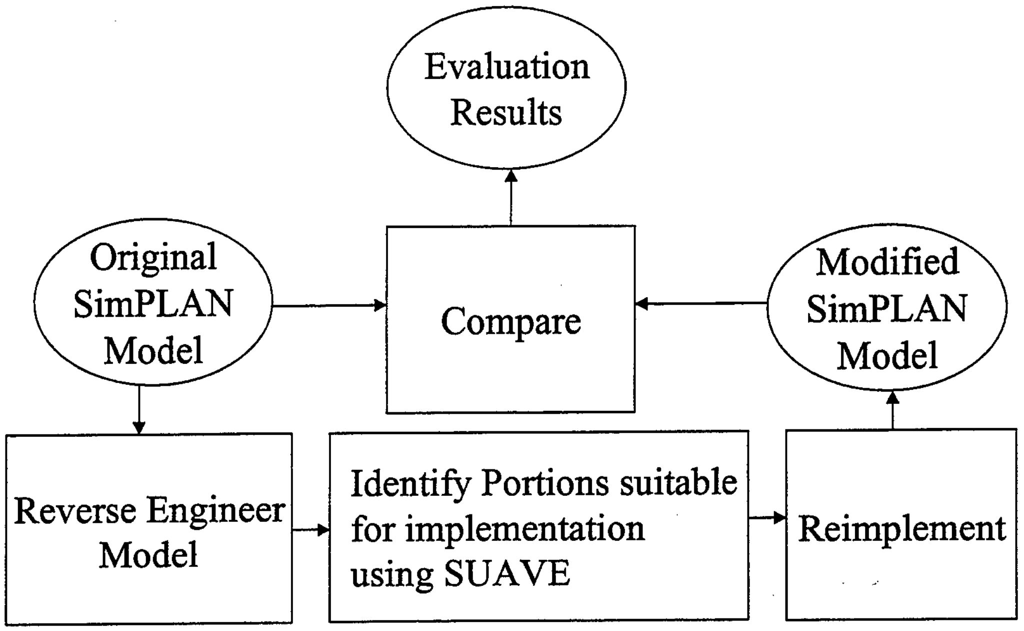


Figure 17. Test Method Used.

3.12.3. Assessment Results

As part of this task, the SUAVE extensions were used to re-implement the benchmark and the metrics were manually measured. The results of the metrics measurement are shown below:

Table 3. Metrics Assessment.

Metric	VHDL Model	SUAVE Model
Lines Of Code	4260	3358
Reuse	70	72
Interface Complexity	88	76
Cohesion	*	*
Coupling	*	*

* SimPLAN would require complete re-design using O-O extensions to realize gains for these metrics

From the results, we can conclude that use of SUAVE extensions led to reduction in size of the model, reduction in interface complexity and increase in reuse of procedures/functions. We believe the results can be explained easily by the following:

1. Use of inheritance and generic types led to reductions in the size of the model. A variety of queues used by the model were reimplemented utilizing SUAVE extensions for inheritance and generic types.
2. Use of inheritance and generic types led to higher reuse.
3. Use of channels led to a reduction in the number of control signals (ports) required to interact between various components of the model. Specifically, coordination required between network stimulants and the network was eased by the use of channels.

The limited test results that we were able to obtain manually indicated that use of SUAVE extensions can reduce the complexity and increase the reuse potential of the model. Further, we believe that if the SimPLAN model could be developed from scratch with the same analysis goals but no constraints on the structure, the resulting model would be greatly reduced in complexity and would have much higher reuse. Specifically, information about the structure of the LAN is contained in the Simulation Coordinator component. We believe that we could completely eliminate the Simulation Coordinator component, as shown earlier in Figure 16, with the use of SUAVE extensions such as Channels and dynamic process creation and deletion. However, that would require us to embed the LAN structure information in other components of the model. This would require a complete overhaul of the design. The effort required to do so was more than was permitted by the resources available.

In spite of the limitations of this study, we believe the results indicate that SUAVE extensions to VHDL are promising and can lead to reduction in complexity and an increase in the reuse potential of designs. With the fast growth of the Intellectual Property (IP) market, the reuse potential of the IP will be increasingly significant. We believe that SUAVE extensions could help in enabling the IP reuse.

3.13 Summary of the Results

In summary, we produced the results that we planned for the SAVANT program and for the Option Task, although not necessarily to the degree of maturity we would have liked in all cases. The core SAVANT components (Analyzer, IF, Code Generator) have undergone several preliminary releases and are fairly mature. The Debugger and the VTK are somewhat less mature. The O-O extensions were implemented to the degree necessary to support their assessment. That assessment showed that the O-O extensions were useful in reducing complexity and promoting reuse.

4.0 CONCLUSIONS

In conclusion, we regard SAVANT as a successful Phase II development program. Under this program and its Option Task, we achieved most of the objectives we defined at program inception.

For the Baseline Program, we developed and released on the WWW throughout the program, versions of the Analyzer, the IF, and the Code Generator, as well as TyVIS and WARPED simulations. In our final release, conformance to acknowledged test standards is high, and others are successfully downloading SAVANT and creating useful simulations. We also developed a debugger and some visualization tools (the VTK). These appear to be useful as well, but are not as mature as the SAVANT core components. Our commercialization paradigm was not vigorously attacked, since MTL's business model diverged from the EDA products area during this time period.

For the Option Task, we implemented O-O extensions in the Analyzer, but did not fully implement them in either the Code Generator or the Simulation Kernel. However, this implementation work will continue at UC under other programs. Using the implementation we did accomplish, we developed models and assessed their performance.

Overall, we find the SAVANT technology to be a useful one, and await its more widespread use among the VHDL/EDA community. SAVANT software for UNIX platforms is available at http://www.mtl.com/projects/com, and is currently free for educational institutions and non-commerical applications. MTL is releasing SAVANT under the GNU Public License.

5.0 RECOMMENDATIONS

Our recommendations follow from the level to which we were able to complete our development, and from our experiences under this program. Basically, we recommend that SAVANT be utilized and tested under several design projects, and compared more rigorously to present, commercial VHDL design suites. We also recommend that the O-O extensions be implemented in a more final form, and made available to the community, much as the core SAVANT technology is being proliferated.

6.0 REFERENCES

[1] Hirsch, Herb, et. al. "Solid State Electronics Directorate Applied Research," SAVANT Phase I SBIR Proposal, MTP94-010, MTL Systems, Inc., January, 1994.

[2] Chawla, Praveen, Herb Hirsch, Hal Carter, and Wilsey, SAVANT Phase I Final Report, MFR95-006, MTL Systems, Inc., April 1995.

[3] Hirsch, Herb, et. al. "SAVANT Phase II," SBIR Proposal, MTP95-016, MTL Systems, Inc., January, 1995.

[4] Ashenden, Peter J. *Designer's Guide to VHDL.* Morgan Kaufman Publishers, 668 pp., December 1995.

[5] Wilsey, Philip, et. al. "Preliminary Investigations and Feasibility Study of Parallel Simulation of Digital Systems," University of Cincinnati, October 1998

[6] P. J. Ashenden, P. A. Wilsey, and D. E. Martin, "Suave: Extending VHDL to Improve Modeling Support," IEEE Design and Test of Computers, April-June 1998.

[7] Brooks, Frederick P. Jr. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley Publishing Co., 322 pp., July 1995.

[8] P. J. Ashenden and P. A. Wilsey, "Considerations on Object-Oriented Extensions to VHDL," VHDL Users' Group Spring 1997 Conference, 109-118, March 1997.

[9] J. C. Willis, P. A. Wilsey, G. D. Peterson, J. Hines, A. Zamfirescu, D. E. Martin, and R. N. Newshutz, "Advanced Intermediate Representation with Extensibility (AIRE)," VHDL Users' Group Fall 1996 Conference, 33-40, October 1996.

[10] Chawla, P., W. Zhou, and H.L. Hirsch. "Performance Modeling and Analysis of a LAN using VHDL." Proceedings of the VIUF. Fall 1993.

APPENDIX A: Published Papers

This appendix lists several papers published by the SAVANT Research and Development Team, as well as ancillary or related papers to the project.

1. P. J. Ashenden and P. A. Wilsey, "Principles for Language Extensions to VHDL to Support High-Level Modeling," VLSI Design. (submitted).
2. P. J. Ashenden and P. A. Wilsey, "Extensions to VHDL for Abstraction of Concurrency and Communication," Proceedings of the Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '98), July 1998.
3. P. J. Ashenden, P. A. Wilsey, and D. E. Martin, "Suave: Extending VHDL to Improve Modeling Support," IEEE Design and Test of Computers, April-June 1998.
4. P. A. Wilsey, D. E. Martin, and K. Subramani "SAVANT/TyVIS/WARPED: Components for the Analysis and Simulation of VHDL," VHDL Users' Group Spring 1998 Conference, 1998.
5. P. J. Ashenden, P. A. Wilsey, and D. E. Martin, "Considerations on System-Level Behavioural and Structural Modeling Extensions to VHDL," VHDL Users' Group Spring 1998 Conference, 1998.
6. P. J. Ashenden and P. A. Wilsey "A Comparison of Alternative Extensions for Data Modeling in VHDL," 31th Hawaii International Conference on System Sciences (HICSS-31), January 1998.
7. P. J. Ashenden, P. A. Wilsey, and D. E. Martin, "Reuse Through Genericity in Suave," VHDL Users' Group Fall 1997 Conference, 170-177, October 1997.
8. P. J. Ashenden, P. A. Wilsey, and D. E. Martin, "Suave: Painless Extension for an Object-Oriented VHDL," VHDL Users' Group Fall 1997 Conference, 60-67, October 1997.
9. P. J. Ashenden, P. A. Wilsey and D. E. Martin, "Suave: A Proposal for Extensions to VHDL for High-Level Modeling," Joint Technical Report, TR-7/97, Dept. Computer Science, University of Adelaide and TR-207/08/97/ECECS, Department of Electrical & Computer Engineering and Computer Science, University of Cincinnati, August 1997.
10. P. J. Ashenden and P. A. Wilsey, "Principles for Language Extension to VHDL to Support High-Level Modeling," Joint Technical Report TR-03/97, Dept. Computer Science, University of Adelaide and TR-204/05/97/ECECS, Department of Electrical & Computer Engineering and Computer Science, University of Cincinnati, May 1997.
11. P. J. Ashenden and P. A. Wilsey, "A Comparison of Alternative Extensions for Data Modeling in VHDL," Joint Technical Report TR-02/97, Dept. Computer Science, University of Adelaide and TR-203/05/97/ECECS, Department of Electrical & Computer Engineering and Computer Science, University of Cincinnati, May 1997.
12. P. J. Ashenden and P. A. Wilsey, "Considerations on Object-Oriented Extensions to VHDL," VHDL Users' Group Spring 1997 Conference, 109-118, March 1997.
13. J. C. Willis, P. A. Wilsey, G. D. Peterson, J. Hines, A. Zamfirescu, D. E. Martin, and R. N. Newshutz, "Advanced Intermediate Representation with Extensibility (AIRE)," VHDL Users' Group Fall 1996 Conference, 33-40, October 1996.

14. D. E. Martin, P. A. Wilsey, and P. Chawla, "SAVANT: An Extensible Object-Oriented Intermediate for VHDL," VHDL Users' Group Spring 1996 Conference, 275-281, March 1996.

15. P. A. Wilsey and D. E. Martin, "Coordinating Joint Cost/No-Cost Rights for Software Developed with SBIR Funding," First Conference on Freely Redistributable Software, 89-94, February 1996.

APPENDIX B: WWW Sites

Additional information on SAVANT may be found at the following sites on the WWW:

http://cs.adelaide.edu.au/users/michael/publications.html

http://web.cs.ualberta.ca/~zhang/vhdl.html

http://dragon.ics.es.osaka-u.ac.jp/LINK.HTM

http://ece.uc.edu/~paw/tyvis/

http://ece.uc.edu/~paw/warped

http://ece.uc.edu/~ramanan/research/research.html

http://ececs.uc.edu/~hcarter/questII.html

http://ececs.uc.edu/~kbse/hepe

http://ececs.uc.edu/~paw/lab

http://ececs.uc.edu/~paw/suave

http://goethe.ira.uka.de/~schneider/other_tools/

http://hpcmo.hpc.mil/ug97/tpapers/mchung/index.htm

http://ftlsystems.com/aire/

http://mtl.com/projects/savant/

http://ececs.uc.edu/~paw/hpc/main.pdf