AUTOMATED REAL-TIME SOFTWARE DEVELOPMENT

Denise R. Jones
NASA Langley Research Center
Hampton, VA 23681

Carrie K. Walker
NASA Langley Research Center
Hampton, VA 23681

John J. Turkovich
The Charles Stark Draper Laboratory, Inc.
Cambridge, MA 02139

ABSTRACT

A Computer-Aided Software Engineering (CASE) system has been developed at the Charles Stark Draper Laboratory (CSDL) under the direction of the NASA Langley Research Center. The CSDL CASE tool provides an automated method of generating source code and hard copy documentation from functional application engineering specifications. The goal is to significantly reduce the cost of developing and maintaining real-time scientific and engineering software while increasing system reliability. This paper describes CSDL CASE and discusses demonstrations that used the tool to automatically generate real-time application code.

INTRODUCTION

Advanced flight vehicles rely heavily on software to accomplish their missions. The cost of designing, developing, testing, and maintaining avionics software (vehicle guidance, navigation, and control) is becoming an increasingly larger part of total vehicle cost. Until recent years, the lack of appropriate tools to aid in the creation of software has led to nonuniform development techniques and software that is difficult to maintain, and is either unreliable or very expensive to verify. Computer-aided software engineering (CASE) is a technology aimed at automating the software development process in order to produce cost effective and reliable software systems.

One of the major problems associated with the development of real-time scientific and engineering software, aside from general software development issues, is communication among the developers. Typically guidance, navigation, and control (GN&C) algorithmic designs are created by engineers with formal educations in aerospace, mechanical, electrical, or other pure engineering disciplines, rarely from a software engineering discipline. These engineers use languages that are indigenous to their disciplines to specify and communicate their designs. These specifications are presented to software engineers who must interpret them, turn them into a software specification and, subsequently, flight code.

GN&C engineers are not programmers and vice-versa. Consequently, they speak different technical languages. Communication between these groups concerning avionics designs is difficult and prone to error. Usually, programmers must confer with engineers extensively before a software specification adequately represents algorithms as conceived by the engineers. One solution to this communication problem is to educate GN&C engineers to cope with a diverse set and growing number of programming issues. Most engineers these days can program to some extent, but to expect them to perfect the art to the degree that programmers have is unreasonable. Another solution is to educate programmers to be able to design GN&C algorithms or more readily assimilate engineers' GN&C algorithms, which are becoming more diverse and complex. This solution is also impractical. Even programmers who have been developing flight code for years, do not truly comprehend the many static and dynamic interactions that must occur for a flight vehicle to carry out its mission.

Problems are magnified when changes to the code are needed. Proper configuration management practices dictate that changes should be made first to the engineer's design, then to the software specification, and finally to the code. Unfortunately this is not always practiced, and documents become inconsistent and incorrect. A tool is needed that automatically transforms the design specifications developed by GN&C engineers into compilable flight code and

the associated system documentation. A Computer Aided Software Engineering (CASE) tool being developed by the Charles Stark Draper Laboratory (CSDL) for the NASA Langley Research Center is such a system.

This paper discusses approaches to software development and defines CASE in general terms. The CSDL CASE software development tool is described and several demonstrations that used the tool are discussed.

APPROACHES TO SOFTWARE DEVELOPMENT

There are several methods of developing software systems, as shown in figure 1. Early software efforts (fig. 1a) were, for all practical purposes, manual. Objectives and algorithms were communicated via combinations of verbal instructions, hand written notes, and, at best, typed documents.

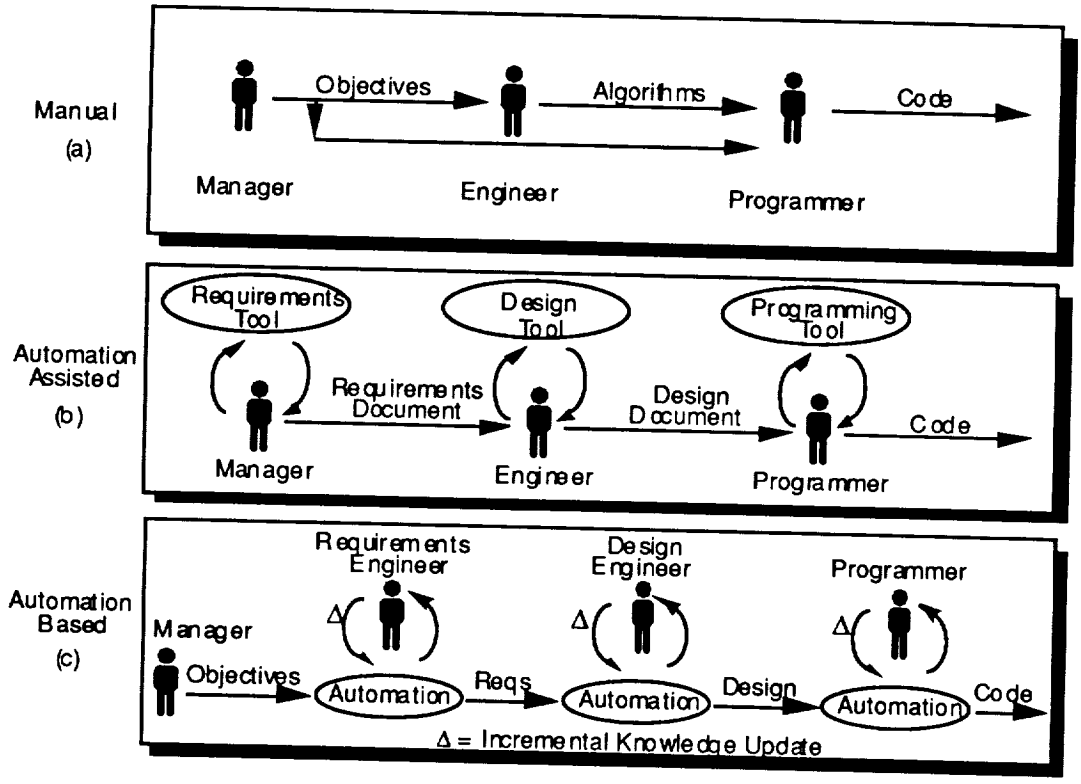

Figure 1. Software development methods.

As expectations, confidence, and perceived flexibility of digital computation grew, so did the notion of computer programs grow into the notion of software and the art of developing these programs into the science of software engineering. Today software development, because of growing size and complexity, is viewed largely as a managerial problem. NASA, DOD, and industry have established standard methodologies for designing, developing, and maintaining software. To support established methodologies, tools that assist the software development process through automation have been and are being developed (fig. 1b).

In order to achieve significant reductions in software costs it is necessary to treat the software problem not merely as a managerial problem but as a technological problem. By doing so, the software development process becomes automation based, as opposed to automation assisted. This automation based system is supported with knowledge acquired by experts as they interact with the system in operation. In this way, software development leverages accumulated knowledge that can be reused from application to application. Instead of people acting as bottlenecks in a flow to analyze functionality (fig. 1b), they serve to fine-tune accumulated knowledge as appropriate on new projects. Notice in figure 1c that people have been removed from the mainstream of software development. The primary function of the mainstream process is to automatically apply accumulated knowledge and secondarily to incrementally acquire knowledge. CSDL CASE takes this approach.

COMPUTER-AIDED SOFTWARE ENGINEERING

Computer-aided software engineering (CASE) is a software technology that began in the early 1980s. CASE is aimed at automating the software development process to improve software productivity and quality. The basic concept is to provide a set of well-integrated software tools and methodologies that automate the entire software development life cycle from analyzing application requirements to maintaining the resulting software system [1].

Hundreds of CASE tools have emerged over the last decade. Many of these tools are workstation-based using graphics and windowing capabilities to interact with the software developer. CASE tools automate a variety of software development and maintenance tasks such as creating structured diagrams and pictorial system specifications, generating executable code and system documentation, and error checking. Generally a CASE tool covers only a portion of the software life cycle. Most CASE tools also support one or more software development methodology, such as Yourden's structured design [2] or DeMarco's structured analysis [3]. These methodologies use notations that are composed of diagrams, graphs, charts, tables, and formal languages.

CASE offers many enhancements to the software development process. Most of these benefits are a result of the automated environment that CASE provides through an interactive graphic user interface. Some of these benefits are as follows [1]:

- enforces software/information engineering,
- makes prototyping practical,
- frees developer to focus on creative part of software development,
- enables reuse of software components (prototypes, data, code, functional designs),
- simplifies program maintenance and reduces maintenance costs,
- reduces software development time and cost,
- improves software reliability and quality, and
- increases productivity.

The introduction of CASE technology has changed the software development process. Traditional development emphasized the later phases of the software life cycle, with 65 percent of the effort placed on the coding and testing phases [1]. CASE automates much of the latter part of the software life cycle so more time can be spent on analysis and design. Table 1 shows the differences between traditional and automated software development [1].

Traditional Development	Automated Development
Emphasis on coding and testing	Emphasis on analysis and design
Paper-based specifications	Rapid iterative prototyping
Manual coding	Automatic code generation
Manual documenting	Automatic document generation
Software testing	Automated design checking
Maintain code	Maintain design specifications

Table 1. Differences between traditional and automated software development.

CHARLES STARK DRAPER LABORATORY CASE TOOL

Background

The foundation of the Charles Stark Draper Laboratory computer-aided software engineering (CSDL CASE) system was developed under the Draper Laboratory's internal research and development (IR&D) program [4]. It was supported by the Advanced Launch System (ALS) Advanced Development Program under the direction of NASA Langley Research Center from 1988 - 1989 and was known as ALS CASE [5]. The advancement of CSDL CASE continues through the sponsorship of the NASA Langley Research Center and the Draper Laboratory Corporate Sponsored Research (CSR) program.

System Description

Although there are many CASE tools available commercially, most support general purpose systems development. Those that do support real-time scientific and engineering software applications generally do so by providing

"extensions" to the notations used to support the general purpose systems development. CSDL CASE supports real-time scientific and engineering software systems development from an engineering perspective.

CSDL CASE views software design, development, and maintenance from the viewpoint of the application engineer [6]. The engineering design or functional specification *is* the software specification from which the source code is automatically produced. The functional specification is input into the CSDL CASE system in the form of hierarchical engineering block diagrams and algebraic equations, notations familiar to avionics and control systems designers. The resulting source code is then maintained by modifying the specification through the block diagrams, therefore, eliminating the need to maintain functional specifications, source code, and documentation separately. This approach enforces consistency, eases maintenance, and increases productivity and reliability since modifications are made solely to the functional specification through an interactive graphical user interface with the software system generated automatically.

The architecture of the CSDL CASE tool is shown in figure 2 [7]. This architecture, which is implemented on an engineering workstation platform, consists of a user interface, an automatic software designer, automatic code generators, and an automatic document generator. The user interface facilitates specification of algorithms and systems of algorithms as engineering block diagrams. The automatic software designer converts the graphical specifications into a machine-dependent design. Code generators automatically apply the syntax of the target language to the software design in order to produce source code. The document generator automatically constructs and prints hard copy documentation of the specification according to a specified format. The following sections describe this architecture in more detail.

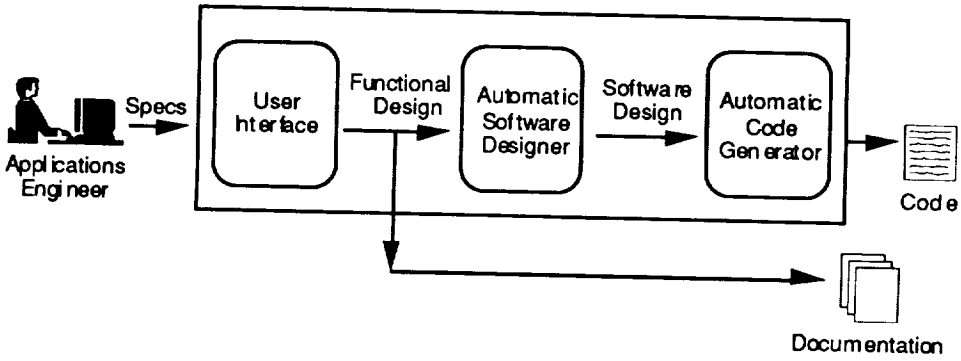

Figure 2. CSDL CASE system architecture.

User Interface

The user interface was designed with the purpose of specifying systems of engineering algorithms as opposed to software designs. This specification emphasizes the functionality and interaction of algorithms in order to convey their meaning to other engineers. Software designs by contrast do not clearly convey the meaning of algorithms and systems of algorithms. Instead, software designs dominantly reflect constraining characteristics of the execution environment such as computational resources, memory resources, input/output resources, and resource connectivity.

Systems of algorithms are specified through extensive use of engineering block diagrams [8]. The computational aspects of a diagram are referred to as "transforms" since they explicitly transform inputs into outputs with no hidden side effects. The data aspects of a diagram are "signals" that carry information from one transform to other transforms. Hierarchies of both transforms and signal types can be built either bottom-up or top-down. For bottom-up design, predefined sets of building blocks for both transforms and signal types are supplied. In the case of transforms, these are called primitive transforms and are comprised of such things as add, subtract, multiply, divide, absolute value, switch, etc. For signal types, these are called predefined types and include integer, float, character, string and boolean. The user can also create his own signal types such as arrays and records. For top-down design, the engineer needs only to specify the input and output characteristics of a transform before using it in a block diagram. The details of the transform's data flow and processing can be deferred until a later time, or a body of existing code can be referenced rather than automatically generating code.

A sample CSDL CASE user interface window is shown in figure 3. The window consists of many subwindows, called panes. The center pane can display either a block diagram or a description of a signal type. The three panes across the top of the window display the name, title, and type of the block diagram that is being edited. The

additional three panes located at the bottom of the window display, from left to right, update messages, a menu of major specification options, and detailed responses to the selection of a major menu option.

Figure 3 also gives an example of an engineering block diagram specification. The diagram is a graphical representation of functionality that is captured in a centralized, object-oriented knowledge base as a result of developing the diagram. The diagram is created by selecting and placing primitive transforms, user-defined transforms, input terminals, output terminals, and constant value terminals and then connecting them with signals. A transform is selected from a pop-up menu (see figure 3) and the resulting graphical representation is placed on the diagram. Input, output, and constant terminals are selected and placed by again calling up a pop-up menu, specifying detailed definitions for them, and then positioning them in the diagram.

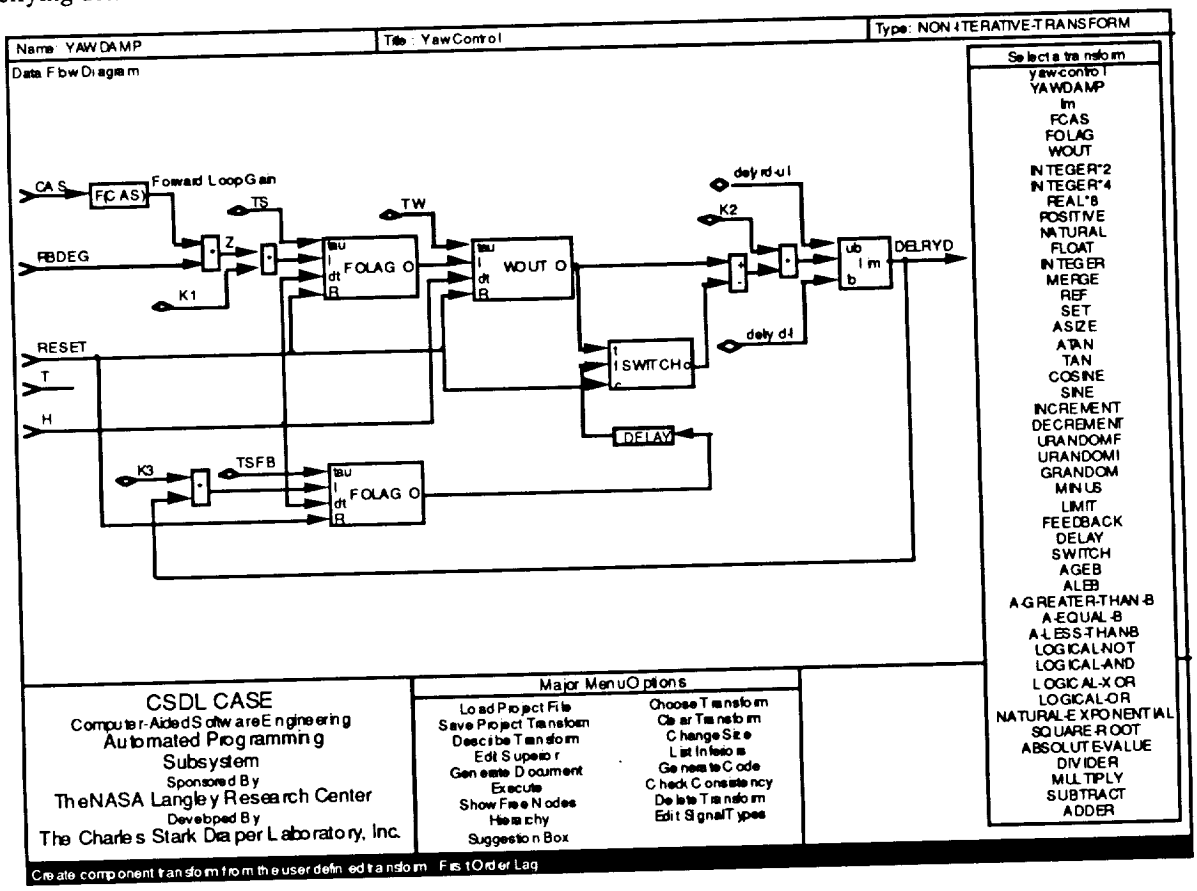

Figure 3. CSDL CASE user interface window.

Automatic Software Designer

The automatic software designer takes as input the object-oriented, functional form of the specification and determines a generic, procedural form which takes into consideration characteristics of the execution environment. During the automatic design process, engineering diagrams are converted into procedures, functions, or in-line code depending on their usage. Each transform in a diagram becomes either a statement or a block of statements. A statement consists of an assignment to one or more variables or a call to a procedure or function. Input, output, and constant terminals are converted into their respective classes of variables. The automatic designer also generates variables when it is necessary to implement state. State variables appear as a result of feedback loops in the graphical specification. In addition, local variables may be needed to reflect the connectivity in a diagram; these variables are also generated by the software designer. Connectivity is also used to determine the execution order of statements. Traversals of diagram connectivity from both outputs to inputs and inputs to outputs are performed to determine which statements must execute in sequence, which may execute in parallel, and which are conditionally executed.

Automatic Code Generator

Ada and C source code generators have been developed. Each automatic code generator takes as input the generic, procedural form of a software design produced by the automatic software designer and creates source code in the syntax of the corresponding target language, while exploiting the functionality of the chosen language.

The generated Ada code is hierarchically structured and contains one Ada function and procedure for each function and procedure in the software design format. Since C is not a hierarchical language and recognizes only function program blocks, the C generator flattens the definition hierarchy and creates functions for both procedures and functions in the software design.

Automatic Document Generator

The documentation process consists of three distinct steps: generation, formatting, and printing. The document generator takes as input the same object-oriented, functional representation of the specification used to generate a software design. The generation process creates both the text and graphics for the title page and body of the document which reflects the block diagram specification. Although the generation is performed in CSDL CASE, the formatting and printing is performed on an engineering workstation using a commercial publishing software package. The final product is a fully collated document which includes a title page, table of contents, list of figures, sections and subsections, appendices, and an index. The entire document is composed and assembled with no manual intervention.

APPLICATIONS OF CSDL CASE

CSDL CASE has been used on both small and moderate size applications [9]. The small applications are a Boeing 737 yaw damping system, a Martin Marietta Lateral Acceleration Sensing System for the Titan IV, and a General Dynamics electromechanical actuator model. The moderate-sized applications are an autoland control system for the Boeing 737 aircraft, an autonomous exploration vehicle simulation, and a guidance and control system for a planetary lander. Each of the small applications resulted in the automatic generation of hundreds of lines of code and tens of pages of documentation. Each of the moderate size applications generated thousands of lines of code and hundreds of pages of documentation. The 737 autoland system is described below as a representative CSDL CASE application.

Boeing 737 Autoland System

Ada code and documentation for a Boeing 737 autoland flight control system [10] were generated using CSDL CASE. The CSDL CASE specification for this control system was reverse engineered from a FORTRAN implementation and documentation for that implementation. This documentation contained both block diagrams and flowcharts. The autoland system controls pitch, roll, yaw, and throttle for the B737 aircraft from about 5000 feet altitude until touchdown. Ada code and documentation were automatically generated using CSDL CASE [11], resulting in approximately 3000 lines of Ada code and 200 pages of documentation.

Execution of the Ada code was tested against execution of the FORTRAN code. The objective of this experiment was to determine deviations in the outputs of the automatically generated Ada code and the manually generated FORTRAN code. The FORTRAN implementation had previously undergone extensive testing. Tests were structured so that all paths within the design would be exercised. A test was conducted by subjecting FORTRAN code to a time varying set of inputs as it executed. Both inputs and outputs were recorded. The Ada code was then executed, using the recorded FORTRAN inputs. Outputs were compared to the FORTRAN outputs.

Thirty-three tests were performed. Eleven discrepancies were detected. Nine discrepancies were traced to the CSDL CASE user; these were errors in the block diagrams that had been specified to the automatic programming system. Two discrepancies were traced to errors in the FORTRAN code. No errors were traced to either the automatic software designer or the automatic Ada code generator.

As a byproduct of this activity, a strategy for testing code that is based on engineering block diagrams was developed [11]. This strategy prescribes a method for designing tests that cover the functionality expressed by block diagrams while minimizing the number of tests. The test design is dependent on the types of transforms in a block diagram and their connectivity.

As an extension of this activity, the Ada code was regenerated and targeted for a flight critical computer system. The objective of this demonstration was to show that the code produced with CSDL CASE is not only suitable for execution on a general purpose computer for prototyping or simulation purposes, but that the code is also appropriate for a typical flight critical architecture. The architecture used for this demonstration was the Advanced Information Processing System (AIPS) [12] being developed at the Draper Laboratory for NASA Langley.

AIPS is a set of hardware and software building blocks that can be configured in various ways to form fault-tolerant distributed computer system architectures. An AIPS configuration can be used for a broad range of applications, including highly-reliable flight critical applications. An AIPS node can consist of a single Fault Tolerant Processor (FTP), duplex FTPs, or triplex FTPs, depending on the function criticality and needed reliability. A node can even contain one or more Fault Tolerant Parallel Processors (FTPPs), if large capacity throughput is needed for the application. Hardware fault detection provides low fault tolerance overhead. System services are provided by an operating system written in Ada. The system complexity is hidden from the applications; therefore, the applications developer does not have to consider redundancy during application development.

The objective of this demonstration was met. The feasibility of using unaltered CASE-generated code on a flight critical architecture was demonstrated. First, enhancements were made to CASE to allow the AIPS FTP to be one of the architectures available for code generation. This enhancement entailed altering the automatic code generator to reflect AIPS system dependencies, particularly to include the appropriate math library instantiations and communication routines.

The steps involved in the actual demonstration are illustrated in figure 4. Specifications for the Boeing 737 autoland system were entered into CSDL CASE using the interactive graphical interface (see figure 3). Ada source code targeted for the AIPS FTP and detailed documentation were generated. The Ada code was compiled on a Digital Equipment Corporation MicroVax using the XDAda compiler [13] and linked with the AIPS operating system. The automatically generated code was successfully executed on the AIPS FTP while interacting with a dynamic simulation of the Boeing 737 executing on the MicroVax. Proper performance was verified visually during execution by a Macintosh display of aircraft state. The displays graphically depicted the aircraft trajectory, attitude, and attitude rates. These displays are shown in figure 5.

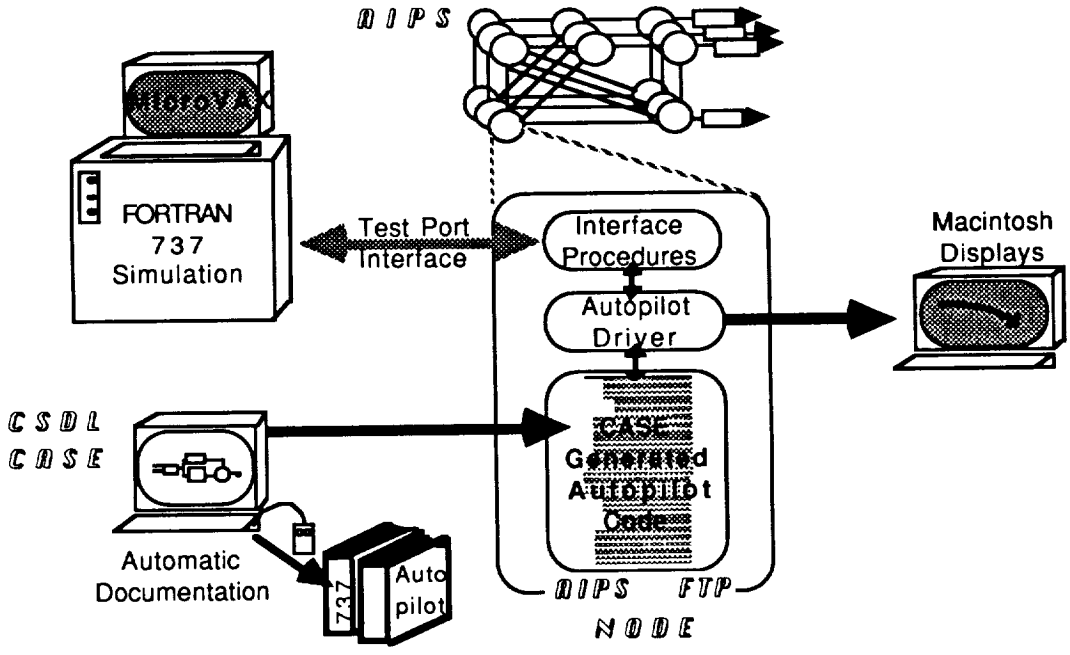

Figure 4. Boeing 737 autoland demonstration.

Table 2 contains data from one run of the demonstration. Initially, the aircraft was flying level at 1500 feet, approximately 118 knots, and was aligned with the runway. The glideslope was intercepted at 34.8 seconds into the simulation and took approximately seven seconds to capture. As is graphically depicted in figure 5, there was no overshoot during glideslope capture and no oscillation along the trajectory. During flare, the system removed all but an acceptable amount of vertical velocity (~3ft./sec.).

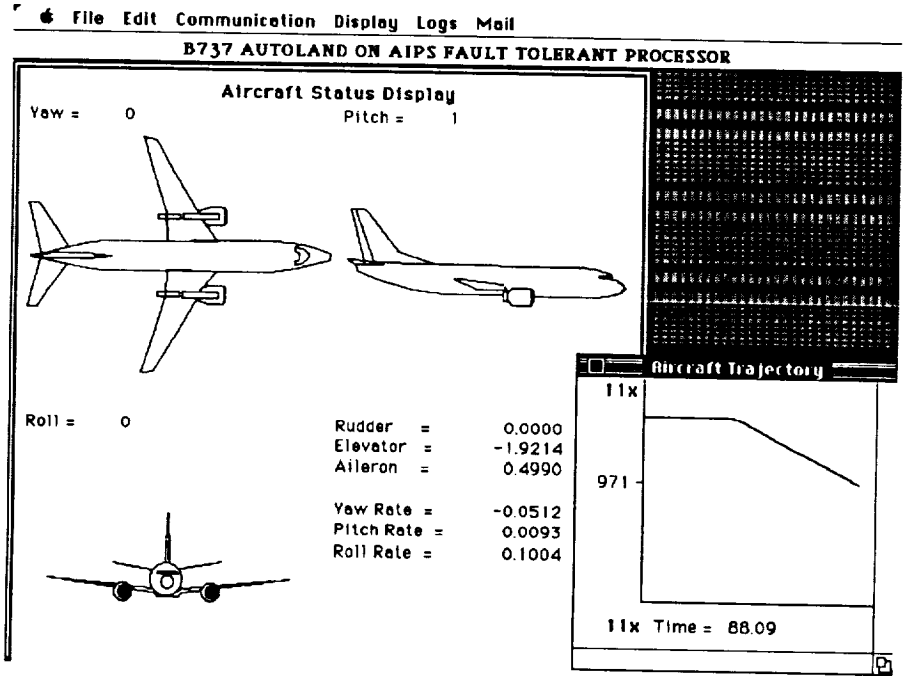

Figure 5. Aircraft status and trajectory displays.

Environmental Conditions:

Atmosphere = STD62	Heading = 67.33 deg.
Winds = None	Elevation = 9 ft.
Runway = Langley 07	Desired Glideslope = -3 deg.

Flight Conditions:

	Initial Condition	Glideslope Intercept	Glideslope Capture	Near Mid-descent	Begin Flair	Touchdown
T (sec)	0.00	34.8	42.4	109.6	175.31	181.41
Alt (ft)	1500.00	1499.75	1461.00	738.73	56.04	18.87
Gamma (deg)	0.00	0.004	-2.995	-2.99	-2.99	-0.87
CAS (kts)	117.85	117.73	119.75	118.01	117.84	108.54
Yaw (deg)	67.33	66.95	67.17	66.90	67.29	67.33
Pitch (deg)	3.97	3.52	-0.17	0.90	------	------
Roll (deg)	0.00	-0.07	0.16	0.03	-0.76	0.26
HDOT (ft/s)	0.00	0.01	-1.16	-10.42	-10.34	-2.76

Table 2. Boeing 737 autoland experiment data.

Advanced Guidance Demonstration

In a follow-on effort to the Boeing 737 autoland experiment, the process of demonstrating automatically-generated Ada code on a *distributed* fault-tolerant computer architecture is being pursued. This demonstration is a more

extensive experiment to further test this technique for developing a flight critical computer system. The application for this demonstration, an advanced adaptive guidance algorithm for a launch vehicle, is much more ambitious.

During analysis, engineers converge on a single algorithm that they consider to be the baseline design. This baseline design is then transcribed into the form of a software specification, since the software specification is typically different from the design form used during analysis. Typically, errors are introduced during 1) the initial transcription from the analysis representation to the software representation and 2) during maintenance of both the analysis and software representations.

In order to avoid these errors and simultaneously reduce development time by eliminating duplicated effort, the algebraic specification capability of CSDL CASE is being modified to accept MATLAB™ scripts. MATLAB™ is an analysis and simulation tool used for the development and test of guidance and control algorithms [14]. This feature will allow both analysis and flight code to be developed from a single specification. When iteration on a design for analysis purposes is complete, the software specification is, therefore, also complete. The analysis and development of the guidance algorithm for the experiment is being performed with MATLAB™. This aspect of the demonstration is intended to show, with MATLAB™ as an example, that it is possible to generate Ada and C code from design specifications, and that design analysis and simulation tools will be usable within CSDL CASE.

Also being demonstrated in this effort is the ability to generate and execute code for a *distributed* architecture, as a network of AIPS FTPs and FTPPs (Fault Tolerant *Parallel* Processors) will be used for execution. As in the previous experiment, the code will be executed on the AIPS while interacting with a dynamic vehicle simulation executing on a MicroVax. The process for this demonstration is shown in figure 6.

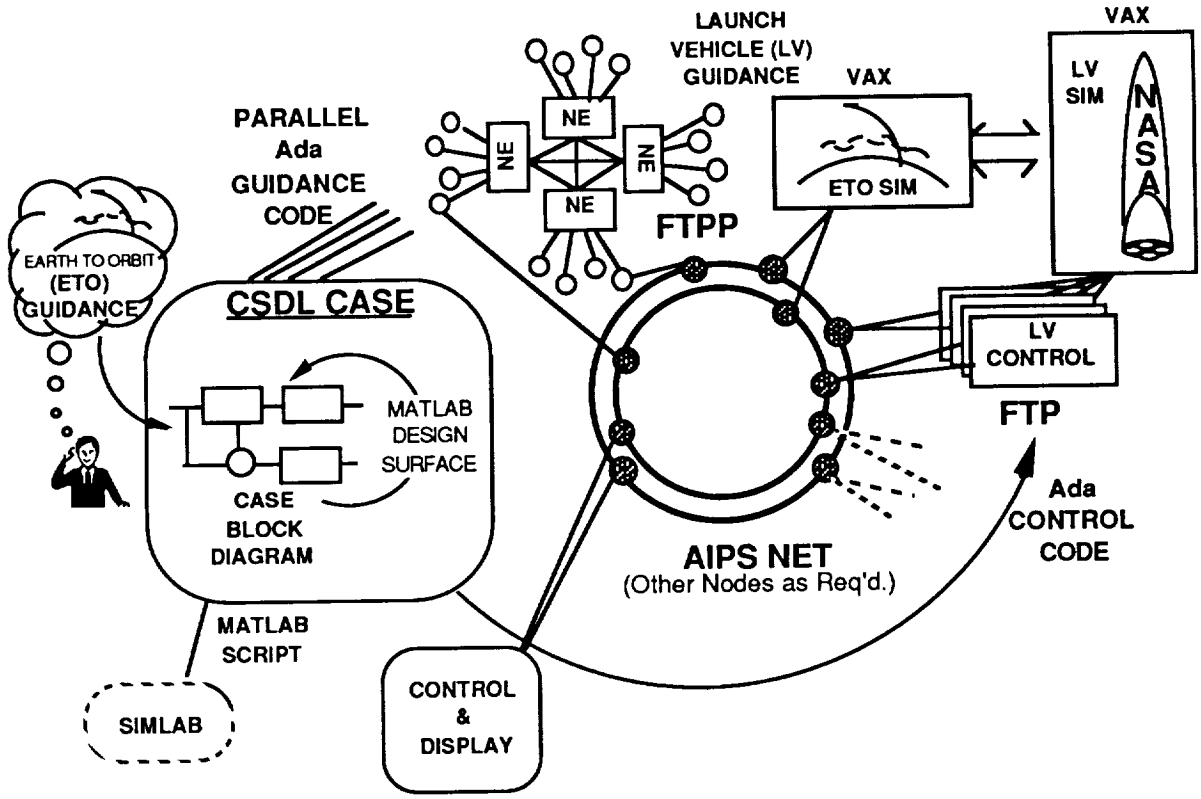

Figure 6. Advanced guidance demonstration on distributed AIPS.

FUTURE ACTIVITIES

Future activities include the development of an Automated Testing Subsystem, a Software Design Methodology User Interface, and the inclusion of formal semantic representations for software designs. Commercialization is also being pursued.

Automated Testing Subsystem

Automating testing would involve automating the five steps common to testing, namely: 1) design; 2) setup; 3) execution; 4) analysis; and 5) documentation. *Design* is the determination of how the code will be exercised and monitored in order to ensure that the code complies with its corresponding requirements. *Setup* is the process of collecting and assembling all of the components that are specified in a test design. These components can include source code, object code, load modules, simulators, input data, specifications on data to be collected during execution, and expected test results. *Execution* is the process of downloading a load module to a target environment, initiating execution, monitoring and collecting data, and terminating execution. *Analysis* is the process of determining if actual results agree with expected results. *Documentation*, obviously, consolidates information in the previous four steps for dissemination and review. Clearly, automated design and thorough analysis are the most complex of the described steps. The remaining three steps and simple analysis are candidates for an initial prototype automated testing system.

Automated test setup will allow test data to be specified for each transform (or selected transforms) in an application design. Test driver code will automatically be generated for the supplied test data. For execution, a load module will be automatically developed for the target environment. A capability will be provided to automatically compile, link, and execute test code, and then monitor execution on the user's workstation while recording data. This process should be controlled from the user's workstation, even if the code is executed on another machine. Simple analysis will be accomplished by comparing achieved results with desired results. The results and discrepancies will be documented in a test report. The display screens and documentation could both vary in complexity from the output of numeric values to graphs, plots, or complex visualization techniques. Naturally, an initial prototype will employ the more simple means of data display.

Software Design Methodology User Interface

The objective of the Software Design Methodology User Interface (SDMUI) is to allow software designers to specify a software design methodology and automatically apply that specified methodology to a functional design. Software design methodologies that are specified through this SDMUI, will be able to be reused on different functional designs. It will be possible to reconfigure a functional specification from one software design into another. Reconfigurations retain the specified functionality of an application design but alter elements of a software design such as memory utilization, execution time, modularity, and compile time in order to optimize execution on a specific execution environment. Formal representations will be used to define software designs, system constraints, and software designing methodologies.

Formal Semantics

Formalisms for representing the semantics of software designs will be investigated. Formalisms will be used to develop a language for representing software designs and methodologies for manipulating software designs. All manipulated versions of a software design should contain the same functionality but would consume different computational resources (i.e., functionally equivalent designs would be represented in different, but equivalent, software designs dependent on the target machine). The design specified by a user of CSDL CASE is the engineer's view of a software application's functionality. As long as the top-level inputs and outputs are the same, it does not matter how the software implementation of this application is executed. It is the intent to be able to transform one method of executing a target software application into another and to be able to demonstrate with mathematical rigor that the two methods are equivalent.

Commercialization

It is the intent of NASA Langley and the Draper Laboratory to pursue commercialization of CSDL CASE in cooperation with a third party. Ideally, present capabilities would be available commercially, supported by a commercial software vendor. The tool would also remain a research vehicle for NASA Langley and the Draper Laboratory to pursue software automation issues like those described in this section.

SUMMARY

The CSDL CASE system views software design, development, and maintenance from the perspective of the application engineer. Unlike most code generators which address the generation of source code from software designs, this approach addresses automated code and documentation generation from specifications of functional application designs. The user interface provides a natural design technique for the specification of real-time software by allowing the functional specifications to be input as hierarchical engineering block diagrams, a notation familiar to application engineers. With this technique, rapid system development and prototyping are possible. Maintenance concerns are reduced since both the code and documentation are maintained by changing the graphical specification. Software reliability and productivity are increased because of the reduction in manual operations, the consistency and completeness checking provided by the system, the automatic translation of specification to code, and the support for reuse of specifications. The application of this CASE tool to many real avionics designs has driven the development of the tool and has proven the viability of the approach. It is felt that CSDL CASE has a place in the commercial arena, particularly in the control systems design environment.

REFERENCES

1. McClure, Carma: *CASE is Software Automation.* Englewood Cliffs, N.J., Prentice-Hall, 1989.

2. Yourdon, Edward; and Constatine, Larry: *Structured Design.* Englewood Cliffs, N.J., Prentice-Hall, 1985.

3. DeMarco, T.: *Structured Analysis and System Specification.* Yourdon Press, New York, 1978.

4. McDowell, M. E.: *Computer-Aided Software Engineering at The Charles Stark Draper Laboratory.* CSDL-P-2802, The Charles Stark Draper Laboratory, Inc., April 1988.

5. Turkovich, John J.; et al: *Advanced Launch System (ALS) Advanced Development Program 2501 Computer-Aided Software Engineering (CASE) Final Report.* NASA Contractor Report, to be published.

6. Turkovich, John J.: Automated Code Generation for Application Engineers. *AIAA/IEEE 9th Digital Avionics Systems Conference*, October, 1990.

7. Walker, Carrie K.; and Turkovich, John J.: Computer-Aided Software Engineering: An Approach to Real-Time Software Development. *AIAA Computers in Aerospace 7*, October, 1989.

8. *ALS CASE User's Guide.* The Charles Stark Draper Laboratory, Inc., Cambridge, MA, June, 1991.

9. Walker, Carrie K.; Turkovich, John J.; and Masato, T.: Applications of an Automated Programming System. *AIAA Computers in Aerospace 8*, October, 1991.

10. *Linear 737 Autoland Simulation for AIRLABS.* Sperry Corporation, SP710-032, June, 1985.

11. Lewin, A. W.; and Turkovich, J. J.: *Testing the ALS CASE Version of the Boeing 737 Autoland Flight Control System.* NASA Contractor Report, to be published.

12. Lala, J. H.; Harper, R. E.; and Alger, L. S.: A Design Approach for Ultra Reliable Real-Time Systems. *IEEE Computer*, May, 1991, pp. 12 - 22.

13. *XDAda Technical Summary.* Digital Equipment Corporation, June, 1989.

14. MATLAB™ High Performance Numeric Computation Software. *MATLAB™ User's Guide,* The MathWorks, Inc., January, 1991.