NAG1-02101

COPY

ADVANCED TOOLS AND TECHNIQUES FOR FORMAL TECHNIQUES IN AEROSPACE SYSTEMS

Final Report

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-2199

Attention:

Benjamin DiVito

Submitted by:

John C. Knight
434.982.2216
knight@cs.virginia.edu

Department of Computer Science
University of Virginia
151 Engineer's Way, PO Box 400740
Charlottesville, VA 22904-4740

June 2005

1. INTRODUCTION

This is the final technical report for grant number NAG-1-02101. The title of this grant was "Advanced Tools and Techniques for Formal Techniques In Aerospace Systems". The principal investigator on this grant was Dr. John C. Knight of the Computer Science Department, University of Virginia, Charlottesville, Virginia 22904-4740. This report summarizes activities under the grant during the period 7/01/2002 to 9/30/2004.

This report is organized as follows. In section 2, the technical background of the grant is summarized. Section 3 lists accomplishments and section 4 lists students funded under the grant. In section 5, we present a list of presentations given at various academic and research institutions about the research conducted. Finally, a list of publications generated under this grant is included in section 6.

2. PROJECT OVERVIEW

Overview

One of the reasons that formal methods are still being adopted only slowly by commercial software companies is the lack of comprehensive tool support. Although many tools have been developed, they are, in almost all cases, outgrowths of research programs and are of limited capability. It is often the case that tools implement a novel concept or an advanced form of analysis yet lack various common functions. Similarly, tools often provide support for a single advanced capability, such as model checking or theorem proving, but offer no integrated support for a range of techniques. This is despite the fact that multiple techniques might be needed in any given circumstance of use.

In previous research, we have identified a number of difficulties with the tools and techniques that have been developed for formal methods, and we have developed preliminary solutions to many of these difficulties. In particular, we have developed a toolset that is designed to provide integrated and comprehensive specification support for developers of critical systems. This existing toolset incorporates support for preparation of specifications that include material in both a natural language and a formal language. Of particular import is the support that the toolset provides for combined analysis of natural and formal languages.

Our current research has built upon this previous research, and we have expanded and extended formal techniques and their applicability. Our research achieved the following goals:

- *Development of advanced tools to support the use of formal methods in aerospace systems.*
  This component of the research program built on our previous research in the area of tools development. We have developed an enhanced versions of our tools by integrating the PVS system into our existing toolset.

- *Development of advanced techniques for the application of formal methods in aerospace systems.*
  This component of the research program has extended our previous research on the practical application of formal techniques by applying previous results and the new tools developed as part of this project to a major aviation safety program,

NASA's Runway Incursion Prevention System (RIPS).

Previous Work

Adoption of formal techniques is viewed as a high risk undertaking by project managers in all areas of aviation and space science. The consequences of failure in these application areas are such that development risks must be minimized. This understandable position makes the challenge of transitioning formal techniques to practice especially difficult. Were formal methods to be applied to a project and there were to be subsequent schedule delays or technical inadequacies, the impact on future use would be extreme.

When formal techniques have been applied to safety-critical systems, the results have generally been very favorable, but there remain many barriers to routine use [4, 1, 2]. Many of these barriers have been addressed, but, in recent research, additional and substantially more fundamental problems have been exposed [3]. One of the primary foci of the problem is tools because: (1) tool support is generally less than it needs to be; and (2) tools are needed to support new ideas in both the theory and process of specification and analysis.

A second area of concern is the detailed process that is followed during the creation of appropriate specifications for safety-critical applications such as those that arise in aviation safety. Formal techniques require a different approach to traditional ad hoc methods based on natural language for several reasons, specifically:

- the formal notations themselves are unfamiliar;
- the use of those notations is unfamiliar;
- analysis capabilities exist for formal notations that are unavailable with natural languages;
- any specification must contain both a formal and informal part, and these parts must be complementary; and
- formal techniques must coexist with informal techniques since wholesale replacement is quite impractical.

Given these issues, it is clear that significant attention must be paid to the way that formal techniques are used. New processes need to be developed, and new comprehensive specification structures need to be developed that integrate all the necessary elements

properly. Tool support for this specific area is also necessary.

Advanced Tools for Practical Formal Specification

Tools to support formal notations must be at least as powerful as those that support informal notations. In addition, they must not impede other existing tools or techniques. These imprecise statements have been made rigorous in a previously developed evaluation framework [4, 1]. The criteria in the evaluation framework were derived from the software lifecycle, and the roles of specific formal techniques were studied in each lifecycle phase. The result of this process was an extensive list of criteria deemed necessary for the successful application of formal techniques, and many of the criteria apply to support tools. There are too many criteria to repeat here, but the following are a representative subset of those that apply to support tools for formal specification†:

- *General:*
  The toolset must be capable of manipulating commercial specifications written in more than one notation using reasonable resources.
  The toolset must provide state-of-the-art analysis of the formal notations that are used and either support analysis of other necessary notations or interface with other tools used in specification documentation.

- *Usability:*
  The toolset must support routine development requirements such as editing, printing and file manipulation, and have the same "look and feel" as other tools being used.
  The toolset must tolerate incomplete specifications and support navigation through the specification.

- *Software Lifecycle:*
  The toolset must support specification development by a team, and the manipulation and analysis of specifications in parts (in the sense of separate compilation of source programs).
  The toolset must support the other phases of software development (design, implementation, verification, etc.) by providing necessary information to and interfacing with support tools in these phases.

†. Some of the criteria list here have been reworded from their original form for clarification and brevity.

The complete list of requirements is quite formidable, and many of them are difficult to meet because of the extensive support that is implied. Developing tools to support the use of formal techniques is a major challenge. A large part of this challenge arises from the need for new tools to provide a wide range of customarily-available features. For example, formal specifications are complex documents, and developers of such specifications appreciate the value of innovative analysis, but must be able to perform routine manipulations conveniently.

Implementing these routine facilities, the "superstructure" of a tool [7], requires enormous resources—frequently more than are needed to implement the innovation that is at the heart of the tool. This has led to many tools being developed with only a limited superstructure thereby limiting the exploitation of their innovation. This situation contrasts sharply with the use of sophisticated desktop publishing systems to manipulate natural language documents—even documents that are artifacts of the software development process.

A second reason why tools to support formal methods are especially challenging is that the tools required are in many ways different from the majority of software tools that a developer uses in a traditional development process. If the full power of a formal notation is to be realized, for example, then this requires the availability of:

- a sophisticated text editor and formatter to deal with the natural language;
- an editing and formatting facility for the formal aspects of the notation;
- a syntax and type-checking system for the formal language; and
- a theorem-proving system.

We have developed an experimental toolset called Zeus that initially supported the development of formal specifications in *Z* and which has been extended to support PVS as a result of this project. Our goal in the design of Zeus is to develop a toolset that: (a) supports the development of complete specifications that contain large formal elements; (b) provides all the routine (mundane) facilities usually found in modern tools; and (c) makes provision for the integration of Zeus with other tools commonly used in the lifecycle.

The design of Zeus is shown in Figure 1. The core of the design is a database that maintains information about an open specification. Zeus can be extended easily by intro-

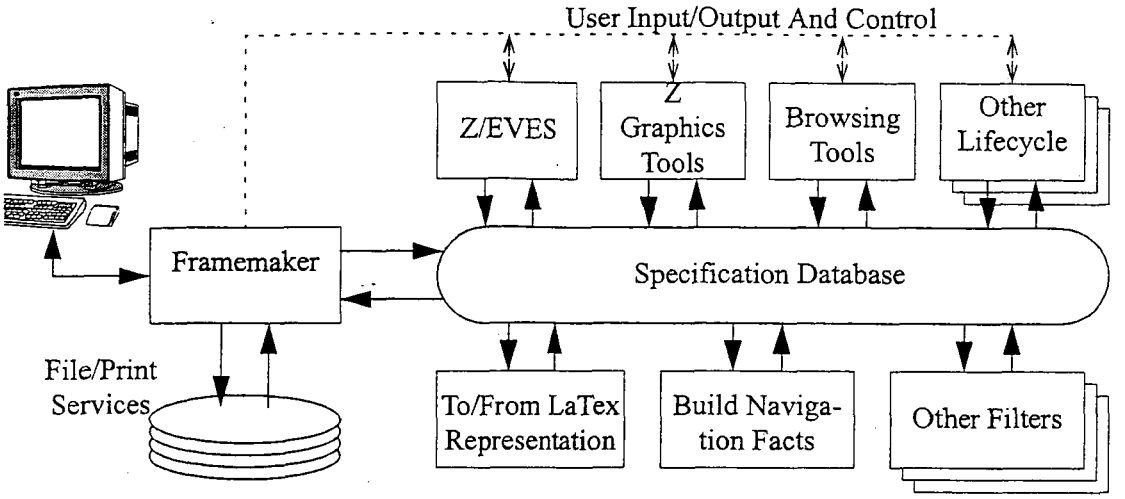

Fig. 1. Present design of the Zeus toolset

ducing new functionality into the database.

Zeus is built using three major software systems as *components*—FrameMaker, an industrial desktop publishing system, Z/EVES, a mature and powerful analysis system for *Z*, and the PVS system. A specification that is manipulated by Zeus is a text document into which *Z* text or PVS text has been included. Zeus provides a seamless connection between FrameMaker and Z/EVES or FrameMaker and PVS, and supplements these systems with a variety of additional functions that facilitate the use of *Z* and PVS.

The present version of Zeus is still a prototype, but it allows: (a) the evaluation of the concept of tool development based on integration of major existing software systems; (b) the evaluation of the utility of a high performance integrated toolset that supports the development of specifications in the *Z* or PVS styles; and (c) the provision of a testbed for evaluation of concepts being developed in the use of natural language in specification.

It has become clear as a result of early experience with Zeus that it meets the goals set for it, and that it facilitates a new paradigm for specification development. The availability of a sophisticated editing facility for a formal notation that also permits syntax and type checking has promoted an approach to specification development that has two basic phases. In the first phase, the developer of the specification operates in a manner much like a programmer in a high-level language. The specification is edited and then the syntax and type rules are checked, and these steps are repeated in an iterative process. Once the

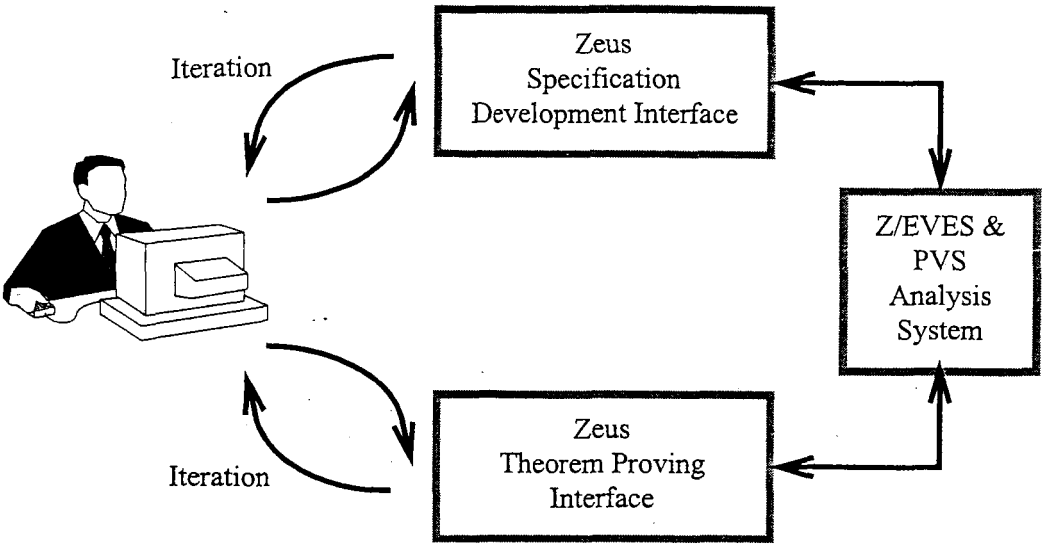

Fig. 2. Comprehensive toolset for formal specification.

developer becomes satisfied that the specification is mostly complete and that preliminary analysis is complete, he or she turns to more extensive analysis of the specification—a phase in which theorems about the specification might be stated and proofs attempted.

In dealing with the introduction of formal techniques into industrial development, we hypothesize that the first of the two phases just outlined will receive the most attention. Industrial developers who are new to formal specification will naturally focus their attention on specification building initially and only progress to analysis after experience is gained. These two phases, and the fact that they will be adopted by industrial developers in series, suggest a structure for specification support tools in which specific interfaces are provided for the two phases. This is exactly the way that Zeus is designed.

Advanced Application Techniques

In previous research, we have developed a linguistic model that explains many of the observed problems in the accurate specification of safety-critical systems [3]. We have also proposed approaches derived from the model that are designed to try to alleviate these problems. Some of these approaches benefit from tool support, and so tools that provide the necessary support and analysis have been built into Zeus.

We have also developed a comprehensive structure for specifications that includes

both formal and informal parts in a precise form. This structure includes models that define the semantics of the operational domain, and that define the mapping between the symbols used in a formal specification and their interpretation in the real world.

The structures that have been worked out do not apply solely to specification. In fact, they should be applied to all software lifecycle artifacts including designs and implementations. The reason is that all artifacts should be thought of as merely a different form of a formal specification. Thus the development of a software system should really be thought of as the development of a more substantial entity that combines a formal part and informal part.

The conclusion that we have reached is that a new type of software artifact should be viewed as the goal in what is now called software development. This artifact, termed a *situated formalism,* combines the following three components into a consistent and comprehensive entity:

- *Function Map*
  The function map is a precise statement of the desired functionality. It is the only part of the situated formalism that can be entirely stated in a formal language although it need not be. The desired functionality can be stated in a formal language such as PVS, *Z*, or C++, but it could also be stated in an informal language but without the benefit of most forms of analysis. The function map is what has been thought of traditionally as the goals in software development. For example, a function map at a high level is what has typically been referred to as a specification. A function map at a lower level is what has typically been referred to as a program.

- *Domain Map*
  The domain map is a comprehensive statement of the domain knowledge needed to understand the real-world meaning of the function map. Since the function map is, by definition, describing a logical function, there is no real-world meaning within the function itself.

- *Model Map*
  The model map documents the relation between symbols used in the model described in the function map and the various concepts in the real world that are described in the domain map. It acts, therefore, as the link between the function

map and the domain map.

The concepts developed for the situated formalism have been evaluated in preliminary experiments [6]. The target of that preliminary evaluation was a production surgical planning system for radiation oncology. That part of the system which computes the radiation dose was developed as a situated formalism and the whole activity evaluated. The situated formalism was initially created at the level of a specification and a refinement was then built at the level of an executable program. The refined version was executed and checked against the production software.

The results of the preliminary experiments were basically in two areas: (1) overall confirmation of the basic concepts behind the situated formalism; and (2) a series of improvements to the original idea that enabled changes to be made thereby bringing the ideas closer to practical use. In the research undertaken for this project, much more experience was gained with the concept of the situated formalism by applying it to the NASA RIPS system. Details are documented in the papers written during this project and in section 3.

Research Summary

The research that we have conducted is in the areas of: (1) advanced tool development; and (2) assessment of the tools and the situated formalism. The tools that we have developed were based on the Zeus toolset, and they primarily involve integration of the PVS system with Zeus. The assessment that we undertook was based upon a case study using the Runway Incursion Prevention System.

Comprehensive Tool Support for Formal Techniques

The present version of the Zeus toolset is fully operational and provides comprehensive support for development of specifications in *Z* or PVS. The necessary syntax checking, type checking, and theorem-proving capabilities are supplied by Z/EVES, a product of ORA (Canada), and the PVS system, a product of SRI International.

We have extended Zeus by adapting it to provide an interface to PVS. The approach taken was to generalize the current architecture so that both analysis systems will be supported by separate versions of Zeus.

The Zeus architecture has been intentionally created to permit extensions of the type

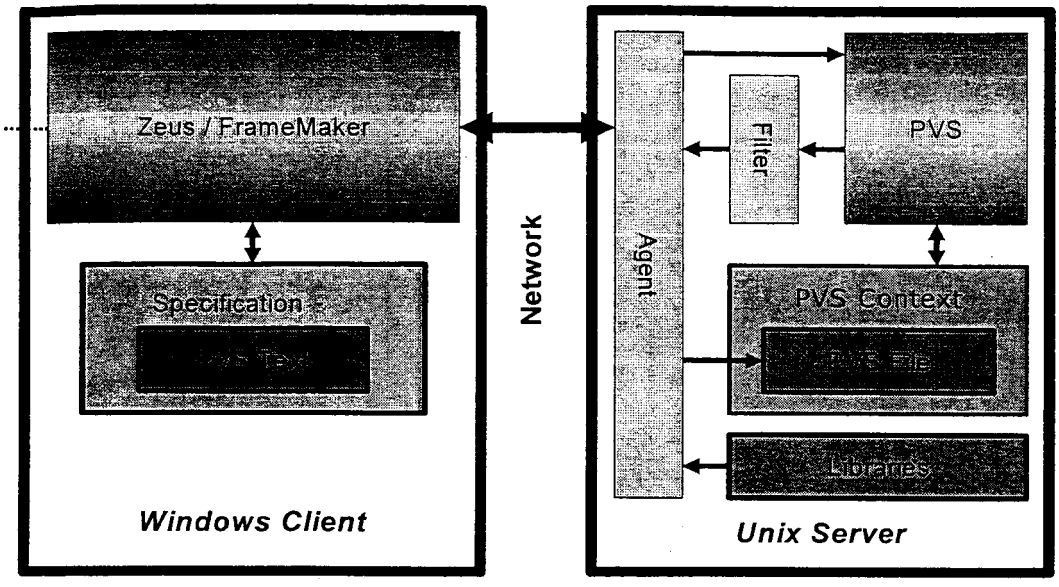

Fig. 3. Zeus PVS Toolset Architecture

proposed. The architecture of the new version is summarized in Figure 3. The platform upon which the Zeus system operates is Microsoft Windows but the analysis system can operate on any platform that is required. This is possible because communication between the major components of Zeus and the analysis system is via standard sockets. Thus, it is possible (and, in fact common) for the Zeus interface to operate on a Windows workstation while Z/EVES or PVS operates on a Unix-based server. Finally, Zeus supports the *Z* and PVS character sets.

The original and current SRI PVS interface is EMACS and is very character oriented. An important aspect of the present interface is its relative simplicity. The way in which PVS is controlled is via simple commands that the user types. With the extensive use of what amount to keyboard shortcuts, PVS is very flexible and convenient for experienced users. The Zeus command interface, on the other hand, is mostly based on menus and similar graphic entities. We have implemented a suitable PVS interface that, as far as we can tell, does not limit the way in which users are able to interact with PVS.

Some minor changes to PVS were made to accommodate this development activity. These changes were made by SRI international working with developers at the University

of Virginia.

The result of the integrated system is a toolset that provides all the WYSIWYG editing and formatting capabilities that Zeus supports along with the analysis capabilities of PVS.

Specification Case Study

In order to make the tools and techniques developed as part of this research project really useful to developers, the ideas were evaluated. We conducted a case study using RIPS as the system studied. The case study consisted of building a situated formalism for RIPS, at the level of a specification. Our goals with this activity were to further refine and demonstrate the situated formalism concept and the Zeus tools.

REFERENCES

1. C.L. DeJong, M.S. Gibble, J.C. Knight, and L.G. Nakano. *Formal Specification: A Systematic Evaluation*. Technical Report CS-97-09, Department of Computer Science, University of Virginia, Charlottesville, VA, June 1997.

2. M.S. Gibble and J.C. Knight. *Experience Report Using PVS for a Nuclear Reactor Control System*. Technical Report CS-97-13, Department of Computer Science, University of Virginia, Charlottesville, VA, June 1997.

3. K. Hanks, J. Knight, and E. Strunk. Erroneous requirements: a linguistic basis for their occurrence and an approach to their reduction. *Proceedings of the 26th Annual IEEE NASA Software Engineering Workshop,* pages 115-119, Greenbelt, MD (2001).

4. J.C. Knight, C.L. DeJong, M.S. Gibble, and L.G. Nakano, Why Are Formal Methods Not Used More Widely? *Fourth NASA Formal Methods Workshop*, Hampton, VA (September 1997).

5. R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 126-133. IEEE Computer Society Press, January 1993.

6. E.A. Strunk. *The Program+: Defining the Role of Natural Language in a Software Product*, MS Thesis, Department of Computer Science, University of Virginia (May 2002).

7. K. Sullivan and J.C. Knight. Experience Assessing an Architectural Approach to Large-Scale Systematic Reuse. *Proceedings ICSE 18: Eighteenth International Conference on Software Engineering*, Berlin, Germany (1996).

3. ACCOMPLISHMENTS

Accomplishments under this grant are summarized in this section. Details are contained in the various publications that have been generated (see section 6) or in documents that are being prepared for publication.

Achievements Specific to the RIPS System

- Reverse engineered a PVS specification from the RSM source code.

  We created a PVS specification from the source code provided by Lockheed-Martin. The purpose of this exercise was to create a starting point for analysis of the Runway Safety Monitor whose properties would not be obscured by implementation detail.

- Built a situated specification of the RSM definition of an incursion zone.

  We are employing survivability as a method to assist in ensuring validity, as well as to assist implementation, in complex systems. To ensure validity, we have built a situated specification of the incursion zone structure. This structure is the key element of the design of the RSM algorithm that must be understood by domain experts if they are to validate the algorithm and decide that it does what they intend it to do. Because PVS is a formal language, it alone is not capable of ensuring that design decisions at that stage are appropriate. We augmented our PVS specification of the RSM algorithm with natural language designating interpretations of many of the basic specification variables, and also a graphical representation of the structure over which the algorithm operates.

- Analyzed the RSM requirements to determine appropriate functionality for survivability levels.

  The idea behind survivability is to separate functionality that is critical to system operation to functionality that is helpful, but not essential for aircraft safety. Because the RSM is designed only to supplement a pilot's existing situational awareness, it is not flight-critical. However, recognizing an error in computation of the system and alerting the pilot that the service is not available is important. We worked with the Lockheed-Martin team to identify different modes of the RSM and those modes' criticality.

- Created a PVS specification of a survivable version of the RSM

  This work aimed to demonstrate how survivability could be used to increase confi-

dence in the correct operation of a piece of software such as the RSM. Part of such demonstration was showing how a survivable specification might be constructed in PVS. We constructed a survivable specification for the RSM algorithm, incorporating the specification of safe programming where it was appropriate in the software. Such implementation decisions directly affect the way the system operates, and how the system is designed to satisfy its requirements. Without noting them in an abstract way such as is supported by PVS, developers cannot be sure that incorporating safe programming will not decrease dependability by failing to deal correctly with borderline conditions (e.g., loops that must execute more than the prescribed number of times).

Achievements in the Development of the Zeus Toolset

- Identified key PVS features that must be accessible from any user interface

  The emacs interface to PVS, as developed at SRI International, includes a plethora of methods for constructing and proving PVS specifications and theorems. For the advanced PVS user, the emacs interface is likely to be quicker and more effective than any interface that has a lower learning curve. Our objective was to develop an interface that would sacrifice some of the flexibility of the emacs interface in order to reduce the learning curve of the tool so that it could be integrated more smoothly into existing development processes. Through a study of a number of advanced PVS users, and the efforts to use the tool by two novice PVS users, we identified a first set of key interface features.

- Developed a system architecture to access PVS running on a UNIX machine from a word processing program running on a Windows machine

  Because PVS cannot be run on a machine with a Windows operating system, an architecture to connect a Windows client to a PVS process running on a UNIX server was required. We accomplished this by creating a UNIX agent program that listens on a particular, customizable network port. When contacted by the Windows Zeus plug-in process, the agent starts PVS in its "raw" mode. The agent then pipes data between the PVS and Zeus programs, sending commands from Zeus to PVS and sending replies from PVS to Zeus. The agent is responsible for stopping the PVS process when the client informs it the process is no longer needed.

- Revised the PVS raw interface to support inclusion of more advanced user interfaces (subcontracted to SRI International)

  Because of the complexity of PVS's interaction with emacs, it was not possible to

use the emacs interface through Zeus. Instead, we used the "raw" mode developed for using PVS as a batch process. Unfortunately, while SRI had previously developed the raw interface for a simpler client user interface, the interface capabilities were far short of those needed to use Zeus effectively, even for a novice user. With SRI as a subcontractor to make the specific changes to the PVS software, we enhanced the interface to become one we could use to implement the necessary Zeus functionality.

- Constructed a user interface design for PVS that is more intuitive to the average user of formal methods than the current user interface

  While emacs is customizable to a significant extent, developing an interface using structures and methods provided by Windows enabled us to design an interface that is more visually appealing and faster to learn for the average user. We designed the interface using input from the advanced and novice users of PVS we consulted in determining key requirements. The design is more easily menu-driven, and provides a more elegant proof interface, than the interface in emacs

- Implemented a plug-in to FrameMaker using the user interface design

  FrameMaker is a powerful document processing software package, and so developing Zeus as a plug-in to it enables the construction of PVS specifications with formatting required for professional review and other use. The plug-in's interface implements the design, which is also more familiar to the Windows user than the emacs design. Using FrameMaker as the base platform means that if a version of Zeus to run on UNIX were desired, it could be implemented for the UNIX version of FrameMaker with less effort than if it were designed for a document processing program specific to one operating system.

- Educated the community in the results of our efforts through presentations and demonstrations at NASA PVS training courses

  NASA Langley periodically offers PVS training courses as a service to practitioners interested in state-of-the-art techniques in software development. The students of this class are primarily of the type targeted by the Zeus effort: they would like a way to strengthen their company's software development practices, but without investing large amounts of money in training employees to use PVS. We have explained to them the principles of the Zeus effort, and demonstrated the more easily accessible interface of the tool so they can see the potential to use it where they need it, without significant up-front expenditures.

4. SUPPORTED STUDENTS

The following students were supported in whole or in part under this grant:

Ph.D.:

Name	-	Elisabeth Strunk
Status	-	Graduated May 2005

5. PRESENTATIONS GIVEN

Seminars (not including conference presentations) describing the research being performed under this grant were presented at the following institution:

- Situated Formalisms: Combining Software Function and Context, Rice University, Houston, TX, 1/2005.
- Situated Formalisms: Combining Software Function and Context, U.C. Irvine, Los Angeles, CA 1/2005.
- Situated Formalisms: Combining Software Function and Context, Politecnico di Milan, Italy 7/2004.
- Situated Formalisms: Combining Software Function and Context, Praxis Critical Systems, Bath, U.K., 12/2003.
- Situated Formalisms: Combining Software Function and Context, Imperial College of Science and Technology, London, U.K., 12/2003.
- Situated Formalisms: Combining Software Function and Context, University of Souther California, 11/2003.
- Situated Formalisms: Combining Software Function and Context, University of Newcastle, Newcastle upon Tyne, U.K., 9/2003.
- Situated Formalisms: Combining Software Function and Context, University of York, York, U.K., 9/2003.

6. PUBLICATIONS

The following papers and documents were prepared from the principal investigators' research program and were funded all or in part by the subject grant. Copies of these papers have been supplied to the sponsor under separate cover.

Papers that were either published or accepted for publication during the reporting period are:

1. Knight, John C., Elisabeth A. Strunk, William S. Greenwell, and Kimberly S. Wasson, Specification and Analysis of Data for Safety-Critical Systems, 22nd International System Safety Conference, Providence RI (August 2004)
   Available at:
   http://www.cs.virginia.edu/~jck/publications/issc.22.pdf

2. Wasson, Kimberly S., John C. Knight, Elisabeth A. Strunk, and Sean R. Travis, Tools Supporting the Communication of Critical Application Domain Knowledge in High Consequence Systems Development, SAFECOMP 2003, The 22nd International Conference on Computer Safety, Reliability and Security, Edinburgh, Scotland (September 2003)
   Available at:
   http://www.cs.virginia.edu/~jck/publications/safecomp.2003.tools.pdf

3. Hanks, Kimberly S., and John C. Knight, Improving Communication of Critical Domain Knowledge in High-Consequence Software Development: An Empirical Study, 21st International System Safety Conference, Ottawa, Canada (August 2003)
   Available at:
   http://www.cs.virginia.edu/~jck/publications/hanks.issc03.final.pdf