AEROSPACE REPORT NO.
ATR-94(4778)-1

Using SDVS to Assess the Correctness of Ada Software used in the Midcourse Space Experiment

30 September 1994

Prepared by

T. K. MENAS, J. M. BOULER, J. E. DONER, I. V. FILIPPENKO,
B. H. LEVY, and L. G. MARCUS
Trusted Computer Systems Department
Computer Science and Technology Subdivision
Computer Systems Division
Engineering and Technology Group

Prepared for

DEPARTMENT OF DEFENSE
Ft. George G. Meade, MD 20744-6000

Engineering and Technology Group

THE AEROSPACE CORPORATION
El Segundo, California

AEROSPACE REPORT NO.
ATR-94(4778)-1

USING SDVS TO ASSESS THE CORRECTNESS OF ADA SOFTWARE USED IN THE MIDCOURSE SPACE EXPERIMENT

Prepared by

T. K. MENAS, J. M. BOULER, J. E. DONER, I. V. FILIPPENKO, B. H. LEVY, and L. G. MARCUS
Trusted Computer Systems Department
Computer Science and Technology Subdivision
Computer Systems Division
Engineering and Technology Group

30 September 1994

Engineering and Technology Group
THE AEROSPACE CORPORATION
El Segundo, CA 90245-4691

Prepared for

DEPARTMENT OF DEFENSE
Ft. George G. Meade, MD 20744-6000

Report No.
ATR-94(4778)-1

Using SDVS to Assess the Correctness of ADA Software Used in the Midcourse Space Experiment

Prepared

[illegible] 4/11/94
T. K. Menas

J. M. Bouler (by [illegible])
J. M. Bouler

J. E. Doner (by [illegible])
J. E. Doner

I. V. Filippenko 4/12/94
I. V. Filippenko

B. H. Levy (by [illegible])
B. H. Levy

[illegible] 4-12-94
L. G. Marcus

Approved

for [illegible] 4/12/94
D. B. Baker, Director
Trusted Computer Systems Department
Computer Assurance Division

[illegible] 4-13-94
J. H. Katz, Principal Director
Computer Science and Technology Subdivision

Abstract

This paper gives an overview of a 1993 project performed at The Aerospace Corporation in cooperation with the Johns Hopkins Applied Physics Laboratory[1] to formally verify, using the State Delta Verification System (SDVS), a portion of the Midcourse Space Experiment (MSX) tracking processor software. SDVS is an automated system developed at The Aerospace Corporation for use in formal computer verification. The tracking processor software is written in Ada and 1750A assembly language. The project has been one of the largest experiments in the formal verification of production Ada code. This paper presents (1) an overview of SDVS, (2) a functional overview of a portion of the MSX tracking processor software (the *target* software), (3) a discussion of the modifications that were made to the MSX software, and (4) a description of the correctness proofs of the modified MSX software and of the two different strategies used in the proofs. The modifications were due primarily to the presence of Ada tasks in the target software.

[1]We are especially indebted to Richard Waddell and Shane Hutton of the Johns Hopkins Applied Physics Laboratory for their help in discussions of issues raised in this report.

Contents

1 Introduction

This report is gives an overview of a 1993 project performed at The Aerospace Corporation (Aerospace) in cooperation with the Johns Hopkins University Applied Physics Laboratory (JHU/APL) to verify a portion of the Midcourse Space Experiment (MSX) spacecraft tracking processor software (the target software) using the State Delta Verification System (SDVS). A detailed discussion of the verification effort is presented in [1].

The State Delta Verification System[2] (SDVS) is an automated system developed at Aerospace for use in formal computer verification. The formal framework of SDVS is based on a form of classical temporal logic that provides an operational semantic representation of computation. A program or hardware description written in one of the three high-level computer languages that SDVS currently supports — subsets of ISPS, VHDL, or Ada — may be translated by the corresponding SDVS language translator into a temporal formula that may then be symbolically executed. Typically, a user writes a specification in SDVS for a program or hardware description written in one of these three languages, and then proves that the SDVS translation of the program implies its specification.

MSX is a near-term Ballistic Missile Defense Organization program whose primary purpose is to conduct tracking-event experiments of targets/phenomena in midcourse. JHU/APL is the prime contractor for the spacecraft and is a key developer of the mission software, which is written in Ada and 1750A assembly language.

In our verification experiment, we focused on a part of the MSX tracking processor software that processes a stream of commands into application-level messages.[3] Our focus was guided by the important role these messages play in a tracking experiment, and was constrained by the pragmatic consideration that SDVS does not handle floating-point numbers and pointers at this time.[4] Nevertheless, we had to deal with Ada software that included tasks, an interrupt-driven procedure, numerous interfaces to 1750A assembly routines,[5] and various Ada constructs that were not implemented in the SDVS Ada translator when we began the experiment. We were able to deal, mostly satisfactorily, with the assembly routines and the unimplemented Ada constructs, but the tasks and the interrupt-driven procedure were a major problem; we had to rewrite tasks as procedures and provide a scheduler for them (the main program). Because of time constraints on this experiment, the scheduler did not encompass a great number of complicated situations that might arise in the execution of the original software.

In spite of the problems we encountered because of the Ada tasking, our walk-through of the code and the correspondence between Aerospace and JHU/APL led to the discovery of two errors in the target software.[6] Once the errors were corrected, we were able to prove

[2]For an overview of SDVS see [2] and [3]. For more detailed accounts see [4] and [5].

[3]See [6] and [7] for a functional overview of the MSX tracking processor software. Our specifications were obtained primarily from [8].

[4]We started to study the inclusion of floating-point numbers and pointers in SDVS (see [9] and [10]).

[5]SDVS has the capability to verify programs written in a subset of 1750A assembly language as well, but we could not include these subprograms in a one-year experiment.

[6]We note that at the time the errors were discovered, JHU/APL had not tested the code extensively.

that 57 types of one class of application-level messages were processed correctly by the code (but under ideal conditions). The interested reader will find the commented code and the batch proof in [1].[7]

In Section 2 of this paper we give a brief overview of SDVS, and in Section 3 we give an overview of the MSX tracking processor software we verified.

In Section 4 we describe our approach to the verification effort, and in Section 5 we present a summary of our proofs and our proof strategy. We have not included the proofs in this paper, because they are quite long. Our proofs rely on a simple scheduler and on restricted input. However, in the latter phase of the project, we started to work on an approach to the verification that, we think, will lead to proofs of correctness for more complex inputs and schedulers.

In Section 6 we list our accomplishments, which include the verification of a substantial, real Ada application program consisting of about 900 lines of uncommented code.

[7]The target software consists of over 900 lines of uncommented code; the batch proof is about 80 pages long; the actual SDVS trace of the proof is several thousand pages long.

2 The State Delta Verification System

The formal framework of SDVS relies on the language and techniques of mathematical logic. SDVS is based on a specialized temporal logic whose characteristic formulas, called *state deltas*,[8] provide an operational semantic representation of computation. Operational verification systems are characterized by computational sequences (or states) that symbolically trace the execution of a program. A state consists of the symbolic values of the program variables at some point in the execution of the program. Thus, a sequence of states constitutes a sequence in time. Temporal logics are used to express and prove propositions involving time. Hence, a proof in SDVS is a proof in SDVS' temporal logic.

Technically, SDVS checks proofs of state deltas. SDVS can handle proofs of claims of the form "if P is true now, then Q will become true in the future." Assuming P represents a program (perhaps with some initial assertions) and Q is an output assertion, this is an input-output assertion about P. SDVS can also be used to prove a claim of the form "if P is true now, then Q is true now;" assuming both P and Q represent programs, this claim asserts the *implementation correctness* of P with respect to Q. This is a claim that one program correctly implements another. Specifications of programs may be directly formulated in state deltas, or may themselves be programs that can be translated into state deltas.

SDVS has a proof checker/theorem prover, knowledge about several computer domains (data types), and a set of application language translators. A user inputs either an Ada, VHDL, or ISPS program together with a specification for that program. The state delta representation of the program is obtained by invoking the appropriate application language translator, with the subject program as its argument. Then the user interacts with SDVS to construct a proof that the state delta representation of the program satisfies the specification. A proof may be developed interactively and then later be executed in batch mode.

The underlying proof method used by SDVS is *symbolic execution.* Symbolic execution essentially involves *executing* a program or machine description from its initial state through successive states using *symbolic* values for the program variables or for the contents of machine registers. Of course, the computation path is often conditional on specific values; in these instances subproofs must be initiated to account for all possibilities. The correctness claims that are proved are all of the form "At certain states some conditions are true." Thus, during a proof there are two kinds of tasks: to go from state to state, and to prove that certain things are true in a given state. These are the dynamic and static aspects of the proof system, respectively.

The dynamic proof language has three basic rules: straight-line symbolic execution (for instances where the path is not data dependent), proof by cases (at branch points), and induction (necessary when the number of times through a loop is data dependent, but could also be used for a large constant number of iterations). We are currently in the process of incorporating a command to handle recursive procedures.

[8]The state delta operator is equivalent in expressibility strength to *until*, the strongest of the classical temporal operators [11]. State deltas were first introduced in [12].

Once SDVS has "arrived" at a state that the user knows (or hopes) will satisfy the conditions to be proved, SDVS must be convinced that these conditions are true. Thus, SDVS has some explicit facts about the state listed in its database, which perhaps do not include verbatim the required condition. The problem is then to prove the "static" theorem that those facts imply the required condition. The user is aided in the proof of the static theorem by SDVS' knowledge about domains used in the programs. A main component of the theorem prover is the SDVS *Simplifier*, which implements these domains as *theories* with complete or partial *decision procedures* (or *solvers*) [13]. The decision procedures are used to deduce properties about domain objects. The complete decision procedures automatically answer queries about propositions, equality, enumeration orderings, fragments of naive set theory, and parts of integer arithmetic. The partial decision procedures are part automatic and part manual, with the user instructing the system to use various axioms to deduce properties. Domains for which there are partial solvers include integer arithmetic, bitstrings, arrays, VHDL time, and VHDL waveforms. The Simplifier handles combinations of theories according to the Nelson-Oppen algorithm for cooperating decision procedures [14].

The adaptation of SDVS for the verification of Ada programs started in 1988 [15]. Prior work on Ada verification had been done elsewhere [16, 17]. The principal research issues that were addressed in adapting SDVS to the verification of Ada programs were (1) formally defining the semantics of Ada (more precisely, the subsets of Ada in which programs will be written and verified in SDVS), and (2) augmenting the SDVS simplifier and data-type theory repertoire with components necessary to support the Ada language.

Rather than trying to deal with the entire language, we found it more appropriate to proceed by degrees by selecting increasingly complex Ada language subsets to incorporate into SDVS. Currently, the subset of Ada that has been incorporated into the SDVS Ada translator is roughly equivalent to Pascal without reals, but with packages.

3 MSX Overview

MSX is a near-term Ballistic Missile Defense Organization program whose primary purpose is to conduct tracking-event experiments of targets/phenomena in midcourse. In a tracking event, the main spacecraft systems are the command processor, the attitude processor, the tracking processor, the sensors, and the data-handling system (telemetry). The command processor receives, buffers, and relays commands for a network consisting of the ground and the spacecraft subsystems. The attitude processor interfaces to the attitude sensors and controllers. Its primary function is to determine and control the spacecraft's attitude. The fundamental function of the tracking processor is to generate sufficient information for the attitude processor to point the spacecraft at the desired target, location, or direction. The tracking processor is designed around a MIL-STD-1750A (1750A) microprocessor with 2K of ROM, 512K of RAM, and 256K of EEPROM.

When the spacecraft is not involved in a tracking event, the tracking processor is turned off to conserve power. During these periods, the direction of the spacecraft is controlled autonomously by the attitude processor and is in "parked mode." Before a tracking event is to take place, the tracking processor is turned on by a real-time or delayed command. Then the ROM is used for power up, the software and data for the tracking event are loaded into storage from EEPROM to RAM, and the event begins.

During the current event and in the preparation for the next tracking event, commands are generally uplinked from the ground to the command processor and relayed to the tracking processor via a serial port. These serial digital commands are combined by the tracking processor to form application-level messages, which are then stored in RAM, EEPROM, or both. There are eleven types of application-level messages. One of these, the data-structure memory-load application message (the data-structure message, for short), can modify up to about 120 tracking parameters used in a tracking event. The number of commands required to form a data-structure message is a function of the type of tracking parameter the data-structure message modifies.

The *target software* we selected for verification is that part of the tracking processor software that processes serial digital commands from the command processor into data-structure messages and then stores the messages into RAM, EEPROM, or both.

3.1 Functional Description of Verified Software

To illustrate the type of serial digital commands required to build a data-structure message and the algorithm used to build the message, consider the Beacon Alignment First Object data-structure message. This message encodes a 3×3 real matrix that is required to be stored in both EEPROM and RAM. The matrix has 9 real number entries, and each real number requires 4 bytes. Therefore, the matrix requires a total of 36 bytes of data.

A byte is 8 bits long; a word is 16 bits long; and a command consists of two words. In the Ada code, bytes and words are represented by integers constrained to specific ranges. Although bytes (words) have an integer parent type, they encode sequences of 0's and 1's that are 8 (16) bits long. For example, if the byte $B = 7$, B encodes the bit sequence $< 00000111 >$.

Fourteen commands are needed for the construction of the Beacon Alignment First Object data structure (see Table 1). The first bit of the first byte of each command is the parity bit (P) for the entire command. The other bits serve the functions we outline below.

(i) Command 1:

- The last seven bits of the first byte encode the op-code of the command. For a command that begins a data structure load message, the op-code is 1.
- The second byte encodes the storage information: EEPROM, RAM, or both. For the Beacon Alignment First Object data structure, which according to the specifications must be stored in both EEPROM and RAM, this byte must have the value 2.
- The third byte is the identification code, ID, of the data structure. For the Beacon First Alignment Object, this code is 1.
- The fourth byte is the first byte of data, d_1, for the matrix.

(ii) Command n where $1 < n \leq 12$:

- The last seven bits of the first byte encode the op-code, which is 8, for a data-structure load continuation command.
- The other three bytes are the final bytes of data for the matrix: d_{3n-4}, d_{3n-3}, and d_{3n-2}.

(iii) Command 13:

- The last seven bits of the first byte encode the continuation op-code 8.
- The next two bytes are the next bytes of data for the matrix: d_{35} and d_{36}.
- The fourth byte is the first byte of the 2-byte checksum.

(iv) Command 14:

- The last seven bits of the first byte encode the continuation op-code 8.
- The second byte is the second byte of the checksum.
- The third and fourth bytes are spares.

Table 2 shows the form of the completed message.

3.2 Ada Implementation of Verified Software

The heart of the verified software consists of three Ada tasks – BUILD, PROCESS_MSG, and MANAGE_MSG_RETRIEVAL – and an interrupt-driven Ada procedure, CMD_IN_HANDLER. When an interrupt occurs signaling that a command is ready to be retrieved from a designated serial port, CMD_IN_HANDLER services the interrupt by retrieving the command and storing it in a command buffer. For an infinite number of times, if the command buffer

Table 1: Beacon Alignment First Object Commands

	Byte 1		Byte 2	Byte 3	Byte 4
Command 1	P	1	2	1	d_1
Command 2	P	8	d_2	d_3	d_4
Command n	P	8	d_{3n-4}	d_{3n-3}	d_{3n-2}
Command 13	P	8	d_{35}	d_{36}	checksum
Command 14	P	8	checksum	spare	spare

Table 2: Beacon Alignment First Object Message Completed

1	0	OP
2	EEPROM/RAM	ID
3	d_1	d_2
4	d_3	d_4
$\vdots$		
n	$d_{2(n-3)+1}$	$d_{2(n-3)+2}$
$\vdots$		
20	d_{35}	d_{36}
21	checksum	checksum

is not empty, BUILD retrieves the command buffer, constructs application-level messages from these commands, places them in a circular message queue, and waits for a rendezvous with MANAGE_MSG_RETRIEVAL. Also for an infinite number of times, if the message queue is not empty, PROCESS_MSG will rendezvous with MANAGE_MSG_RETRIEVAL to retrieve and process a message from the message queue. PROCESS_MSG stores data-structure messages in EEPROM, RAM, or both. As its name indicates, MANAGE_MSG_RETRIEVAL synchronizes the other two tasks.

Note that the interplay of the three tasks and the interrupt-driven procedure can be quite complex. At any particular time in the execution of the code the following may occur:

- The command buffer may contain enough commands to build several messages, or to build several messages plus a portion of another, or to finish the construction of a message whose construction has not been completed.
- The message queue may contain any number of messages up to the maximum number (30). If it already contains 30 messages, then the next message may be lost.

4 Our Approach to the MSX Verification Experiment

In this section we briefly discuss our approach to the verification project. In the first phase of the project, we studied and modified the target software, and in the second phase, we stated and proved correctness assertions for increasingly complex scenarios.

4.1 Ada Modifications to the Software

After studying the documentation for the tracking processor software ([6], [7] and [8]), we examined 18 library units that fell either in the area of or on the periphery of our target software and selected the units and the parts thereof that were needed to receive, build, process, and store data-structure messages in EEPROM. After this process of elimination, we were left with target software consisting of (1) two packages containing the three tasks and the interrupt-driven procedure mentioned above, (2) three packages containing type definitions, (3) one package containing Tartan-supplied[9] functions for bit manipulations, and (4) four packages only marginally related to data-structure processing. The parts we needed from these packages constituted our new target software and consisted of around 900 lines of code.

We then examined the target software to note the following:

(i) the Ada constructs not handled by the SDVS Ada translator,

(ii) the subprograms written in 1750A assembly language, and

(iii) the nonstandard Ada functions provided by the Tartan compiler for bit manipulations.

We added some of the Ada constructs in item (i) to the SDVS Ada translator (integer subtypes and integer definition types, and for these types: type conversions, length representation clauses, and UNCHECKED_CONVERSION). For the rest of the Ada constructs in item (i), we substituted either equivalent Ada code (for example, decimal literals for hexadecimal literals) or nonequivalent Ada code (for example, procedures for tasks) that nevertheless behaved similarly under certain conditions. Our replacements were chosen so that the following would be true: if the modified code did not satisfy a required property, then the original code would not satisfy that property. For example, if our order of execution of the procedures that were formerly tasks did not process a specific message correctly, then the same order of execution of the actual tasks in the original code would also not process the message correctly.

For the subprograms written in 1750A assembly language, we substituted Ada code that functioned similarly (for example, Ada type conversions replaced assembly routines used for the conversions[10]) or Ada code that, in general, functioned differently, but correctly

[9]Tartan is the compiler used by JHU/APL to compile the MSX software.

[10]In the original MSX software, type conversions were done by assembly routines because the Tartan compiler did not handle Ada type conversions properly.

under the right conditions. For example, our version of the CHECK_PARITY function, whose original body was written in 1750A assembly language, returned the boolean value *true*, whereas the original function returned either *true* or *false*, depending on the parity of the object to which it was applied. This change in the function altered the behavior of the main program. However it did not alter the behavior of the main program under the assumption that all inputs have good parity, which was, in fact, one of our assumptions.

We wrote Ada code for the Tartan-supplied bit-manipulation functions, so that we could compile and test the target software with the Verdix Ada compiler available at Aerospace. In our SDVS correctness proofs, calls to these functions were characterized by the SDVS *adalemma* facility. An *adalemma* is an assertion in SDVS about the behavior of an Ada subprogram. Adalemmas may either be proved in SDVS and used in a proof involving the main program, or simply asserted and used without proof. In the latter case, SDVS warns the user that the proof was completed with unproved adalemmas.

We next wrote schedulers (main programs) for the procedures that replaced the three tasks and the interrupt-driven procedure in the original MSX software.

Our modifications altered the functionality of the original target software. We did not verify the correctness of the original code. However, we verified certain properties of the modified code that should also be true of the original code; we think the original code would not have satisfied these properties if the modified code had not.

Our approach to the verification effort was to prove specifications of increasing complexity. Complexity arises primarily from the execution paths allowed by the scheduler and the restrictions on the input to the program.

4.2 Incremental Proof Complexity

In our first attempt at a proof, we wrote a scheduler that processed exactly three commands, and we restricted the input to a block of three commands encoding a specific type of data-structure message. The specification stated that this one block of input was processed correctly as a data-structure message and stored in EEPROM.

In our second proof, we assumed the input consists of an infinite sequence of blocks of commands, each block encoding one of two types of data-structure messages. In an infinite cycle, the scheduler called CMD_IN_HANDLER precisely the number of times needed to retrieve the next block from the input, then BUILD, and finally PROCESS_MSG. The specification stated that for each block of commands in the input, there is a message written in EEPROM corresponding to the input block in the manner stipulated by the MSX documentation. Furthermore, the correspondence of blocks of commands to messages is one-to-one, onto, and order-preserving.

Our attempts to generalize the specification in the second proof to one allowing an input consisting of an infinite sequence of blocks of commands, each block encoding *any* one of the 61 data-structure messages, were thwarted by the amount of time required for its proof to execute: the system on which we ran the proof never stayed up long enough to complete it. We estimated it would take at least seven days for the proof to terminate.

The problem was that in proving that the nth block of commands was processed correctly, each of the 61 possible cases for this block had to be considered separately (at least in our approach). Each case added to the execution time of the proof. One of our solutions was to run many separate but similar proofs. In each proof we considered only a few of the cases for the nth block of commands, proved only those cases, and deferred the proof for the other cases. But we still could not prove that four especially long data-structure messages were processed correctly; the proof for each one of these messages required not only more storage than that allowed by the SDVS image (about 70 megabytes), but even more storage than that allowed by the SDVS image we created specifically for the MSX verification project (300 megabytes). We were thus unable to verify the correctness for these four messages.

The proofs we were constructing in SDVS were long and repetitive. To facilitate their construction, we prototyped three new proof commands in SDVS; this greatly reduced the amount of work required to develop subproofs for the 61 cases.

Towards the latter part of the project, we realized that many of our time and storage problems would be solved if we proved adalemmas for some of the key subprograms in BUILD and in CMD_IN_HANDLER. We were able to develop and prove adalemmas for two of the main procedures in BUILD.

5 Summary of the Correctness Theorem and the Two Proof Strategies

In this section we discuss the correctness proofs for the final version of the modified target software: the Ada program MSX_PROGRAM_FINAL_VERSION. The body of this main program (i.e., the scheduler) consists of a simple (infinite) Ada loop that (1) calls CMD_IN_HANDLER precisely the number of times needed to obtain the number of bytes required for the construction of the next data-structure message; (2) calls BUILD to construct the message; and (3) calls PROCESS_MSG to write the message either in EEPROM, RAM, or both.

Stripped of detail and roughly stated in English, the correctness assertion that we proved for this program is, "the Ada program MSX_PROGRAM_FINAL_VERSION correctly extracts and reformats embedded data-structure messages from an infinite input stream of uncorrupted bytes." The proof of correctness, then, is a proof of the above assertion (expressed formally). More specifically, the formal assertion states that eventually the block of words output by MSX_PROGRAM_FINAL_VERSION for the nth message is the appropriately reformatted version of the relevant input bytes for that message. Proving this correctness assertion involves symbolically executing through the program to a point where the stated relationship between input and output is true. The formal specification and the complete proofs are documented in [1].

We attempted two different approaches to the proof:

1. the symbolic execution of the Ada portions of the program, including symbolic execution through all invocations of the Ada subprograms, and the use of (prototype implementations of) meta-level proof commands to develop subproofs of similar cases;

2. the abstract characterization and proof of properties for major Ada subprograms, and then the use of these abstract characterizations upon invocation of the subprograms.

The first approach was attempted because we thought that the second approach would not be feasible within the time constraints for this experiment. Although the first approach was conceptually easier and the newly implemented meta-proof commands greatly assisted the proof construction, it took what we felt was an inordinate amount of time to execute the proof. In spite of the lengthy execution time, most of the proof of our correctness assertion was completed in SDVS: 57 out of the 61 types of data-structure messages were proved correct. Towards the end of the verification experiment, we embarked on the second approach, which was not as difficult as we first imagined.[11] The characterization and proof for two major subprograms were completed and a third was partially completed. This approach greatly reduces the time/space "explosion" and will permit the verification of more general schedulers and input. Below we discuss each of these approaches in more detail.

[11]The second approach is not disjoint from the first: adalemmas may be substituted for the invocation of selected subprograms.

5.1 Discussion of First Proof Strategy

First, consider verification approach (1) above. The first step in symbolically executing through
MSX_PROGRAM_FINAL_VERSION is to elaborate all of the declarations. After this is done, we are at the main body of the program. Since the main body of the program is an infinite loop that processes exactly one message at a time, when the loop completes its nth iteration, the block of input bytes for the nth message has been processed and output. Thus after the nth iteration of the loop, the relationship between the input and output stated in the correctness assertion should hold, and so we want to execute symbolically to this point. However the n in the correctness assertion represents an arbitrary number, and so its value is symbolic, not concrete. This means we cannot simply execute symbolically through the loop n times, but must use a form of induction known as *loop induction*, which is closely related to the familiar form of mathematical induction. The following brief discussion illustrates the similarities.

In the usual form of induction, to prove the statement

$$\forall n P(n)$$

we must first prove the base case

$$P(0)$$

and the step case

$$P(n) \rightarrow P(n+1)$$

In loop induction, to prove

P(n) is true after n iterations of the loop, where n is an arbitrary number

we must first prove the base case

P(0) is true before the loop is executed, i.e., after 0 iterations

and the step case

If P(n) is true after n iterations of the loop, then P(n+1) is true after symbolically executing through the loop one more time, i.e., after $n+1$ iterations

The formula $P(n)$ above is known as the *loop invariant.* We chose the loop invariant to be strong enough to prove the goal. Hence most of the work in developing the proof of correctness was in the construction of the loop invariant.

The base case in the loop induction is straightforward, involving only two proof commands. Since many of the variables in the loop invariant are assigned values when they are declared, these variables have concrete values when the beginning of the loop is reached for the first time. Thus much of the work in proving the invariant for the base case involves simplifying expressions involving only concrete values, and the Simplifier can verify most of these automatically.

However, in the step case, we cannot simply execute symbolically through the loop. The reason for this is that the number of times certain loops are executed and if certain branches are taken are dependent on the length of the message, which in turn is a function of the message identifier. In fact, the value of the message identifier completely determines the execution path through the loop, since the values of all variables that determine the execution path through the loop and that are not already fixed by the invariant depend on the message identifier. Whenever there is only one execution path through a portion of code, the proof of that portion of code is straightforward (or at least involves no case splits).

Motivated by the above observations, we split the proof of the step case into 61 cases based on the possible range of message identifiers for data-structure messages. As pointed out above, this case split ensures that there is only one execution path through the loop in each case, and thus that the subproof for each case has a simple structure. Thus, while this way of proving the step case has the disadvantage of creating 61 separate cases for this proof, it has the advantage that the proof of each case is relatively simple. In fact, the proofs for the different cases were so simple and similar in structure, differing primarily in the number of times certain functions were called, that we decided to implement new proof commands so that we could write a single meta-level proof for each of the cases.

Because of computer crashes, we became concerned that we would not be able to execute the proof completely, as this execution took roughly a week. Hence, we elected to do multiple runs of the main proof, with each run handling a few cases at a time. However, we were unable to complete 4 of the 61 cases. In these cases (which corresponded to the longest messages and hence those that required the most resource-intensive proofs), SDVS failed to complete the proof, in spite of efforts such as increasing the amount of storage in the SDVS image and changing the garbage-collection strategy.

5.2 Discussion of Second Proof Strategy

We now turn to verification approach (2). The first proof strategy had two distinct disadvantages: it was time-consuming, and it did not allow for easy generalization if we changed the order in which the procedures in the body of the main loop were called. One approach we explored to overcome both of these problems was to write and prove abstract characterizations (known as adalemmas) for various procedures. This approach saves time in that using an adalemma for a procedure instead of symbolically executing through that procedure accomplishes in one proof step what typically takes many proof steps. Also, since an adalemma for a procedure characterizes the effect of that procedure abstractly, it can be used whenever that procedure is called in the program. Thus the adalemmas we wrote could be used in proofs of the software with different schedulers.

Specifically, we wrote and proved adalemmas not only for the procedure that processes the first command of a message, but also for the procedure that is repeatedly called to process every remaining command of a message. We also made substantial progress in creating and proving an adalemma for the procedure that uses these two procedures to extract the relevant input bytes of a message and reformat them prior to additional processing. For one of the types of data-structure messages, we were able to modify the initial proof to employ these adalemmas when the procedures they characterized were called (rather than symbolically executing through them), and to run this modified proof successfully. For more information about both of these approaches, see [1].

6 Main Accomplishments and Results

This project is one of the largest experiments in the formal verification of production Ada code.[12] The importance to the verification of the target software would have been much greater if the latter did not involve tasking.[13]

We had previously tested SDVS on relatively short textbook-type Ada programs. The size of the MSX code and its complexity unveiled problems in SDVS that had not been detected in the proofs of smaller examples. The main problem encountered was the time and storage explosions that accompanied the execution of the long proofs. This problem was at least partially caused by the large number of object declarations in the MSX software.[14]

We were pleased by the way we were able to specify the input and output conditions for an infinite array of blocks of commands encoding correct data-structure messages. This specification was initially neither obvious nor entirely trivial.

Our initial walk-through of the software and the correspondence between Aerospace and JHU/APL were instrumental in the discovery of two errors in the target software. We suspect that if there are any errors in the code, they are in those parts that we did not verify: our scheduler was restrictive to the point of not allowing complex situations to arise. For example, the message queue was always either empty or contained precisely one message. Furthermore, the documentation we examined for the specifications never specified precisely what was to be done in unusual situations, e.g. how lost commands should be handled.

The verification effort, although restricted in scope, did add some measure of confidence in the correctness of the code. We were able to prove that 57 of the data-structure messages were received, built, and stored in EEPROM according to the specifications.

As a result of the verification effort, we enhanced the SDVS Ada environment by extending the SDVS Ada translator and enhancing the SDVS Ada proof capabilities.

[12] A recent substantial experiment in the formal verification of Ada software has been the application of Penelope (a formal verification system for Ada developed by Odyssey Research Associates, Inc.) in a proof of correctness of a portion of Grady Booch's Calendar_Utilities package (about 300 lines of code) [18].

[13] We know of no formal verification system that handles Ada tasking.

[14] We tested the system with a trivial Ada program having 1000 object declarations, and ran into the same storage problems.

References

[1] T. K. Menas, J. M. Bouler, and J. E. Doner, "Specifications and correctness proofs for portions of the MSX Ada software," Tech. Rep. ATR-93(3778)-5, The Aerospace Corporation, Sept. 1993.

[2] J. V. Cook, I. V. Filippenko, B. H. Levy, L. G. Marcus, and T. K. Menas, "Formal Computer Verification in the State Delta Verification System (SDVS)," in *Proceedings of the AIAA Computing in Aerospace Conference*, (Baltimore, Maryland), pp. 77–87, American Institute of Aeronautics and Astronautics, Oct. 1991.

[3] B. Levy, I. Filippenko, L. Marcus, and T. Menas, "Using the State Delta Verification System (SDVS) for Hardware Verification," in *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience: Nijmegen, The Netherlands* (V. Stavridou, T. F. Melham, and R. T. Boute, eds.), pp. 337–360, North-Holland, June 1992.

[4] L. G. Marcus, "SDVS 12 Users' Manual," Tech. Rep. ATR-93(3778)-4, The Aerospace Corporation, Sept. 1993.

[5] T. K. Menas, "SDVS 11 Tutorial," Tech. Rep. ATR-92(2778)-12, The Aerospace Corporation, Sept. 1992.

[6] G. Heyler, S. Hutton, and R. L. Waddell, "Midcourse space experiment (MSX) tracking processor software detailed design document," Tech. Rep. S1A-084-91, JHU/APL, 1991.

[7] R. L. Waddell and S. Hutton, "Midcourse space experiment (MSX) tracking processor software functional design," Tech. Rep. S1A-031-90, JHU/APL, 1990.

[8] S. Hutton, "Tracking processor/command processor interface design specification (Version 2)," Tech. Rep. S1A-136-91, JHU/APL, 1991.

[9] L. G. Marcus, "Preliminary Investigations into Specifying and Proving Ada Floating-Point Programs in the State Delta Verification System (SDVS)," Tech. Rep. ATR-91(6778)-4, The Aerospace Corporation, Sept. 1991.

[10] L. G. Marcus, "The Semantics of Ada Access Types (Pointers) in SDVS," Tech. Rep. ATR-92(2778)-5, The Aerospace Corporation, Sept. 1992.

[11] T. K. Menas, "The Relation of the Temporal Logic of the State Delta Verification System (SDVS) to Classical First-Order Temporal Logic," Tech. Rep. ATR-90(5778)-10, The Aerospace Corporation, Sept. 1990.

[12] S. D. Crocker, *State Deltas: A Formalism for Representing Segments of Computation.* PhD thesis, University of California, Los Angeles, 1977.

[13] T. Redmond, "Simplifier Description," Tech. Rep. ATR-86A(8554)-2, The Aerospace Corporation, Sept. 1987.

[14] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Trans. Programming Languages and Systems*, vol. 1, pp. 245–257, Oct. 1979.

[15] D. F. Martin and J. V. Cook, "Adding Ada program verification capability to the State Delta Verification System (SDVS)," in *Proceedings of the 11th National Computer Security Conference*, National Bureau of Standards/National Computer Security Center, Oct. 1988.

[16] D. Guaspari, C. Marceau, and W. Polak, "Formal Verification of Ada Programs," *IEEE Trans. Software Engineering*, vol. SE-16, pp. 1058–1075, Sept. 1990.

[17] D. C. Luckham, F. W. von Henke, B. Krieg-Bruckner, and O. Owe, *ANNA - A Language for Annotating Ada Programs.* Berlin: Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 260.

[18] C. T. Eichenlaub, C. D. Harper, and G. Hird, "Using Penelope to assess the correctness of NASA Ada software: A demonstration of formal methods as a counterpart to testing," Tech. Rep. Contractor Report 4509, NASA, May 1993.