SOFTWARE METRICS, ADA, AND THE B-2 ATD

Paul E. McMahon, Staff Scientist
Dennis W. Meehl, Section Head
CAE-Link Corporation
Binghamton, New York

ABSTRACT

Many believe the greatest benefit of Ada is that it encourages software engineers to explore new design approaches leading to higher quality software. However, Ada's primary goal is to reduce the life cycle cost of software. Furthermore, the relationship between cost and modern software techniques is not always evident. This paper addresses the cost of Ada software. How long does it take an engineer to develop software when using Ada and modern software engineering techniques? How much computational capacity does Ada require? This paper provides answers to these questions based on data from the B-2 Aircrew Training Device (ATD). Lines of code, development time, and computational resources are provided for selected B-2 ATD software systems. Key contributing factors include the cost of training engineers in modern software techniques and the impact caused by developing and using more modern software tools. This paper identifies key factors found on the B-2 ATD to be influential in affecting today's software cost and explains what we are doing to reduce this impact in the future.

INTRODUCTION

The B-2 ATD was the first major training device at Link to use Ada. With over 1.7 million lines of code, it was also one of the most complex. Factors contributing to software cost on a first Ada project include on-the-job training in new design techniques and costs associated with new tools. The impact of these factors is expected to decrease on future Ada projects.

There are no simple answers to software metrics with Ada. Simple formulas fail to consider many of the possibilities. Ada metrics can be sensitive to design techniques and compiler implementations. These situations must be identified and managed. This paper is intended to provide information that can help in estimating, and also understanding, software costs with Ada.

Ten software components (CSCs in DOD-STD-2167A terminology) were selected for this study. Eight real-time and two non-real-time systems were investigated. The real-time systems were selected from the disciplines of aerodynamics and avionics. For the systems selected, the design engineer's experience in simulation ranged from 2 to 25 years. Ada experience ranged from the first to the fourth Ada assignment. The following data were obtained:

- Development Hours – Development hours were derived from our Management Control and Information System (MCIS). MCIS is a performance measurement system and is a validated Cost/Schedule Control System (C/SCS).
- Lines of Code – Lines of code were measured by a source code scanning tool. Both "carriage return" lines of code and "semicolon" lines are reported.
- Bytes of Memory – Memory requirements were derived from computer vendor load maps and measured stack space needs for program execution.
- Execution Time – Execution time was measured using a microsecond clock on the real-time target system.

HISTORY OF PROJECT

Resource Estimation

Initial computational resource estimates on the B-2 ATD were based on a Fortran to Ada translated benchmark known as Mainflt. Translating Mainflt resulted in Ada code employing primarily integer, float, and boolean data types.

This benchmark required 40 bytes per line of code and 1.25 times the equivalent Fortran execution time. Additional resources for new design techniques with Ada were unknown at the start of the project. The cost of new design techniques with Ada is discussed later in the paper.

Development Environment

The B-2 ATD software was developed using one of the most mature Ada development environments available today. After test, the software is transferred through a local area network to the target system, where it is compiled again and linked. The target computer has its own distinct operating system and Ada compilation environment. A single processor on the target system provides approximately 6 VUPS (VAX units of processing speed) of computing power. The B-2 Weapon System Trainer (WST) requires approximately 20 processors, or 120 VUPS. The target provides a real-time non-virtual operating environment with limits, such as memory, that do not exist in the development environment.

Our experience indicates that the resource demands of Ada compilers may differ. It is not recommended that the numbers presented in this paper be applied to estimates for other Ada environments. This information should be viewed as trend data only. Benchmarking one's chosen compiler and target hardware is necessary to arrive at accurate resource estimates.

SOFTWARE STRUCTURE MODEL

Many simulation design issues are common, providing an opportunity for reuse. One vehicle that aids us in applying reuse at Link is our software structure model developed specifically for use with Ada. This model has been developed and coordinated with members of the Software Engineering Institute (SEI) staff.

In this section a brief description of the model and real-time environment is provided. This subject is discussed here because understanding the software structure is helpful in understanding some of the new cost trends with Ada. The structure model can also be used in reducing software costs.

Real-Time Environment

The B-2 simulation software runs in a tightly-coupled parallel-processor environment. Separate processors communicate through a global memory system. Application software does not directly access global memory except for time-critical applications. Communication through global memory, common file I/O services, and executive control are provided by automatically generated software.

Interface Management

During the design phase, global interfaces are defined through a data base referred to as the Interface Management Data Base (IMDB). An off-line processor uses the IMDB to generate GLOBAL DATA PACKAGES and IMPORT and EXPORT PROCEDURES. These procedures move the data in real time to and from the global packages at rates specified in a control file. Imports are moved to local IMPORT PACKAGES and exports are moved from DECLARATION PACKAGES. The automatically generated IMPORT and EXPORT procedures are referred to as CONNECTION MANAGERS.

Application Software

CONTROL MANAGERS are called at rates specified in a control file by an automatically generated EXECUTIVE. A CONTROL MANAGER is the top-level user procedure and usually controls software the equivalent of a DOD-STD-2167A CSC. However, a single control manager may control multiple CSCs, or a single CSC may have multiple control managers.

"Objects", in an object-oriented design (OOD) sense, are defined in OBJECT DEFINITION PACKAGES. These packages define Ada data types and Ada procedures that operate on these types. The CONTROL MANAGERS invoke these "objects", passing data from the IMPORT PACKAGES and DECLARATION PACKAGES. OBJECT DEFINITION PACKAGES may contain only types or types and procedures together.

Real-time application designers develop CONTROL MANAGERS, DECLARATION PACKAGES, IMPORT PACKAGES and DEFINITION PACKAGES. The EXECUTIVE, IMPORT and EXPORT CONNECTION MANAGERS, and GLOBAL DATA PACKAGES are automatically generated.

Real-time file I/O services required by the application code are created, using Ada's generic capability, based on the application types. Direct I/O is provided to application software for both local and remote file access. Figure 1 is a diagram of the Structure Model Components.

DEFINITIONS

Closure

Closure consists of all the Ada units required in the library for a given unit to compile (compilation closure) or link (execution closure).

Design Phase

The design phase includes the development of the CONTROL MANAGER specifications, DEFINITION PACKAGE specifications, DECLARATION PACKAGES, and IMPORT PACKAGES.

Code and Test Phase

In the code and test phase the Ada bodies are completed for the CONTROL MANAGERS and the DEFINITION PACKAGES. Simulation algorithms reside in the bodies.

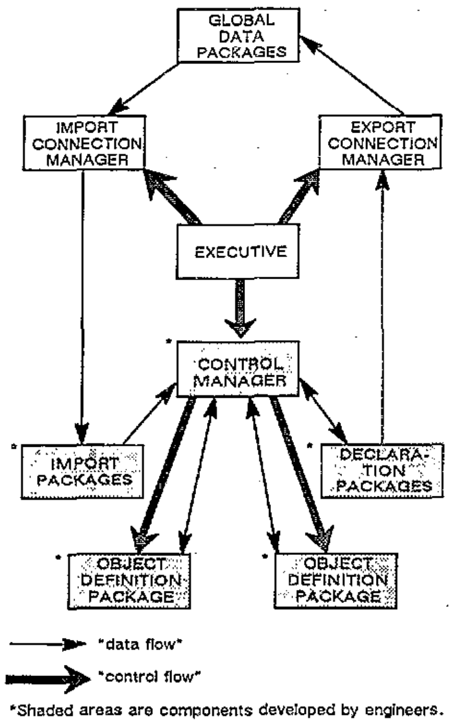
GLOBAL DATA PACKAGES
IMPORT CONNECTION MANAGER
EXPORT CONNECTION MANAGER
EXECUTIVE
CONTROL MANAGER
IMPORT PACKAGES
DECLARATION PACKAGES
OBJECT DEFINITION PACKAGE
OBJECT DEFINITION PACKAGE
"data flow"
"control flow"
*Shaded areas are components developed by engineers.

Figure 1 Structure Model Components

Ada Lines of Code

In this study we report both the number of semicolons and the number of lines with carriage returns containing Ada compilable statements. Lines of code generated automatically by off-line processors and generic instantiations are reported separately and are not used in productivity calculations since engineers do not manually generate this software.

Compiler-Generated Code

Compiler-Generated Code consists of all executable instructions generated by the compiler. This includes elaboration code and user code.

User Code

User code consists of all compiler-generated code produced from Ada statements found after the BEGIN in procedure and function bodies. The code that carries out the simulation algorithms is user code.

Elaboration Code

Elaboration code consists of all compiler-generated code used solely to carry out MIL-STD-1815A elaboration rules. Elaboration code can be thought of as set-up code. It is only run in preparation for executing user code.

Static Data

Static data includes all Ada variables allocated to dedicated memory locations. Static data remain fixed in size and location throughout the simulation exercise.

Stack

The stack has two parts. First, the stack is used to elaborate packages. This only occurs once prior to the start of simulation. Secondly, the stack consists of temporary data used during simulation. This temporary data on the stack does not retain its value between program calls, does not remain fixed in location between program calls, and may not be fixed in size.

DATA ANALYSIS

For each of the systems analyzed, Table 1 provides numbers of Ada units, lines of code, and memory requirements. Table 2 provides experience levels of the software engineers assigned to develop these systems. Development time is discussed later in the paper.

This data indicates that traditional methods used to estimate computational resources may fall short with Ada. This is because there are new and influential factors with Ada. Such factors include the use of composite types (records and arrays), elaboration code, overhead for packaging system services, stacks, and the use of generics. These five factors are discussed in the following paragraphs.

Composite Types in Ada

The resources required for a single Ada statement can vary dramatically as a result of Ada's composite type capabilities. Consider the data for Forces_And_Moments, Aero_Coefficients, and Nav_Geography in Table 1. These systems average 66, 41, and 39 bytes per semicolon per declaration package. The data in these systems consists primarily of "small" records each containing 5-10 scalar components. On the other hand, the Mass_Storage_Unit CSC averages 1930 bytes per semicolon per declaration packages. This is due to a single declaration requiring over 150 kilobytes.

In many cases with Ada, we are seeing complexity moving out of the code and into the data. Data structures can become complex rapidly. Evidence of this fact is found in the size of the declaration packages.

Table 1 B-2 ATD Lines of Code and Memory Requirements

Structure (Note 1)	No. of Ada Units	No. of Ada Lines <CRs>	No. of ;s	Constants	Static Data	Compiler Generated Code	Total Bytes	Bytes Per ;	Avg. ; Per Unit
Real-Time Systems									
Forces and Moments									
Defns	5	388	209	140	544	196	880	4	42
Declar	2	108	51	40	1488	1848	3376	66	26
Bodies	5	449	216	20	80	6260	6320	29	43
Exp Imp	2	586	270	112	48	2272	2432	9	135
Total	12	945	476	200	2112	8264	10576	22	40
Mass Storage Unit									
Defns	6	1049	795	1752	368	24	2144	3	133
Declar	4	136	85	2184	161504	424	164112	1930	21
Bodies	5	4640	2321	3400	4848	70168	78416	34	464
Exp Imp	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Rtio	3	36	21	28	10736	164	10928	520	7
Total	15	5825	3201	7336	166720	70616	244672	76	213
Weight and Balance									
Defns	2	193	141	136	160	184	480	3	71
Declar	2	374	130	116	4200	6844	18160	140	65
Bodies	10	744	387	136	160	10360	10656	28	39
Exp Imp	2	2058	1059	636	32	6116	6787	6	530
Total	14	1311	658	388	11520	17388	29296	45	47
Aero Coefficients									
Defns	7	423	329	144	1872	2312	4328	13	47
Declar	2	60	57	24	720	1608	2352	41	29
Bodies	19	2494	1065	196	1328	30488	32012	30	56
Exp Imp	2	575	286	252	48	6100	6400	22	143
Total*	28	2977	1451	364	3920	34408	38692	27	52
Nav Geography									
Defns	2	345	297	48	96	16	160	1	74
Declar	2	183	81	760	1408	952	3120	39	41
Bodies	2	2198	856	1244	400	16468	18112	21	428
Exp Imp	2	428	215	16	48	2540	2604	12	108
Total	6	2726	1234	2052	1904	17436	21396	17	154
Motion									
Defns	10	713	453	992	352	848	2192	5	45
Declar	2	181	42	24	2432	216	2672	64	21
Bodies	10	1402	703	204	224	17700	18128	26	70
Exp Imp	2	428	250	224	80	4960	5264	21	125
Gen Ins	11	N/A	N/A	92	192	7620	7904	N/A	N/A
Total	22	2296	1198	1220	3008	18764	22992	19	55

Note 1:
Exp Imp means EXPORT/IMPORT PROCEDURES. See Software Structure Model.
Rtio means Real-Time I/O Auto-Generated Software
GenIns means Generic Instantiation
Defns means DEFINITION PACKAGES. See Software Structure Model.
Declar means DECLARATION and IMPORT PACKAGES. See Software Structure Model.

Note 2:
Totals do not include automatically generated software.
This includes Export/Import software, Real-Time I/O, and Generic Instantiations.

Table 1 B–2 ATD Lines of Code and Memory Requirements (Cont'd)

Structure (Note 1)	No. of Ada Units	No. of Ada Lines <CRs>	No. of ;s	Constants	Static Data	Compiler Generated Code	Total Bytes	Bytes Per ;	Avg. ; Per Unit
Motion Supplement									
Defns	18	810	443	1036	1792	68	2896	7	25
Declar	2	269	99	2356	3376	460	6192	63	50
Bodies	16	1274	665	204	816	15236	16256	25	42
Exp Imp	2	1103	424	84	48	4796	4828	12	212
Gen Ins	14	N/A	N/A	324	292	6312	6928	N/A	N/A
Total	36	2353	1207	3596	5984	15764	25344	21	34
Seat Shaker									
Defns	9	278	212	268	240	308	808	4	24
Declar	2	117	47	140	336	196	672	14	24
Bodies	7	497	244	108	160	7684	7952	33	35
Exp Imp	2	298	190	120	64	3912	4096	22	95
Gen Ins	2	N/A	N/A	16	32	688	736	N/A	N/A
Total	18	892	503	516	736	8188	9432	19	28
LFI Data (Sample)									
1 Var	N/A	32	16	92	176	468	736	46	N/A
2 Var	N/A	48	16	364	672	564	1600	100	N/A
3 Var	N/A	855	68	11160	21984	3608	36752	540	N/A
Real–Time Totals									
Defns	59		2879				13888	5	
Declar	18		592				200656	339	
Bodies	74		6457				187852	29	
Exp/Imp	14		2694				32508	12	
Gen Ins	27		N/A				15568	N/A	
Non–Real–Time Systems									
Master Bootstrap									
Defns	9	461	292	1112	12480	1732	15324	52	32
Declar	6	30	30	444	12544	1872	14860	495	5
Bodies	40	2290	1343	18024	1148	73700	92870	69	34
Total	55	2781	1665	19580	26172	77304	123056	74	30
Slave Bootstrap									
Defns	2	12	8	24	48	8	80	10	4
Declar	1	7	5	12	16	4	32	6	5
Bodies	22	1154	660	2996	608	25352	28956	44	30
Total	25	1173	673	3032	672	25364	29068	43	27

Note 1:
Exp Imp means EXPORT/IMPORT PROCEDURES. See Software Structure Model.
Rtio means Real–Time I/O Auto–Generated Software
GenIns means Generic Instantiation
Defns means DEFINITION PACKAGES. See Software Structure Model.
Declar means DECLARATION and IMPORT PACKAGES. See Software Structure Model.

Note 2:
Totals do not include automatically generated software.
This includes Export/Import software, Real–Time I/O, and Generic instantiations.

Table 2 Personnel Experience

System	Engineer	Years of Experience in Simulation	Years of Prior Ada Experience
Forces and Moments, Aero Coefficients, Weight and Balance	A	25	0
Mass Storage Unit	B	2	3
Slave Bootstrap, Master Bootstrap	C	1	1
Nav Geography	D	15	0
Motion, Seat Shaker, Supplemental Motion	E	9	0

Declaration packages tend to require significantly more memory per semicolon than other structural components. On average, the B–2 data indicates that our estimate of 40 bytes per semicolon, based on our Mainflt benchmark, is valid for definition packages and bodies. For declaration packages this estimate may be off by a factor as great as 10 or more. The average number of bytes per semicolon of all measured declaration packages was 375 (339 for real-time alone).

However, by not including the large data structure within the Mass_Storage_Unit, the average number of bytes per semicolon for declaration packages drops to 77. This analysis may indicate that large data structures within the application software should be accounted for together with other large data structures such as global/hardware interfaces. As a result, one can use a single "bytes per semicolon" estimate treating all structural components as code. This simplifies the resource estimation process for the application designer.

Elaboration Code

In the data analyzed, only about half of the memory required for declaration packages was allocated for user data. The other half was elaboration code. Any compiler–generated code in a declaration or definition package is elaboration code. User code resides only in bodies.

Ada compilers today can generate significant amounts of code in carrying out the elaboration rules of Ada. Elaboration code can be particularly costly in initializing data of a composite type. As an example, compilers may generate elaboration code to initialize records and arrays even when the initial values are known at compilation time. Compiler enhancements in this area can result in significant memory savings.

OVERHEAD COSTS

The memory demands of Ada extend beyond the application software itself. Memory required for stacks, generic instantiations, and "closure" units adds to the total resource picture.

Closure

All units in another unit's closure do not necessarily need to be loaded into memory at execution time. A package specification that contains only type information may be needed only during compilation. However, when "withed" units contain information that may change at run time or cannot be determined at compilation time, this unit may need to be loaded at run time.

When multiple procedures and/or functions are placed within a single Ada package, all of the software or only those procedures or functions actually referenced may require memory. These memory issues are dependent on compiler vendor implementations and can result in different memory demands between compilers.

An Example of Overhead Costs

On the B–2 ATD, Bootstrap is a set of interactive OS processes providing a menu–driven capability to initiate various simulator functions, such as simulator loading. Slave_Bootstrap provides control for a single OS environment and Master_Bootstrap provides common control over all Slave_Bootstraps. The total amount of memory required by the Slave_Bootstrap application software is 95 kilobytes. This includes 65 KB of software reused from Master_Bootstrap. However, the OS services required for such functions as loading and starting tasks require another 120 kilobytes. The package text_io requires 200 kilobytes. The stack requires 110 kilobytes and closure units add 35 kilobytes. As a result, the total task size of Slave_Bootstrap is 465 kilobytes. Master_Bootstrap requires 400 kilobytes.

Stacks

Most simulation processes on the B–2 ATD require approximately a 500 KB stack. Stack demands, however, are highly dependent on user software characteristics. It is not unusual for some applications to require stacks as large as 2 MB or larger. This includes the stack space required to initially elaborate packages as well as the space needed to execute the simulation programs. Determining worst–case stack requirements may demand special tools. We have used a specially developed vendor tool reporting worst–case stack needs to assist us in managing this resource.

Generics

Generic instantiations result in complete software units from a single Ada statement. Dependent on the compiler, these units may require computational resources similar to manually generated units.

Most of the systems analyzed do not use generics. Weight_and_Balance, Aero_Coefficients, Nav_Geography, and Forces_and_Moments do not utilize generics. It is not uncommon for first and second Ada systems to make little use of generics. Generics tend to be used by more experienced Ada engineers. However, when generics are employed, their resource demands can be substantial. Large quantities of code requiring significant resources can be generated rapidly when using generics. Our Motion and Motion Supplemental CSCs utilize generics (see Table 1).

Unconstrained Types

Unconstrained types in Ada provide simulation software engineers with a powerful new capability for managing data. The resource demands of this new feature, however, may be more costly than expected.

An Example

On the B-2 ATD project, an off-line processor called the LFI (Linear Function Interpolation) Compiler is used to transform aircraft data sets into an Ada-compilable format for use in the training device. The data is interpolated in real time, frequently at high rates. Since each data set may vary in size, the use of an unconstrained Ada type was chosen.

We found with our real-time compiler that objects of an unconstrained record type required more computational time and memory than expected. In certain cases the maximum size of an object rather than the actual size was allocated both on the stack and in memory. This maximum was almost 2 MB. Furthermore, additional computational time was required since these objects were being passed on the stack as parameters to an interpolation routine. Objects of a constrained record type are passed as parameters more efficiently.

Limits and Automatically Generated (Auto-Gen) Software

We also found that some of the larger LFI data sets caused a constant table limit to be exceeded within the compiler and used excessive stack space during elaboration which was never recovered.

The LFI data sets are not the only Ada software automatically generated (Auto-Gen) on the B-2. Import and Export Procedures are automatically generated from the Interface Management Data Base (IMDB). Real-time disk I/O software employs generics. The Executive software is partially generic and partially automatically generated.

Automatically generated software can enhance productivity and reliability and is highly recommended. However, our experience indicates that real-time software employing unconstrained types and Auto-Gen software may have an increased risk of encountering target compiler limitations or inefficient use of resources. These conditions may not be evident when initially developing the software in a virtual non-real-time environment.

ADA CODE CHARACTERISTICS

Counting Ada Lines

Today there is not a single accepted standard for measuring lines of code in Ada. In this paper both lines with carriage returns and lines with semicolons are reported. We have seen approximately a 2 to 1 ratio between "carriage return" lines of Ada and semicolon lines. However, this relationship can vary depending on the particular Ada constructs employed and coding style. As an example, in the case of a 3-variable LFI (see Table 1) containing a large aggregate, this ratio is more than 10 to 1. Nevertheless, our experience indicates that managing size is best accomplished by focusing on terminating semicolons.

Comparing Ada to Other Languages

Our experience indicates that Ada may require more lines of code than languages such as Fortran. This is partially a result of design techniques and coding style, but it is also a result of the language itself.

Table 3 indicates that the ratio of code contained in the DEFINITION AND DECLARATION PACKAGES to the total manually generated code averages 41% for the real-time software analyzed. This may indicate that we can expect 40% more lines of code with Ada. This statistic also indicates that 40% of the Ada code generated occurs in the design phase.

Table 3 Percentage of Ada Generated During Design

System	Percentage of Design Code (DEFNS, DECLARS) to Total
Forces_And_Moments	54%
Mass_Storage_Unit	38%
Weight_And_Balance	41%
Aero_Coefficients	27%
Nav_Geography	31%
Motion	41%
Motion_Supplement	45%
Seat_Shaker	51%
Average	41%

In Fortran a preprocessor to compilation added interface declarations from a symbol dictionary. These declarations were not part of lines of code management. In Ada, these declarations are included in lines of code management.

EXECUTION TIME

Execution time can vary depending on loops and branch paths. Nevertheless, when code is primarily straight line and the data used is small records or scalars, estimates based on source line counts are possible.

Table 4 indicates a range of 0.5 microseconds to 4.1 microseconds per semicolon for the systems measured. Be advised that all of the instructions in these systems may not have been exercised during timing. However, this data indicates that Ada compilers today can generate code that meets stringent real-time simulation needs. In fact, we have found that managing real-time computational time is actually more of a simulation software design issue than a compiler issue.

Table 4 Real-Time Software Execution Time

System	Measured Execution Time*	Time* Per Semicolon**
Forces_And_Moments	240	1.1
Motion	320	0.5
Seat-Shaker	440	1.8
Nav_Geography	1482	1.7
Weight-And_Balance	1570	4.1

* Execution Time is reported in microseconds

** Manually generated bodies only are used for this calculation

The reader is cautioned against applying this data to code with varying design techniques and styles. Deep nesting of small procedures, the use of large composite data types, or the use of unconstrained types can dramatically alter timing results. For example, in a separate case study of a high-rate CSC on the B-2 ATD, 6-10 microseconds per semicolon was measured. Characteristics of this CSC included many array references, procedure calls, and loops.

DEVELOPMENT TIME

The ultimate success of Ada may rest with how favorably engineering productivity compares with previous generation languages. Table 5 provides the hours required to develop the eight real-time systems studied from the B-2 ATD. Productivity ranges from 0.7 to 4.0 Ada statements (semicolons) per hour. In all the cases analyzed productivity improved (frequently by 100% or more) moving from the design stage to code and test. This is believed to be due to two factors.

First, designing the definition packages requires considerably more thought than coding the bodies. Secondly, on a first and second Ada assignment, on-the-job training in Object Oriented Design (OOD), costs associated with new tools, and rework due to immature compilers all impact cost, particularly in the early design stages.

We have found that the OOD cost frequently includes a redesign of the first system and a refinement cost on second and third assignments. Despite this situation, our data indicates that developing software with Ada can be cost-competitive today. Once Ada experience has been gained, and a process and mature toolset put in place, further productivity gains can be expected. While individual results will vary, dramatic productivity gains can occur, as seen from our experience with the Mass_Storage_Unit.

RECOMMENDATIONS

Use Small Team Early

Ada's goal is to reduce the life cycle cost of software. New factors with Ada can increase complexity leading to higher, rather than lower, software maintenance costs. The use of small teams with focused goals can play a key role in managing this complexity early.

We recommend that a small team investigate the factors discussed in this paper as early in the project as possible. This activity must occur on the chosen real-time target system.

Our goal is to keep the software process simple. Software designers need clear and concise rules to achieve maintainable software. These rules must be based on the structure model, and on target-specific factors that can be learned only through prototyping. Rules for estimating resources can also be kept simple by managing large data structures as system data, allowing a common approach to all application structural components.

Table 5 Development Time

System	Design Time*	Code and Test Time*	; Per Hour Design	; Per Hour Bodies	; Per Hour Total
Mass_Storage_Unit	436	360	2.0	6.4	4.0
Nav_Geography	1200	700	0.3	1.2	0.7
Weight_And_Balance, Aero_Coefficients, Forces_And_Moments	1753	2140	0.5	0.8	0.7
Motion, Motion_Supplement, Seat_Shaker	1695	939	0.8	1.7	1.1

* Time reported in hours

Data types used for system interfaces must be established and controlled early. Typing strategies must consider both the value of modern software techniques and the "side effects" that may occur on one's chosen machine. The impact of data structures (such as structures using unconstrained and composite types) on stacks, elaboration code, and closure software must be known early and considered in establishing the rules.

Use Compiler Options When the Software Is Mature

We have found that once the real-time software is mature, significant computational resource savings can be gained through compilation options. While individual results may vary, we have seen execution time reductions of 30% when disabling run-time checks and a 25% savings in memory. By in-lining small units (less than 5 semicolons) another 10% reduction in execution time can be achieved, although in-lining does increase memory.

To gain these benefits it is important not to "design in" Ada's run-time checks. For example, one should not employ constraint exception processing to limit data. Otherwise, the software will not operate properly when the checks are disabled.

We recommend that these options be employed only toward the end of the project when the software is mature and the full impact is clear. Ada's run-time checks (e.g., range checking of interface data, ranging of indexes, etc.) are invaluable during test and in-lining can increase recompilation during development when the software needs to be modified frequently.

DISCIPLINE AND COMPUTER SYSTEMS SUPPORT

Although hardware/software integration (HSI) time is not included in this study, our experience on the B-2 ATD project indicates that HSI time is considerably shorter with Ada. While the time to build a load is longer, fewer loads are required to attain functionality. Although this is partially due to Ada itself, it may also be partially a result of the discipline the host-target environment brings to the software process.

Most, if not all, projects face schedule pressures, especially late in integration. Mounting pressure to integrate more software faster doesn't change with Ada. However, providing development tools in a different environment from the real-time target motivates engineers to test more thoroughly prior to software release. On the B-2 ATD we have found the software released for integration to be significantly more mature, leading to shorter integration schedules.

However, we have also found that the application engineers require more computer systems support in the integration stage. This is due to the fact that the development tools they have become familiar with may not be available in the target environment. Having spent most of their time in development, they are simply not as familiar with the target tools. As a result, we have found a need to provide more target computer systems support during integration than on traditional programs where development occurs on the real-time target machine.

CONCLUSIONS

Ada is complex and introduces new software measurement factors and trends. With Ada we are seeing more lines of code, but we are also seeing higher productivity rates. Complexity is moving out of the code and into the data. Data declarations are requiring factors of 10 or more times traditional memory requirements. Design time is increasing, but code and test time is decreasing. There are new costs associated with language features such as composite types, unconstrained types, and generics. There are also many new factors to consider in managing computational resources with Ada; stack space, closure software, system services, generics, and structural overhead all must be closely managed.

Today, compilers, tools, and Ada environments are rapidly maturing. The cost of computational hardware is continually decreasing. Clear and simple rules supporting modern software techniques can lead to increased productivity and reduced software costs with Ada. This is being realized today on the B-2 ATD project.

ABOUT THE AUTHORS

Paul E. McMahon is a Staff Scientist at the Binghamton Operations of CAE-Link Corporation. Mr. McMahon has been with Link since 1973 and has held various management and technical positions within the company. He has been involved with Ada development on the B-2 project since 1985. He received his BA in Mathematics from the University of Scranton in 1971 (Magna Cum Laude) and his MA in Mathematics from the State University of New York at Binghamton in 1973. Mr. McMahon has published numerous papers on Ada dating back to 1985. His most recent publications include a paper entitled "On the Fringe of Ada", presented at the 1989 NAECON Conference, a paper entitled "Lessons Learned on the Fringe of Ada", which was nominated for best paper at the 1989 Interservice/

Industry Training Systems Conference, a paper entitled "Ada: Experience It Again for the First Time", presented at the 1990 Tri-Ada Conference, and a paper entitled "Ada in the 90's," presented at the 1990 Interservice/Industry Training Systems Conference.

Dennis W. Meehl is the Department Manager for Special Programs Computer Systems at the Binghamton Operations of CAE-Link Corporation. Mr. Meehl started work with Link in 1977 and has held various technical positions in real-time simulation systems and software development environments. In 1986, he began investigating the use of Ada in real-time simulation and worked technical issues until 1987, when he was assigned as Department Manager. Mr. Meehl received his BS degree in Engineering at Case Western Reserve University in 1971.