N94- 36494

Software Engineering With Application-Specific Languages

1994031987

David J. Campbell
Unisys Corporation
Valley Forge Engineering Center
P.O. Box 517, Paoli, PA 19301-0517

Linda Barker
Deborah Mitchell
Unisys Corporation
Space Systems Division
Mail Stop U04D
600 Gemini Ave, Houston, TX 77058

Robert H. Pollack
Unisys Corporation
Valley Forge Engineering Center
P.O. Box 517, Paoli, PA 19301-0517

510-61
12692
p- 21

Abstract

Application-Specific Languages (ASLs) are small, special-purpose languages that are targeted to solve a specific class of problems. Using ASLs on software development projects can provide considerable cost savings, reduce risk, and enhance quality and reliability. ASLs provide a platform for reuse within a project or across many projects and enable less-experienced programmers to tap into the expertise of application-area experts.

ASLs have been used on several software development projects for the Space Shuttle Program. On these projects, the use of ASLs resulted in considerable cost savings over conventional development techniques. Two of these projects are described.

1 Introduction

An application-specific language is a special purpose language that is oriented towards writing programs for a specific class of problems. An ASL presents the programmer with a higher level of abstraction than a general-purpose programming language, and, as a result, the programmer needs to write much less code to implement a software system.

The ASL code written by a programmer is called a *specification*: it describes the requirements for a software system. A *translator* reads a specification, as shown in Figure 1, and automatically generates software and perhaps other related products, such as accompanying

PRECEDING PAGE BLANK NOT FILMED

PAGE 252 INTENTIONALLY BLANK

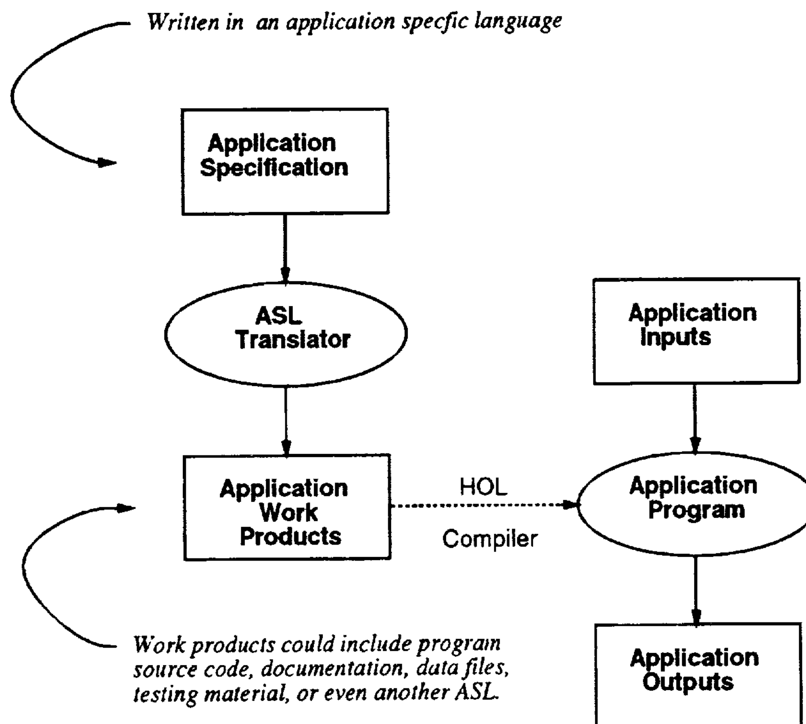


Figure 1: The ASL translator generates software and other related products based on a specification written in a high level language.

design documentation, that satisfy the specification. Usually, the generated software is in a high-order language such as C or Ada.

Today, there are many ASL based commercial off-the-shelf products (sometimes called 4GLs), that address such diverse application areas as data base applications, spread sheets, and graphical user interfaces. If a COTS ASL can be found which meets the needs of a software development project, it will often produce seemingly miraculous results. If such a tool cannot be found, however, an ASL approach is usually abandoned.

This is unfortunate because custom ASLs can be created rather inexpensively, and they can provide considerable advantages to projects that are developing software with certain characteristics. ASLs can increase productivity and reliability by shifting more of the tedious work and mechanical details to the computer, freeing programmers to spend more time addressing the decisions that require creative thinking. ASLs also provide a single point of control for a large amount of software. This enables requirements and design decisions to change with minimal impact on cost and schedule.

2 An Overview of ASL-Based Software Engineering

ASL-based software engineering is a software engineering technique for creating software through automatic code generation. It is not suited to all projects, but there is a large

class of applications where its use can dramatically reduce cost and schedule. For any given project, many different techniques may be applicable, and the best approach may be a combination of techniques. Since software engineers are relied upon to identify the most cost-effective approach, they should be knowledgeable of this technique.

An ASL approach is indicated for a software system when it has recurring similar requirements, especially if there is a large number of them. For example, the requirements might define a series of screens that a system uses to interact with its user. While each screen is different, they are also similar, e.g., each screen contains editable fields for data entry and data validation must be done on each field. If these similar requirements can be implemented with similar code, and an algorithm to transform the requirements into the code can be found, then an ASL can be used.

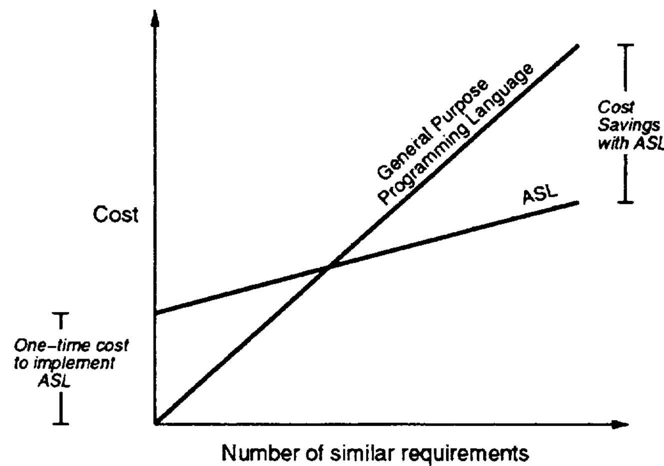


Figure 2: This graph compares the cost of using an ASL versus the cost of using a general-purpose programming language, based on the number of similar requirements. Initially, the ASL is more expensive, because of the one-time only cost to develop the translator. With sufficient repetition in the requirements, however, the cost to develop translator pays for itself.

With ASLs, there is a one-time cost to implement the translator. After the translator is implemented, a specification still must be written to obtain any application code. However, compared with a general-purpose programming language, fewer lines of ASL code are required to implement a corresponding amount of the system's functionality. Moreover, a programmer can typically write more lines of ASL code per day, because, with ASLs, the programmer is transcribing already written requirements into the syntax of the ASL, whereas with a general-purpose programming language, the programmer must write code which describes how to implement the requirements. Consequently, as the amount of repetition in the requirements increases, the cost of implementation with an ASL falls below the cost of implementing software with general-purpose programming language. This relationship is shown in Figure 2

Even if there is not enough repetition to produce a dramatic cost difference, other factors may warrant the use of an ASL. For example, can the ASL be reused on other projects?

Is the algorithm to transform requirements into implementation so complex, that it is best handled by a computer? Are the requirements volatile? Are there risk factors that might cause a possible re-design of the software, e.g., performance issues? If there is significant risk that the requirements or the design may change, then using an ASL will make the software more manageable, because the code is controlled from a single point.

Implementation of an ASL requires a team of engineers with collective expertise both in the application area being addressed and in language implementation. This team must design a generic solution to the problem, which is expressed as a set of reusable code templates and an algorithm to instantiate these templates based on requirements. This design, i.e., the templates and the translation algorithm can be reviewed just like any other form of design.

The language expert designs a language for expressing the information required to instantiate the templates. This language will typically incorporate terminology and notations used by the application experts so that they can easily write or review the specifications. The language enables the *variances* in the similar requirements to be expressed. For example, while each screen consists of a set of unique fields and control buttons, they may also contain a set of standard controls, e.g., controls that return to the previous screen or quit the program. Since the standard controls appear on all screens, they do not need to be specified in the ASL; instead, the translator can automatically supply them.

The language expert also builds the translator. The translator reads an input specification, extracts the information needed by the translation algorithm, and generates the output products by instantiating the templates. The translator may perform semantic checks on the input specification to check that it describes a valid application.

In order to produce other related products from the same specification, such as design documents, test plans, or test data, templates for these products must be designed and logic must be added to the translator to instantiate these templates. The ASL may be enhanced to include additional information that is necessary to instantiate these templates.

Based on our experiences at Valley Forge Engineering Center implementing many different ASLs over the past decades, implementing an ASL, i.e., designing the language and implementing the translator, typically takes from a few weeks to several months, depending on the complexity of the specification language. This cost includes designing the language and implementing the translator only; it does not include the cost of writing any required support software which the generated code may call upon. Since the support software (or software with similar functionality) is usually required whether or not an ASL is used, it is not be figured into the cost of implementing the translator.

There are two reasons why ASLs are relatively inexpensive to implement. First, the underlying technology and theory used to build ASL translators comes from the well-understood software domain of compilers. Many automated tools exist for this domain, e.g., code generators for building lexical analyzers and parsers. Besides automated tools, there are standard architectural designs for translator programs and libraries of commonly used components.

Second, ASLs are much easier to implement than compilers. The generated code is a high-level language, instead of a machine language. The generated code can interface with other software components to implement its functionality. Also, ASLs are much simpler

languages than general-purpose programming languages. Since the design of the language is under the control of the implementer, language constructs which are hard to implement can be avoided. It, therefore, is possible to design and implement a translator for a small, special-purpose language, at lower cost and risk than most other types of software.

The Benefits of ASL-Based Software Engineering

ASL-based software engineering provides a number of benefits, including:

- Increased Productivity
- Increased Reliability
- Better Control
- Lower Maintenance Cost
- Increased Reusability

Increased Productivity

First, there is less code to write, because a software description written in an ASL is much shorter than that same software written in a general-purpose programming language. Second, more lines of ASL code can be written per day than lines of a general-purpose programming language, because, when an ASL is used, the programmer writes a description of *what* the application does, instead of writing a description of *how* it does it.

Moreover, ASLs can be use to capture the expertise of an experienced programmer and transfer it to less experienced programmers. For example, an ASL that allows programmer to build screens for X-windows by just describing their appearance, enables the coding of the screens to be done by a programmer that does not know X-windows. The translator contains the knowledge of an X-windows expert on how to transform the descriptions into the appropriate X-windows code.

Increased Reliability

Generated code is more reliable than hand-written code. Since all of the code is based on the same set of templates, once the templates are correct, all of the code will be correct. The computer can be counted on to perform the repetitive task of instantiating the templates accurately.

Better Control

The form and content of the generated code is controlled from a single-point, the translator; consequently, all of the generated code can easily be changed. A single point of control reduces risk by allowing many design decisions to be deferred. For example, if a generated system interfaces with another complex system, e.g., X-Windows, the design of the generated system can be fine-tuned later, after more experience is gained, by simply

changing the generator. On the other hand, when there is large amount of hand-written code, it is desirable to completely decide on the design before the code is written, because of the cost of retrofitting a change in all of the code.

Also, if the translator generates multiple products, then the products are kept in synchronization automatically. For example, if a translator generates a program and a structure chart which describes the design of the program, then the design documentation and program always parallel each other.

Lower Maintenance Cost

Perhaps the biggest benefit of using an ASL approach is realized in the maintenance phase of the life cycle. There is less code to maintain. Moreover, the capability evolve the system to accommodate new requirements is built into the system; features can be added or modified by making changes to the specification.

Sometimes, over the lifetime of a program, fundamental changes must be made to its overall design, e.g., porting the program to a different hardware platform, operating system, windowing system, database, or even programming language. ASLs facilitate this, because the specification and translator maintain a clean separation between what a program does and how a program does it. In order to retarget a program, only the translator must change. All of the code invested in the specification is still valid because it is independent of the implementation.

Increased Reusability

ASLs extend the scope of reuse beyond what is possible with conventional development techniques and general-purpose programming languages. When an software component is implemented in a general-purpose programming language, the amount of customization that can be done is limited by the parameterization methods available in the language. When a component is generated, however, more possibilities for customization exist, because the generator can add, modify or omit code.

3 Examples of ASLs

ASL technology has been applied on several software development project at NASA/Johnson Space Center. The work was performed under the Space Transportation System Operations Contract (STSOC) on which Unisys Space Systems Division is a subcontractor to Rockwell Space Operations Company.

In this section, we present the work done on two projects to give examples of two ASLs that address completely different kinds of problems. On one project, done for the Payload Operations branch, a command editor for the Tethered Satellite was implemented using ASLs. On the other project, done for the Shuttle Flight Design and Dynamics branch, an ASL was implemented to serve as a general-purpose tool for analyzing data files used during flight design.

3.1 Tethered Satellite Command Editor

Approximately 500 Tethered Satellite System (TSS) payload commands required editing. The ground control specialist uses menus to select commands for editing. Menu buttons either display a submenu or a screen for editing commands. A sample of a menu and a screen is shown in Figure 3.

Satellite RF
Autoreconfiguration
Override Telemetry
AMCS 32-bit Constants - I
AMCS 32-bit Constants - II
AMCS 16-bit Constants
Gyro Constants
Memory Dump
DRBS
Time Tag Command
Hold/Spin Mode
PREVIOUS

Satellite Hold/Spin Mode (RF)
Hold Angle DEG
Spin Rate RPM
MSG FLD:
QUIT PREVIOUS REFRESH STORE

Figure 3: The command editor provides a GUI for selecting and editing commands. A sample menu and and a screen for editing two commands is shown.

Screens have varying requirements for grouping of commands; some screens process one command, while others process 35 or more commands. Each command must be retrieved from a database and stored again after it has been modified. Five different command formats must be processed, each with a unique checksum calculation. Some commands required values to be converted to engineering units, and most commands require values to be displayed both symbolic and in hexadecimal. A single group of commands can be optionally loaded from an external file, rather than the database.

Rather than assigning many programmers to build 140 or so screens—having each programmer code similar sorts of things, but each doing it differently—we invested in the design of special specification language, in which each command and screen layout can be described. A sample of this language is shown in Figure 4. Common capabilities such as the need for certain buttons on each screen, the retrieval of data, and conversion and checksum calculation were built directly into the associated generation process. The specification had

```
Command Format RF_32_bit_degrees is
   Format : RF;
   (7,0)[32] Degrees : Sat_Degrees;
end RF_32_bit_degrees;

Command format RF_16_bit_RPM is
   Format : RF;
   (7,0)[16] RPM : Sat_RPM;
end RF_16_bit_RPM;

Satellite_Hold_Mode_On_RF: P13K1020L RF_32_bit_degrees;
Satellite_Spin_Mode_On:    P13K1022L RF_16_bit_RPM;

Form Hold_Spin_Mode_RF is
   title : "Satellite Hold/Spin Mode (RF)";
   "Hold Angle", Satellite_Hold_Mode_On_RF.Degrees, "DEG";
   "Spin Rate",  Satellite_Spin_Mode_On.RPM,        "RPM";
end Hold_Spin_Mode_RF;

Menu RF_Menu is
   title : "Satellite RF";
   "Autoreconfiguration"        => Auto_Reconfiguration_RF;
   "Override Telemetry"         => RF_Override_Telemetry_Form;
   "AMCS 32-bit Constants - I"  => amcs_constants_32_RF_page1;
   "AMCS 32-bit Constants -II"  => amcs_constants_32_RF_page2;
   "AMCS 16-bit Constants"      => AMCS_Constants_16_RF;
   "Gyro Constants"             => RF_Gyro_Constants;
   "Memory Dump"                => RF_Memory_Dump_Form;
   "DRBS"                       => DRB_Menu;
   "Time Tag Command"           => RF_Time_Tag_Command;
   "Hold/Spin Mode"             => Hold_Spin_Mode_RF;
end RF_Menu;
```

Figure 4: This is the specification for the screens shown in Figure 3. Besides displaying the menu, the code generated for this specification fetches two commands from the database (P13K1020L and P13K1022L), extracts the Degrees and RPM field from each command respectively, and displays their values on the screen for editing by the user. If the user presses the STORE **button, the commands in the database will be updated with the last value the user entered.**

all the implementation details for each command; the generator integrated all special process requirements with common capabilities.

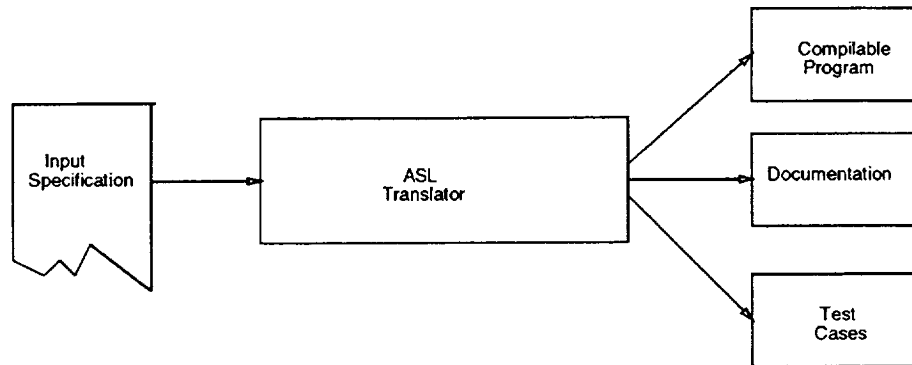


Figure 5: The TSS ASL translator generates a command editor, a user's manual for the command editor, and test program from a single specification.

The translator generates three significant products for this project as shown in Figure 5. The main product consists of several thousand lines of high quality, maintainable C code. In addition, a 200 page user's manual and test program are produced. The user's manual describes how to use the editor and the screens that editor is capable of displaying. The test program validates that each TSS command exists in the database and is defined as specified. Additionally, high and low value entry is simulated for each editable data value.

The ASL approach accommodated introducing new requirements in the unit testing phase with *no* impact to schedule. During this phase, about 40 new screens were requested by the customer to handle science commands. To accommodate this request, no actual C coding was required, only descriptions of the new screens had to be added to the specification. Then a new editor, user's manual, and test program were generated automatically.

The productivity for the command editor application was not tracked in detail. The translator consists of 7K lines of code, 4K lines were hand written for this project and 3K lines were reused or generated; the level of effort to produce the translator was 3 person months, including the design of the templates for the generated code. The TSS Command Editor is 12K lines of code, 7K lines are generated by the translator, and 5k lines are hand written. The hand-written code is used by the generated code and is not changed to accommodate new specifications. The generated test program for the TSS editor is 6K lines of code, and the generated user's manual is 12K lines of `troff` and `pic` commands. Additional productivity gains have been achieved, because the command editor generator has been used for other payloads, e.g., SSBUV and Wake Shield.

3.2 Strip Manipulate and Merge Tool

The Strip Manipulate and Merge (STMM) tool was created by the Common Software task as part of its overall goal to reduce maintenance cost by creating a common set of tools for use by flight designers, since many of the existing tools duplicate functionality.

STMM accepts a specification that describes operations to be performed on standard flight design data files. There are several different types of data files used for flight design. Each data file type has its own physical format; however, all of the data files are logically similar—Each file consists of a collection of records; each record is the same type, consisting of a set of named fields; and each file has a data dictionary which describes the structure of the records, i.e., the names of the fields in the record, the number of bytes allocated to the field, the type of data in the field (e.g. ASCII or binary), and the engineering units represented by the data.

STMM replaces an existing set of forty or so tools that perform similar, but specific, operations on flight design data files, such as converting from one file format to another; creating a file from selected records of another file; or omitting, reordering, renaming, or adding fields to the records of a file. In addition, some tools perform operations on multiple data files, such as concatenating, merging or joining them. Each tool did some specific combination of the above operations on a specific set of data files. With STMM, these forty custom tools are replaced by forty small specifications and the STMM tool itself.

Originally, STMM was to be implemented using a COTS product that manipulates flat files. After analysis, it was found that the COTS product could not adequately replace the existing set of tools. The COTS product did not support the number fields that records in some of the data files had. It did not support operations such as joining or merging files based on a tolerance for the key fields. And finally, it could not convert from one file type to another. The additional support code required to use the COTS solution made the COTS implementation unfeasible, so a custom ASL was implemented.

```
merge "run1.cff"(cff) and "run2.cff"(cff) giving "out.merge"(fcff);

   record selection for "run1.cff"(cff) is
      range : Number in 1.0e6 .. 2.0e6;
   end;

   key is Pressure;
end;
run
```

Figure 6: This sample language specification merges two data files, *run1.cff* and *run2.cff*, producing a the result file *out.merge*. The files are merged on the key field *Pressure*. The only records selected from *run1.cff* are records where the value of the field *Number* is in the range one million to two million.

One of goals of STMM, was to make the language easy to use by flight design engineers, who are not necessarily computer programmers, so that new file manipulation programs could easily be created by them. The language designed for STMM allows the user to express operations on data files using an is English-like syntax, which is easy to read and write. A sample of the STMM language is shown in Figure 6. Also, extensive error checking was built into the translator to make it easier for the user to debug specifications.

The architecture of STMM is slightly different from the other ASLs that we have been discussing. Instead of translating the user specification into an HOL program, which must then be compiled, the translator generates an internal, intermediate language that represents the user's program. A component called an *interpreter* executes this intermediate language.

The interpreter for STMM makes use of a library that defines a class of objects called *filters*. There are several types of filters; each type of filter can be connected to one or more input streams of data and produces an output stream of data. In addition, each type of filter is capable of doing some kind of transformation on its input streams to produce its output stream. For example, there are filters which select records based on parameterizable criteria, strip fields from records, or concatenate, merge, or join multiple streams of data. The STMM translator translates the specification into the appropriate chain of filters. Once the filter chain has been constructed, the translator turns to control over to the filters to executed the operations.

Summary

The way in which software is produced has changed several times since the invention of electronic computers. All of these changes consist of transferring an increasing amount of work from human beings to the machine itself. Application-specific languages are a step in this trend. They enable software engineers to leverage the tools and techniques from a well-understood domain—compilers—against problems of developing new software.

Application-specific languages provide many important benefits to a project during implementation and maintenance phases. They increase productivity, increase reliability, provide control of a large amount of software and related products from a single point, and enhance the ability of a system to adapt changing requirements.

Because of the success of ASLs on these and other STSOC software development projects, ASL training was given to a team of about twenty STSOC software engineers. These engineers will assess new projects and existing maintenance efforts to find areas where ASLs can reduce cost.

Biographical Sketch

David J. Campbell has over seventeen years experience in compiler, operating system, and support tools development. In addition to his work at Unisys, he is a part-time instructor for the Mathematical Sciences Department/Computer Science Division, Villanova University. For the past seven years, the main focus of his work has been on automatic generation of software, chiefly through the use of compiler development technology. His work includes the implementation of many software generators and the creation of tools to build software generators. He has also been involved with many tasks on the STARS program, including porting a Sun Unix version of the Common APSE Interface Set, revision A, to the MACH operating system, and serving as chief programmer on the rapid software modeling task.

Mr. Campbell is currently a Staff Engineer in the Research and Development Division of Valley Forge Laboratories. He holds an B.S. degree in computer science from Wichita State University.

Linda Barker has over seventeen years experience in the computer industry. She is currently Supervisor of Software Engineering for the Mission Control Center, Data Systems Software Section, which is responsible for maintaining several applications used in the ground support operations for space shuttle flights. She is also a charter member of the Houston–SSO Software Engineering Process Group (SEPG).

Deborah A. Mitchell has over fourteen years experience in programming and software support on a variety of hardware systems. For the past six years, she has worked on the Space Transportation System Operation Contract in the Flight Design and Dynamics Department. Her work includes project management of Common Software applications, the development of two ASL applications, the General Purpose Input Processor and the Strip Manipulate and Merge, Generic Report Writer.

Deborah Mitchell is currently a project manager in the Reconfiguration Department of the Unisys, Houston, division. She holds a Bachelor of Science in Electrical Engineering (BSEE) from Prairie View A&M University.

Robert H. Pollack has over twenty years experience in programming and software support, on a variety of hardware and operating systems. For the past nine years, the main focus of his work has been on the automated creation of application software, chiefly through the use of compiler development technology. His work includes the creation of a system to generate Ada message validation code from abstract specifications of the message formats, a system which is used for software development in several Unisys projects. He is also the creator of a major subsystem of an interpreter for the Ada language developed under the STARS program.

Mr. Pollack is currently a Staff Engineer in the Research and Development Division of Valley Forge Laboratories, where he is assigned to the Re-Engineering IR&D project. He holds an M.S.E. (Computer and Information Science) from the University of Pennsylvania.

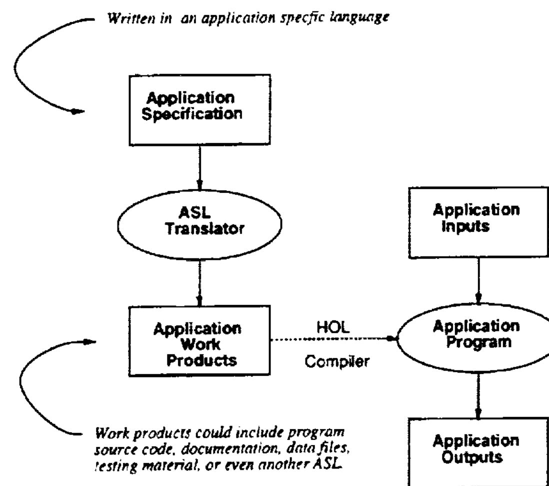
Software Engineering With Application-Specific Languages

David J. Campbell
Unisys Corporation
PO Box 517
Paoli, PA 19301
Campbell@VFL.Paramax.COM

Application-Specific Languages (ASLs)

- Special-purpose languages targeted to solve a specific class of problems
- Present programmers with a higher level of abstraction than general-purpose languages, allowing a programmer to write less code
- Used to automatically generate required software or other related work products
- Inexpensive to produce (typically, from a few weeks to a few months)

Automatic Software Generation With ASLs



Automatic Software Generation With ASLs (Cont.)

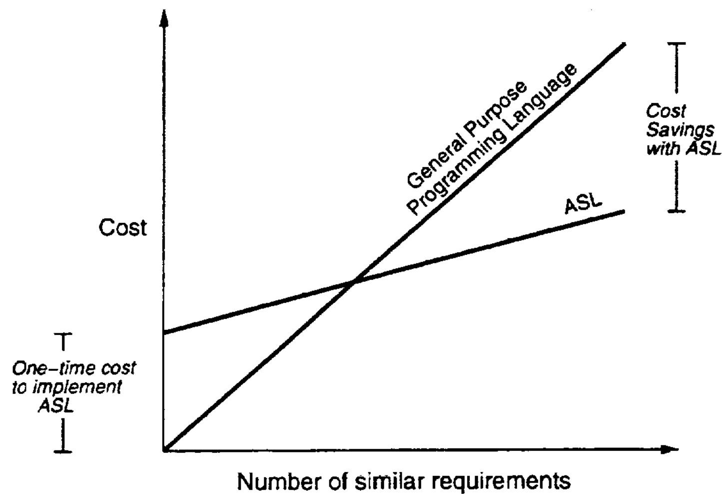
- Specification and translator maintain a clean separation between *what* software does, and *how* it does it
- Generic solution to problem is formulated as a set of reusable code templates
- Translator executes an algorithm that instantiates templates from a specification which describes the requirements for the software

Evaluation process

- Determine if a software component is a candidate for ASL implementation
  - Repetitive coding tasks
  - Complex or error-prone coding tasks
  - Requirements subject to change
  - Recurring problem (i.e., ASL is reusable on other projects)
- Perform tradeoff analysis, ASL vs other approaches

Application-Specific Languages(29 November 1993) Foil 5

Cost Tradeoff



Application-Specific Languages(29 November 1993) Foil 6

ASL Development Activities

- Language Design
  - Design a language for specifying requirements in terms familiar to the application expert
- Translator Development
  - Develop a translator that checks the input specification for errors and generates code that satisfies the requirements
- Product Generation
  - Write specification for the required work products and generate the actual components

Application-Specific Languages(29 November 1993) Foil 7

Benefits

- Increased Productivity
  - Less code to develop and maintain
- Increased Reliability
  - All code based on same templates
  - Computer accurately instantiates templates
- Increase Manageability
  - Translator provides a single-place for controlling a large amount of code and related work products
  - Design decisions are encapsulate in the translator
  - Less impact to evolve design or tune implementation

Application-Specific Languages(29 November 1993) Foil 8

Benefits (Cont.)

- Related work products are always consistent
- Less impact to handle anticipated requirements changes

- Increased Reusability
  - Generated components are more tailorable than components implemented in programming languages

Application-Specific Languages(29 November 1993) Foil 9

Examples of ASLs

- Editor Generator (Egen)
- Strip Manipulate and Merge (STMM)

Application-Specific Languages(29 November 1993) Foil 10

Egen (Editor Generator)

- Egen is an ASL that generates a payload command editor from a high-level specification
- Initially developed for the TSS payload, subsequently used on the SSBUV and Wake Shield payloads

Application-Specific Languages(29 November 1993) Foil 11

Command Editor

- Fetches and stores commands from a data base
- Enables the user to display and change the variable fields of commands
- Converts values to engineering units
- Handles different command formats and computes checksum required by formats
- Provides a GUI for editing commands

Application-Specific Languages(29 November 1993) Foil 12

Example of User Interface

Satellite RF

Autoreconfiguration
Override Telemetry
AMCS 32-bit Constants - I
AMCS 32-bit Constants - II
AMCS 16-bit Constants
Gyro Constants
Memory Dump
DRBS
Time Tag Command
Hold/Spin Mode
PREVIOUS

Satellite Hold/Spin Mode (RF)

Hold Angle DEG
Spin Rate RPM
MSG FLD:
QUIT PREVIOUS REFRESH STORE

Application-Specific Languages(29 November 1993) Foil 13

The Egen Specification

```
Command Format RF_32_bit_degrees is
   Format : RF;
   (7,0)[32] Degrees : Sat_Degrees;
end RF_32_bit_degrees;

Command format RF_16_bit_RPM is
   Format : RF;
   (7,0)[16] RPM : Sat_RPM;
end RF_16_bit_RPM;

Sat_Hold_Mode_On_RF: P13K1020L RF_32_bit_degrees;
Sat_Spin_Mode_On:    P13K1022L RF_16_bit_RPM;

Form Hold_Spin_Mode_RF is
   title : "Satellite Hold/Spin Mode (RF)";
   "Hold Angle", Sat_Hold_Mode_On_RF.Degrees, "DEG";
   "Spin Rate",  Sat_Spin_Mode_On.RPM,        "RPM";
end Hold_Spin_Mode_RF;
```

Application-Specific Languages(29 November 1993) Foil 14

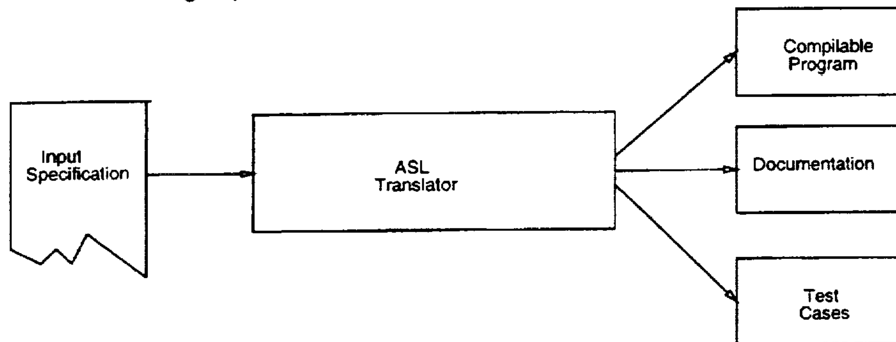
The Egen Specification (Cont.)

```
Menu RF_Menu is
   title : "Satellite RF";
   "Autoreconfiguration"          => Auto_Recon_RF;
   "Override Telemetry"           => RF_Override_Telm;
   "AMCS 32-bit Constants - I"    => amcs_32_RF_page1;
   "AMCS 32-bit Constants -II"    => amcs_32_RF_page2;
   "AMCS 16-bit Constants"        => AMCS_16_RF;
   "Gyro Constants"               => RF_Gyro_Constants;
   "Memory Dump"                  => RF_Memory_Dump_Form;
   "DRBS"                         => DRB_Menu;
   "Time Tag Command"             => RF_Time_Tag_Command;
   "Hold/Spin Mode"               => Hold_Spin_Mode_RF;
end RF_Menu;
```

Application-Specific Languages(29 November 1993) Foil 15

Egen Translator

- Egen produces multiple work products



Application-Specific Languages(29 November 1993) Foil 16

STMM (Strip Merge and Manipulate)

- STMM programs describe operations to be performed on flight design data files
  - Create files from selected records of other files
  - Omit, rename, reorder, or add additional fields to records
  - join, concatenate, or merge files
  - convert files from one format to another
- It replaces forty programs that perform specific operations on given files

Example of a STMM Specification

```
merge "run1.cff"(cff) and "run2.cff"(cff)
   giving "out.merge"(fcff);

   record selection for "run1.cff"(cff) is
      range : Temperature in 1.0e4 .. 2.0e4;
   end;

   key is Pressure;
end;
run
```