A Specification and Code Generation Tool for Message Translation and Validation

Charles Plinta & Richard D'Ippolito
Accel Software Engineering
Roger Van Scoy
TTFN Software, Inc.

This paper describes the Message Translation and Validation (MTV) problem common to most computer-based applications, and a model-based solution to the problem that is supported by a tool. An early version of the MTV model was used by Granite Sentry Phase II in the late eighties and later by the Air Force PRISM program at ESC in the nineties. Accel obtained an SBIR Phase I and II contract to extend the MTV model, convert it from Ada83 to Ada95, and to provide tool support to ease the specification of message formats and automate the MTV code generation. This paper will describe the solution and benefits, its usage, some lessons learned, and future directions.

In this paper we address a data conversion problem common to most applications. We begin by describing the problem, presenting a model-based solution with tool support, and characterizing the tool's users and discussing their expected benefits. We then provide some insight into how the tool is used to demonstrate the power of model-based technologies. Finally, we discuss some of our lessons learned in developing the model-based solution.

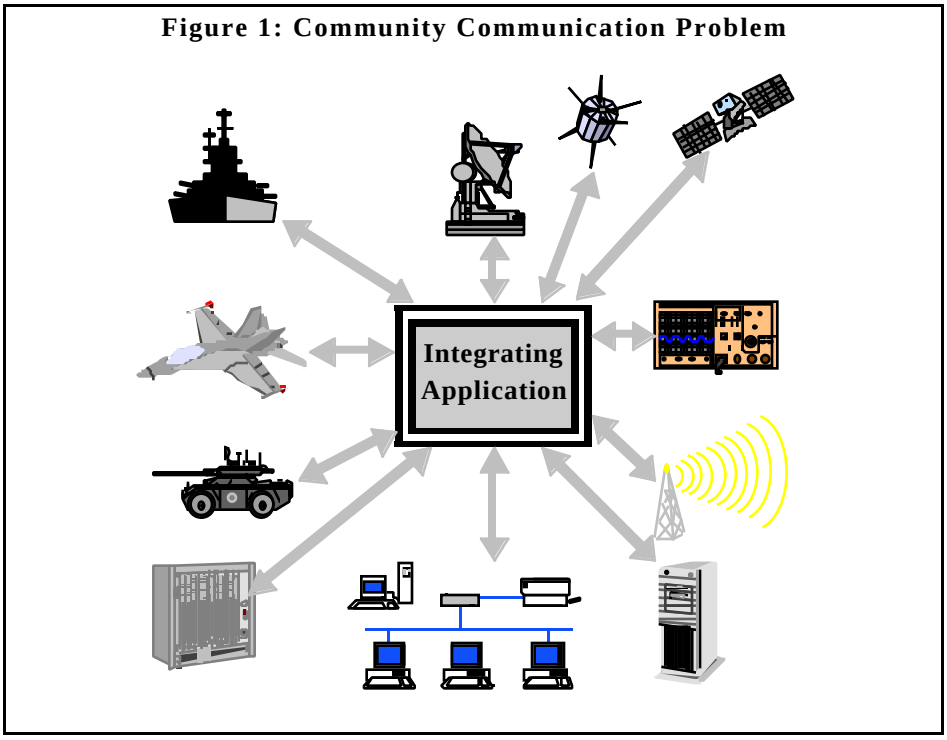
Overview

What is the problem?

Most useful computer-based applications must communicate with a diverse community of systems and devices. The systems are typically hosted on a variety of hardware and operating system platforms and were originally developed in one of many programming languages. The devices typically communicate in their own device-specific language.

Traditional communication within the community is via messages. Messages are streams of ASCII text or bits having structural components such as fields. Integrating this message-based community of disparate systems and devices into a new or existing application is a very time-consuming, tedious, and error-prone process. Application developers often get consumed in this activity and can be distracted from focusing on the real problem of defining and implementing new application capabilities to address mission-critical needs.

Figure 1: Community Communication Problem



Systems need to be linked in real-time or near real-time, and the amount of information shared between dissimilar systems is growing dramatically. Military systems exchange things such as troop locations, equipment information, radar track data, enemy intelligence data, dates, latitude and longitude information. Commercial information systems exchange things such as customer names, account information, dates, amounts of money, invoices and advance-shipment notices. These items are represented as character strings, integers, floating point numbers, and data structures composed of several simpler items. Each system might have its own representation for character strings, integers, and so on. The challenge is to find a general solution that allows all connected systems to exchange information without loss of meaning.

A model-based solution.

Accel has developed a product called MTV Builder™. It is an easy to use, PC-based tool that simplifies message specification and automates software generation to facilitate communication between systems and devices.

Users specify message formats using the GUI and the tool automatically generates Ada95 code that provides message translation and validation (MTV) capabilities that are easily integrated into an application. The generated MTV software can be ported to any hardware and operating system platform that hosts an Ada95 compiler.

The tool contains an internal model, or pattern, of what message translation and validation means. This model is implemented in scaleable code segments which have been tested and verified. Like a pattern for a shirt, the model can be scaled within a range of workable sizes. Where size, style, material, and color are the important specifications of a shirt, message elements (delimiters, fields, and groups) and element order are the important specifications of a message. The MTV Builder accepts these message specifications from the user and applies them to the internal models to produce the working code to translate and validate each message.

Who are the users?

The tool can be used by domain engineers familiar with the message set protocols of the disparate systems and devices in the community that are targeted for integration, and by software engineers responsible for evolving existing applications and developing new ones.

Domain engineers use a visual interface to specify the message protocols necessary for communication among the disparate systems and devices in the community. Software engineers specify application specific data-types necessary to process incoming and outgoing message protocols within the community. MTV Builder automatically generates interface control documents that describe the message formats, Ada95 software to translate and validate messages, and test drivers and data to verify the software's translation and validation capabilities. Software engineers then integrate the generated translation and validation code into the overall application.

What are the benefits?

MTV software is a vital component when trying to integrate a diverse community of systems and devices. Traditionally, developers have taken a message-by-message, hand-coding approach to creating MTV software. The resulting development and integration efforts have been witness to the time consuming, error prone, and tedious nature of this approach, as well as the drain on vital resources better spent addressing mission critical needs.

MTV Builder antiquates the traditional approach used to develop MTV software by providing System Integration professionals with a truly engineered solution for constructing MTV software *quickly*, *inexpensively*, and *reliably*.

How does MTV Builder accomplish this? User's simply specify message formats using the GUI and it *quickly* generates the message translation and validation software and test drivers with sample messages. Generating this software eliminates development time, dramatically reduces test time, and allows maintenance to be performed at the message specification level using the GUI. These time reductions result in *reduced costs.* Finally, the MTV software is generated according to user specifications using proven building blocks (models). These blocks are built on Accel's pre-engineered foundation software that remains constant, thereby increasing *reliability.*

How did the solution evolve?

The original MTV software model (foundation software and code templates) was developed in Ada83 at the Software Engineering Institute in 1989 by the Accel founders. It was developed for Granite Sentry Phase II , an Air Force command and control program. We developed this model to demonstrate our model-based software engineering techniques. The model was used by Granite Sentry and several other command and control programs.

In the early 1990's, the model was adopted by the Air Force PRISM program at ESC, Hanscom AFB. The PRISM Program's objective was to change the way the Air Force procured command centers in the future by using a generic command center concept. According to the PRISM program, 80% of the functionality of most command centers is generic in nature, and only about 20% is unique from application to application. PRISM's approach was to gather Commercial-Off-The-Shelf (COTS), Government-Off-The-Shelf (GOTS), and public domain software modules to support the generic command center portion of the application. New models would be produced only when necessary. PRISM had used the original MTV software model as the fundamental model for message translation and validation and were very interested in having a commercially supported product.

Founded in 1993, Accel was awarded several research contracts in 1994 and 1995 that seeded our commercialization efforts:

- *Small Business Innovative Research (SBIR) Phase I Contract*
  Technology research, development, and proof of concept.
- *Ben Franklin Challenge Grant*
  Prototype tool development and proof of concept.
- *Oregon Graduate Institute (OGI)*
  Training course development and delivery to support experiment.

In 1996, Accel was awarded an *SBIR Phase II Contract* to evolve a market-sensitive tool called MTV Builder. This effort is currently building on the prior research and development results and is aimed at strengthening and

extending the foundation Ada95 software, producing a commercial product, and developing usage and training material.

Where are we going?

In November of 1997, Accel released Version 1.0 of MTV Builder. Our SBIR Phase II contract runs through June 1998. In the months that remain we intend to integrate interface technologies, such as SQL and CORBA to accommodate any implementation technology (e.g., Ada95, C, and Java). Integrating these technologies will enable us to generate *custom* standalone message processor applications.

Using MTV Builder

In this section we will provide some insight into using the tool to specify message formats and how easy it is to integrate the generated code.

The development process for creating MTV code using MTV Builder is depicted in Figure 2 and contains the following steps:

1. Install tool and compile foundation code into your development library.
2. Use the GUI to specify messages
3. Generate message format reports to verify proper message specification, Generate MTV code and test drivers.
4. Move MTV code to target platform. Compile, link, and run tests.
5. Integrate MTV code into application. Compile, link, and run application.

How do I specify messages?

Step 2 in the development process described above involves using the GUI to specify message formats. This process can be summarized by:

1. Define **delimiters** in a message.
2. Define **fields** in a message by specifying low-level characteristics.
3. Define **groups** by organizing and characterizing fields and sub-groups.
4. Define **messages** by assembling and characterizing groups and fields.

Figure 2: MTV Builder Development Process

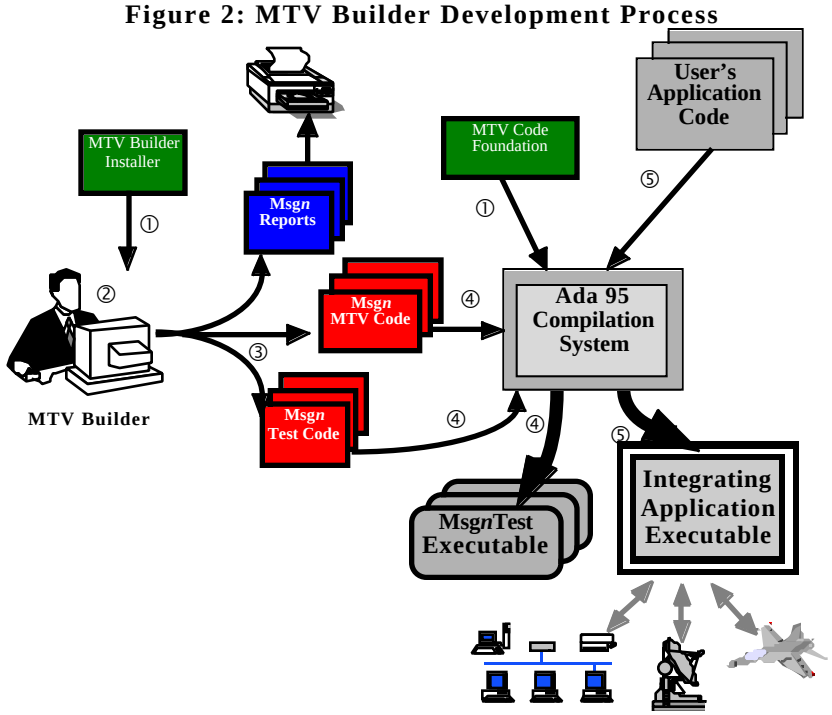


Figure 3: MTV Builder - Message Set Editor Overview Screen

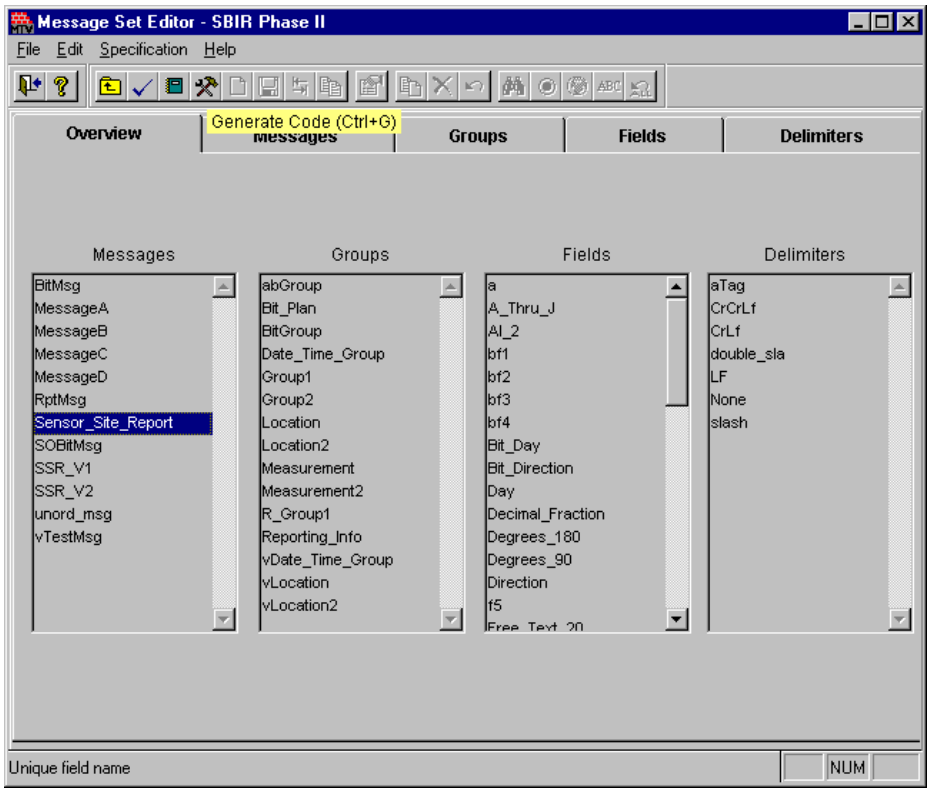


Figure 3 shows a snapshot of the overview screen in the tool. Each tabbed page is used to specify different parts of the message.

Once messages are specified by the user, this specification information becomes the basis for all report, code, and test generation.

This GUI captures the essence of the MTV model. It provides the means for developers to describe message formats in a manner that the tool can then generate code based on the underlying MTV model. The users are specifying information about message formats in a manner that is natural to them and the underlying MTV model requires information structured in a specific fashion. The tool is a mediator between the user and the model.

Show me the MTV code!

The following Ada95 package specifications shows the API generated by the MTV Builder which provides the message translation and validation capabilities for a specific message. The Ada data type declarations for the message are contained in the package <MsgName>_INR shown in Figure 4.

The translation and validation interface routines are defined in the child package <MsgName>_INR.MTV shown in Figure 5. The Image and Value procedures defined in this package mimic the Ada 'IMAGE and 'VALUE attributes defined for scalars types, but they work on more complex data structures to accommodate the structure and format of a message.

The tool also generates the code that does all the work required to provide the translation and validation capabilities specified by the interface. This generated code is built upon a foundation of Ada utilities that provide the core parser, translation, and validation capabilities.

Figure 6 depicts this building block approach to developing systems.

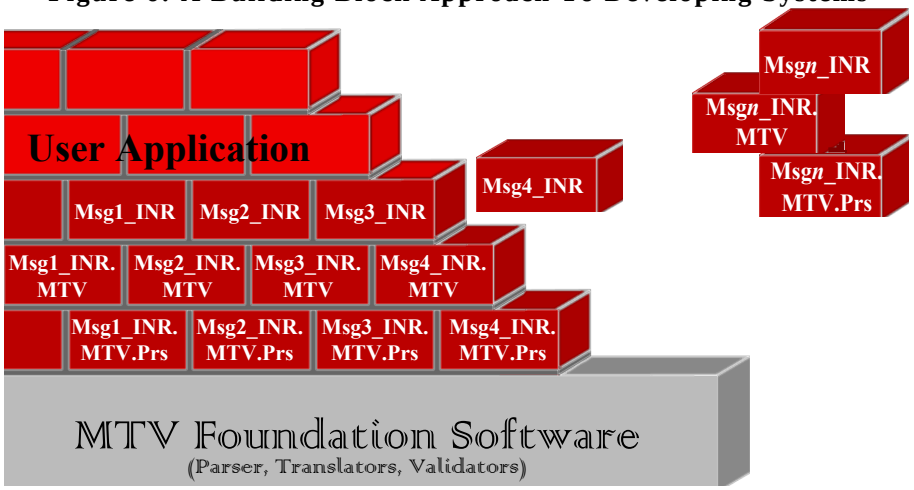
Figure 4: INR Ada Package Specification

```
package <MsgName>_INR is
   -- ALL discrete types necessary for <MsgName>
   subtype NInt_Type is Integer range 1..99;
   type NInt_Type_Pointer is access all NInt_Type;

   type NEnu_Type is (North, South, East, West);
   type NEnu_Type_Pointer is access all NEnu_Type;

   -- ALL composite types necessary for <MsgName>
   type Group_Type is record
           ...
   end record;
   type Group_Type_Pointer is access all Group_Type;

   -- Record types definitions for <MsgName>
   type <MsgName>_Message_Type is record
       Group  : Group_Type_Pointer;
       Nint   : NInt_Type_Pointer;
   end record;
end <MsgName>_INR;
```

Figure 5: MTV Ada Package Specification

```
with Ada.Strings.UnBounded; use Ada.Strings.UnBounded;
with TV;
package <MsgName>_INR.MTV is
   procedure Value(Image_In : in  UnBounded_String;
                   Value_Out: out <MsgName>_Message_Type);
   procedure Image(Value_In : in  <MsgName>_Message_Type;
                   Image_Out: out UnBounded_String);
   procedure Check(Image_In : in  UnBounded_String;
                   Error_Info:out UnBounded_String;
                   Validity : out TV.Validity_Rep_Type);
end <MsgName>_INR.MTV;
```

Figure 6: A Building Block Approach To Developing Systems



How do I test the MTV code?

Because the MTV code generated by the tool has a standard API, and the message format specification information is entered by the user and captured by the tool, the tool can also generate test data and test drivers to exercise the generated code. This provides developers with an additional level of confidence in the correctness of the generated code as well as regression test capabilities.

How do I integrate MTV code?

Integrating the MTV capabilities simply involves “with”ing the generated packages <MsgName>_INR and <MsgName>_INR.MTV for each message that a system must handle and calling the appropriate message-based translation and validation routines (Image or Value) when necessary.

How do I maintain MTV code?

You don’t. It is not necessary to maintain the MTV code generated by the tool. What you maintain is the message specifications captured by the tool. If a message format changes or a new message is added, the resulting changes or additions are made at the specification level via the tool. From here the code would be re-generated, re-tested, and re-integrated.

Lessons Learned

Effective software reuse has been the subject of considerable effort, especially since the hardware costs of a system have been reduced to an insignificant part of the total system costs for many applications. For example, the PRISM program and many commercial organizations have tried to establish libraries of reusable code. In our experience, the ease of doing this is dependent on two things:

1. the need to build a number of similar systems having similar performance and functional characteristics
2. the ability to identify a common set of building blocks that can be used to compose those systems.

Roughly speaking, the first item means that a library makes sense only if you expect to do the same system more than once. The second item means that you need to be able to develop a consistent style of building (architecture) the systems from a set of modules whose final functional characteristics can be tailored within limits to the exact application. This is consistent with general architecture and engineering practice, where reuse occurs at the architecture and design levels, and not at the implementation level. What is saved is the set of building blocks (adjustable patterns) and the methods for tailoring and constructing new instances of the components, and not previously built components.

PRISM was a good candidate for creating a model set because of the 80% common functionality content from system to system. Identifying and drawing a box on paper labeled “message translation and validation” as one of several blocks in a command system block diagram is not enough to allow you to say that you now have a system design (or architecture!). You need to be able to create that block upon request and know that it will communicate with the adjacent blocks and will work with all of the messages expected. From Granite Sentry, we learned that a poor way to construct this block was to parcel out the messages in groups to several programmers and have them code a unique translator for each message. The result was a collection of code segments each written and documented in the unique style of the programmer. Integration and maintenance becomes a real nightmare—who does it and who adds the next message when the need arises, as it surely will.

The solution was to identify the common characteristics of all current and possible messages and to create a model-based way of creating code that would handle them. This was not as hard as it would seem. Only two types of messages were identified: streams of characters and streams of bits, and each of these can be composed of the same common message elements. So, given an general translator that can be told ahead of time which messages to expect and how to translate them into their compositional elements, we can define the requirements of the block in a way that it will work in any target system.

But, we aren't quite done yet. There are three more lessons to go.

One is that once we have created a translator composed of proven, architecturally consistent internal components for a particular system and its expected messages, we can't accommodate changes to either individual messages or the addition of new messages. This is the argument for a built-in code generator. This does two things: It saves the manual labor of re-coding and re-testing the module, and it insures that each implementation is created to the same standard of quality from trusted components.

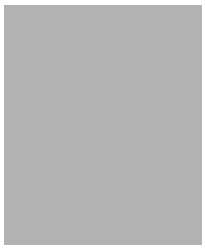
The next lesson concerns testing and documentation. We can integrate a test function that can be used to verify that the specification for each message and element is consistent and correct. We can also bundle a documentor with the tool that creates up-to-date tables of all specified messages and message elements on demand. We learned early on that documentation takes a back seat to programming, and that even when done correctly, documentation is seldom modified to keep pace with system changes.

Finally, it makes sense to save the specifications of the messages in a form that can be used as electronic input to the tool for the next system. This allows a developer to archive a completed translator module as a disk file and to use all or portions of that disk file as input to the next project without having to re-enter data from the keyboard. Therefore, we have included message set import and export features in the tool.

Summary

Software reuse will not just happen—we need to make it easy to reuse software and models. Intuitive specification tools, such as the MTV Builder, with built-in code generators will facilitate reuse. These tools will do this by removing the focus from the miserable and thankless job of cataloging and maintaining code modules (artifacts) and placing the focus more appropriately on the maintenance of module specifications (requirements), where changes are more easily implemented and controlled. Additionally, having a built-in code generator assures that with each change of the specification during the product lifetime, the quality of the implemented code is maintained. Finally, the built-in code generator saves labor and programming costs.

About the Authors



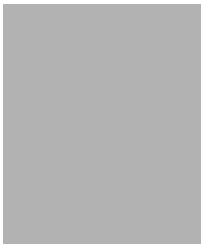
Mr. Charles Plinta is a founder and managing partner of Accel Software Engineering. At Accel he manages the research, development, and marketing of MTV Builder. He also designs and develops shop floor integration systems.

Mr. Plinta previously worked on the Software Architectures Engineering project at the Software Engineering Institute (SEI) and at the Defense Electronics Center of Westinghouse Electric Corp. He is a member of the IEEE and holds a bachelor's degree in computer science and mathematics from the University of Pittsburgh.

Accel Software Engineering
9 Mellon Road
Export, PA 15632-8904
Voice: 724-733-8800
Fax: 724-733-8820
Email: cplinta@accelse.com
Web: www.accelse.com

References

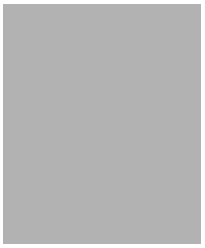
1. Plinta C., *SBIR Phase I Final Report, Development and Demonstration of an OCU-Style Message Translator and Validator Model*, for USAF ESC/ENS, April 1995.
2. Oregon Graduate Institute, *Software Design for Reliability and Reuse: Phase I Final Scientific and Technical Report*", for USAF ESC/AVK, February 1995
3. Plinta C., C. Stanley, "An Executive Model Set Built Upon a Software Architectural Foundation", *Proceedings of the 1994 Technical Innovations Symposium*, Sept. 1994, pp. 31-56.
4. D'Ippolito R., et. al., "Putting the Engineering into Software Engineering", *6th Conference on Software Engineering Education*, October 1992.
5. Plinta C., "Considerations for Defining Software Architectures", *Proceedings of the DARPA Workshop on Domain-Specific Software Architectures*, July 1990.
6. Plinta C., K. Lee, "A Model Solution for the C^3I Domain", *Proceedings of Tri-Ada '89*, Nov 1989, pp. 56-67.
7. D'Ippolito R., C. Plinta, "Software Development Using Models", *Proceedings, Fifth International Workshop on Software Specification and Design*, May 1989, p140-142.
8. Plinta C., K. Lee, M. Rissman, *A Model Solution for C^3I Message Translation and Validation*, Technical Report CMU/SEI-TR-89-12, 1989
9. Polya G., *How to Solve It*, Princeton University Press, Princeton NJ 1984.
10. Jones J. C., *Design Methods – seeds of human futures*, John Wiley & Sons Ltd., New York NY, 1981.
11. Alexander C, *Notes on the Synthesis of Form*, Harvard University Press, Cambridge MA, 1964.



Dr. Richard D'Ippolito is a founder and managing partner of Accel Software Engineering. At Accel he manages, designs, and develops shop floor integration systems.

Dr. D'Ippolito previously lead the Software Architectures Engineering project at the SEI and has worked for several companies focused on capturing and integrating factory process control information. He is a member of the IEEE and a registered professional engineer in the Commonwealth of Pennsylvania. He holds a bachelor's degree in electrical engineering from Carnegie Institute of Technology, and a master's and doctorate in engineering design from Carnegie Mellon.

Accel Software Engineering
449 Maple Avenue
Pittsburgh, PA 15218-1501
Voice: 412-731-3293
Fax: 412-731-2036
Email: rsd@accelse.com



Mr. Roger Van Scoy is president of TTFN Software, Inc. At TTFN he consults on software risk management and performs interactive multimedia development.

Mr. Van Scoy previously worked on Professional Education and Risk projects at the SEI and at the Defense Electronics Center of Westinghouse Electric Corp. Mr. Van Scoy holds a bachelor's degree in computer science and physics from the Hiram College, a masters degree in Physics from John Hopkins University, and a masters in Computer Science from the University of Pittsburgh.

TTFN Software, Inc.
503 Woodland Road
Pittsburgh, PA 15237
Voice & Fax: 412-367-3945
Email: rlvs@TTFNsw.com
Web: www.TTFNsw.com