A Model Solution for the C³I Domain

Charles Plinta and Kenneth Lee

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

ABSTRACT

This paper[1] briefly describes a specific portion of recent work performed by the Domain Specific Software Architecture (DSSA) Project at the Software Engineering Institute (SEI) -- the development and use of a model solution for message translation and validation in the C³I domain. Based on this experience and our involvement with programs in the C³I domain, future considerations are described. These considerations involve identifying potential models within a domain and making recommendations for developing and documenting model solutions which will enable the models to be reused.

BACKGROUND

The work was performed by Kenneth Lee, Charles Plinta, and Michael Rissman, in conjunction with the Granite Sentry (GS) Program. GS is a phased hardware and software replacement of some of the systems in the Cheyenne Mountain complex of North American Aerospace Defense Command (NORAD). The DSSA Project supports the GS program office by providing advice on technical issues. The DSSA Project members participate in design discussions and working group meetings with the lead designers. As part of our involvement, the DSSA Project developed a model solution to perform message translation and validation (MTV). The MTV model is currently being used by GS Phase II in its design specification and the MTV model solution will be used to implement that portion of the design. The MTV model solution is also being used by other programs developing systems in the C³I domain: Army WWMCCS Information System (AWIS) and Mobile Command and Control System/Mission Support Segment (MCCS/MSS).

AN OVERVIEW OF C³I SYSTEMS

Figure 1 shows a high-level block diagram of a typical C³I system. The *Gateway* is an interface between the C³I system and all external systems. The *Gateway* sends messages to and receives messages from the external systems. The messages enter and leave the C³I system as *external representations (EXR)* of the information, whose formats, EXR Descriptions, are defined by the external systems.

The *Mission Processor* maintains a view of the world, in a mission database, based on the *EXR* provided by the external systems. This world view is kept in *internal representations (INR)* which allow processing of the information based upon the C³I system's mission requirements. The *INR* are a set of Ada values. The Ada values are defined by a set of Ada types, called the INR Description. The world view is available to other systems via the *EXR* of the information and to the user via *user representations (USR)* of the information. The *USR* is a string representation of the *INR*.

[1] This work is sponsored by the U.S. Department of Defense. The views and conclusions contained in this paper are solely those of the author(s) and should not be interpreted as representing official policies, either expressed or implied, of Carnegie Mellon University, the U.S. Air Force, the Department of Defense, or the U.S. Government.

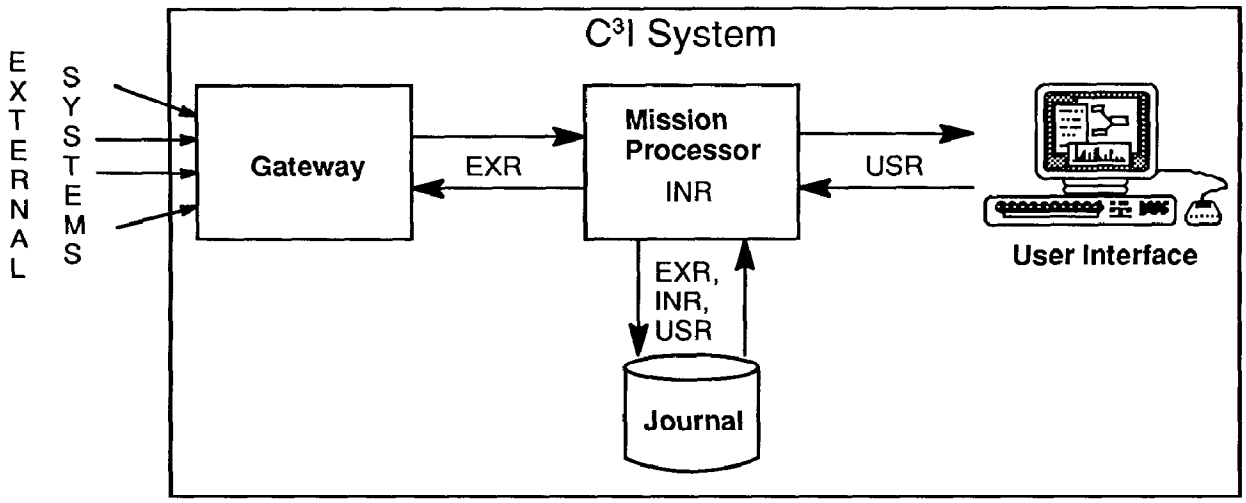

Figure 1: C³I System Block Diagram

The *User Interface* provides a window into the *Mission Processor's* view of the world. It presents all or a subset of the world view, as requested by the user, in a form which is understandable to the user. The user can also add information to the *Mission Processor's* view of the world. The messages enter and leave the *User Interface* as *USR* of the information.

The *Journal* is a storage device used for safe storage of all representations of messages for recovery, analysis, and testing purposes.

Figure 2 shows an EXR Description for a sample message. This example message will be used throughout the paper. The EXR Description describes the field name, field size, range of external values, and the meaning of the external values for the message. For example, field 1 is the Reporting Location. The field size is three characters. The valid external values are "KJL" and "CPP" and "MMR" and the meanings of the values are Andrews AFB, Peterson AFB, and Wright Patterson AFB, respectively. In addition, the EXR Description specifies field separators, if they exist, and an end-of-message indicator.

Figure 3 shows an INR Description for the sample message. The Ada types represent the information in the fields of the EXR Description.

Finally, Figure 4 is an example of the message in its EXR, INR, and USR forms.

Sample Message Format				
Field Number	**Field Name**	**Field Size (chars)**	**Range of Values**	**Amplifying Data**
1	Reporting Location	3	KJL CPP MMR	Andrews AFB Peterson AFB Wright Patterson AFB
	Field Separator	1	/	Slash
2	Direction	1	N S E W	North South East West
3	Date/Time Group	 3 2 2	 001-366 00-23 00-59	Julian Date & Time Julian Day Hours Minutes
	End Of Message	1	<cr>	Carriage Return

Figure 2: Sample Message External Representation Description

```
-- field 1
type Reporting_Location_Type is (Andrews_Afb, Peterson_Afb, Wright_Patterson_Afb);

-- field 2
type Direction_Type is (North, South, East, West);

-- field 3
subtype Julian_Day_Type is Integer range 1 .. 366;
subtype Hour_Type is Integer range 0 .. 23;
subtype Minute_Type is Integer range 0 .. 59;

type Julian_Date_Time_Record_Type is record
    Julian_Day     : Julian_Day_Type;
    Hour           : Hour_Type;
    Minute         : Minute_Type;
end record;

type Sample_Message_Type is record
    Reporting_Location :Reporting_Location_Type;          -- field 1
    Reporting_Direction :Direction_Type;                  -- field 2
    Reporting_Time     :Julian_Date_Time_Record_Type;     -- field 3
end record;
```

Figure 3: Sample Message Internal Representation Description

External Representation:

"CPP/N1810244<cr>"

Internal Representation:

```
Message :=  (Reporting_Location    =>  Peterson_Afb,
             Reporting_Direction   =>  North,
             Reporting_Time        =>  (Julian_Day   => 181,
                                        Hour         => 2,
                                        Minute       => 44));
```

User Representation:

" Peterson_AfbNorth 181 2 44"

Figure 4: Message Representations

RECURRING PROBLEMS IN C³I SYSTEMS

Recurring problems are those problems which appear repeatedly within a system or from system to system. Typical recurring problems in C³I systems are:

- **Packet Unbundling**: packing and unpacking a group of EXR messages. Messages are grouped for ease of transmission.
- **Message Translating and Validating**: translating messages from one representation to another. Validation is performed as part of the translation process.
- **Message Analyzing:** processing information in an INR, updating a mission database, and generating alarm and display information to support the user interface.
- **Journaling:** storing and retrieving message information for generating reports, testing, and restart/recovery processing.
- **Report Generating:** formatting of message information.
- **User Interface Processing:** gathering/presenting USR information from/to users at interactive workstations.

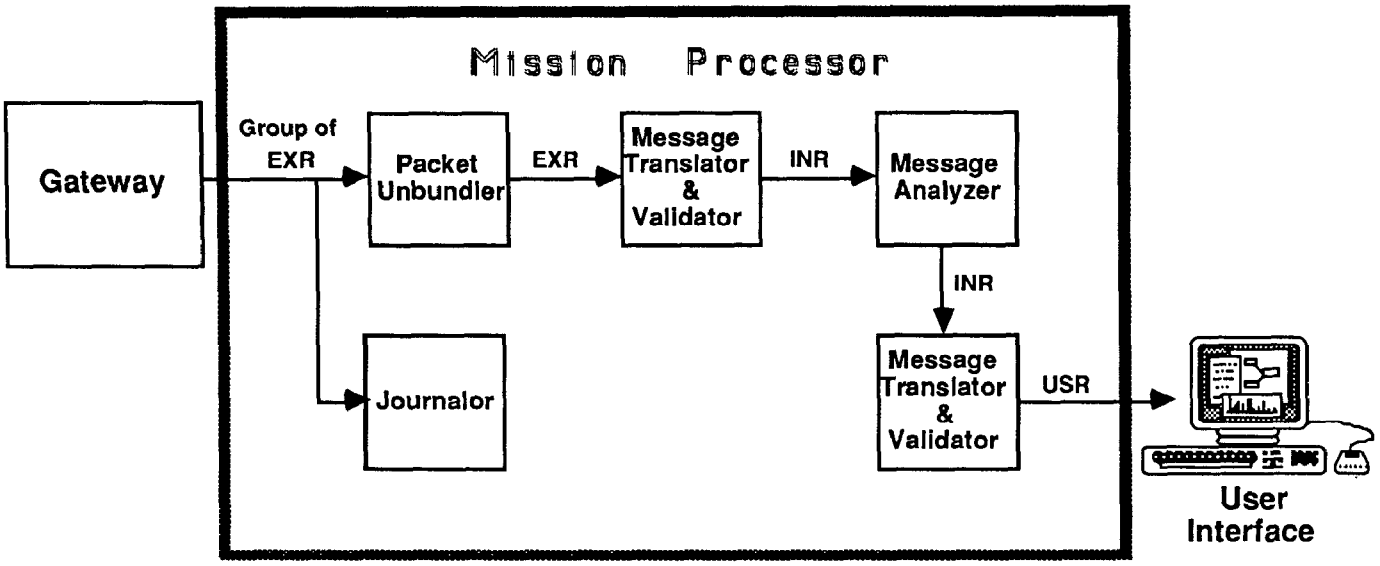

Figure 5: EXR Message Processing Thread Diagram

Models represent reusable, generalized solutions to problems and have their greatest impact on recurring problems. **Thread diagrams** show how models for recurring problems are connected to satisfy certain requirements. For a typical C³I system, several thread diagrams are needed to capture all stimulus-response processing requirements. Figure 5 shows an EXR message processing thread diagram. This diagram shows the Packet Unbundler, Message Translator and Validator, Message Analyzer, and Journalor models in the Mission Processor. A group of messages enters the C³I system from the Gateway and is recorded in a journal file. Packet Unbundler removes one message at a time from the group and makes the message available for translation and validation. MTV converts the message to an INR. Message Analyzer processes the information in the message, updates the mission database, and creates new INR with alarm and display information. MTV converts the INR to a USR for display at a workstation.

Other thread diagrams would show the necessary models which satisfy different stimulus-response processing requirements.

The next two sections of this paper will focus on one recurring problem, Message Translating and Validating, and present a model solution for this problem.

THE MTV RECURRING PROBLEM

The MTV recurring problem is pervasive throughout C³I systems. At GS, MTV occurs on the Mission Processor, on user workstations, and on analyst workstations. Based on an analysis of GS specifically, and the C³I domain in general, we arrived at the following MTV requirements:

1. **Support real-time requirements:**
   a. Translation and validation between EXR and INR to support mission processing.
   b. Translation and validation of all message representations to support writing to a journal.
2. **Support non-real-time requirements:**
   a. Generation of external message representations to support construction of simulation scripts for training purposes.
   b. Generation of all message representations to support system testing.
   c. Translation and validation of all message representations to support reading from a journal.
3. **Support interactive requirements:**
   a. Translation and validation between EXR and INR to support manual entry of information at a workstation and to support presentation and correction of invalid messages received from a mission processor.
   b. Translation and validation between USR and INR to support manual entry of information at a workstation and to support presentation and correction of invalid messages received from a mission processor.

The MTV model, in Figure 5, represents the solution to the real-time requirements. The solution to the non-real-time

and interactive requirements would be represented on other thread diagrams.

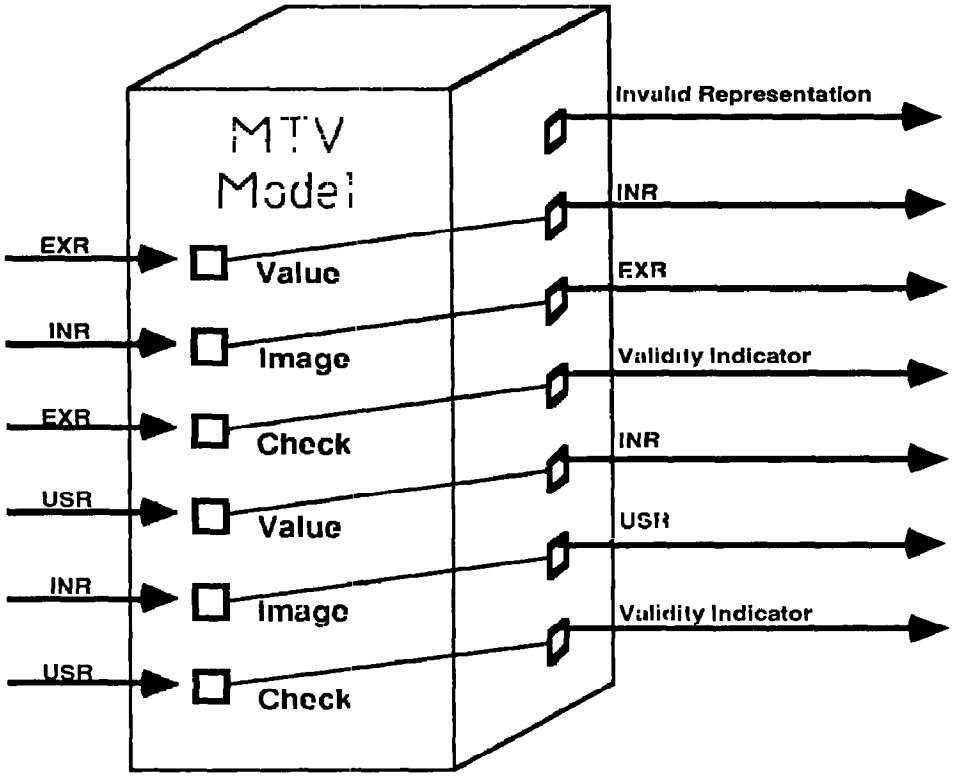

Figure 6: MTV Model Black Box Diagram

A MTV MODEL SOLUTION

A **model solution** is produced in response to a recurring problem and represents an architectural building block which is implemented, in a specific form, for each instance of the recurring problem. A model solution must convey to the system designer an adequate sense of the general problem it solves, and must provide a means to create an instance of the model which solves the specific problem.

This model solution[2] for the MTV recurring problem has the following characteristics:

- The solution addresses all requirements for all instances of the recurring problem throughout the GS Phase II system.
- The solution is based on building blocks. Building blocks allow for the creation of standardized translation and validation solutions for different message formats.
- The solution is consistent. The building blocks provide a consistent interface regardless of the information translated.

MTV Model Functional Description

This section, describing the functionality, conveys to the system designer the problem that the model solves.

Figure 6 shows a black box diagram of the MTV model. The MTV model provides the capability to convert between either the EXR or USR of a message and the INR of a message. The *Value* functions convert from an EXR to an INR, and from an USR to an INR; the *Image* functions convert in the other direction. The conversion includes real-time validation with respect to the range of possible values for the fields and with respect to any inter-field dependencies. If a problem is found, the conversion process is stopped, and an *Invalid Representation* exception is raised.

The MTV model also supports a diagnostic, non-real-time syntactic analysis of both USR and EXR. A diagnostic indicator is returned which supports error detection and cor-

[2] An SEI technical report entitled "A Model Solution for C³I Message Translation and Validation", SEI-89-TR-12 , is forthcoming.

rection. The *Check* functions diagnose USR and EXR and return the diagnostic indicator.

The next three sections describe the means provided to create instances of the model solution.

MTV Model Solution Building Blocks

Building blocks allow for the creation of consistent, standardized solutions. For example, many message fields contain information which can be represented as enumerated values, such as Field 1 and Field 2 in the sample message (see Figure 2). A building block for translating between field representations and enumerated value representations need only be parameterized to specify the mapping between the representations.

The building blocks of the MTV model solution fall into three categories. All the components are necessary to provide the functionality of the MTV model described above.

1. **Discrete Typecaster Generics** are Ada generic packages which serve as the foundation of the MTV model solution. The generic packages convert between Ada discrete values, INR, and strings representing these values, EXR and USR. For example, there is a generic package for converting integer values and another generic package for converting enumerated values.[3] The generic packages must be compiled into the Ada library for use by other portions of the MTV model solution.
2. **Discrete Typecaster Templates** are Ada coding templates[4] which are some of the building blocks of the MTV model solution. The discrete templates convert between Ada discrete values, INR, and strings representing these values, EXR and USR. The templates make use of the generics. There is a one-to-one relationship between the templates and the discrete typecaster generics. The templates insulate the existence of the generics from the application developer. The templates also provide a test procedure which does exhaustive testing, based on the range of the Ada discrete type, and interactive testing. Figure 7 is an example of part of a discrete typecaster template. Engineering points are represented by tokens enclosed in curly brackets, such as *{Type}*, and by double question marks. Figure 8 is the instantiated discrete typecaster using the template in Figure 7.
3. **Composite Typecaster Templates** are Ada coding templates which are the rest of the building blocks of the MTV model solution. The composite templates convert between Ada composite values, INR, and strings representing these values, EXR and USR. For example, there is a composite typecaster template for converting records and another for converting arrays.[5] Instances of these are layered upon both discrete typecasters and other composite typecasters, as shown in Figure 9. The templates also provide a test procedure which does canned testing based on test cases supplied when the template is instantiated.

The use of the template building blocks, to create software for translating and validating messages, guarantees consistency for all instances of the model solution.[6] The generated test procedures allow for easier unit testing of the instances.

MTV Model Solution Building Plan

The following are the steps involved in applying the MTV model solution to a set of messages which need to be translated and validated:

1. **Compile Foundation Utilities.** Compile the utilities which form the foundation of the MTV model solution. These are the components in the Discrete Typecaster Generics category.
2. **Analyze Message.** Define the INR Description, as in Figure 3, based on the information provided in the EXR Description, Figure 2. An Ada type for each field must be defined.

[3] Other generics exist as well. These handle other kinds of representation mappings. See the forthcoming report, SEI–89–TR–12, for more information.

[4] A template is a file containing an Ada package specification, body, and test procedure. The file contains engineering points for the name of the package, the Ada type used in the template, and so on. The template is instantiated by supplying information, in place of the engineering points, via global editor substitutions. Global replacement affects the specification, the body, and the test procedure.

[5] Other composite templates exist as well. For example, there are record and array templates which guarantee inter-field data integrity using private data structures. There is also a wrapper template. The wrapper maps to a discriminated record which allows for null or not present values. See the forthcoming report, SEI–89–TR–12, for more information.

[6] The model solution has been extended to handle the conversion of bit-based EXR to INR. The model was sufficient to account for the new requirements. Two new discrete typecasters (and the associated generics) were created: an integer–bit typecaster and an enumeration–bit typecaster.

```
with Integer_Typecaster;
package {Type}_Typecaster is

    -- The range of values corresponding to an integer image
    --
    subtype {Type}_Type is Integer range {First}..{Last};
    -- The instantiation of an Integer Typecaster
    --
    package {Type}_Tc is new Integer_Typecaster
    (Type_To_Be_Cast => {Type}_Type);

end {Type}_Typecaster;

with {Type}_Typecaster;
procedure {Type}_Typecaster_Test is
?? Enter Test Cases Here
    Test_Cases : is array (1..??) of Test_Record := (??);
begin
    .
    .
    .
end {Type}_Typecaster_Test;
```

Figure 7: Example Discrete Template

```
with Integer_Typecaster;
package Hour_Typecaster is

    -- The range of values corresponding to an integer image
    --
    subtype Hour_Type is Integer range 0..23;
    -- The instantiation of an Integer Typecaster
    --
    package Hour_Tc is new Integer_Typecaster
    (Type_To_Be_Cast => Hour_Type);

end Hour_Typecaster;

with Hour_Typecaster;
procedure Hour_Typecaster_Test is
    Test_Cases : is array (1..2) of Test_Record := ("04", "23");
begin
    .
    .
    .
end Hour_Typecaster_Test;
```

Figure 8: Example Discrete Template Instance

3. **Instantiate MTV Model Solution.** Use the templates provided by the MTV model solution to create an instance of the model solution based on the message analysis performed in the previous step.

    a. **Identify and Build the Discrete Typecasters.** The discrete typecasters needed to translate and validate the discrete elements of a message are identified based on the INR Description constructed in Step 2. Check to see if any of the needed typecasters already exist; some may have been created for other messages. Generate the discrete typecasters which don't exist, using the appropriate discrete typecaster templates. Run the generated test routines to check the discrete typecasters.

For the sample message, five discrete typecasters are needed. Three will use the integer typecaster template. These allow for converting Hour, Minute, and Julian Day values. The remaining two will use the enumeration typecaster template. These will convert Direction and Reporting Location.

b. **Identify and Build Composite Typecasters.** The composite typecasters needed to group discrete and composite elements of the message are identified based on the INR Description constructed in Step 2. Check to see if any of the needed typecasters already exist; some may have been created for other messages. Generate the composite typecasters which don't exist, using the appropriate composite typecaster templates. Run the generated test routines to check the composite typecasters.

   For the sample message one composite typecaster is needed. This Julian Date Time record typecaster is built using the record typecaster template.

c. **Build the Message Typecaster.** The message typecaster is generated using the appropriate composite template, usually the record typecaster template. Run the generated test routine to check the instance of the MTV model solution for the message.

   For the sample message, the message typecaster is generated from the record typecaster template.

The application developer need not be concerned with the generics unless the code performance (sizing or timing) is inadequate to meet the system performance requirements. The application developer need only be concerned with the discrete and composite templates and instantiating them, as necessary, to obtain the MTV capabilities required by the system under development.

MTV Software Architecture

Figure 9 shows the general software architecture for the MTV model solution. Figure 10 shows the software architecture which results when the MTV model solution is applied to a specific message format. The example message is that shown in Figure 2. The software architecture is shown as Ada packages and the dependencies among them.

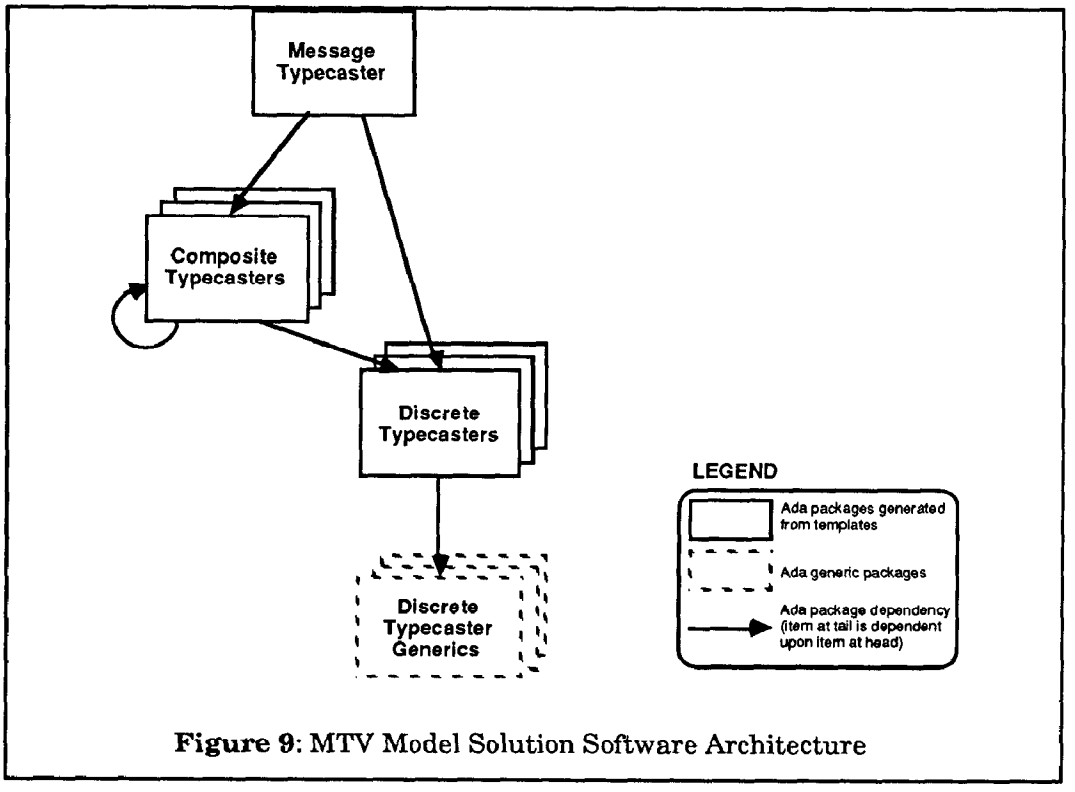

Figure 9: MTV Model Solution Software Architecture

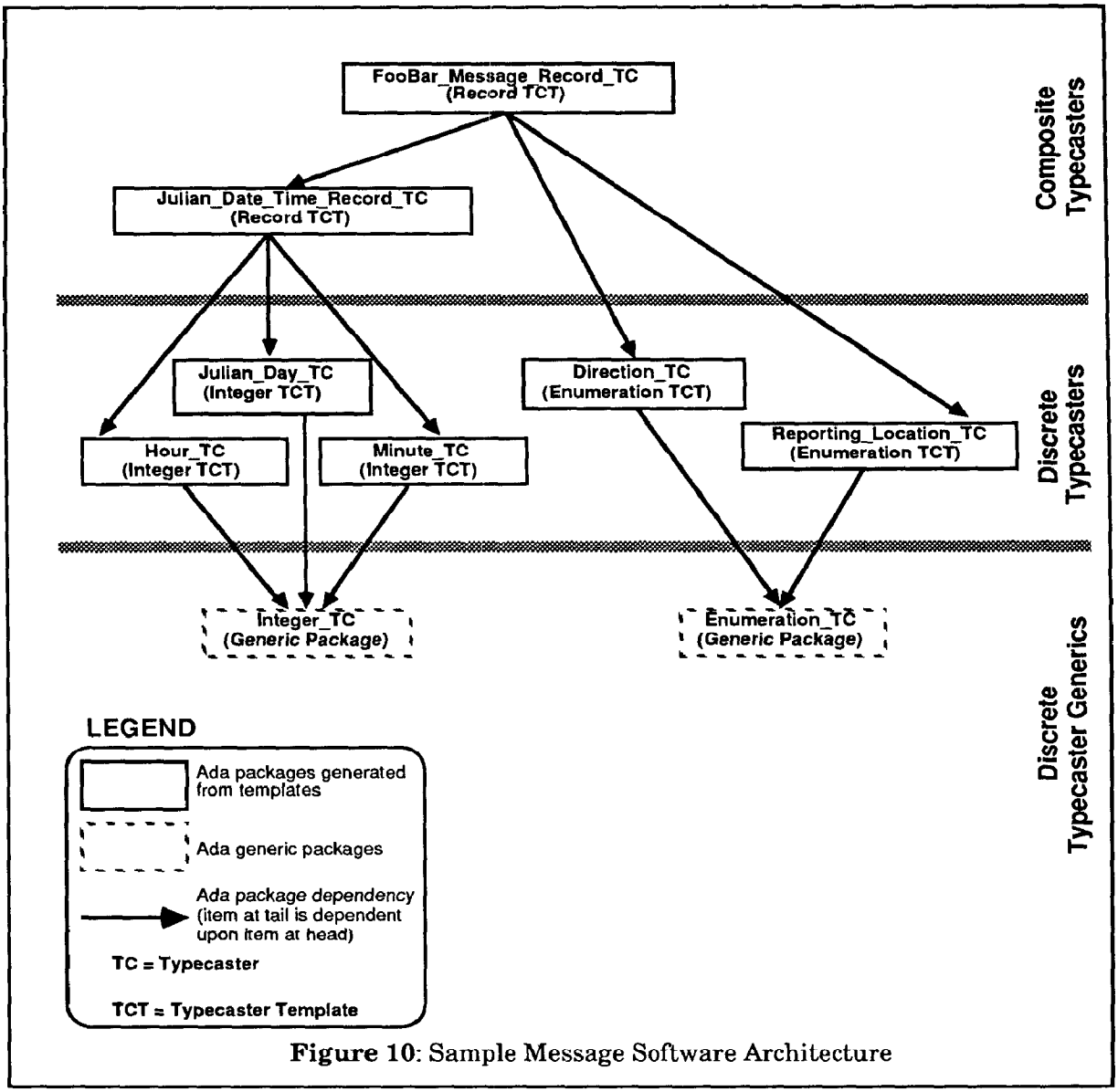

Figure 10: Sample Message Software Architecture

The software architecture is based upon the structure of the Ada types. When the MTV model solution is instantiated for a particular message, the resulting architectural components are instances of the discrete typecaster templates and composite typecaster templates, one for each Ada type used to describe the INR of the message. The MTV architecture is hierarchical in nature. The discrete typecasters are dependent upon the discrete typecaster generics. The composite typecasters and the message typecaster may be dependent upon instances of both discrete typecasters and composite typecasters.

MTV MODEL SOLUTION IN USE

The MTV model solution was developed while the DSSA Project members were involved with Phase I of the GS Program. GS Phase II has adopted the model solution and, as of this writing, has generated instances of the model solution for translating and validating 11 of 26 messages. This section summarizes Phase II's experiences using the MTV model solution.[7]

The Use of the MTV Model Solution

The EXR <—> INR translation and validation functionality is currently being used to check and convert message EXR received from the NORAD Computer System to INR for analysis. The formats of the EXR are specified in the GS

[7] We would like to thank Major Mike Goyden and Lt. Jordie Harrell of Air Force Space Command, and Guy Cox of Martin Marietta, for their support and cooperation in providing feedback regarding the use of the MTV model solution. See Goyden's paper, *The Software Lifecycle with Ada: A Command & Control Application*, also in this conference.

Phase II Interface Control Document. The INR are specified by the application developers and capture the information in a form which can be analyzed by the GS system. The information is used to update the world view and to raise appropriate alarms which notify the users of critical events.

The USR <—> INR translation and validation functionality is being used to check and convert USR, received from the user workstations, to INR for analysis. This functionality is also being used to convert the INR, which are the result of message analysis, to USR which are sent to the user workstations. The USR is the basis for displaying information to the user and obtaining information from the user at the workstation.

Finally, both sets of translation and validation functionality (EXR <—> INR and INR <—> USR) are being used to to support journalling and report generation. Messages are journalled in all representations. At a later time, messages are retrieved from journal files, in the various representations, and must be converted to USR so that the information can subsequently be formatted into humanly readable reports.

The use of the model solution in multiple places demonstrates that the model solution solves the MTV problems found in different parts of the GS Phase II system.

The MTV software should meet the performance requirements for MTV for GS Phase II based upon preliminary timing runs performed on parts of the model solution at the SEI. GS Phase II has performed no formal timing studies.

The MTV model solution was used "as delivered" by GS Phase II except for the following:

1. A new generic discrete typecaster, String Typecaster, was developed for conversion between Ada string values (INR) and free text EXR fields. This was necessary when no validation of the field was needed and the information in the field did not affect the message analysis. The String Typecaster also ensures that no non–displayable characters are sent to the display.

2. A new generic discrete typecaster, Fixed Point Typecaster, was developed for conversion between Ada fixed point values (INR) and strings representing these values (EXR and USR).

3. The Record_Typecaster template's engineering points were increased to allow 32 elements in the record. This was done because of the large amount of information contained in some messages.

Benefits of the Use of MTV Model Solution

The following are a few of the benefits reported by the GS Phase II team:

1. Less inline documentation is required of the MTV model solution.

   The delivered version of the model solution had inline documentation for all discrete and composite templates. This documentation made up a good portion of the total number of characters in the template. GS Phase II engineers reported that this documentation was examined, initially, for an understanding of the templates, but once the templates were understood the documentation was no longer necessary. This is especially true when the documentation present in each of the instances of the templates is the same, except for the engineering points.

   Based on this feedback, the header documentation and most inline documentation was removed from the Ada code and was incorporated in the report currently being developed by the DSSA Project.[8] The header documentation for each template points to the report for the general description of the template and only contains a description of the engineering points used to create an instance of the template.

2. Less time is spent on code reviews and walkthroughs for the instances of the model solution. Code review and walkthrough effectiveness has increased.

   The templates were reviewed initially, before each was used, for correctness of the code and the coding style. Once the templates passed the review process, instances of the templates were not fully reviewed. They were only reviewed based on the information used to instantiate the templates, i.e., the engineering points.

3. Reliability of message translation and validation code has improved.

   Reliability comes from the use of the generic discrete typecasters, which were developed and tested, and which form the foundation of the MTV model solution, and from the use of the templates, discrete and composite, which constrain how the developer uses the generics. The use of these building blocks assures that solutions, for each message, are consistent in structure, behavior, and functionality.

[8] See the forthcoming report, SEI–89–TR–12.

Two errors were found in the software delivered to GS Phase II. These were minor errors found early when testing instances of the templates using the test procedures included in each template. The errors were corrected in the templates, so subsequent instances could benefit from the early testing.

4. Productivity is increased.

The model solution, embodied in the templates, provides a means of specifying instances at a high level of abstraction. The high level of abstraction insulates a developer from the implementation details of the building blocks. Generating an instance merely involves selecting the appropriate templates and substituting for the engineering points. All other relationships and dependencies are inherent in the instantiated solution.

Working at a higher level of abstraction, like the move from assembly language to higher-level languages, allows one to be more productive. The building blocks used embody the implementation and developers only need to understand what functionality the building blocks provide and how to use them.

Similar to the move from assembly language to higher–order languages, the move from language constructs to model solutions removes the need for determining productivity based on language constructs, e.g., counting lines of code.

Productivity is a measure of the effort spent performing code generation, documentation, test generation and execution, reviews and walkthroughs, and so on. Generating the code for one message from the templates requires about one work–day. The test drivers are part of the templates and the only effort involved is the specification of test cases. Documenting the instance involves specifying the engineering points for each template used. Reviews only need to consider the choice of substitutions for the engineering points.

But, for purposes of illustration, some line of code numbers are provided: in 18 work–weeks, MTV code for 11 messages was generated, documented, tested, and reviewed. This included 150 instances of the templates for a total of 9600 lines of code (semi-colon count) or about 100 lines/work–day.

5. Consistency of the model solution makes using the model easier and consistency of the resulting software should aid maintenance.

Training implementors to use the model solution is easier because the building blocks are similar in structure, behavior, and functionality. Once the features of one are understood, development using any of them is straightforward. Similarly, quality assurance can be performed more easily because all instances are derived from the building blocks.

To date, GS Phase II has not performed maintenance on developed software. But, it is anticipated that the consistency, embodied in the building blocks, will enable maintainers to understand the model solution, to localize where changes need to be made, and to modify the software more effectively.

FUTURE CONSIDERATIONS

To realize these payoffs, model databases must be populated and the software development process must be refined to take advantage of existing pools of model solutions. This should occur as an evolutionary process.

First, domain experts need to identify recurring problems in their domains. We will support this by validating and refining the recurring problem approach for identifying target models.

Second, model solutions need to be developed and verified. Based on our experience with GS, prototype solutions should be built using a representative subset of the data for each recurring problem. Verification is based on both functionality and performance. In addition, the system should be prototyped by integrating the initial solutions to demonstrate that the integrated models will meet system requirements. After the solutions are verified, the prototype solutions are generalized to produce code templates and generics. The templates and generics help to insure that each instantiation of the model provides the functionality specified by the model. They also promote code and comment consistency. These characteristics encourage reuse.

Third, models solutions need to be documented and published so they are recognizable, usable, and adaptable. We propose the following documentation outline:

1. **Problem Description.** Describes the problem the model solves.
2. **Model Description.** Provides a functional description, an interface description and a description of resource requirements of the model. This is equivalent to a page from an engineering handbook describing a standard component.
3. **Model Solution Overview.** Provides an high-level overview of the model solution. Lists the building

blocks, how to apply them, and architectural ramifications of the use of the model solution.

4. **Model Solution Application Description.** Describes how to use the model solution to solve a problem. This is equivalent to a user's manual.
5. **Model Solution Detailed Description.** Describes the implementation details of the model solution.
6. **Model Solution Adaptation Description.** Describes how to adapt the model solution if it doesn't quite solve your problem.
7. **Open Issues.** Addresses issues of interest. These include functional limitations, performance limitations, etc.

We are working with GS Phase II, GS Phase III, and AWIS to capture models in this form and group the descriptions in a C³I Model Handbook.

Finally, the development process needs to be refined to encourage systems to be designed by selecting the appropriate models from the model databases, verifying designs based upon model solutions, and building the system using the model solutions.

CONCLUSIONS

The DSSA Project has developed a MTV model solution for a problem which recurs in the C³I domain. GS Phase II is using the MTV model solution. The functionality provided by the MTV model solution meets their needs and, based on early timing and sizing analysis, it also satisfies their performance requirements. The GS Phase II development team is more productive in generating MTV code and is producing a reliable, maintainable, and consistent product.

While developing the MTV model solution and participating in design reviews at GS, we developed a process for identifying models. This process entails identifying problems which recur on a project or across similar projects in one domain. Once identified, detailed solutions to these problems are developed, depth-first, for a representative subset of the data. After this prototyping, the solutions are generalized to model solutions which are used to generate instances for the rest of the data, i.e., to complete the system. Also, while developing the MTV model solution, we developed a way of documenting models to make them recognizable, usable, and adaptable[9]

Based on our experiences developing, documenting, and transitioning the MTV model solution in the C³I domain, we feel that the development and use of models in the software engineering field will provide high payoffs.

[9] Rich D'Ippolito, of the SEI, was instrumental in helping to define how models should be documented to make them reusable at both the design and implementation levels. See his paper, *Using Models in Software Engineering*, also in this conference.

About the Author: *Charles P. Plinta is a member of the technical staff (MTS) on the Software Architectures Engineering (SAE) Project at the Software Engineering Institute (SEI).*

This project is applying engineering methods to the design of software. Plinta's previous assignments at the SEI were as a member of the Dissemination of Ada Software Engineering Technology (DASET) Project and the Domain Specific Software Architectures (DSSA) Project.

Plinta's work at the SEI has included consulting and working with defense contractors and DoD program offices on technical issues related to system design and the use of Ada in real-time systems. This work has been focussed on the flight simulator and C³I domains.

Before joining the SEI, Plinta was an engineer at the Defense Electronics Center of Westinghouse Electronic Corporation. In that position he participated with and lead software teams in the design and development of real-time radar systems, hardware test and diagnostic languages, and simulations.

Plinta holds a bachelor's degree in computer science and mathematics from the University of Pittsburgh. He is a member of the IEEE Computer Society, and is the coauthor of several papers and presentations on design issues and their relationship to Ada-based systems.

About the Author: *Kenneth J. Lee is a member of the technical staff (MTS) for the Software Engineering Institute (SEI) where he is currently a member of the Software Architectures Engineering (SAE) Project. This project is applying engineering methods to the design of software. Lee's previous assignments at the SEI were as a member of the Dissemination of Ada Software Engineering Technology (DASET) Project and the Domain Specific Software Architectures (DSSA) Project.*

Lee has worked on Ada-based designs for simulators, for command, control, communications, and intelligence systems (C³I systems), and for embedded systems and other real-time systems. He is the author of several SEI Technical Reports, papers, and conference presentations on software design issues in Ada-based systems.

His interests include: Ada-based software engineering, real-time systems, distributed systems, pattern-based modeling for software design, and automation of the design efforts, through tools and automated methods. Prior to joining the SEI, Lee was a post-doctoral student in the Department of Engineering and Public Policy at Carnegie Mellon University.

Lee received a bachelor's degree in chemistry from Carleton College in Northfield Minnesota, and a masters and doctorate in organic chemistry from University of California, Los Angeles.