Technical Report

CMU/SEI-89-TR-12

ESD-89-TR-20

Carnegie-Mellon University

Software Engineering Institute

A Model Solution for C^3I Message Translation and Validation

Charles Plinta
Kenneth Lee
Michael Rissman

December 1989

Technical Report
CMU/SEI-89-TR-12
ESD-89-TR-20
December 1989

A Model Solution for C^3I Message Translation and Validation

Charles Plinta
Kenneth Lee
Michael Rissman
Software Architectures Engineering Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

[illegible]

Karl H. Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Table of Contents

List of Figures

A Model Solution for C^3I Message Translation and Validation

Abstract:

This document describes an artifact, the Message Translation and Validation (MTV) model solution. The MTV model solution is a general solution, written in Ada, that can be used in a system required to convert between different message representations. These message representations can be character-based, bit-based, and internal (i.e., Ada values).

This document provides designers with enough information to determine whether this solution is applicable to their particular problem. It gives detailed designers the information needed to specify solutions to their particular problem using the MTV model solution. Finally, it describes the MTV model solution in enough detail to enable a maintainer or adapter to understand the solution.

1. Introduction

1.1. Purpose

The purpose of this document is to describe an artifact, the Message Translation and Validation (MTV) model solution. The MTV model solution is a general solution, written in Ada, that can be used in a system when the system must convert between different representations of a message. These message representations can be character-based, bit-based, and internal (i.e., Ada values). This document enables designers to determine whether this solution is applicable to their particular problem and shows detailed designers how to specify solutions to their *particular problem using* the MTV model solution. Finally, it describes the MTV model solution in enough detail to enable a maintainer or adapter to understand the solution.

1.2. Intended Audience

This document was written specifically for developers of software systems in the Command, Control, Communications, and Intelligence (C^3I) domain where there is a need to translate and validate incoming and outgoing messages. Enough C^3I context information is provided so that developers in other domains, where a need to translate and validate messages also exists, can understand and use the MTV model solution.

The authors assume that the reader has some knowledge of the Ada programming language [Ada 83] because the MTV model solution is implemented using Ada. The early chapters of the document support designers and therefore require less Ada knowledge than the later chapters that are targeted for the detailed designers, implementors, maintainers, and model adapters.

1.3. Background

Work being produced by Software Architectures Engineering (SAE) Project has been on-going at the Software Engineering Institute (SEI) since 1986 [Lee1 88, VanScoy 87, Lee2 89, D'Ippolito2 89, Lee3 88, Plinta 89, D'Ippolito1 89]. The primary goal of the project is to encourage the creation and use of canonical design solutions for typical recurring problems in an application domain. The MTV model solution is one such canonical design solution to the message translation and validation problem recurring in the C^3I domain.

SAE's involvement in the C^3I domain is primarily with the Granite Sentry Program [Goyden 89, GSSDP 89]. The Granite Sentry Program is a phased hardware and software replacement of the systems in the Cheyenne Mountain Complex of the North American Aerospace Defense Command (NORAD). The Air Force Electronic Systems Division (ESD) of Wright Patterson Air Force Base is the contracting office, and Air Force Space Command of Peterson Air Force Base is the development office. The MTV model solution is being incorporated into the current phases of the Granite Sentry Program and several other C^3I programs.

This document was initiated by the Domain-Specific Software Architectures (DSSA) Project at the SEI.

1.4. Reader's Guide

This document is broken into nine chapters and four appendices. Each chapter is targeted to a different portion of the software systems development audience (i.e., designers, detailed designers, implementors, maintainers, model adapters). If the audience is not specified, the chapter or appendix is of general interest.

Chapter 1: Introduction
Defines the purpose of the document, the intended audience, and sets the context for the work documented in this report.

Chapter 2: The Recurring Problem
Characterizes C^3I systems from a message point of view and defines the MTV problem found in many C^3I systems.

Chapter 3: TV Model Description
Describes the Translation and Validation (TV) model as a black box. It describes what the TV model does and how to access its capabilities. This is the model used to solve the message translation and validation problem.

Chapter 4: MTV Model Description
Analyzes the MTV recurring problem with respect to the TV model. Instances of the model are generated to solve the problem, and these instances form the MTV model. This chapter is targeted for a designer responsible for designing the system. The designer needs a pool of models from which to choose solutions to populate the design space. These solutions need to be documented such that the system designer can make educated decisions when selecting particular models.

Chapter 5: MTV Model Solution Overview

Provides an overview of the MTV model solution. This overview is needed by the designer to understand, at a high level, the consequences of choosing this MTV model solution. The overview is also provided for the detailed designer, implementor, maintainer, and model adapter as an introduction to the MTV model solution.

Chapter 6: MTV Model Solution Application Description

Describes how to apply the MTV model solution that is described in Chapter 7 to the message translation and validation requirements of a particular application. This chapter is targeted for the detailed designers responsible for specifying (and actually in this case implementing) all software for translating and validating messages.

Chapter 7: MTV Model Solution Description

Describes the internals of the MTV model solution and exactly how the model solution provides the capabilities described in Chapter 4, MTV Model Description. This chapter is targeted for the maintainer and the model adapter, that is, those responsible for understanding the implementation of the system for either maintenance purposes or for the purpose of adapting the model for use in a particular application for which the current state of the model solution does not suffice. Adaptation may be necessary because of efficiency problems (size and speed) or because of the need for additional functionality.

Chapter 8: MTV Model Solution Adaptation Description

Describes how to adapt the MTV model solution described in Chapter 7. This chapter is targeted for the model adapter.

Chapter 9: Open Issues

Describes issues that may need to be addressed by the designer and model adapter.

Appendix A: Definitions

Contains a glossary of the terms and acronyms used throughout the document. Terms defined in this appendix are bold and italicized the first time they appear in the document.

Appendix B: Detailed Description of the Templates

A reference manual for MTV model solution application. It describes how to apply all templates individually in detail. This appendix is targeted for the detailed designer.

Appendix C: MTV Model Solution Ada Code

Contains the Ada code that comprises the MTV model solution. Appendix C is self-contained.

Appendix D: FooBar Message Ada Code

Contains examples instances of the MTV model solution. The same example is used throughout the document. Appendix D is self-contained.

1.5. Acknowledgments

We thank Richard D'Ippolito for his help with the organization and layout of the information in this document. He was instrumental in helping define how models should be documented and in separating the concerns of the designer, implementor, and maintainer.

We also thank Doug Resetar, a student from Grove City College who worked with us during the summer. Doug helped develop and test portions of the Ada code and got the Ada code into a publishable format. Doug also helped to write Section 7.3 and provided the timing data found in Section 5.5.

We thank the reviewers for their time, effort, and constructive criticisms that kept the authors on their toes and helped shape the final publication of this document. People from the SEI include: Judy Bamberger, Tim Coddington, Richard D'Ippolito, Bob Holibaugh, Stan Rifken, Jeff Stewart, Roger Van Scoy, and Rob Veltre. People related to the Granite Sentry Program include Lt. Jordie Harrell of AFSPACECOM and Guy Cox of Martin Marietta. Other reviewers from industry include Bill Schmidt of SofTech in association with the Rapier Program and Chuck Williams of CONTEL Federal Systems Division. Finally, we thank Major Mike Goyden for his willingness to consider and evaluate this model solution for application on Granite Sentry Phase II.

2. The Recurring Problem

This chapter briefly describes the C^{3}I domain, the role of messages in it, and the recurring message translation and validation problem found in it. This chapter sets the context for the remainder of the document to discuss the MTV model solution.

2.1. Characteristics of the C^{3}I Domain

A C^{3}I system accepts messages (information) that describe a view of the world as seen by other systems (sensors, electronic, human, etc.). The C^{3}I system transforms these sensor views into an internal view. The internal view is used to update the system's view of the world. It allows the users to examine a portion of the world view in a manner that is understandable to them. The C^{3}I system or user can also react to new information and send information about a particular view of the world back out to other systems. It allows for interactive entry of messages and correction of errant messages. It allows for the recovery of journalled messages for analysis purposes, and for the generation of simulation scripts (i.e., messages) for training and system testing purposes.

Figure 2-1 is a high-level block diagram of a typical C^{3}I system. The following paragraphs describe the entities shown Figure 2-1. These entities are italicized in the text.

The *Gateway* sends/receives messages to/from all systems external to the C^{3}I system. The *Gateway* is an interface between the C^{3}I system and all other systems. Messages (information) are communicated between the systems. The messages enter and leave the C^{3}I system as *external representations* of the information whose formats are defined by the other systems.

The *Mission Processor* maintains a view of the world based on the views (*external representations*) provided by the other systems. This world view is kept in an *internal representation* to allow processing of the information based upon the C^{3}I systems mission requirements. This view is available to other systems via *external representations* of the information and to the user via *user representations* of the information.

The *User I/F* (user interface) provides a window into the *Mission Processor*'s view of the world. It presents the information about a user-selectable view in a manner that 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 I/F* as *user representations* of the information whose format is understandable to the user.

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

The different representations of a message are discussed in more detail in the next sections.

General system constraints that are normally placed on the described C^3I system can be summarized as follows:

- C^3I systems tend to consist of distributed processors upon which the required functionality must be allocated.
- C^3I systems tend to have soft real-time processing requirements for processing a message and getting the information to the user.[1]
- C^3I systems also need to keep journal entries of messages off-line for recovery, analysis, and simulation purposes.

[1]Timing requirements tend to be on the order of *seconds* to process a message. The peak load of messages to be processed per second varies.

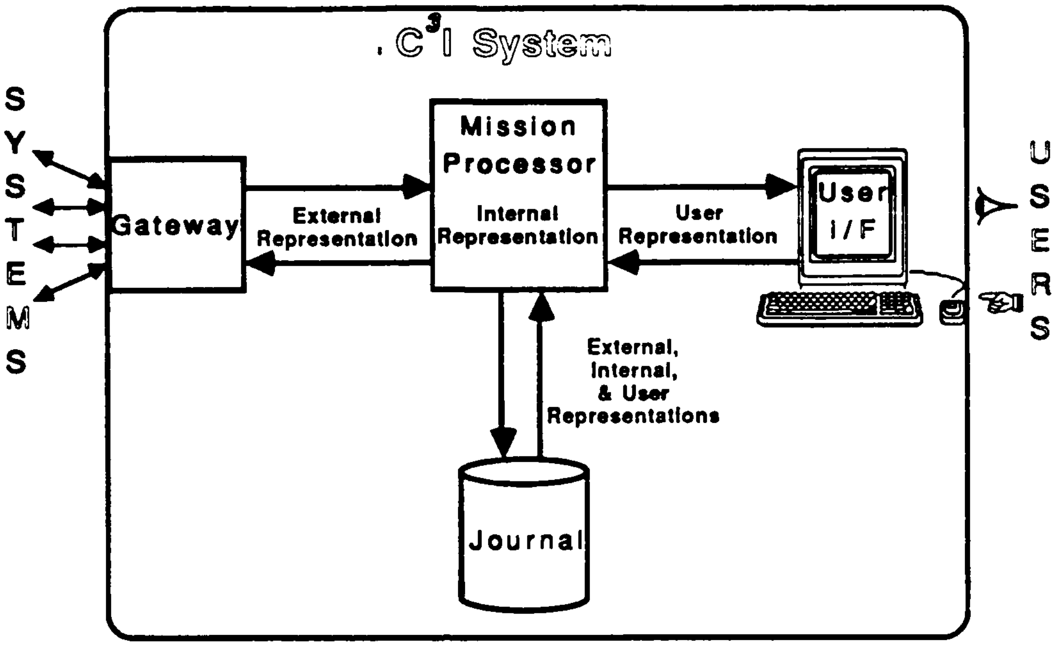

Figure 2-1: C³I System Block Diagram

2.2. Messages in the C^{3}I Domain

A ***message*** is defined as pieces of related information. Based on the characteristics of systems in the C^{3}I domain, messages need to be expressed in three forms:

1. external representation
2. internal representation
3. user representation

Conversion between these forms of a message must be provided. These forms are described in detail in the following sections.

2.2.1. External Message Representation

An ***external representation (EXR)*** of a message is a string.[2]This representation is received from (or sent to) systems outside the C^{3}I system being developed. The string is normally cryptic for the purpose of facilitating fast communications. The string contains multiple ***fields***, each of which contains the related information in the message. The fields can be separated by ***punctuation*** or be in a fixed position in the string. The fields in the string can be interpreted as ASCII character strings or bit strings.

An ***external representation description*** (***EXR description***) is a textual description of the external representation of a message. It defines the size and location of the fields and punctuation. It also describes the information found in the message and how to interpret the information. The EXR description is normally defined outside the scope of the C^{3}I system being developed and becomes part of the requirements levied upon the C^{3}I system. An EXR description exists for each message that the C^{3}I system is required to process. Figure 2-2 is an example of a textual EXR description. Note that all fields in this EXR are character-based. This example will be used throughout the document.

Figure 2-3 is an example of a message described by the EXR description shown in Figure 2-2.

String values a field can contain are called ***symbolic images***. Symbolic images encode information of a given field. For example, the *Direction* field shown in Figure 2-2 has a set of valid symbolic images associated with it: the ASCII characters "N", "S", "E", and "W". The meaning or value of the symbolic images is North, South, East, and West.

Other characteristics of fields include recurrence of fields in a message set and possible varying length fields. Varying length fields result from the field being null (i.e., not present) or a variable length symbolic image. These characteristics of a field are discussed in detail in the context of the model solution in Chapter 6.

[2]A string is defined as a contiguous set of bytes that can be interpreted as ASCII characters or bit streams.

FooBar 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	<cr>	Carriage Return
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
	Field Separator	1	/	Slash
4	Status	1	0 1	Operational Non-Operational
	End-of-Message	1	<cr>	Carriage Return

Note: If the Status field indicates Non-Operational, then the Date/Time Group field must have a value indicating January 1 at 12:00 am (i.e., "0010000").

Figure 2-2: Example Char-Based EXR Description: FooBar Message

```
"CPP<cr>E1831407/0<cr>"
```

Figure 2-3: Example Char-Based EXR: FooBar Message

Messages also have associated with them the notion of validity. There are three concepts involved with message validity:

intra-field validity	The symbolic image in a field is in the range of valid symbolic images specified in the EXR description for that field. For example, valid symbolic images for the *Direction* field show in the EXR description in Figure 2-2 are the ASCII characters "N", "S", "E", and "W".
inter-field validity	The symbolic images of interdependent fields conform to constraints that are specified in the EXR description. For example, the note specified in the EXR description in Figure 2-2 states that if the *Status* field is an ASCII character "1", then the *Date/Time Group* field must be exactly the following ASCII characters, "0010000". If not, the message is not valid.
inter-message validity	The message received is not a redundant message or a message whose time tag is older than a previously received message.[3]

Figure 2-4 shows an example of an EXR description whose fields are bit-based. Notice that there are two parts to the description. The first deals with the field information descriptions: the number of bits a field occupies and the meaning of various bit patterns. The second part deals with the organization of the fields in the message string (i.e., bit positions). For example, bits 0-1 of word 1 are the most significant bits of the *Julian Day*, bits 0-5 of word 2 are the next set of significant bits of the *Julian Day*, and bit 5 of word 3 is the least significant bit of the *Julian Day*. Figure 2-5 shows an example EXR of the bit-based EXR description in Figure 2-4. The values are the same as in Figure 2-3.

[3]The model solution described in this document does not address the problem of inter-message validity. The C^3I system must provide this functionality.

FooBar Bit-Based Message Format				
Field Number	**Field Name**	**Field Size (bits)**	**Range of Values**	**Amplifying Data**
1	Reporting Location	2	1-3	1 - Andrews AFB 2 - Peterson AFB 3 - Wright Patterson AFB
2	Direction	2	0-3	0 - North 1 - South 2 - East 3 - West
3	Date/Time Group	 9 5 6	 001-366 00-23 00-59	Julian Date & Time Julian Day Hours Minutes
4	Status	1	0-1	0 - Operational 1 - Non-Operational

Note: If the Status field indicates Non-Operational, then the Date/Time Group field must have a value indicating January 1 at 12:00 am.

```
--------------------------------------------------------------------
                   FooBar Bit-Based Message Data Format
--------------------------------------------------------------------
Byte                              Bits
--------------------------------------------------------------------
          MSB                                            LSB
           7      6      5      4      3      2      1      0
         +-----------------------------------------------------
1        |  *      *    {   1   }   {   2   }   { 3.day }
         |
2        |  *      *    {            3.day                  }
         |
3        |  *      *  {3.day} {          3.hour             }
         |
4        |  *     {               3.minute            }   {4}


         * = Bits that have no meaning
```

Figure 2-4: Example Bit-Based EXR Description: FooBar Message

2.2.2. Internal Message Representation

An ***internal representation (INR)*** of a message, for the purpose of this solution, is an Ada typed value. This value is used internal to the C^3I system being developed for processing and analysis purposes. The value is based on the information contained in a message and is a more natural representation of the information than the EXR. The value is of an Ada composite type containing elements that group the information in the message. There is at least one element for each field in the EXR. The types of the elements can be Ada discrete types or Ada composite types.[4]

The purpose of the INR is to allow the information to be organized in a manner that is usable, natural, and efficient to the implementor and not constrained by the format of the message as specified by the EXR description. It also allows the information in the message to be processed and analyzed by the C^3I system for presentation to the user or to be sent to other systems.

An ***internal representation description*** (***INR description***) is the set of Ada type declarations needed to define an internal representation of a message. The Ada types are defined by the detailed designer and are part of of the detailed design specification. Each Ada type defines the type for one of the elements. An INR description must exist for each message to be translated and validated.

Figure 2-6 is an example of an INR description corresponding to the EXR descriptions shown in Figure 2-2 and Figure 2-4. This example is used throughout the document. Figure 2-7 is an example the INR described by the INR description in Figure 2-6.

An assumption made by the model solution described in this document is that the INR is always valid as defined by the Ada types. Once an INR of a message exists, the Ada runtime (or application) will assure its validity at the intra-field level, and the application will assure validity at the inter-field and inter-message level.

The descriptions of external message representations and internal message representations begin to define the initial part of the problem, i.e., being able to convert between the EXRs and the INRs of a message.

[4]Types of the elements can also be fixed or floating point. The MTV model solution does not address these. See Chapter 8 for more details on how the solution can be extended.

```
byte1:  00101001
byte2:  00011011
byte3:  00101110
byte4:  00001110
```

Figure 2-5: Example Bit-Based EXR: FooBar Message

```
  -- 1
  type Reporting_Location_Type is (Andrews_AFB, Peterson_AFB, Wright_Patterson_AFB);

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

  -- 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 ;

  -- 4
  type Status_Type is (Operational, Non_Operational);

type Foobar_Message_Type is record
  Reporting_Location  : Reporting_Location_Type;        -- 1
  Reporting_Direction : Direction_Type;                 -- 2
  Reporting_Time      : Julian_Date_Time_Record_Type;   -- 3
  Reporting_Status    : Status_Type;                    -- 4
end record ;
```

Figure 2-6: Example INR Description: FooBar Message

```
Foobar_Message : Foobar_Message_Type := (
  Reporting_Location => Peterson_AFB,
  Reporting_Direction=> East,
  Reporting_Time     =>(
    Julian_Day     => 183,
    Hour           => 14,
    Minute         => 7),
  Reporting_Status    => Operational);
```

Figure 2-7: Example INR: FooBar Message

2.2.3. User Message Representation

Finally, because information must be presented to the user of a C^3I system, and information must be obtained from the user, we also need a ***user representation (USR)*** of the information. This representation is a user-readable, fixed-length character string representation of the information.

The purpose of the USR is to *support* the presentation of the information in a manner that is natural to the user and is not constrained by the EXR. The USR supports the interactive portions of the C^3I system requirements. Figure 2-8 is an example of a USR of the message described by the INR description shown in Figure 2-6.

```
"          Peterson_AFB East 183 14   7     Operational"
```

Figure 2-8: Example USR: FooBar Message

The USR is derived from the INR. Because of this the USR is broken up into elements, one for each element of the INR. We call these elements of the USR ***natural images*** because they are more natural representations of the information than those found in the EXR. For example, the *Direction* element shown in the INR description in Figure 2-6 has a set of valid natural images associated with it, the ASCII character strings "**North**", "**South**", " **East**", and " **West**". The meaning or value of the natural images is North, South, East, and West.

The natural images in the USR are images of the INRs shown in Figure 2-6. Although this does not look extremely user readable, it is more so than the EXR, which relies on symbolic images. How the USR in our model solution supports a user interface is described in more detail in Chapter 7.

The description of the user message representation defines the final part of the problem, i.e., being able to convert between the USRs and the INRs of a message.

2.3. The MTV Recurring Problem

The different representations of a message described in the previous sections are representations that a C^{3}I system receives or produces. All three representations are necessary for the C^{3}I system to function properly.

The recurring problem becomes evident when we realize that ***all*** messages that the application must translate and validate must be representable in these forms.

We can therefore summarize the MTV recurring problem requirements. These are defined in two parts. The first part contains the functional requirements levied upon a solution, and the second part contains software engineering requirements placed upon the solution.

To meet the functional requirements imposed by C^{3}I systems in general, a solution must:

1. Support these real-time activities:
    a. Translation and validation of external message representations to internal message representations.
    b. Translation of internal message representations to external message representations. Validation of internal message representations is the responsibility of the C^{3}I system.
    c. Translation and validation of all message representations to support writing to a journal.
2. Support these non-real-time activities:
    a. Generation of external message representations to support 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 these interactive activities:
    a. Translation and validation of external message representations to internal message representations (and vice versa) to support manual entry of information and presentation of invalid information to allow correction of the invalid information received.
    b. Translation and validation of user message representations to internal message representations (and vice versa) to support manual entry of information along with presentation of invalid information to allow correction of the invalid information received.

To meet the imposed software engineering and implementation requirements a solution must:

1. Reuse design concepts and implementations (i.e., use a modeling approach).
2. Provide Ada code generation capabilities through the use of Ada generics and Ada coding templates, and incorporate testing as part of the templates.
3. Use good software engineering practices (e.g., information hiding, abstraction, separation of concerns).

3. TV Model Description

This chapter describes the Translation and Validation (TV) model as a black box. It describes what the model does (not how it does it) and how to access the capabilities provided by the model.

This model is described because it is used to solve the message translation and validation recurring problem described in Chapter 2. It can also be used to extend the solution or as a building block to other solutions.

This information allows a designer to evaluate the model in the context of the functional requirements for which a solution is being sought.

3.1. Functional Description

The TV model provides two abstractions of some information: a primary representation and a secondary representation. The TV model provides the capability to convert between a primary and a secondary representation. The conversion entails a real-time validation of the conversion. The validation is real-time in the sense that if a problem is found, the conversion process is stopped, and the caller is notified.

The primary representation is a representation of some information upon which the conversion is based. Objects of a primary representation are assumed to always be valid therefore no checking of this representation is needed. The secondary representation is another representation of the same information provided in the primary representation.

The model also supports a diagnostic, non-real-time analysis of the secondary representation. A diagnostic indicator is returned that supports error detection for secondary representations of a message.

Figure 3-1 shows a black box diagram of the TV model. The next section describes the interface shown in the black box in more detail.

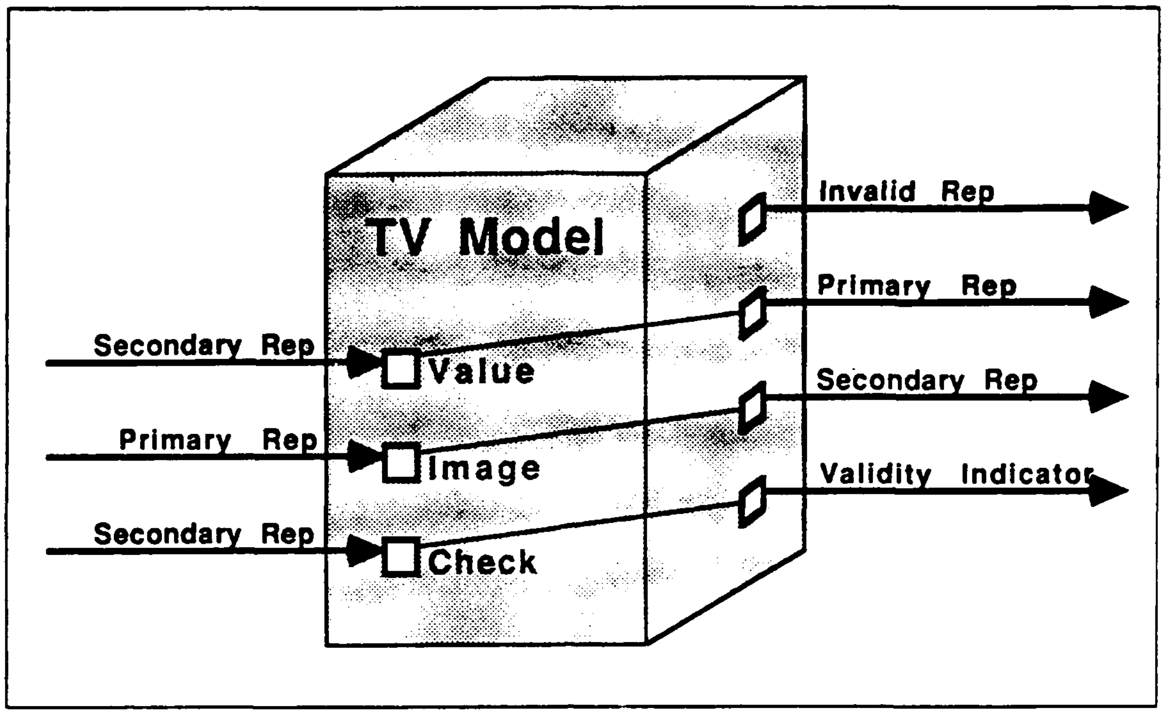

Figure 3-1: TV Model Black Box Diagram

3.2. Interface Description

Figure 3-2 shows the interface provided by the TV model expressed as incomplete Ada PDL.[5] The parts of the PDL that are ***bold, italicized, and boxed*** represent those parts of the model that are adjustable based upon the nature of the primary and secondary representations. The model exports a primary representation (*Primary_Rep*) and a secondary representation (*Secondary_Rep*). The primary representation is the base representation from which a mapping to the secondary representation is defined. This mapping is defined in *The_Map*. The primary representation description (*Primary_Rep_Descr*) describes the format of the primary representation and the secondary representation description (*Secondary_Rep_Descr*) describes the format of the secondary representation.

The *Image* function provides the capability to convert from a primary representation to a secondary representation. Conversely, the *Value* function provides the capability to convert from a secondary representation to a primary representation. If either conversion fails, the *Invalid_Representation* exception is raised. These operations provide real-time validation and conversions.

[5]Ada PDL (Program Design Language) is provided to show that the TV model is more than a concept. The model can be specified and implemented in Ada.

The *Check* function provides the capability to check the secondary representation for validity, based on the legal values specified by the primary representation. The *Check* function returns a validity indicator that provides diagnostic support to indicate the invalid fields. This operation supports non-real-time diagnostic analysis of secondary representations.

3.3. Style Characteristics

The *Image* and *Value* functions are complementary translation and validation operations, and the *Check* function is a diagnostic analyzer. All functions are independent, separate, autonomous, and designed to be re-entrant.

The designer has freedom in selecting the primary and secondary representations but must tailor the model to allow translation and validation to occur. An important part of the tailoring process is specifying a mapping between the two representations. The *Value*, *Image*, and *Check* functions depend upon this mapping to provide their functionality.

A small set of simple concepts are provided by the model. Once these concepts are understood, the functionality that the model provides and the means of accessing the functionality are easily understood.

```
package Secondary_Rep_TV is
    -- Primary Representation Description and Primary Representation type
    Primary_Rep_Descr Note 1
    type Primary_Rep is Note 2

    -- Secondary Representation Description and Secondary Representation type
    Secondary_Rep_Descr Note 3
    Secondary_Rep_Width : constant Integer := Note 3
    type Secondary_Rep is array (1..Secondary_Rep_Width) of Bytes;

    -- Mapping between Primary Representation and Secondary Representations
    The_Map Note 4

    -- Functions for converting between primary and secondary representations
    function Image(Value_In : Primary_Rep) return Secondary_Rep;
    function Value(Image_In : Secondary_Rep) return Primary_Rep;

    -- Function for checking the validity of a secondary representation
    type Validity_Indicator is Note 5
    function Check(Image_In : Secondary_Rep) return Validity_Indicator;
    -- Indicators of a valid or inconsistent secondary representation
    Valid_Secondary_Rep : constant Validity_Indicator := Note 5
    InValid_Secondary_Rep : constant Validity_Indicator := Note 5
    Inconsistent_Secondary_Rep : constant Validity_Indicator := Note 5

    -- Real-time constraint raised by Value and Image functions
    Invalid_Representation : exception;
end Secondary_Rep_TV;
```

NOTES:

These notes describe the items boxed in the PDL. The items should be tailored by the model adapter to meet the specific translation and validation requirements imposed by the two different representations.

Note 1:	*Primary_Rep_Descr* describes the element(s) of the primary representation.
Note 2:	*Primary_Rep* is an Ada type used to declare objects to hold primary representations. The format of the objects is based upon the *Primary_Rep_Descr*.
Note 3:	*Secondary_Rep_Descr* describes the position of the element(s) of the secondary representation. *Secondary_Rep_Width* defines the length of the secondary representation and must be consistent with the *Secondary_Rep_Descr*.
Note 4:	*The_Map* defines the one-to-one mapping between primary representations and secondary representations.
Note 5:	*Validity_Indicator* is a type that describes the validity indicators: *Valid_Secondary_Rep*, *InValid_Secondary_Rep*, and *Inconsistent_Secondary_Rep*.

Figure 3-2: Interface Provided by TV Model (Incomplete Ada PDL)

4. MTV Model Description

This chapter analyzes the message translation and validation recurring problem (described in Chapter 2) with the intent of applying the TV model (described in Chapter 3) to solve the problem. The result is a description of the MTV model.

4.1. MTV Problem Analysis

Examination of the functional requirements of the MTV problem indicate that the following translation and validation relationships must exist in the MTV model:

1. Conversion between external representations (EXR) and internal representations (INR) of a message.
2. Conversion between internal representations (INR) and user representations (USR) of a message.

As a goal, we did not want to tie the detailed designer's specification of the INR description (Ada type) directly to the EXR description that is part of the requirements levied upon the C^3I system. Based on further examination of formats of the predefined EXR descriptions and definitions of INR descriptions proposed by detailed designers, we saw the need to introduce another message representation, the ***universal representation*** (***UNR***).

A UNR is an intermediate representation of the information in a message. It is a fixed-length string containing a symbolic image in a predefined place for each field in the EXR with the ***punctuation*** removed. The symbolic images of character-based fields that are of varying length, null, or optional are expanded to their maximum length and padded with blanks. The symbolic images of bit-based fields are expanded to a length of multiples of one byte (i.e., a bit-based field of four bits is expanded to one byte, and a bit-based field of 12 bits is expanded to two bytes). Also, the symbolic images are ***cut*** (rearranged) to match the INR, if necessary. Figure 4-1 is an example of a UNR of the FooBar message described in Chapter 2.

The introduction of the UNR stems from our desire to eliminate the dependency of the INR description of a message from the format of the corresponding EXR description. Both representations should contain the *same* information, but in their own format. The UNR contains the information as symbolic images as defined by the EXR, but the symbolic images are in the order defined by the INR.

```
"CPPE18314070"
```

Figure 4-1: Example UNR: FooBar Message

Finally, the UNR description is derived from both the EXR description (as shown in Figure 2-2) and the INR description (as shown in Figure 2-6).

Therefore, the MTV model contains three instances of the TV model:

1. **External Representation TV Model** - Conversion between EXRs and UNRs of a message.
2. **Universal Representation TV Model** - Conversion between UNRs and INRs of a message.
3. **User Representation TV Model** - Conversion between USRs and INRs of a message.

Each instance of the TV model is described as a black box, similar to the way the TV model was described. The description first tells what the TV model instances do (but not how they do it). Next, it describes how to access the capabilities provided by the TV model instances. This is done by building upon the incomplete Ada PDL of the TV model.[6] The adjustable parts of the TV model are further refined to handle the primary and secondary representations of the three instances defined.[7] Those parts of the Ada PDL that still need to be tailored to instantiate an implementation of the particular model are ***bold, italicized, and boxed***.

The MTV model is formed by integrating the TV model instances. Figure 4-2 shows how the primary and secondary representations supported by each TV model instance allow the instances to be integrated to meet the MTV requirements described in Chapter 2.

Finally, conversion between EXRs and INRs; EXRs and USRs; and UNRs and USRs are not directly supported because they are not part of the required functionality as described in Chapter 2. Although, the conversions are supported indirectly as a multiple step process.

[6]Ada PDL is provided to show that the TV model is more than a concept. The model can be specified and implemented in Ada.

[7]Because the instances are customized, each begins to use predefined common Ada types whose definitions can be found in the package ***Casting_Common_Types*** (CCT) in Appendix Section C.1.

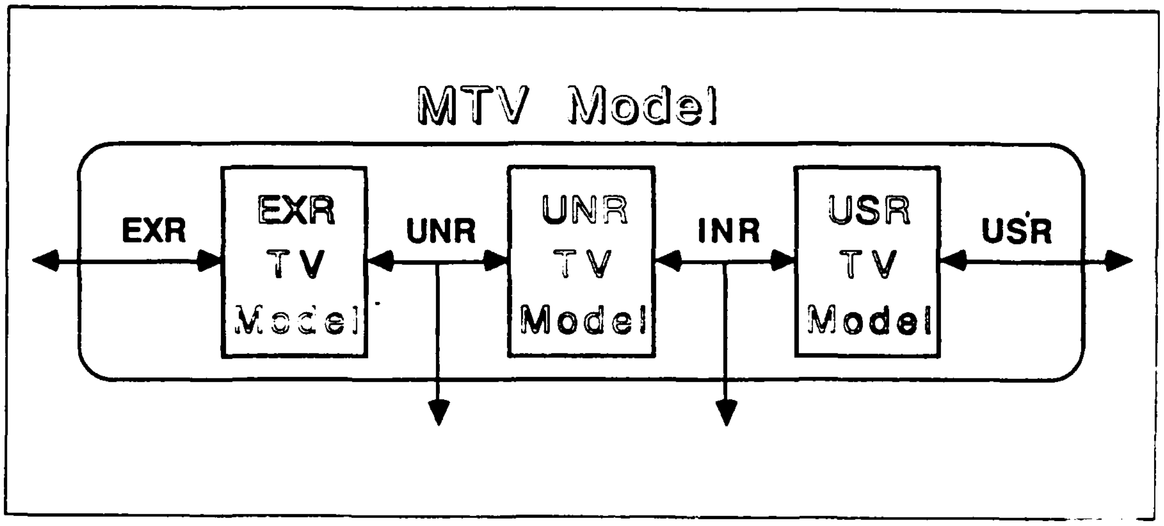

Figure 4-2: MTV Model Block Diagram

4.2. External Representation TV Model

4.2.1. Functional Description

The EXR TV model provides the capability to convert between the EXR of a message (as shown in Figure 2-3) and the UNR of a message (as shown in Figure 4-1). The conversion entails a real-time syntactic check of the message based on the EXR description. For example: if a field or punctuation is missing, or if a field is the wrong length, the conversion process is stopped and the caller is notified. The model also supports a diagnostic, non-real-time syntactic analysis of EXRs. A diagnostic indicator is returned that supports error detection for EXRs of a message.

Figure 4-3 shows a black box diagram of the EXR TV model. The next section describes the interface shown in the black box in more detail.

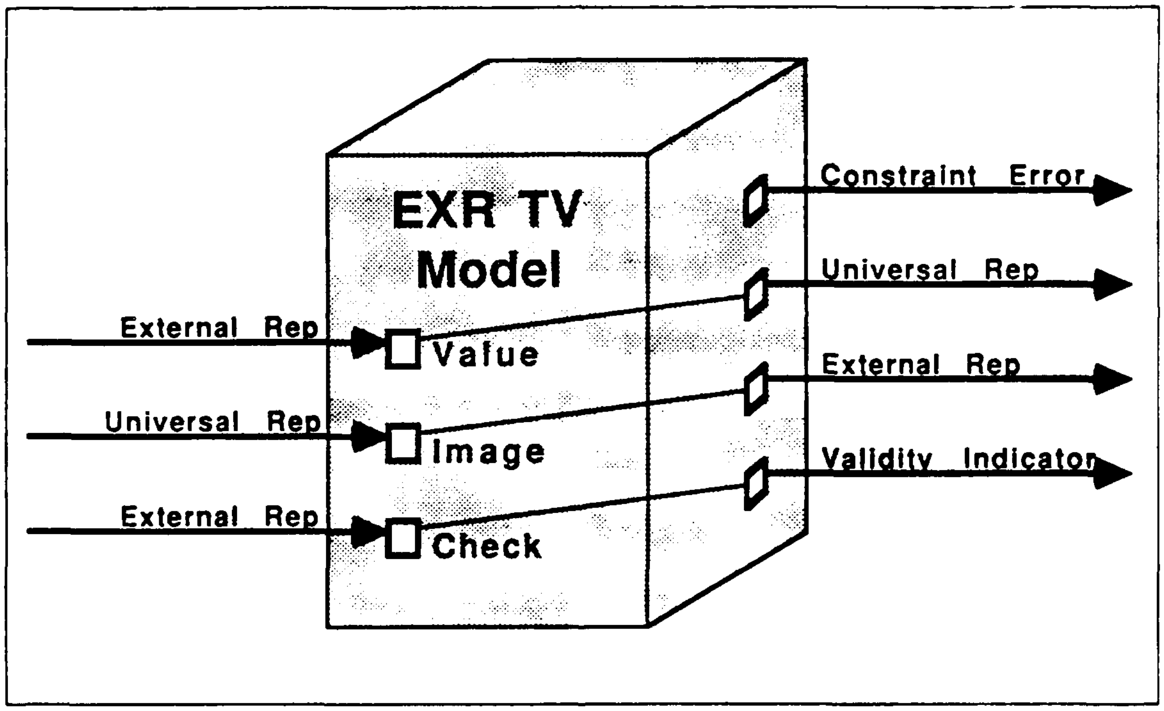

Figure 4-3: EXR TV Model Black Box Diagram

4.2.2. Interface Description

Figure 4-4 shows the interface provided by the EXR TV model expressed in incomplete Ada PDL. The model is represented as an Ada package, *External_Representation_TV*. The package exports the following abstractions.

The *EXR_Descr*, a codification of the textual description of the EXR of a message (as shown in Figure 2-2), is supplied by the detailed designer for each message. Figure 6-3 shows an example of how the textual EXR description of the Foobar message is codified for the model solution described in this document. This is discussed in more detail in Chapter 6, MTV Model Solution Application Description. EXRs are stored using the *EXR* type. UNRs are stored using the *UNR* type.

The *Image* function provides the capability to convert, in real-time, from a UNR of a message to an EXR of a message. Conversely, the *Value* function provides the capability to convert, in real-time, from an EXR of a message to a UNR of a message. If either conversion fails, the *Constraint_Error* exception is raised.

Finally, the *Check* function provides the capability to check an EXR of a message for syntactic correctness based on the EXR description defined in *EXR_Descr*. In the case of an invalid EXR, the *Check* function returns a validity indicator that points to the invalid field.

By definition, the EXR of a message contains fields that are an aggregation of the information in the message. These fields are accessed by obtaining positional information with respect to a particular field from the *EXR_Descr*. The array *Cuts* is also defined by the detailed designer and describes how repetitive information in the EXR of a message should be partitioned to provide a more natural view based on an INR description (as shown in Figure 2-6). Also, by definition, the UNR of a message contains elements that are an aggregation of the symbolic images of each of the fields contained in the EXR. These elements are accessed via a combination of the positional information obtained from the *EXR_Descr* and *Cuts*.

```
with Casting_Common_Types;                        -- renamed as CCT
package External_Representation_TV is
  -- EXTERNAL REPRESENTATION
    -- Maximum width of the EXR and string type to it.
    EXR_Width : constant Integer := Note 1;
    subtype EXR is CCT.Byte_Array (1..EXR_Width);

    -- Describes each field in the message as per the EXR textual description
    type Field_Names is (Note 2);
    EXR_Descr : array (Field_Names) of CCT.Field_Description_Type := Note 3;

  -- UNIVERSAL REPRESENTATION
    -- Width of the UNR and string type to hold it.
    UNR_Width : constant Integer := Note 4;
    type UNR is CCT.Byte_String (1..UNR_Width);

    -- Describes any reorganization of repetitive information
    Cuts : array (Number_Of_Cuts,2) of Field_Names := Note 5;

  -- CONVERSION AND VALIDATION FUNCTIONS
    -- Functions for converting between EXRs and UNRs
    function Image(Value_In : in UNR)  return EXR;
    function Value(Image_In : in EXR)   return UNR;

    -- Function for checking the validity of a EXR
    procedure Check(Image_In           : in EXR
                    Validity_Indicator  : out Boolean,
                    Bad_Position        : out Integer);

  -- Real-time constraint raised by Image and Value functions
    Constraint_Error : exception;
end External_Representation_TV;
```

NOTES:

These notes describe the items boxed in the PDL. The items must be tailored by the detailed designer for each instance of the EXR TV model. There will be one instance for each message.

Note 1:	*EXR_Width* defines the maximum width of the EXR of a particular message.
Note 2:	*Field_Names* is an enumerated type whose literals name the fields as described in the textual EXR description.
Note 3:	*EXR_Descr* is an array indexed by *Field_Names*. Each element of the array describes an individual field as per the textual EXR description. Each element also contains information regarding the positions of each elemental symbolic image in the UNR.
Note 4:	*UNR_Width* defines the width of the UNR of a particular message.
Note 5:	*Cuts* defines the repetitive fields that are to be reordered. Each cut is defined as a starting field and stopping field.

Figure 4-4: EXR TV Model Interface (Incomplete Ada PDL)

4.3. Universal Representation TV Model

4.3.1. Functional Description

The UNR TV model provides the capability to convert between the UNR of a message (as shown in Figure 4-1) and the INR of a message (as shown in Figure 2-7). The conversion entails a real-time validation of the conversion that includes syntactic analysis of the range of values possible for the elements and checking of any inter-element dependencies. If a problem is found, the conversion process is stopped, and the caller is notified. The model also supports a diagnostic, non-real-time syntactic analysis of UNRs. A diagnostic indicator is returned that supports error detection for UNRs of a message.

Figure 4-5 shows a black box diagram of the UNR TV model. The next section describes the interface shown in the black box in more detail.

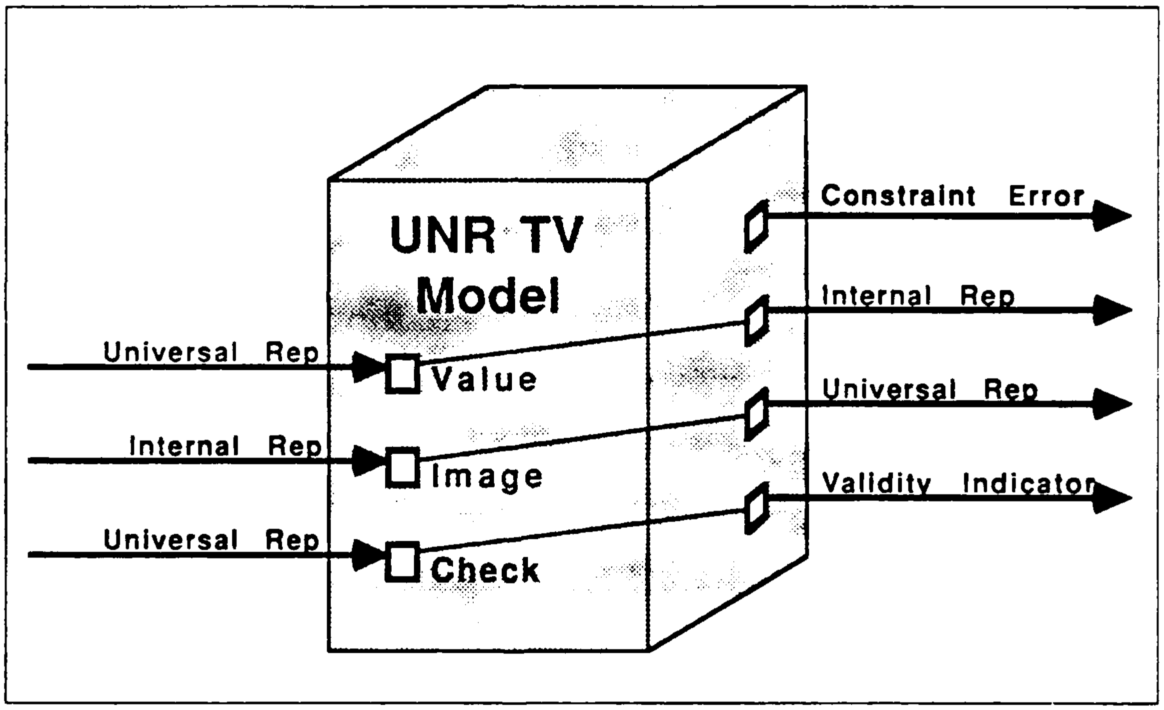

Figure 4-5: UNR TV Model Black Box Diagram

4.3.2. Interface Description

Figure 4-6 shows the interface provided by the UNR TV model expressed in incomplete Ada PDL. The model is represented as an Ada package, *Universal_Representation_TV*. The package exports the following abstractions.

The *INR* type is specified by the detailed designer and is the INR description for a message (see the FooBar_Message_Type in Figure 2-6). INRs are stored using the *INR* type. UNRs are stored using the *UNR* type.

The *Image* function provides the capability to convert, in real-time, from an INR of a message to a UNR of a message. Conversely, the *Value* function provides the capability to convert, in real-time, from a UNR of a message to an INR of a message. If either conversion fails, the *Constraint_Error* exception is raised.

The *Check* function provides the capability to check, in non-real-time, the UNR of a message for syntactic correctness based on the legal values specified by the INR description and any inter-element consistency checks based on the textual EXR description.

In the case of an invalid indication, the *Check* function returns indicators that point to the invalid fields. *Valid_UNR* and *Inconsistent_UNR* are indicators that specify the validity of the UNR of a message as analyzed by the *Check* function. If the *Check* function returns something other than the above two indicators, it is assumed to be an *Invalid_UNR*.

By definition, the INR of a message contains elements that are an aggregation of the information in the message. These elements are accessed by using named association on the INR (an Ada composite type). Also, by definition, the UNR of a message contains elements that are an aggregation of the symbolic images of each field contained in the EXR. These elements are accessed via *UNR_Descr*. This array defines the positions in the UNR of each element's symbolic image. This abstraction also provides access to the results of the *Check* function that allows an application to determine the validity of individual elements of a UNR.

```
with Casting_Common_Types;                          -- renamed as CCT
package Universal_Representation_TV is

  -- INTERNAL REPRESENTATION
    type INR is Note 1;

  -- UNIVERSAL REPRESENTATION
    -- Width of the UNR and string type to hold it.
    UNR_Width : constant Integer := Note 2;
    type UNR is CCT.Byte_Array (1..UNR_Width);

    -- Elemental symbolic image positional information.
    type Element_Names is (Note 3);
    UNR_Descr is array (Element_Names) of CCT.Position_Type := Note 4;

  -- CONVERSION AND VALIDATION FUNCTIONS
    -- Functions for converting between UNRs and INRs
    function Image(Value_In : in INR) return UNR;
    function Value(Image_In : in UNR) return INR;

    -- Function for checking the validity of a UNR
    subtype Validity_Indicator is UNR;
    function Check(Image_In : in UNR) return Validity_Indicator;
    -- Indicators of a valid, or inconsistent UNR
    Valid_UNR : constant Validity_Indicator := Note 5;
    Inconsistent_UNR : constant Validity_Indicator := Note 5;

  -- Real-time constraint raised by Image and Value functions
    Constraint_Error : exception;

end Universal_Representation_TV;
```

NOTES:

These notes describe the items boxed in the PDL. The items must be tailored by the detailed designer for each instance of the UNR TV model.

Note 1:	*INR* is an Ada type that defines the INR description for a particular message. It also is used to define objects that hold INRs of the information in a particular message.
Note 2:	*UNR_Width* defines the width of the UNR for a particular message.
Note 3:	*Element_Names* is an enumerated type whose literals name the elements described in *INR* and *UNR_Descr*.
Note 4:	*UNR_Descr* is an array indexed by *Element_Names*. Each element of the array describes the position of an elemental symbolic image in the UNR.
Note 5:	The validity indicators *Valid_UNR* and *Inconsistent_UNR* are defined as the aggregation of the validity indicators of the elemental symbolic images. Elemental symbolic image validity is defined by the constants *CCT.Valid_Symbolic_Image*, *CCT.Invalid_Symbolic_Image*, and *CCT.Inconsistent_Symbolic_Image*.

Figure 4-6: UNR TV Model Interface (Incomplete Ada PDL)

4.4. User Representation TV Model

4.4.1. Functional Description

The USR TV model provides the capability to convert between the USR of a message (as shown in Figure 2-8) and the INR of a message (as shown in Figure 2-7). The conversion entails a real-time validation of the conversion that includes syntactic analysis of the range of values possible for the elements and checking of any inter-element dependencies. If a problem is found, the conversion process is stopped, and the caller is notified. The model also supports a diagnostic, non-real-time syntactic analysis of USRs. A diagnostic indicator is returned that supports error detection for USRs of a message.

Figure 4-7 shows a black box diagram of the USR TV model. The next section describes the interface shown in the black box in more detail.

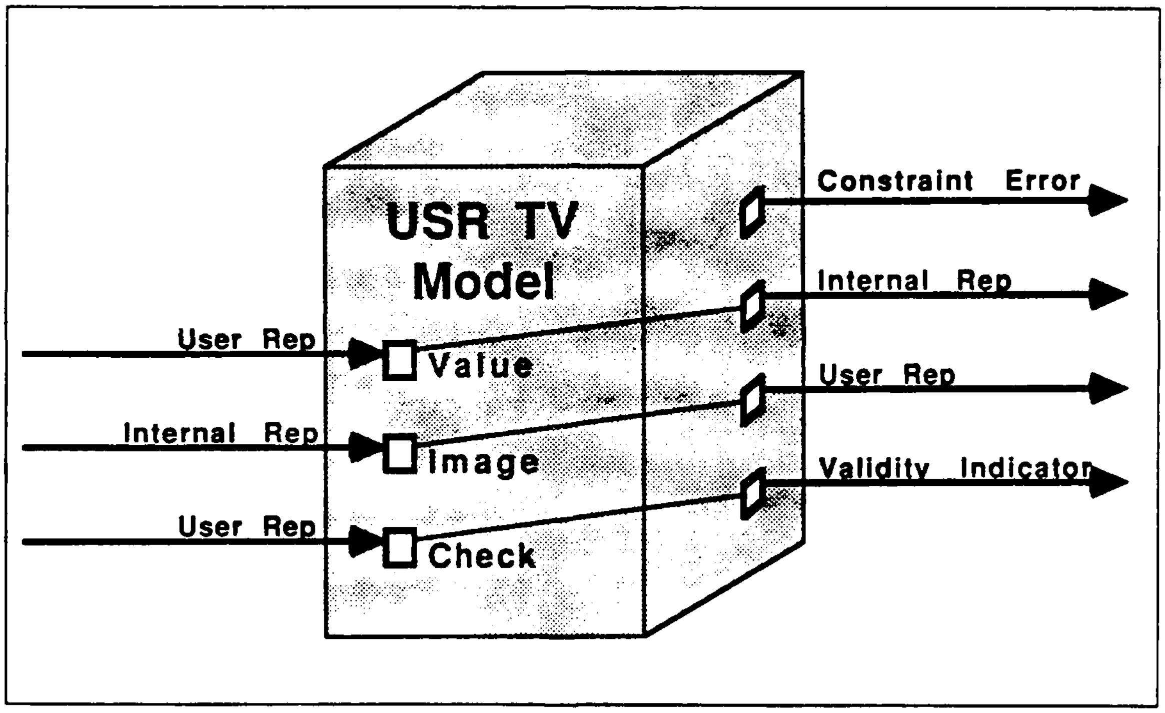

Figure 4-7: USR TV Model Black Box Diagram

4.4.2. Interface Description

Figure 4-8 shows the interface provided by the USR TV model expressed in incomplete Ada PDL. The model is represented as an Ada package, *User_Representation_TV*. The package exports the following abstractions.

The *INR* type is specified by the detailed designer and is the INR description for a message (see the FooBar_Message_Type in Figure 2-6). INRs are stored using the *INR* type. USRs are stored using the *USR* type.

The *Image* function provides the capability to convert, in real-time, from an INR of a message to a USR of a message. Conversely, the *Value* function provides the capability to convert, in real-time, from a USR of a message to an INR of a message. If either conversion fails, the *Constraint_Error* exception is raised.

The *Check* function provides the capability to check, in non-real-time, the USR of a message for syntactic correctness based on the legal values specified by the INR description and any inter-element consistency checks based on the textual EXR description.

In the case of an invalid indication, the *Check* function returns indicators that point to the invalid fields. *Valid_USR* and *Inconsistent_USR* are indicators that specify the validity of the USR of a message as analyzed by the *Check* function. If the *Check* function returns something other than the above two indicators, it is assumed to be an *Invalid_USR*.

By definition, the INR of a message contains elements that are an aggregation of the information in the message. These elements are accessed by using named association on the INR (an Ada composite type). Also, by definition, the USR of a message contains elements that are an aggregation of the natural images of each element of the INR. These elements are accessed via *USR_Descr*. This array defines the positions in the USR of each element's natural image. This abstraction also provides access to the results of the *Check* function that allows an application to determine the validity of individual elements of a USR.

```
with Casting_Common_Types;                          -- renamed as CCT
package User_Representation_TV is

  -- INTERNAL REPRESENTATION
    type INR is Note 1;

  -- USER REPRESENTATION
    -- Width of the USR and string type to hold it.
    USR_Width : constant Integer := Note 2;
    type USR is String (1..USR_Width);

    -- Elemental natural image positional information.
    type Element_Names is (Note 3);
    USR_Descr is array (Element_Names) of CCT.Position_Type := Note 4;

  -- CONVERSION AND VALIDATION FUNCTIONS
    -- Functions for converting between USRs and INRs
    furction Image(Value_In : in INR) return USR;
    function Value(Image_In : in USR) return INR;

    -- Function for checking the validity of a USR
    subtype Validity_Indicator is USR;
    function Check(Image_In : in USR) return Validity_Indicator;
    -- Indicators of a valid, or inconsistent USR
    Valid_USR : constant Validity_Indicator := Note 5;
    Inconsistent_USR : constant Validity_Indicator := Note 5;

  -- Real-time constraint raised by Image and Value functions
    Constraint_Error : exception;

end User_Representation_TV;
```

NOTES:

These notes describe the items boxed in the PDL. The items must be tailored by the detailed designer for each instance of USR TV model.

Note 1:	*INR* is an Ada type that defines the INR description for a particular message. It also is used to define objects that hold INRs of the information in a particular message.
Note 2:	*USR_Width* defines the width of the USR for a particular message.
Note 3:	*Element_Names* is an enumerated type whose literals name the elements described in *INR* and *USR_Descr*.
Note 4:	*USR_Descr* is an array indexed by *Element_Names*. Each element of the array describes the position of a elemental natural image in the USR.
Note 5:	The validity indicators *Valid_USR* and *Inconsistent_USR* are defined as the aggregation of the validity indicators of the elemental natural images. Elemental natural image validity is defined by the constants *CCT.Valid_Natural_Image*, *CCT.Invalid_Natural_Image*, and *CCT.Inconsistent_Natural_Image*.

Figure 4-8: USR TV Model Interface (Incomplete Ada PDL)

4.5. Typecaster Model

Upon examination of the three instances of the TV model described in the previous sections and the message representations they deal with, one can see how the pieces fit together to form the MTV model. Figure 4-2 shows the three instances of the TV model, the message representations each part supports, and the integration of the instances to form the MTV model.

Based on our knowledge of the Ada language and results from prototyping, we chose to combine the UNR TV model and the USR TV model. This combination is possible because both models perform conversions on INRs. The rationale for this decision is summarized in the following points:

- If the models were implemented separately, one of the following would have to be done:
    1. The existing Ada packages for both the UNR TV model and the USR TV model would each have to encapsulate the INR (i.e., Ada type). This duplication of Ada types would not allow us to fully utilize the strong type checking provided by the Ada runtime when common types are used. Reliance on common types and the Ada runtime insures consistent representations of the information. The duplication of Ada types also causes a configuration/version management problem.
    2. Ada packages would have to be created whose sole purpose would be to hold the INR (i.e., Ada type). The Ada packages for the UNR TV model and the USR TV model would "with" the packages, each holding an Ada type declaration. This solves the problem described in the previous point, but the additional packages for holding an Ada type and the added package layering are not necessary and not aesthetically pleasing.
- It seemed natural for the INR to be encapsulated and be the basis of a model. This model would be capable of producing two images of the information captured in the INR: the UNR to support the EXR of the information and the USR to support a user interface.

We call the model resulting from combining the UNR TV model and the USR TV model a ***Typecaster***. Figure 4-9 shows the results of the combination in the form of a black box diagram. Figure 4-10 shows the Typecaster model interface expressed in incomplete Ada PDL. Chapter 7 describes the Typecaster model in more detail.

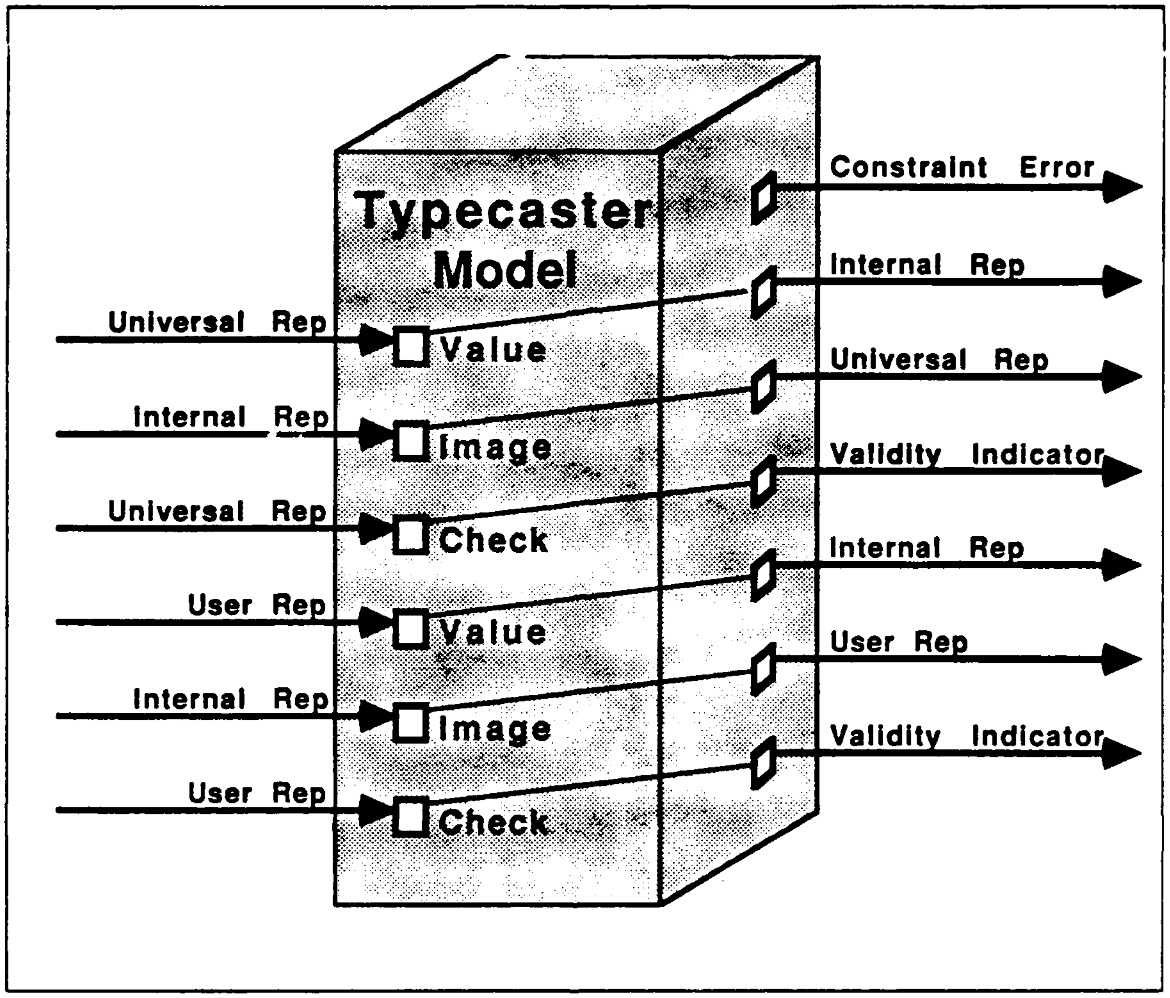

Figure 4-9: Typecaster Model Black Box Diagram

4.6. The MTV Model Description

The MTV model is based on the EXR TV model and the Typecaster model. The EXR TV model deals with the abstraction of an EXR of a message, while the Typecasting model deals with the abstraction of an INR of a message. This separation allows for an independent analysis of the two representations as long as the interface between the two (UNR) is agreed upon.

The remaining chapters of this document describe the model solution, how to apply it given a set of messages, and how to adapt it given that the model solution doesn't meet a given set of requirements.

```
with Casting_Common_Types;                         -- renamed as CCT
package Typecaster is
  -- INTERNAL REPRESENTATION
    type INR is Figure 4-6 Note 1;
    type Element_Names is (Figure 4-6 Note 3);

  -- UNIVERSAL REPRESENTATION
    -- Width of the UNR and string type to hold it.
    UNR_Width : constant Integer := Figure 4-6 Note 2;
    subtype UNR is CCT.Byte_Array (1..UNR_Width);
    -- Elemental symbolic image positional information.
    UNR_Descr is array (Element_Names) of CCT.Position_Type := Figure 4-6 Note 4;

    -- Functions for converting between UNRs and INRs
    function Image(Value_In : in INR) return UNR;
    function Value(Image_In : in UNR) return INR;

    -- Function for checking the validity of a UNR
    subtype UNR_Validity_Indicator is UNR;
    function Check(Image_In : in UNR) return UNR_Validity_Indicator;
    -- Indicators of a valid, or inconsistent UNR
    Valid_UNR : constant UNR_Validity_Indicator := Figure 4-6 Note 5;
    Inconsistent_UNR : constant UNR_Validity_Indicator := Figure 4-6 Note 5;

  -- USER REPRESENTATION
    -- Width of the USR and string type to hold it.
    USR_Width : constant Integer := Figure 4-8 Note 2;
    subtype USR is String (1..USR_Width);
    -- Elemental natural image positional information.
    USR_Descr is array (Element_Names) of CCT.Position_Type := Figure 4-8 Note 4;

    -- Functions for converting between USRs and INRs
    function Image(Value_In : in INR) return USR;
    function Value(Image_In : in USR) return INR;

    -- Function for checking the validity of a USR
    subtype USR_Validity_Indicator is USR;
    function Check(Image_In : in USR) return USR_Validity_Indicator;
    -- Indicators of a valid, or inconsistent USR
    Valid_USR : constant USR_Validity_Indicator := Figure 4-8 Note 5;
    Inconsistent_USR : constant USR_Validity_Indicator := Figure 4-8 Note 5;

  -- Real-time constraint raised by Image and Value functions
    Constraint_Error : exception;
end Typecaster;
```

Figure 4-10: Typecaster Model Interface Description (Incomplete Ada PDL)

5. MTV Model Solution Overview

This chapter briefly presents the parts of the MTV model solution and how to apply them by describing the categories of parts available and outlining the steps involved to use the parts to generate code for translating and validating a message. The chapter also describes what software architecture will result from the application of the MTV model solution and provides some performance characteristics of the MTV model solution.

This chapter is a road map to the rest of the document.

5.1. Fundamental Concepts

Before proceeding with an overview of the MTV model solution, a few concepts must be defined.

5.1.1. Templates

The MTV model solution consists of a set of utilities and Ada coding ***templates***. The Ada coding templates are provided to guarantee identical instantiations of the MTV model solution and enhance the probability for reuse. Figure 5-1 shows an example of a template.

A template is a file containing an incomplete (i.e., not fully defined, thus not compilable) Ada package specification, package body, and test procedure. The package specification captures the functional interface defined by a model, and the package body implements the functionality defined by the same model. This is the ***model solution***. The test procedure tests the functionality provided by a model solution. All templates also contain a header that identifies the template by name and version number, and lists the engineering points to be supplied by the user of the template. The instructions for instantiating individual templates are found in Appendix B.

The incomplete Ada code in the file contains ***placeholders***. Placeholders are embedded in the Ada code where a general, replaceable piece of information is required to make the template unique. The templates are instantiated by making a copy of the file and performing editor substitutions on the placeholders. There are two types of placeholders:

1. The first is of the form *<Type>* or *<First>*, i.e., a phrase enclosed in brackets. The entire phrase (including the brackets) must be replaced. For example, *<Type>_Type* becomes *Hour_Type* for all instances of *<Type>_Type* in a file when *<Type>* is replaced by "*Hour*".
2. The second form is the double question mark, *??*. This form means that some special action must be taken by the detailed designers, such as supplying a function body, supplying test cases, or removing some lines from the template, e.g., the instructions at the beginning of each template.

Replacement of the placeholders affects the package specification, the package body, and the test procedure. Once the package specification and package body are compiled and linked, the test procedure allows for canned and/or interactive testing of the instantiated template. Figure 5-2 shows an instance of the example template shown in Figure 5-1.

```
--|*******************************************************************
--| Template Used : Integer_Template, Text-Based, Version 1.0
--|
--| Documentation :      CMU/SEI-89-TR-12
--|    "A Model Solution for C3I Message Translation and Validation"
--|
--| Engineering Points :
--|     Package                    : <Type>_Typecaster
--|     Integer type to be cast    : <Type>_Type
--|     First in integer range     : <First>
--|     Last in integer range      : <Last>
--|     Is symbolic image signed   : <Is-signed>
--|     Test procedure             : <Type>_Typecaster_Test
--|
--|*******************************************************************
with Integer_Typecaster;
package <Type>_Typecaster is

    subtype <Type>_Type is integer range <First>..<Last>;

    package <Type>_Tc is new Integer_Typecaster
        (Type_To_Be_Cast => <Type>_Type,
         Is_signed => <Is-signed>);

end <Type>_Typecaster;


with <Type>_Typecaster;
procedure <Type>_Typecaster_test is

??Enter test cases
    Test_Cases : array (1..??) of Test_Record := (??);

begin

end <Type>_Typecaster_Test ;
```

Figure 5-1: Template Example

```
--|*********************************************************************
--| Template Used : Integer_Template, Text-Based, Version 1.0
--|
--| Documentation :      CMU/SEI-89-TR-12
--|    "A Model Solution for C3I Message Translation and Validation"
--|
--| Engineering Points :
--|     Package                     : Hour_Typecaster
--|     Integer type to be cast     : Hour_Type
--|     First in integer range      : 0
--|     Last in integer range       : 23
--|     Is symbolic image signed    : False
--|     Test procedure              : Hour_Typecaster_Test
--|
--|*********************************************************************
with Integer_Typecaster;
package Hour_Typecaster is

    subtype Hour_Type is integer range 0..23;

    package Hour_Tc is new Integer_Typecaster
        (Type_To_Be_Cast => Hour_Type,
         Is_signed => False);

end Hour_Typecaster;


with Hour_Typecaster;
procedure Hour_Typecaster_test is

    Test_Cases : array (1..2) of Test_Record := ("04", "23");

begin

end Hour_Typecaster_Test ;
```

Figure 5-2: Instantiation of Template Example

5.1.2. Validity Indicators

The Typecasters' notion of validity indicators is that each individual primitive element, in both UNR and USR form, has associated with it a character string indicating whether the element is valid or invalid (i.e., "V " or "I "). The length of the validity indicator string is equal to the length of the primitive UNR or USR. Indication of message validity is then achieved by concatenating the elemental validity indicators. The validity indicators for elements can then be accessed the same way the elements of the UNR and USR are accessed, via the UNR description and USR description that define the positions of the elements. Inter-element inconsistencies are indicated by the character string "? ".

Figure 5-3 shows valid, invalid, and inconsistent representations for UNRs and USRs of a FooBar message. It also shows the validity indicator associated with each.

```
Valid UNR:               "CPPE18314070"
Validity Indicator:      "V  VV  V V V"

Invalid UNR              "LEPE60014075"
Validity Indicator       "I  VI  V V I"

Inconsistent UNR         "CPPE18314071"
Validity Indicator       "V  V?       "


Valid USR                "        Peterson_AFB East 183 14  7    Operational"
Validity Indicator       "V                    V    V   V  V  V              "

Invalid USR              "           Elgin_AFB East 600 14  7SemiOperational"
Validity Indicator       "I                    V    I   V  V  I              "

Inconsistent USR         "        Peterson_AFB East 183 14  7Non_Operational"
Validity Indicator       "V                    V    ?                        "
```

Figure 5-3: Valid FooBar Message

5.1.3. Terminology

Because of the transition from describing the design to describing the implementation, it is important to note other terms used to describe the implementation that are synonymous with terms used to describe the design.

- *ICD Formatted Message* is used interchangeably with EXR.
- *Symbolic Image* is used interchangeably with UNR.
- *Natural Image* is used interchangeably with USR.
- *Ada Value* is used interchangeably with INR.

5.2. List of Parts

Figure 5-4 lists the contents of the MTV model solution. The figure lists the source files by category, what each source file contains, where to find hardcopy listings of the source files in this document, and finally, if the source file contains a code template, where to find directions for using (instantiating) the template.[8] All components listed are necessary to provide the functionality of the MTV model described in Chapter 4. The components are grouped into six categories. These categories are listed and briefly described below and are elaborated upon in Chapter 7.

1. **Casting Common Types.** An Ada package that contains global type declarations and constants that are used by the components of the MTV model solution. This package is analogous to the Ada package *Standard*. This package must be compiled into the Ada library for use by other portions of the MTV model solution. This part of the solution is described in Section 7.1.1, and the code is presented in Appendix Section C.1.
2. **Discrete Typecaster Generics.** Ada generic packages that are the foundation of the Typecaster model solution. These must be compiled into the Ada library for use by other portions of the MTV model solution. This part of the solution is described in Section 7.2.1, and the code is presented in Appendix Section C.2.
3. **Discrete Typecaster Templates.** Ada coding templates that are the building blocks of the Typecaster model solution. The templates provide the capability to perform typecasting on Ada discrete types. Instances of these templates are layered upon the discrete typecaster generics. This part of the solution is described in Section 7.2.2, and the code template is presented in Appendix Section C.3.
4. **Composite Typecaster Templates.** Ada coding templates that are the building blocks of the Typecaster model solution. The templates provide the capability to perform typecasting on Ada composite types. Instances of these are layered upon instances of both discrete and other composite typecasters. This part of the solution is described in Section 7.2.3, and the code template is presented in Appendix Section C.4.
5. **ICD Utilities.**[9] Ada packages that are the foundation of the EXR TV model solution. These must be compiled into the Ada library for use by other portions of the MTV model solution. This part of the solution is described in Sections 7.3.2 and 7.3.1, and the code is presented in Appendix Sections C.6.1 and C.6.2.
6. **External Representation TV Template.** An Ada coding template that is part of the EXR TV model solution. It provides the capability to convert between EXRs and UNRs. Instances of these are layered upon the ICD Utilities. This part of the solution is described in Section 7.3.3, and the code template is presented in Appendix Section C.5.

[8]If the file does not contain a template (i.e., it contains utilities), the column will be specified as N/A (Not Applicable).

[9]ICD stands for Interface Control Document. For the Granite Sentry Program, this document contained the EXR descriptions for the Granite Sentry message set. The acronym is used in the solution because the solution was developed based mainly on information obtained from the Granite Sentry Program.

Category & *FILE NAME*	Contents	Code Appendix Section	Temp Use Appendix Section
Casting Common Types			
CCT_.ADA	package spec	C.1	N/A
Discrete Typecaster Generics			
INTEGER_TYPECASTER_.ADA	generic package spec	C.2.1	N/A
INTEGER_TYPECASTER.ADA	generic package body	C.2.1	N/A
MATH_ON_INTEGER_TYPECASTER_.ADA	generic package spec	C.2.3	N/A
MATH_ON_INTEGER_TYPECASTER.ADA	generic package body	C.2.3	N/A
INTEGER_BIT_TYPECASTER_.ADA	generic package spec	C.2.2	N/A
INTEGER_BIT_TYPECASTER.ADA	generic package body	C.2.2	N/A
ENUMERATION_TYPECASTER_.ADA	generic package spec	C.2.4	N/A
ENUMERATION_TYPECASTER.ADA	generic package body	C.2.4	N/A
MATH_ON_ENUMERATION_TYPECASTER_.ADA	generic package spec	C.2.6	N/A
MATH_ON_ENUMERATION_TYPECASTER.ADA	generic package body	C.2.6	N/A
ENUMERATION_BIT_TYPECASTER_.ADA	generic package spec	C.2.5	N/A
ENUMERATION_BIT_TYPECASTER.ADA	generic package body	C.2.5	N/A
STRING_MAP_TYPECASTER_.ADA	generic package spec	C.2.7	N/A
STRING_MAP_TYPECASTER.ADA	generic package body	C.2.7	N/A
Discrete Typecaster Templates			
INTEGER_TEMPLATE_.ADA	template package spec & test proc	C.3.1	B.3.1
MATH_ON_INTEGER_TEMPLATE_.ADA	template package spec & test proc	C.3.3	B.3.3
INTEGER_BIT_TEMPLATE_.ADA	template package spec & test proc	C.3.2	B.3.2
ENUMERATION_TEMPLATE_.ADA	template package spec & test proc	C.3.4	B.3.4
MATH_ON_ENUMERATION_TEMPLATE_.ADA	template package spec & test proc	C.3.6	B.3.6
ENUMERATION_BIT_TEMPLATE_.ADA	template package spec & test proc	C.3.5	B.3.5
STRING_MAP_TEMPLATE_.ADA	template package spec & test proc	C.3.7	B.3.7
Composite Typecaster Templates			
RECORD_TEMPLATE.ADA	template package spec, body, & test proc	C.4.1	B.4.3
PRIVATE_RECORD_TEMPLATE.ADA	template package spec, body, & test proc	C.4.2	B.4.4
ARRAY_TEMPLATE.ADA	template package spec, body, & test proc	C.4.3	B.4.1
PRIVATE_ARRAY_TEMPLATE.ADA	template package spec, body, & test proc	C.4.4	B.4.2
WRAPPER_TEMPLATE.ADA	template package spec, body, & test proc	C.4.5	B.4.5
ICD Utilities			
FIELD_UTILITIES_.ADA	package spec	C.6.2	N/A
FIELD_UTILITIES.ADA	package body	C.6.2	N/A
ICD_UTILITIES_.ADA	generic package spec	C.6.1	N/A
ICD_UTILITIES.ADA	generic package body	C.6.1	N/A
External Representation TV Template			
MSG_ICD_TEMPLATE.ADA	template package spec, body, & test proc	C.5	B.5.1

Figure 5-4: MTV Model Solution Contents

5.3. Building Plan

The following are the steps involved in applying the MTV model solution to a set of messages that need to be translated and validated. These steps are elaborated in Chapter 6.

1. **Compile Foundation Utilities.** The utilities that form the foundation of the solution need to be compiled. These include the components in the following categories: Casting Common Types, Discrete Typecaster Generics, and ICD Utilities. Section 6.1 describes this step in more detail.

2. **Analyze Message.** The EXR description for a message needs to be analyzed to determine how to:

   a. Codify the EXR description using the Msg_ICD template. The EXR description needs to be coded in a grammar that describes the characteristics of each field.

   b. Define the INR description based on the information provided by the EXR description. The Ada types for each field need to be defined.

   Section 6.2 describes this step in more detail.

3. **Instantiate MTV Model Solution.** The building blocks (templates) of the MTV model solution are used to create an instance of the model solution for a message.

   a. **Identify and Build the Discrete Typecasters.** The discrete typecasters needed to translate and validate discrete elements of a message are identified based on Step 2. Check to see if any instances of them already exist; some may have been created for other messages. Generate the discrete typecasters that do not exist using the appropriate discrete typecaster templates. Run the generated test routines to check the discrete typecasters. Section 6.3.1 describes how and when to apply the discrete typecaster templates. Appendix Section B.3 provides the details for applying individual discrete typecaster templates.

   b. **Identify and Build Composite Typecasters.** The composite typecasters needed to group discrete and composite elements of the message are identified based on Step 2. Check to see if any of them already exist; some may have been created for other messages. Generate the composite typecasters that do not exist using the appropriate composite typecaster templates. Run the generated test routines to check the composite typecasters. Section 6.3.2 describes how and when to apply the composite typecaster templates. Appendix Section B.4 provides the details for applying individual composite typecaster templates.

   c. **Build the External Representation TV.** The EXR TV for the message is generated using the MSG_ICD template. The information entered, the field description array, and the cut array are critical for the proper conversion between the UNR and the EXR. Run the generated test routine to check the instance of the EXR TV model solution for the message. Section 6.3.3 describes how and when to apply the MSG_ICD template. Appendix Section B.5 provides the details for applying the MSG_ICD template.

   d. **Tie Together the Two Model Solutions.** The instances of the Typecaster model solution and EXR TV model solution for the message must be tied together. The communication interface between the two model solutions is the UNR. Section 6.3.4 describes this step in more detail.

   Section 6.3 describes all these steps in more detail.

5.4. Architectural View

Figure 5-5 shows the general software architecture that results when the MTV model solution is applied. The software architecture is shown as Ada packages and the dependencies among them.[10]

When the EXR TV model solution is instantiated for a set of messages, the resulting architectural components are instances of the Msg_ICD template, one per message, all of which depend upon the ***ICD Utilities*** packages.

The Typecaster portion of the software architecture is based upon the structure of the INR description (i.e., Ada type). When the Typecaster 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 type used to describe the INR description of the message. The Typecaster architecture is therefore hierarchical in nature. The discrete typecasters are dependent upon the discrete typecaster generics. The composite typecasters are dependent upon both discrete typecaster instances and composite typecaster instances.

The software architecture for an instance of the MTV model (specifically for the FooBar message described in Chapter 2) is shown in Figure 6-13. Chapter 7 discusses the software architecture in more detail.

[10]Remember that the MTV model solution is comprised of the EXR TV model solution and the Typecaster model solution.

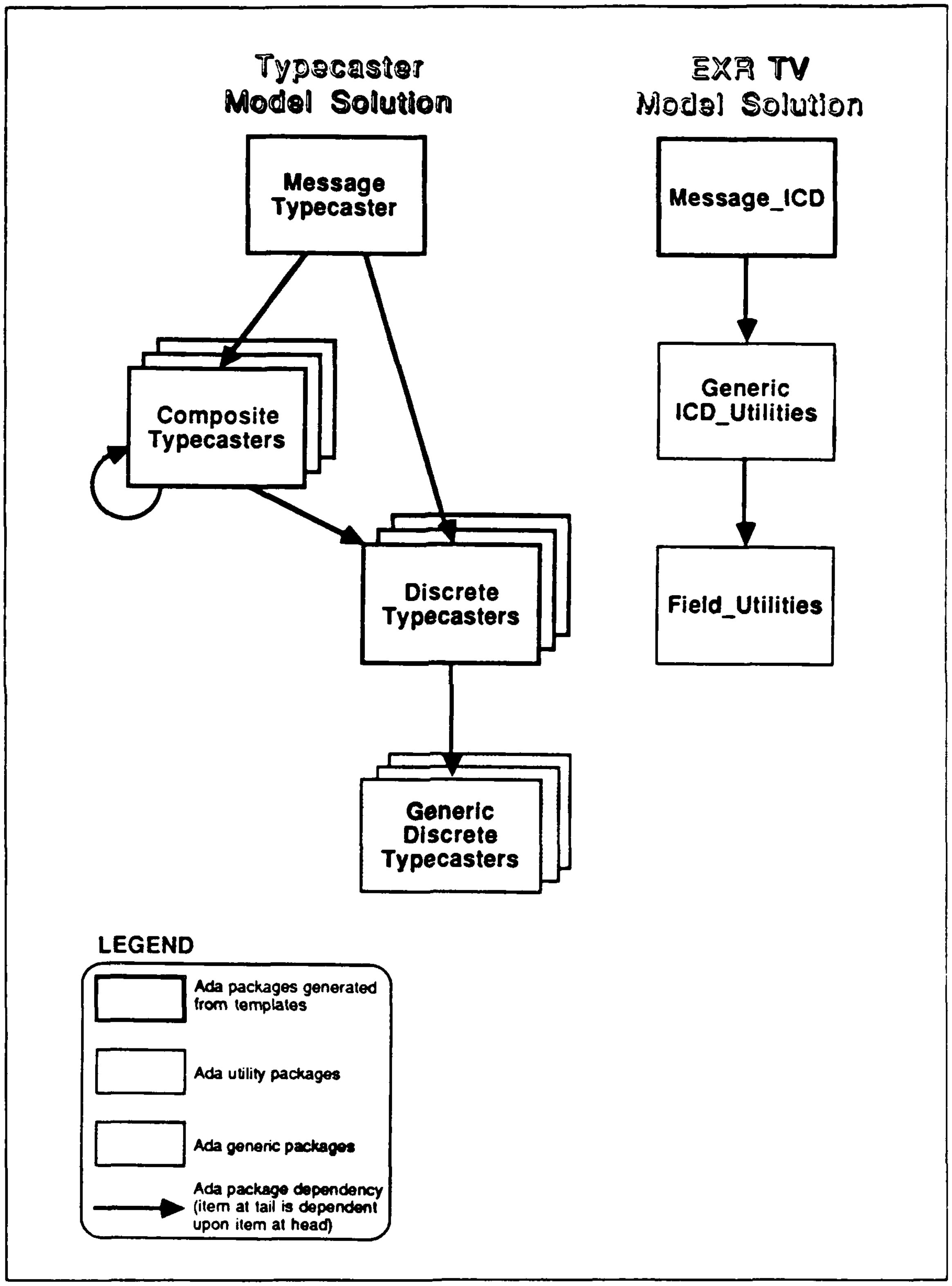

Figure 5-5: MTV Model Solution Architecture

5.5. Performance Characteristics

Timing performance characteristics of the MTV model solution were measured using the FooBar message example shown throughout this document.

Timing measurements were made on a MicroVAX II with 16 megabytes of memory running Version 5.1 of the VAX/VMS operating system. The code was compiled under VAX Ada Version 1.5. All of the code was optimized at both compilation and link time.

5.5.1. Discrete Typecaster Measurement

The discrete typecasters were measured in the following manner. All functions exported by the discrete typecaster (value, image, and check, for both natural and symbolic images) were addressed individually. Each was called and measured 10,000 times for each discrete value in the range of valid discrete values to take care of the coarseness of the clock. A running total was kept for each function exported by the discrete typecaster, and averages were derived from the total times. It is important to note that the times measure the loop as well as the function call. The amount added to the total should be negligible.

Performance measurements for discrete typecasters are shown in Figure 5-6. The discrete typecaster templates (and thus the discrete typecaster generics) from which the discrete typecaster instances were generated are specified in parenthesis.

5.5.2. Composite Typecaster Measurement

The composite typecasters were measured in the following manner. All functions exported by the composite typecaster (value, image, and check, for both natural and symbolic images) were addressed individually. Each was called and measured 10,000 times for test cases that addressed the upper and lower bounds of the discretes they grouped. Again, the calls were performed 10,000 times to take care of the coarseness of the clock. A running total was kept for each function exported by the composite typecaster, and averages were derived from the total times.

Performance measurements for composite typecasters are shown in Figure 5-7. The composite typecaster templates from which the composite typecaster instances were generated are implicit in the names of the typecasters.

5.5.3. EXR Translation and Validation Measurement

The EXR Translation and Validation was measured in a fashion similar to the composite typecasters.

Performance measurements for EXR Translation and Validation are shown in Figure 5-8.

Discrete Typecaster Performance Summary
(Time in milli-seconds)

	INR <--> UNR			INR <--> USR		
Discrete Typecaster	**Image**	**Value**	**Check**	**Image**	**Value**	**Check**
Hour_Typecaster (Integer TCT)	0.93	0.48	0.63	0.59	0.22	0.31
Minute_Typecaster (Integer TCT)	0.93	0.47	0.63	0.59	0.23	0.31
Julian_Day_Typecaster (Integer TCT)	0.95	0.51	0.66	0.60	0.26	0.33
Direction_Typecaster (Enumeration TCT)	0.16	0.31	0.38	0.27	0.35	0.45
Reporting_Location_Typecaster (Enumeration TCT)	0.19	0.35	0.44	0.31	0.51	0.62
Status_Typecaster (String Map TCT)	0.28	0.16	0.33	0.29	0.47	0.57
Scaled_Integer_100_500_Typecaster (Math_On_Integer TCT)	1.22	0.74	0.87	0.58	0.25	0.32
Scaled_Integer_100_1000_Typecaster (Math_On_Enumeration TCT)	0.32	0.46	0.72	0.60	0.29	0.38
Hour_Bit_Typecaster (Integer_Bit TCT)	0.30	0.32	0.51	0.56	0.21	0.31
Minute_Bit_Typecaster (Integer_Bit TCT)	0.30	0.32	0.51	0.57	0.21	0.32
Julian_Day_Bit_Typecaster (Integer_Bit TCT)	0.31	**0.43*	**0.60*	0.56	0.26	0.33
Direction_Bit_Typecaster (Enumeration_Bit TCT)	0.35	0.34	0.53	0.29	0.35	0.46
Reporting_Location_Bit_Typecaster (Enumeration_Bit TCT)	0.32	0.34	0.52	0.31	0.51	0.61
Status_Bit_Typecaster (Enumeration_Bit TCT)	0.33	0.35	0.51	0.31	0.48	0.58

Notes: * = Timing measurements were performed on code that was not optimized because of compiler problems with optimized code.

TCT = Typecaster Template

Figure 5-6: Discrete Typecaster Performance Summary

Composite Typecaster Performance Summary (Time in milli-seconds)						
	INR <--> UNR			INR <--> USR		
Composite Typecaster	Image	Value	Check	Image	Value	Check
Julian_Date_Time_ Record_Typecaster	2.98	1.71	1.92	1.88	0.94	1.02
FooBar_Message_ Private_Record_Typecaster	4.17	3.18	3.44	3.03	2.91	3.01
Julian_Date_Time_Bit_ Record_Typecaster	1.20	1.25	1.49	1.76	0.93	1.02
FooBar_Message_Bit_ Private_Record_Typecaster	2.61	2.91	3.34	2.95	2.93	3.04
Probability_ Wrapper_Typecaster	0.93	1.08	0.74	0.62	0.50	0.56
Probability_ Private_Array_Typecaster	6.77	9.19	6.66	5.58	4.76	4.75

Figure 5-7: Composite Typecaster Performance Summary

EXR Translation and Validation Performance Summary (Time in milli-seconds)			
	EXR <--> UNR		
ICD Typecaster	Extract	Construct	Check
FooBar_Message_ICD	2.13	3.47	2.50
FooBar_Bit_Message_ICD	3.11	4.93	3.49

Figure 5-8: EXR Translation and Validation Performance Summary

6. MTV Model Solution Application Description

This chapter describes how to apply the components of the MTV model solution, introduced in Chapter 5, to generate Ada code that translates and validates messages, as described in Chapter 4. This chapter is targeted for a detailed designer and/or implementor.

6.1. Foundation Utilities Compilation

Before the Ada software for translating and validating a specific message can be compiled, tested, and used, the following sets of utilities must be compiled and placed in an Ada library.

The file CCT_ADA, listed in Figure 5-4 under the category **Casting Common Types**, contains the Ada package *Casting_Common_Types* (CCT). The package contains global type declarations and constants that are used by the components of the MTV model solution. This package is synonymous with the Ada package *Standard*. This is the first package that must be compiled. Detailed designers will reference this package during the application of the MTV model solution. They may need type information and will need to find the names of predefined constants to apply the solution.

The files listed in Figure 5-4 under the category **Discrete Typecaster Generics** contain either an Ada generic package specification or an Ada generic package body. These packages are the foundation of the Typecaster model solution and are used by other portions of the MTV model solution. Detailed designers do not need to concern themselves with these packages other than compiling them into the Ada library. The generic package specifications and bodies must be compiled in the order the files are listed in Figure 5-4. Finally, the files listed in Figure 5-4 under the category **ICD Utilities** contain Ada packages. These are the foundation of the EXR TV model solution and are used by other portions of the MTV model solution. Detailed designers do not need to concern themselves with these packages other than compiling them into the Ada library. The package specifications and bodies must be compiled in the order the files are listed in Figure 5-4.

Figure 6-1 shows the software architecture of the MTV model solution after the foundation utilities have been compiled.

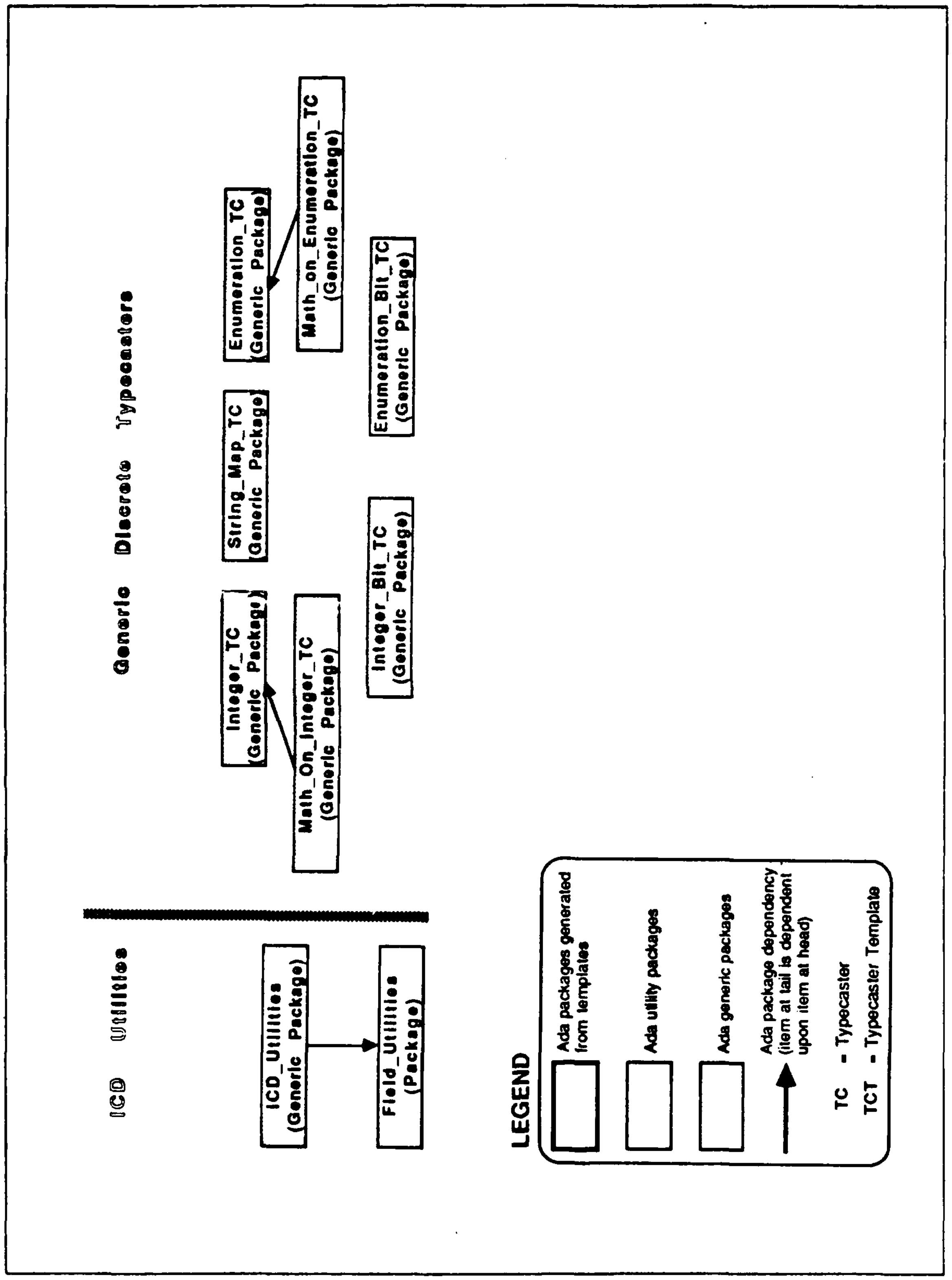
ICD Utilities
ICD_Utilities (Generic Package)
Field_Utilities (Package)
Generic Discrete Typecasters
Integer_TC (Generic Package)
Math_On_Integer_TC (Generic Package)
String_Map_TC (Generic Package)
Enumeration_TC (Generic Package)
Math_on_Enumeration_TC (Generic Package)
Integer_Bit_TC (Generic Package)
Enumeration_Bit_TC (Generic Package)
LEGEND
Ada packages generated from templates
Ada utility packages
Ada generic packages
Ada package dependency (item at tail is dependent upon item at head)
TC - Typecaster
TCT - Typecaster Template

Figure 6-1: MTV Model Solution Software Architecture After Foundation Utilities Are Compiled

6.2. Message Analysis

The requirements for translating and validating messages are expressed in the form of textual EXR descriptions[11] (as shown in Figure 2-2 and Figure 2-4). To apply the MTV model solution, detailed designers must analyze the EXR descriptions with the MTV model solution in mind.

Because the MTV model solution separates the concept of an EXR and an INR, the descriptions for these two must be codified separately, but an interface (UNR) between the two must exist.[12] How to codify the EXR description is described first. Second, a description of how to specify the transformation of the ordering from that of the EXR to that of the desired INR is provided (i.e., through the UNR). These first two analysis steps are used to instantiate the Msg_ICD template discussed in Section 6.3.3. Finally, how to specify the INR description (i.e., the Ada types) is described. This last analysis step is used to instantiate the typecaster templates discussed in Sections 6.3.1 and 6.3.2.

6.2.1. Codifying the EXR Description

Detailed designers should first make sure that the EXR description can be codified given the *Fields* data structure supplied by the Msg_ICD Template. The definition of the *Fields* data structure is the main step in instantiating the Msg_ICD template as described in Section 6.3.3.

Fields is an array where each element contains the description of a field (both its EXR and UNR). Two types of fields are possible: character-based and bit-based. The elements of the array are variant records where the variant part differs based on whether the field is character-based or bit-based.

The following information is required to describe a character-based field:

Base	The discriminant for the variant record that describes a field. It specifies whether the field is character-based or bit-based. Values are of the type *CCT.Base_Type*.[13]
Can_Be_Last	A boolean that indicates whether the field can be the last field in the EXR. This must be set *True* for at least one character-based field. The end of message punctuation marker and the length of the message are used to determine wh.:n the end of message has been reached.
Position	Describes the position of the symbolic image of this field in the UNR. This information is *derived by the software* and does not need to be specified by the detailed designers. Values are of the type *CCT.Position_Type*.

[11]The message EXR descriptions were contained in the ICD (Interface Control Document) for the Granite Sentry Program.

[12]Note that the form of each representation is different but the message information content captured by both representations must be the same.

[13]*CCT* refers to the Ada package *Casting_Common_Types*. See Appendix Section C.1.

Kind	Specifies whether the field is single, repetitive, or echoed. A single field appears once in a message. A repetitive field repeats itself consecutively in a message. An echoed field has the value of the field repeated with no more information content. Values are of the type *CCT.Field_Types*.
Width	An integer that specifies the number of characters in the field. For fields that are varying length, repetitive, or echoed, it specifies the maximum number of characters in the field.
Element_Size	An integer that specifies the number of characters contained in each element in the field. If the kind of field is single, then Element_Size is equal to Width. If the kind of field is repetitive, then Element_Size is equal to the Width divided by the number of times the element is repeated. If the kind of field is echoed, then Element_Size is equal to the Width divided by the number of times the element is echoed.
Null_Possible	A boolean that specifies whether the field can have a null value, i.e., no character physically present.
Odd	Specifies whether the field can have possible widths other than that specified by Width and Null_Possible. If true, then a list of integers specifying the alternate widths must be provided. Values are of the type *CCT.Odd_Description*.
Separator	Specifies the punctuation found at the end of a field. Values are of the type *CCT.Little_String_Type*. Also, constants are defined that specify several common punctuation marks, including a constant for specifying when no punctuation will be present in a EXR of a message. These can be found in the package *CCT*.

Figure 6-3 shows how *Fields* should be defined based on the EXR description for the FooBar message in Figure 2-2. Note that this information does not need to be entered yet. The framework of this description has been captured in the Msg_ICD template (shown in Appendix Section C.5). Using the Msg_ICD template is discussed in Section 6.3.3.

Because of the different possibilities related to the specification of Null_Possible and Odd_Possible, the truth table in Figure 6-2 enumerates the different possibilities and their meaning. It is assumed that a fields has a maximum length n, specified by Width in the description.

Null_Possible=>	Odd=>(Possible=>)	Meaning
True	True, Odd_Lengths=> $(i_1, i_2, \dots i_m)$	Variable length string $(0, (i_1, i_2, \dots i_m)$, or $n)$
True	False	Variable length string (0 or n)
False	True, Odd_Lengths=> $(i_1, i_2, \dots i_m)$	Variable length string $((i_1, i_2, \dots i_m)$, or $n)$
False	False	Fixed length string (n)

Figure 6-2: Variable Length Field Truth Table

```
type Field_Names is (Reporting_Location, Direction, Date_Time_Group, Status);

Fields : Icd_Util.Description_Array :=
  (Reporting_Location =>
    (Can_Be_Last => False,
     Position => (0,0),          -- Filled in at elaboration time
     Base => Cct.Char_Field,
        Kind => Cct.Single,
        Width => 3,
        Element_Size => 3,
        Null_Possible => False,
        Odd => (Possible => False),
        Separator => Cct.Cr
   ),
  Direction =>
    (Can_Be_Last => False,
     Position => (0,0),          -- Filled in at elaboration time
     Base => Cct.Char_Field,
        Kind => Cct.Single,
        Width => 1,
        Element_Size => 1,
        Null_Possible => False,
        Odd => (Possible => False),
        Separator => Cct.No_Punctuation
   ),
 Date_Time_Group =>
    (Can_Be_Last => False,
     Position => (0,0),          -- Filled in at elaboration time
     Base => Cct.Char_Field,
        Kind => Cct.Single,
        Width => 7,
        Element_Size => 7,
        Null_Possible => False,
        Odd => (Possible => False),
        Separator => Cct.Slash
   ),
  Status =>
    (Can_Be_Last => True,
     Position => (0,0),          -- Filled in at elaboration time
     Base => Cct.Char_Field,
        Kind => Cct.Single,
        Width => 1,
        Element_Size => 1,
        Null_Possible => False,
        Odd => (Possible => False),
        Separator => Cct.No_Punctuation
   )
 );

 EO_Text : constant Cct.Little_String_Type := Cct.CR;
```

Figure 6-3: Example Codification of Char-Based EXR Description: FooBar Message

The following information is required to describe a bit-based field:

Base	The discriminant for the variant record that describes a field. It specifies whether the field is character-based or bit-based. Values are of the type *CCT.Base_Type*.[14]
Can_Be_Last	A boolean that indicates whether the field can be the last field in the EXR. This must be set *True* for at least one bit-based field. The end of message punctuation marker and the length of the message are used to determine when the end of message has been reached.
Position	Describes the position of the symbolic image of this field in the UNR. This information is *derived by the software* and does not need to be specified by the detailed designers. Values are of the type *CCT.Position_Type*.
Number_Of_Bytes	An integer specifying the number of bytes that the field spans.
Byte_Positions	Specifies the byte number, and start and stop bits within that byte, for each byte that the field spans. Byte numbers are in the range 1..*n*, and bits are numbered 0..7, zero being the least significant bit. Values are of the type *CCT.Byte_Position_Record*.

Figure 6-4 shows how *Fields* is defined based on the bit-based EXR description for the FooBar message in Figure 2-4. Note that field three, *Date_Time_Group*, is described in three parts and not one part as in the text-based example, to extract each piece of information individually; each piece must be allocated to its own string.

Finally, *EO_Text* is of the type *CCT.Little_String*, and should specify the end-of-message punctuation marker for both character-based and bit-based messages. Possible end-of-message punctuation markers are defined in *CCT*, including one for specifying when no end-of-message marker will be present.

Also, the solution does handle messages that contain both character-based and bit-based fields, but care should be taken when describing the fields using the *Fields* array. When describing the fields, note that bit-based fields are expected to be in specific byte and bit locations and that character-based fields can be defined as varying length. This dynamic characteristic of character-based field descriptions could potentially conflict with the static bit-based descriptions.

[14]*CCT* refers to the Ada package *Casting_Common_Types*. See Appendix Section C.1.

```
type  Field_Names is
   (Reporting_Location, Direction, Date_Time_Group_Day,
    Date_Time_Group_Hour, Date_Time_Group_Minute, Status);

Fields : Icd_Util.Description_Array :=
  (Reporting_Location => (
      Can_Be_Last => False,
      Position => (0,0),          -- Filled in at elaboration time
      Base => Cct.Bit_Field,
      External_Format => (Number_Of_Bytes => 1,  Byte_Positions => (
                        1 => (Byte_Number => 1, Bit_Position => (
                                Start => 4,
                                Stop => 5))))),
   Direction => (
      Can_Be_Last => False,
      Position => (0,0),          -- Filled in at elaboration time
      Base => Cct.Bit_Field,
      External_Format => (Number_Of_Bytes => 1, Byte_Positions => (
                        1 => (Byte_Number => 1, Bit_Position => (
                                     Start => 2,
                                     Stop => 3))))),
   Date_Time_Group_Day => (
      Can_Be_Last => False,
     Position => (0,0),           -- Filled in at elaboration time
      Base => Cct.Bit_Field,
      External_Format => (Number_Of_Bytes => 3, Byte_Positions => (
                        1 => (Byte_Number => 1, Bit_Position => (
                                Start => 0,
                                Stop => 1) ),
                        2 => (Byte_Number => 2, Bit_Position => (
                                Start => 0,
                                Stop => 5) ),
                        3 => (Byte_Number => 3, Bit_Position => (
                                Start => 5,
                                Stop => 5))))),
   Date_Time_Group_Hour => (
      Can_Be_Last => False,
      Position => (0,0),          -- Filled in at elaboration time
      Base => Cct.Bit_Field,
      External_Format => (Number_Of_Bytes => 1, Byte_Positions => (
                        1 => (Byte_Number => 3, Bit_Position => (
                                Start => 0,
                                Stop => 4))))),
   Date_Time_Group_Minute => (
      Can_Be_Last => False,
      Position => (0,0),          -- Filled in at elaboration time
      Base => Cct.Bit_Field,
      External_Format => (Number_Of_Bytes => 1, Byte_Positions => (
                        1 => (Byte_Number => 4, Bit_Position => (
                                Start => 1,
                                Stop => 6))))),
   Status => (
      Can_Be_Last => False,
      Position => (0,0),          -- Filled in at elaboration time
      Base => Cct.Bit_Field,
      External_Format => (Number_Of_Bytes => 1, Byte_Positions => (
                        1 => (Byte_Number => 4, Bit_Position => (
                                Start => 0,
                                Stop => 0))))));

 EO_Text : constant Cct.Little_String_Type := Cct.No_Punctuation;
```

Figure 6-4: Example Codification of Bit-based EXR Description: FooBar Message

6.2.2. Interfacing the EXR and INR Descriptions

If the ordering and grouping of the information in the EXR description is a natural and acceptable one, then proceed to Section 6.2.3 and define the INR description (Ada type). If, however, the ordering of information is not appropriate, cutting (i.e., reordering) may be necessary.

For example, suppose the EXR description contained the field descriptions shown in Figure 6-5, that is, there are eight pieces of information grouped in each field. The goal, though, was to represent the information with the INR description containing the Ada types, also shown in Figure 6-5 — that is, as an array where each element of the array contains information from fields four, five, and six.

Cuts is an array where each element contains the names of the first and last fields to cut. To cut information is to reorder related repetitive information from the order specified in the EXR description to a more natural order, as specified by the INR description.

The initial step is to specify the *Fields*. Figure 6-6 shows how a repetitive field is specified. The next step is to specify the *Cuts* array, also shown in Figure 6-6. The information in the *Cuts* array specifies the fields where cutting is to start and stop. The definition of the *Cuts* data structure is a step in instantiating the Msg_ICD template as described in Section 6.3.3.

Finally, Figure 6-7 shows the EXR and the UNR of the information after it has been cut. Note that the information in the UNR has been reordered to match that of the INR.

6.2.3. Specifying the INR Description

After it is determined that the textual EXR description can be codified, the INR description must be specified, i.e., the Ada types necessary to represent the information in the message must be specified. These Ada types are the basis for instantiating the typecaster templates described in Sections 6.3.1 and 6.3.2.

Although the two representations are separate, they are still related with respect to the information they represent. Therefore, to specify the INR description, one must start by identifying the primitive elements of the fields (fields that can be represented with discrete values) in the EXR. In the EXR description shown in Figure 2-2, fields one, two, and four are primitive (i.e., they cannot be broken down further). Field three, *Date_Time_Group*, however, is not primitive but has three elements that are in primitive form.

Once the primitive elements are identified, the Ada discrete types of these elements must be specified based on the "meaning" (or amplifying data) specified for the element, not based on the symbolic information expected in the EXR:

- Primitive fields whose meaning can be represented by integers should be defined as a subtype of integer whose range is defined according to the "Range of Values" specified in the EXR description (see Figure 2-2).
- Primitive fields whose meaning can be represented by enumerations should be defined as an enumeration whose range is defined according to the "Amplifying Data" specified in the EXR description (see Figure 2-2).

Partial Message Format				
Field Number	**Field Name**	**Field Size (chars)**	**Range of Values**	**Amplifying Data**
4	Detection Confidence (DC)	8	aaaaaaaa	The DC in each 7.5 deg radar barrier segment reported in sequence. Values are: H = High M = Medium L = Low N = None
	Field Separator	1	<cr>	Carriage Return
5	Probability of Detect (PD)	16	nnnnnnnn nnnnnnnn	The PD in each 7.5 deg radar barrier segment Values are 00..99
	Field Separator	1	<cr>	Carriage Return
6	Barrier Segment (BS)	16	anananan anananan	The BS in each 7.5 deg degree radar segment Letter is start range: A=100 F=600 B=200 G=700 C=300 H=800 D=400 I=900 E=500 J=1000 Number is barrier width 1..5 = 100..500
	End of Message	1	<cr>	Carriage Return

```
type Detection_Confidence_Type is (High, Medium, Low, None);
subtype Integer_0_99_Type is Integer range 0..99;
subtype Scaled_Integer_100_1000_Type is Integer range 100..1000;
subtype Scaled_Integer_100_500_Type is Integer range 100..500;

type Barrier_Segment_Record_Type is record
  Detection_Confidence   : Detection_Confidence_Type;
  Probability_Of_Detection   : Integer_0_99_Type;
  Start_Range        : Scaled_Integer_100_1000_Type;
  Width              : Scaled_Integer_100_500_Type;
end Barrier_Segment_Record_Type;

Barrier_Segments_Array_Type is array (1..8) of Barrier_Segment_Record_Type;
```

Figure 6-5: Example of EXR and INR Descriptions Where Cutting Is Necessary

```
type Field_Names is
  (... Detection_Confidence, Probability_Of_Detect, Barrier_Segment);

Fields : Icd_Util.Description_Array :=
  (Detection_Confidence =>
    (Can_Be_Last => False,
     Position => (0,0),          -- Filled in at elaboration time
     Base => Cct.Char_Field,
        Kind => Cct.Repetitive,
        Width => 8,
        Element_Size => 1,
        Null_Possible => False,
        Odd => (Possible => False),
        Separator => Cct.Cr
    ),
   Probability_Of_Detect =>
    (Can_Be_Last => False,
     Position => (0,0),          -- Filled in at elaboration time
     Base => Cct.Char_Field,
        Kind => Cct.Repetitive,
        Width => 16,
        Element_Size => 2,
        Null_Possible => False,
        Odd => (Possible => False),
        Separator => Cct.Cr
    ),
   Barrier_Segment =>
    (Can_Be_Last => True,
     Position => (0,0),          -- Filled in at elaboration time
     Base => Cct.Char_Field,
        Kind => Cct.Repetitive,
        Width => 16,
        Element_Size => 2,
        Null_Possible => False,
        Odd => (Possible => False),
        Separator => Cct.Cr
    )
  );


  Cuts : Icd_Util.Cut_Array :=
     (1 => (Start => Detection_Confidence,
            Stop => Barrier_Segment)
     );
```

Figure 6-6: Example Codification of EXR Description with Repetitive Information and Specification of Cuts

External Representation

```
     |Field4|    |   Field 5     |     |   Field 6    |
    "aaaaaaaa    nnnnnnnnnnnnnnnn     ananananananananan"
    "HMLNHMLN<cr>1122334455667788<cr>A1B2C3D4E5F5G5H5<cr>"
```

Universal Representation

```
     |       Barrier_Segments_Array_Type       |
     |                                         |
 Barrier_Segment_Record_Type                   |
     |   |                                     |
    "annanannanannanannanannanannanannanannan"
    "H11A1M22B2L33B3N44D4H55E5M66F5L77G5N88H5"
```

Figure 6-7: Example of EXR and UNR Where Cutting Was Specified

While specifying the discrete Ada types, keep in mind that the character string images of the integer ranges and enumeration literals selected to represent the information (i.e., natural images) are the basis for the USR. Chapter 8 contains a discussion of this constraint.

After all Ada discrete types are defined for a message, the discrete types should be grouped using composite types (arrays and records) as follows:

- Physically related elements (i.e., represented by the same Ada type) can be grouped using an array type, for example, a list of 8 *Barrier Segments*, as shown in Figure 6-5.
- Logically related elements can be grouped using a record type, for example, a grouping of information pertaining to a *Julian Day*, as shown in Figure 2-6.
- Elements that are of varying length or can possibly be null are grouped using a variant record type, where the variant specifies whether the field is present or not.
- Finally, elements related by inter-element dependencies (as defined in the EXR description) are grouped using the appropriate composite type (record or array) and will be declared as a private type. The Ada private type is the mechanism used to preserve the validity of the internal representation based upon inter-element dependencies.

The ordering of the information in the INR description of the message must be preserved as per the EXR description and any cutting that is to be performed. See Chapter 8 for a discussion of this constraint.

After the discrete grouping is complete, group both composite types and discrete types using composite types. The same rules defined above can be applied here. This discrete/composite grouping continues until the composite for the message (usually a record) is defined. Figure 2-6 shows the INR description resulting from the analysis of the EXR description for the FooBar message shown in Figure 2-2.

6.3. MTV Model Solution Instantiation

The analysis of the message representations as defined in the previous section is the basis for creating an instance of the MTV model solution. The results of the analysis drives the detailed designers' choice of available building blocks (templates) and defines how the choices are to be instantiated.

As described in Chapter 5, there are two parts to the MTV model solution:

- Typecaster model solution
- EXR TV model solution

The templates that comprise the Typecaster model solution fall into two categories: discrete typecasters and composite typecasters. These templates are described in Sections 6.3.1 and 6.3.2. The EXR TV model solution is captured in one template, the Msg_ICD template. This is discussed in Section 6.3.3.

6.3.1. Discrete Typecaster Template Application

Once the INR description is defined (as described in Section 6.2.3), the first of two steps must be taken to create an instance of the Typecaster model solution. The first step in creating an instance of the Typecaster model solution involves generating the appropriate discrete typecasters for the discrete types defined by the INR description for a message. The discrete typecasters are generated by selecting from one of the seven discrete typecaster templates provided by the Typecaster model solution. The names and descriptions of when to use them are listed below. Also shown is an example of the conversions that result when the functionality of the discrete typecaster template is used. It is of the form:

symbolic image <--> *value* <--> *natural image*

1. **Integer Typecaster Template**. Use when converting between symbolic images that are character strings representing integers (zero padded and either signed or unsigned) and INRs that are integer values. Natural images, by definition, are character strings based upon images of the INR.[15]
   Example: "01" <--> 1 <--> " 1"

2. **Math_On_Integer Typecaster Template**. Use when converting between symbolic images that are character strings representing integers (zero padded and either signed or unsigned) and INRs that are scaled integer values. Natural images, by definition, are character strings based upon images of the INR.[15]
   Example: "1" <--> 100 <--> " 100"

3. **Enumeration Typecaster Template**. Use when converting between symbolic images that are character strings that can be represented as enumeration literals and INRs that are discrete values. Natural images, by definition, are character strings based upon images of the INR.[15]
   Example: "N" <--> North <--> "North"

4. **Math_On_Enumeration Typecaster Template**. Use when converting between symbolic images that are character strings that can be represented as enumeration literals and INRs that are scaled integer values. Natural images, by definition, are character strings based upon images of the INR.[15]
   Example: "A" <--> 100 <--> " 100"

5. **String_Map Typecaster Template**. Use when converting between symbolic images that are character strings and INRs that are discrete values where the relationship between the two representations is not supported by other character-based discrete typecaster templates. More specifically, use when:

   - The symbolic images are Ada key words.
     Example: "do" <--> DO_Air_Force_Base <--> "DO_Air_Force_Base"
   - The symbolic images represent integers but the INR desired is an enumeration type.
     Example: "0" <--> Operational <--> " Operational"
   - The symbolic images are case sensitive.
     Example: "N" <--> North <--> " North"
     "n" <--> No_Direction <--> "No_Direction"

   Also note that the String_Map typecaster can be used in the place of all character-based discrete typecasters templates. Natural images, by definition, are character strings based upon images of the INR.[15]

6. **Integer_Bit Typecaster Template**. Use when converting between symbolic images that are bit strings representing integers and INRs that are integer values. Natural images, by definition, are character strings based upon images of the INR.[15]
   Example: 00001001 <--> 9 <--> " 9"

7. **Enumeration_Bit Typecaster Template**. Use when converting between symbolic images that are bit strings representing enumeration literals and INRs that are enumeration values. Natural images, by definition, are character strings based upon images of the INR.[15]
   Example: 00000001 <--> South <--> "South"

[15]The natural image of discrete Ada types is defined using the *'image* attribute defined by the Ada language.

The choice of a discrete typecaster template is dependent upon the answers to the following two questions:

1. What is the desired INR description (Ada type)?
2. What is the nature of the symbolic image as a string (bit, alphanumeric character, or numeric character)?

The table shown in Figure 6-8 describes which typecaster template to choose based on answers to the two questions. The rows of the table describe the nature of the symbolic image as a string. The columns describe the INR description (Ada type) selected to represent this information.

For example, the symbolic images of the *Directions* field shown in Figure 2-2 are the characters "N", "S", "E", and "W". The information in the *Direction* field is represented as the Direction_Type enumeration type as shown in Figure 2-6, i.e.,

```
type Direction_Type is (North, South, East, West);
```

Therefore, upon examining the discrete typecaster template selection table shown in Figure 6-8, one should choose either the Enumeration typecaster template or the String_Map typecaster template. Before making a choice, the detailed designers should examine the details involved in applying each possible discrete typecaster template. More detailed descriptions of when to use each discrete typecaster template are found in Appendix Section B.3.

Once a selection is made, the detailed designers create an instance of the discrete typecaster template chosen by making a copy of the template in a new (empty) file and making the appropriate editor substitutions. The substitutions are based on placeholders located throughout the template. The placeholders are listed in the header comments at the beginning of the template. Some templates also use the "??" placeholder for special instructions, for example, replacement rather than substitution, and code fragment selection. The "??" placeholder is *never* listed in the header comments, but one should *always* search for it. The "??" placeholders should be located and replaced with the appropriate information as described by the instructions found with the "??". After the instructions are followed, the instructions must be removed. If the instruction lines are not removed, the file will not compile. This insures that all special instructions are examined. More detailed descriptions of how to instantiate each discrete typecaster template are in Appendix Section B.3.

After the template is instantiated, the file that contains a typecaster package specification and a test procedure can be compiled, and the test procedure can be linked and run. The test procedure performs two types of testing. First, it performs exhaustive testing on the typecaster operations based on the range of possible discrete values specified by the INR description (discrete Ada type). Second, it performs interactive testing, allowing the detailed designers to enter valid and invalid forms of the different representations. (i.e., symbolic images, natural images, and values).

Symbolic Images	Internal Representations		
	Integer	**Enumeration**	**Scaled Integer**
Numeric Char String	Integer TCT	String_Map TCT	Math_On_Integer TCT
AlphaNumeric Char String	Enumeration TCT	Enumeration TCT	Math_On_Enumeration TCT
Any Char String	String_Map TCT	String_Map TCT	String_Map TCT
Bit String	Integer_Bit TCT	Enumeration_Bit TCT	*Not Available**

Note:

TCT = Typecaster Template

* = This functionality was not necessary for Granite Sentry Phase II, although it could be easily implemented.

Figure 6-8: Discrete Typecaster Template Selection

Figure 6-9 shows examples of discrete typecaster template selection based on symbolic images and Ada types.[16]

Symbolic Image	Internal Representation	Discrete Typecaster Template Selection
"00"-"15"	Integer range 0..15	Integer TCT
"00"-"15"	(North..NorthNorthWest)	String_Map TCT
"00"-"15"	Integer range 0..1500	Math_On_Integer TCT
"NNN"-"NNW"	Integer range 0..15	Enumeration TCT
"NNN"-"NNW"	(North..NorthNorthWest)	Enumeration TCT
"NNN"-"NNW"	Integer range 0..1500	Math_On_Enumeration TCT
00000000-00001111	Integer range 0..15	Integer_Bit TCT
00000000-00001111	(North..NorthNorthWest)	Enumeration_Bit TCT
00000000-00001111	Integer range 0..1500	*Not Available*

Note: TCT = Typecaster Template

Figure 6-9: Discrete Typecaster Template Selection Examples

Figure 6-10 shows the typecaster software architecture of the FooBar message defined in Figure 2-2 after the discrete typecasters have been generated. At the bottom of the architecture, the foundation of the Typecaster model solution, are the discrete typecaster generics. Above these are the instances of the discrete typecaster templates, each dependent upon a discrete typecaster generic. The instances of the discrete typecaster templates also show (in parenthesis) which discrete typecaster template was used to generate them.

Finally, before proceeding, detailed designers should study the FooBar message example using Figure 2-2, Figure 2-6, and Figure 6-10 with the goal of understanding why specific templates were chosen, and become familiar with the discrete template descriptions found in Appendix Section B.3.

[16]Note that the String_Map typecaster template could be chosen for all character-based symbolic image possibilities.

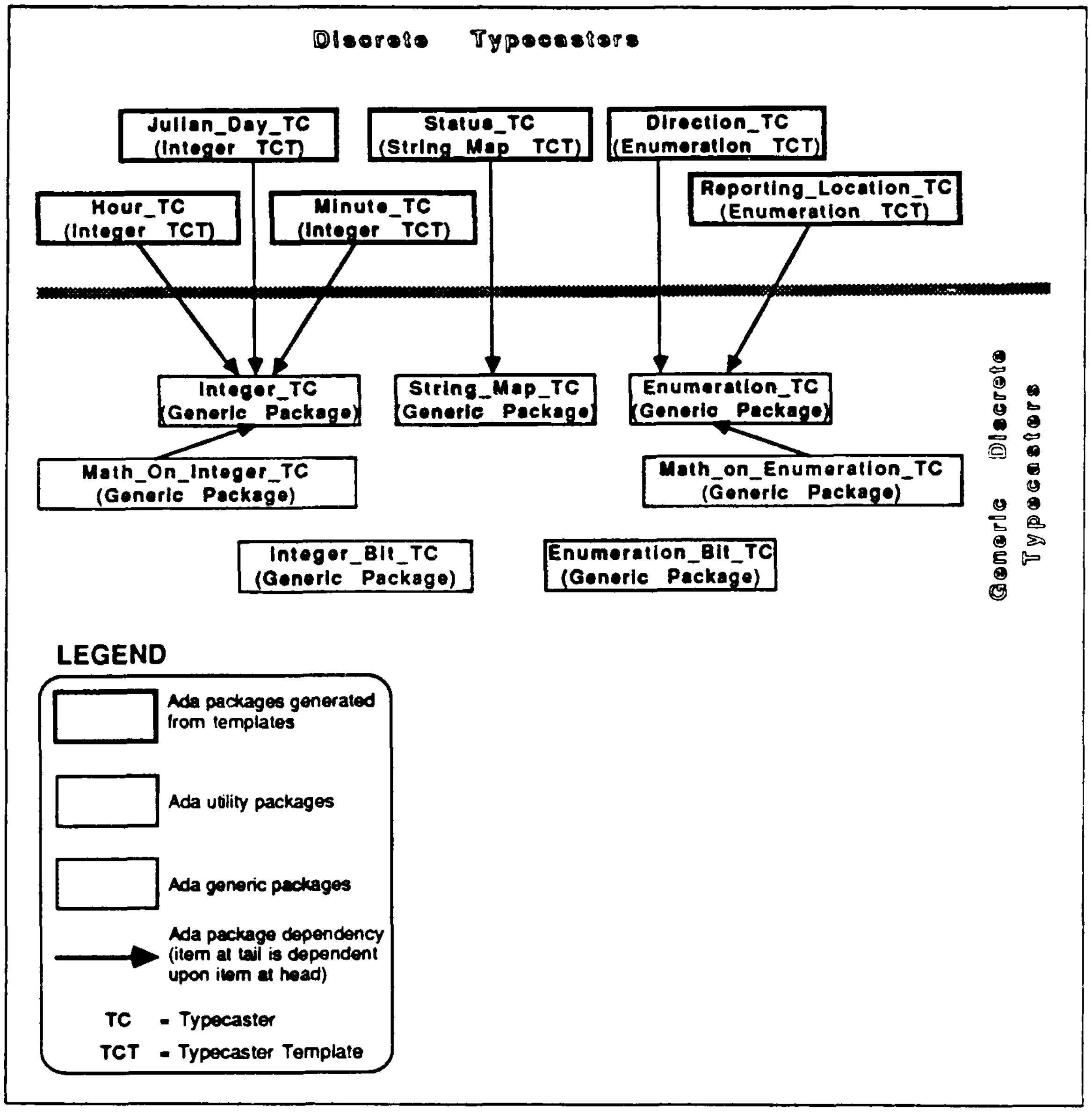
Discrete Typecasters
Julian_Day_TC (Integer TCT)
Status_TC (String_Map TCT)
Direction_TC (Enumeration TCT)
Hour_TC (Integer TCT)
Minute_TC (Integer TCT)
Reporting_Location_TC (Enumeration TCT)
Integer_TC (Generic Package)
String_Map_TC (Generic Package)
Enumeration_TC (Generic Package)
Math_On_Integer_TC (Generic Package)
Math_on_Enumeration_TC (Generic Package)
Integer_Bit_TC (Generic Package)
Enumeration_Bit_TC (Generic Package)
Generic Discrete Typecasters
LEGEND
Ada packages generated from templates
Ada utility packages
Ada generic packages
Ada package dependency (item at tail is dependent upon item at head)
TC - Typecaster
TCT - Typecaster Template

Figure 6-10: FooBar Message Typecaster Software Architecture After Generation of Discrete Typecasters

6.3.2. Composite Typecaster Template Application

Once the necessary discrete typecasters for a message have been instantiated, detailed designers can proceed to the second step in creating an instance of the Typecaster model solution. This second step involves generating the appropriate typecasters for the composite types defined by the INR description for a message and building these from the bottom of the composite type hierarchy to the top, the top being the type for the message. The composite typecasters are generated by selecting from one of the five composite typecaster templates provided by the Typecaster model solution. The names and descriptions of when to use them are listed below:

1. **Record Typecaster Template.** Use when the INR description is a record.
2. **Private_Record Typecaster Template.** Use when the INR description is a record and inter-element dependencies exist (as defined in the textual EXR description).
3. **Array Typecaster Template.** Use when the INR description is an array.
4. **Private_Array Typecaster Template.** Use when the INR description is an array and inter-element dependencies exist (as defined in the textual EXR description).
5. **Wrapper Typecaster Template.** Use when the INR description is a variant record where the discriminant indicates whether the information exists or not, as defined by the EXR description, i.e., a field that can be null.

More detailed descriptions of when to use each composite typecaster template are found in Appendix Section B.4.

Once a selection is made, the detailed designers create an instance of the composite typecaster template chosen by making a copy of the template in a new (empty) file and making the appropriate editor substitutions. The substitutions are based on placeholders located throughout the template. The placeholders are listed in the header comments at the beginning of the template. All templates also use the "??" placeholder for special instructions, for example, replacement rather than substitution, and code fragment selection. The "??" placeholder is *never* listed in the header comments, but one should *always* search for it. The "??" placeholders should be located and replaced with the appropriate information as is described by the instructions found with the "??". After the instructions are followed, the instructions must be removed. If the instruction lines are not removed, the file will not compile. This insures that all special instructions are examined. More detailed descriptions of how to instantiate each composite typecaster template are found in Appendix Section B.4.

After the template is instantiated, the file that contains a typecaster package specification, body, and a test procedure can be compiled and the test procedure can be linked and run. The test procedure performs canned testing based upon valid and invalid symbolic images entered by the detailed designers during template instantiation.

Figure 6-11 shows the typecaster software architecture of the FooBar message defined in Figure 2-2 after the composite typecasters have been generated. Notice that the instances of the composite typecaster templates needed for the FooBar message are dependent upon both discrete and composite typecasters. The instances of the composite typecaster templates also show (in parenthesis) which composite typecaster template was used to generate them.

After all composite typecasters have been generated, the detailed designers have an instance of the Typecaster model solution. This is one of two parts of the MTV model solution. The other part, the EXR TV model solution is discussed in Section 6.3.3.

As a review, the natural image of the message is the USR. The symbolic image of the message is the UNR. This representation is the interface between the EXR TV model solution and the Typecaster model solution. At this point one can proceed with the generation of an instance of the EXR TV model solution.

Finally, before proceeding, detailed designers should study the FooBar message example using Figure 2-2, Figure 2-6, and Figure 6-11 with the goal of understanding why specific templates were chosen, together with the composite template descriptions found in Appendix Section B.4.

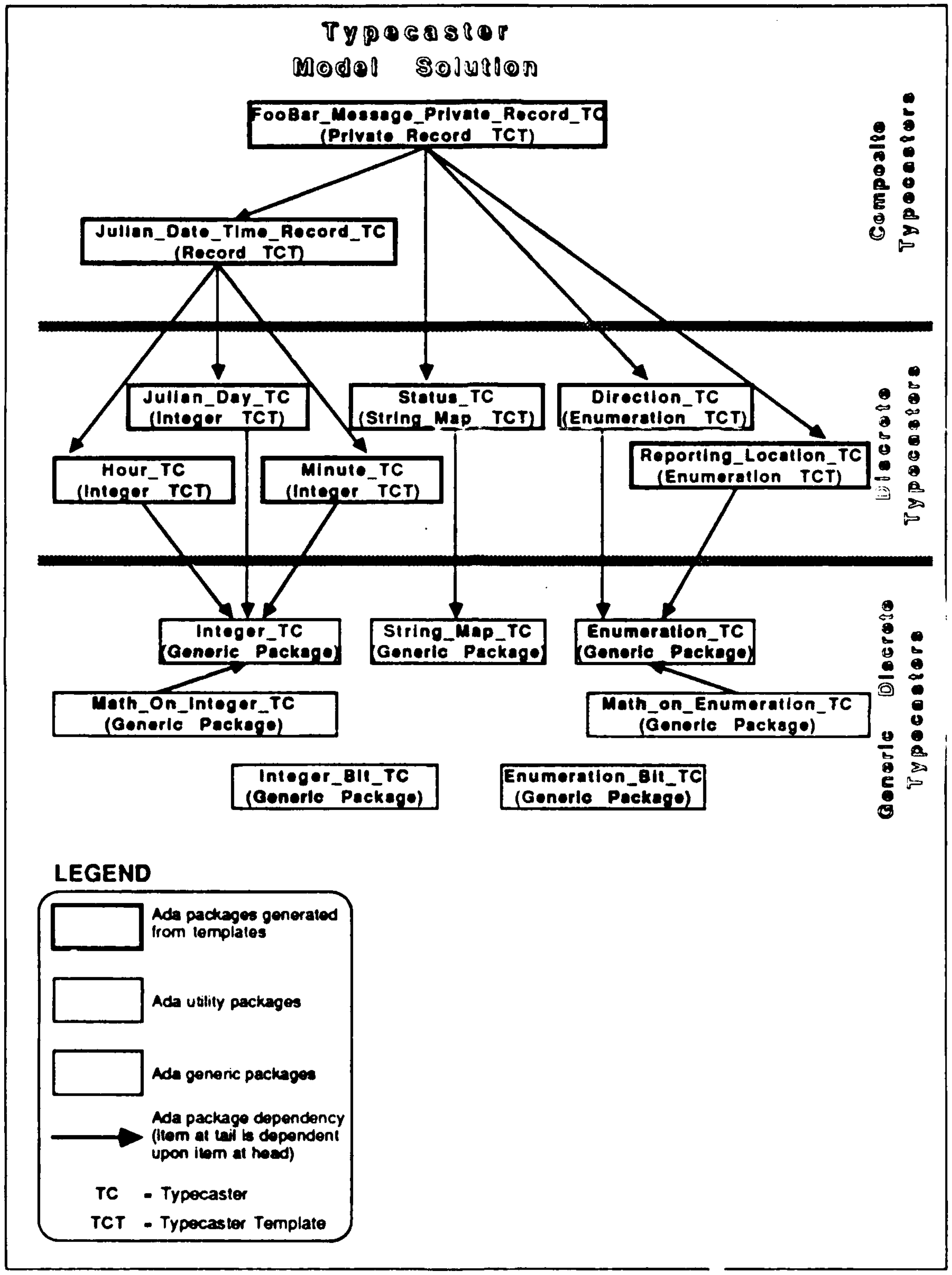
Typecaster
Model Solution
FooBar_Message_Private_Record_TC (Private Record TCT)
Julian_Date_Time_Record_TC (Record TCT)
Julian_Day_TC (Integer TCT)
Status_TC (String_Map TCT)
Direction_TC (Enumeration TCT)
Reporting_Location_TC (Enumeration TCT)
Hour_TC (Integer TCT)
Minute_TC (Integer TCT)
Integer_TC (Generic Package)
String_Map_TC (Generic Package)
Enumeration_TC (Generic Package)
Math_On_Integer_TC (Generic Package)
Math_on_Enumeration_TC (Generic Package)
Integer_Bit_TC (Generic Package)
Enumeration_Bit_TC (Generic Package)
Composite Typecasters
Discrete Typecasters
Generic Discrete Typecasters
LEGEND
Ada packages generated from templates
Ada utility packages
Ada generic packages
Ada package dependency (item at tail is dependent upon item at head)
TC - Typecaster
TCT - Typecaster Template

Figure 6-11: FooBar Message Typecaster Software Architecture After Generation of Composite Typeca[illegible]rs

6.3.3. External Representation TV Template Application

After producing an instance of the Typecaster model solution for a message, an instance of the EXR TV model solution for the same message can be created. The EXR TV model solution is captured in one template, the Msg_ICD template. It provides the capability to convert between the EXR and UNR of a message. The UNR is the interface between the Typecaster model and the EXR TV model.

Instantiation of the Msg_ICD template requires the definition of the *Fields* and the *Cuts* data structures. These are the results of message analysis and are described in Sections 6.2.1 and 6.2.2.

Detailed designers create an instance of the Msg_ICD template by making a copy of the template in a new (empty) file and making the appropriate editor substitutions. The substitutions are based on placeholders located throughout the template. The placeholders are listed in the header comments at the beginning of the template. This template also uses the "??" placeholder for special instructions, for example, replacement rather than substitution, and code fragment selection. The "??" placeholder is *never* listed in the header comments, but one should *always* search for it. The "??" placeholders should be located and replaced with the appropriate information as is described by the instructions found with the "??". After the instructions are followed, the instructions must be removed. If the instruction lines are not removed, the file will not compile. This insures that all special instructions are examined. More detailed descriptions of how to instantiate the Msg_ICD template are found in Appendix Section B.5.

After the template is instantiated, the file that contains a Msg_ICD package specification, body, and a test procedure can be compiled and the test procedure can be linked and run. The test procedure performs canned testing based upon the valid and invalid EXR entered by detailed designers during template instantiation.

Figure 6-12 shows the EXR TV software architecture of the FooBar message defined in Figure 2-2 after an instance of the Msg_ICD template has been generated.

The detailed designer does not need to create an instance of the Msg_ICD template if the EXR and the UNR are the same, i.e., a fixed-length, character-based string with punctuation removed, or a fixed-length, bit-based string with at most one field present in each byte of information.

At this point instances of both model solutions that comprise the MTV model solution exists. The UNR, or symbolic image, is the interface between the EXR TV model solution and the Typecaster model solution. What remains is to tie the two solutions together.

Finally, before proceeding, it is recommended that the FooBar message example using Figure 2-2, Figure 6-3, and Figure 6-12 be studied with the goal of understanding how to use the Msg_ICD template described in Appendix Section B.5 to codify an EXR description.

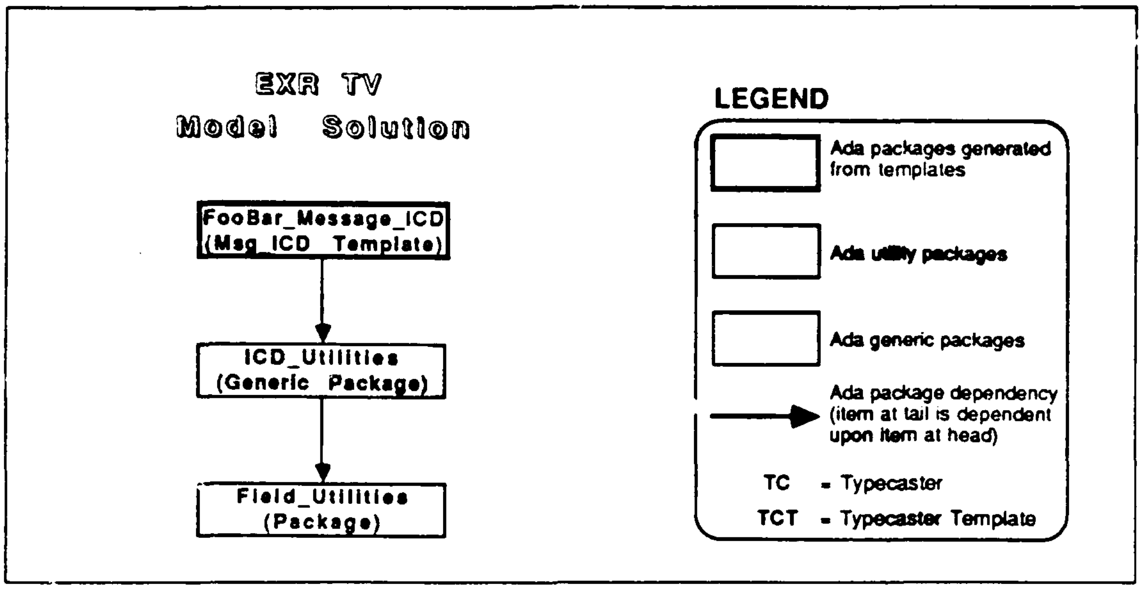
EXR TV
Model Solution
FooBar_Message_ICD (Msg_ICD Template)
ICD_Utilities (Generic Package)
Field_Utilities (Package)
LEGEND
Ada packages generated from templates
Ada utility packages
Ada generic packages
Ada package dependency (item at tail is dependent upon item at head)
TC = Typecaster
TCT = Typecaster Template

Figure 6-12: FooBar Message EXR TV Software Architecture

6.3.4. Tying Together the Two Model Solutions

Figure 6-13 shows the software architecture resulting from applying the MTV model solution to the the FooBar message. Notice that the instance of the EXR TV model solution is separate from the instance of the Typecaster model solution.

Now that the two model solutions have been generated and tested for a particular message, the two solutions need to be tied together. This part of the MTV solution must be coded manually.[17]

The detailed designers must generate an executive that makes the appropriate calls to the two model solutions based on the message type and the direction that the information is flowing (i.e., INR to EXR or vice versa). The interface between the two model solutions is the UNR. The Ada code fragment in Figure 6-14 shows a portion of an example executive.

[17]Although a template for this is feasible, one is not supplied with this solution.

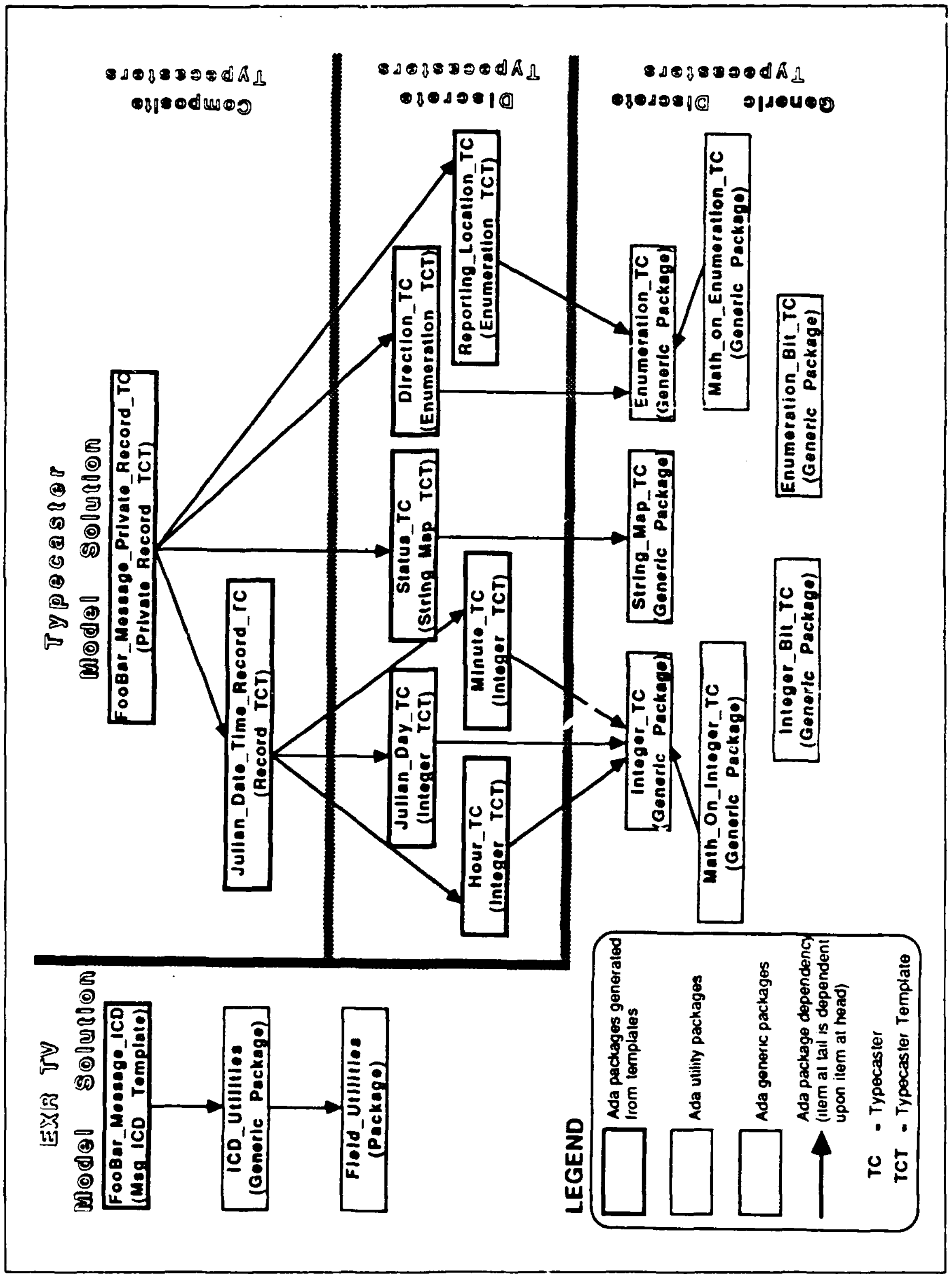
Typecaster Model Solution
Composite Typecasters
Discrete Typecasters
Generic Discrete Typecasters
FooBar_Message_Private_Record_TC (Private Record TCT)
Julian_Date_Time_Record_TC (Record TCT)
Status_TC (String Map TCT)
Direction_TC (Enumeration TCT)
Reporting_Location_TC (Enumeration TCT)
Julian_Day_TC (Integer TCT)
Minute_TC (Integer TCT)
Hour_TC (Integer TCT)
Enumeration_TC (Generic Package)
Math_on_Enumeration_TC (Generic Package)
Enumeration_Bit_TC (Generic Package)
String_Map_TC (Generic Package)
Integer_TC (Generic Package)
Math_On_Integer_TC (Generic Package)
Integer_Bit_TC (Generic Package)
EXR TV Model Solution
FooBar_Message_ICD (Msg ICD Template)
ICD_Utilities (Generic Package)
Field_Utilities (Package)
LEGEND
Ada packages generated from templates
Ada utility packages
Ada generic packages
Ada package dependency (item at tail is dependent upon item at head)
TC - Typecaster
TCT - Typecaster Template

Figure 6-13: FooBar Message MTV Software Architecture

```
 type Message_Ids is (..., Foobar, ...);

 Id: Message_Ids;
 A_Message : Cct.Byte_Array;
 Icd : Cct.Icd_Message_Type;
begin

 Get_Next_External_Message(A_Message);

   -- identify the message (Id) and
   -- store it for translation and validation (Icd)
 Message_Id(A_Message, Id, Icd);

 case Id is
  .
  .
  .

  when Foobar =>
  declare
    External_Rep  : FooBar_Message_Icd.External_Rep := Icd;
    Universal_Rep : FooBar_Message_Record_Typecaster.Universal_Rep;
    Internal_Rep  : FooBar_Message_Record_Typecaster.Foobar_Type;
    User_Rep      : FooBar_Message_Record_Typecaster.User_Rep;

  begin
    Universal_Rep := FooBar_Message_Icd.Extract(External_Rep);
    Internal_Rep  := FooBar_Message_Record_Typecaster.Value(Universal_Rep);

    -- process the Internal_Rep

    User_Rep := FooBar_Message_Record_Typecaster.Image(Internal_Rep);
  end ;

  .
  .
  .

 end case ;
```

Figure 6-14: Example Fragment of an MTV Executive

7. MTV Model Solution Description

This chapter describes the implementation of the MTV model solution and how the model solution provides the capabilities described in Chapter 4. This chapter is targeted for the maintainer and the model adapter, i.e., the people responsible for understanding the implementation of the model for:

- Integration purposes, i.e., integrating instances of the model solution into an application.
- Maintenance purposes, i.e., fixing errors or enhancing an application.
- Adaptation purposes, i.e., modifying the model solution for use in a particular application for which the current state of the model solution does not suffice due to efficiency reasons (size and speed) or possibly the need for additional functionality.

Fundamental Concepts

Ada packages are used for abstraction purposes, i.e., the encapsulation of data type and operations on the data type.

Ada generic packages are used when multiple instances of an abstraction are needed and the common information shared by instances of the abstraction (that makes each instance unique) can be formally generalized using generic formal parameters.

Ada package templates are used when Ada generic packages are not powerful enough, i.e., one cannot formally specify, using generic formal parameters, the common information that is shared by instances of an abstraction. Ada package templates are also used to ease the use (hide the use) of Ada generics and encapsulate the objects, types, and operations needed to instantiate the generics. The templates, in general, also facilitate testing by containing test drivers.

One key to the success of the MTV model solution is the consistent instantiation of the Typecaster model for each category of parts or templates described briefly in Chapter 5.

7.1. Foundation Type and Constant Definitions

7.1.1. Common Casting Types

All packages in the MTV model solution are dependent upon the types and constant declarations defined in the package *Casting_Common_Types* (*CCT*). The types and constants provide a consistent view of common information used throughout the model solution from the foundation utilities through the templates. They are the foundation upon which the MTV model solution is built.[18] See Appendix Section C.1 for the complete listing of the *CCT* package.

There are three sections of the package that are divided by comment lines. They are:

1. Definitions used by the Typecaster model solution.
2. Definitions used by the EXR TV model solution.
3. Definitions used by both Typecaster and EXR TV model solutions.

Typecaster Type and Constant Definitions

This section of the *CCT* package specification contains the following definitions that support the Typecaster model solution:

- The maximum length of the symbolic images and natural images (i.e., UNR and USR). These parameters can be adjusted based on the message set.
- Constant string declarations upon which all validity indicators, defined in the typecasters, are based (both symbolic image validity indicators and natural image validity indicators). The use of these constant string declarations insures validity indicator consistency across all typecasters.

EXR TV Type and Constant Definitions

This section of the *CCT* package specification contains the following definitions that support the EXR TV model solution:

- The maximum length of an ICD message, i.e., EXR. This parameter can be adjusted to different message set requirements.
- A type for holding ICD messages, i.e., a record that has as its elements a string and an integer indicating the length of the EXR.
- Constant string declarations that define common punctuation found in EXRs. Others can be added to support different message set requirements.
- Type definitions that support field extraction utilities.
- Type definitions that support the *Fields* data structure found in the ICD_Message template. This data structure is defined by the template instantiator and contains the codification of the EXR description for a message. *Fields* supports conversion between EXRs and UNRs.

[18]This package should be considered synonymous to the package *Standard* supplied by Ada.

Typecaster & EXR TV Type and Constant Definitions

This section of the *CCT* package specification contains the following definitions that support both the Typecaster and EXR TV model solutions:

- The notion of a string (*Byte_Array*) as an array of unsigned bytes. The symbolic image (UNR) is defined using this form.
- The concept of a position that has the attributes start and stop. The position of fields in the EXR and elements in the UNR and USR are expressed using this concept.

7.2. Typecaster Model Solution

This section discusses the various parts of the Typecaster model solution. The categories of parts that make up the Typecaster model solution, as discussed in Chapter 5, are:

- discrete typecaster generics
- discrete typecaster templates
- composite typecaster templates

Because the typecasters are all based upon the Typecaster model, and the implementations of individual parts within a category are similar, each part will not be examined individually. Instead, general solutions for each category will be examined.

For each category, the Ada PDL for the specification, body, and test procedure, where they exist, will be presented. Those portions of the PDL that differ from typecaster to typecaster in a category will be ***bold, italicized, and boxed***. How the abstractions (provided by a category of typecasters) are implemented is then discussed. Items to be examined include:

- what abstractions are provided
- what the package exports to provide the abstractions
- what is imported and inherited to support the abstractions
- how the abstractions are provided

Individual typecasters (either generics, templates or instances) can be examined in detail in the appendices of this document.

7.2.1. Foundation Utilities: Discrete Typecaster Generics

These are utility packages upon which the Typecaster model solution is built. This is shown abstractly in Figure 5-5 and via example in Figure 6-1. All discrete typecaster generics are Ada generic packages based upon the Typecaster model described in Chapter 4.

Their implementations must be understood when modifications or enhancements to the model solution are necessary. The Ada code for each discrete typecaster generic is shown in Appendix Section C.2.

Each discrete typecaster generic has a template associated with it that provides an "interface" for instantiating the generic. The template also includes a test procedure for testing particular instances of the discrete typecaster generic. Figure 7-1 shows the association of discrete typecaster generics to discrete typecaster templates and also lists the Appendix Section in which each discrete typecaster generic can be found. The discrete typecaster templates are discussed in the next section.

The job of the discrete typecaster generics is to provide a mapping between symbolic images or natural images, and typed discrete Ada values. The existence of a discrete typecaster generic (and thus a discrete typecaster template) is driven by the characteristics of the symbolic image and the desired INR. This is shown in Figure 6-8. If a relationship exists between symbolic images and values that is not represented in Figure 6-8, other discrete typecaster generics and corresponding templates need to be generated. This is discussed further in Chapter 8

Discrete Typecaster Generic	Appendix Section	Discrete Typecaster Template
Integer_Typecaster	C.2.1	Integer Typecaster Template
Math_On_Integer_Typecaster	C.2.3	Math_On_Integer Typecaster Template
Enumeration_Typecaster	C.2.4	Enumeration Typecaster Template
Math_On_Enumeration_Typecaster	C.2.6	Math_On_Enumeration Typecaster Template
String_Map_Typecaster	C.2.7	String_Map Typecaster Template
Integer_Bit_Typecaster	C.2.2	Integer_Bit Typecaster Template
Enumeration_Bit_Typecaster	C.2.5	Enumeration_Bit Typecaster Template

Figure 7-1: Discrete Typecaster Generics

Figure 7-2 shows the package specification PDL for a discrete typecaster generic. Figure 7-3 shows the package body PDL for a discrete typecaster generic. All discrete typecaster generic specifications and bodies are identical except where the PDL is $\boxed{\textbf{\textit{bold, italicized, and boxed}}}$.

The discrete typecaster generics provide the following abstract capabilities:

- Convert from a symbolic image to typed discrete Ada value.
- Convert from a typed discrete Ada value to a symbolic image.
- Check a symbolic image for validity (i.e., can it be converted to a value?).
- Convert from a natural image to typed discrete Ada value.
- Convert from a typed discrete Ada value to a natural image.
- Check a natural image for validity (i.e., can it be converted to a value?).
- Exception is raised if conversion from a symbolic image or natural image to a value fails.
- Typed discrete Ada values are always valid (enforced by the Ada runtime) and can always be converted to a symbolic image or a natural image.

These abstractions make up the Typecaster model described in Chapter 4.

Two pieces of information are needed to instantiate a particular discrete typecaster generic. The first is the *Type_To_Be_Cast*. This is an Ada type that is the INR of the information. The second piece of information is *The_Map*. This defines the mapping between symbolic images and values. This mapping information is expressed differently among discrete generic typecasters. For example, the *Integer_Typecaster* (see Appendix Section C.2.1) has no *The_Map* because the mapping from symbolic images to values is implicit.[19] The *Enumeration_Typecaster* (see Appendix Section C.2.4), on the other hand, defines *The_Map* as an Ada enumeration type where the enumeration literals are the symbolic images, and the mapping between a symbolic image and value is defined via positional correspondence of the values defined by the two discrete types. For more information on how *The_Map* is defined for individual discrete typecaster generics, see the Ada generic code in Appendix Section C.2. For more information on specifying the mapping, see the directions for instantiating individual discrete typecaster templates in Appendix Section B.3.

Based upon *Type_To_Be_Cast*, *The_Map*, and some information obtained from the package *Casting_Common_Types*, all discrete typecaster generics provide the abstract functionality described above. The remainder of this section discusses how the functionality is provided (i.e., implemented).

The *Symbolic_Image_Width* is defined as an integer constant. Its value is based upon information about the symbolic images provided in *The_Map*, whether it is implicit or explicitly defined. *Symbolic_Image*, the type used to hold a symbolic image, is defined as an array of bytes of length *Symbolic_Image_Width*. The type that defines an array of bytes comes from *CCT.Byte_Array*. This promotes consistency among the discrete typecaster generics.

Similarly, the *Natural_Image_Width* is defined as an integer constant. Its value is based on

[19]The mapping is implicit because Ada provides *'Image* and *'Value* functions.

the size of the image of an Ada value as defined by the Ada *'Image* and Ada *'Width* attributes. *Natural_Image*, the type used to hold a natural image, is defined as a character string of length *Natural_Image_Width*. The type that defines a character string comes from *Standard.String*. This promotes consistency among the discrete typecaster generics.

Two *Value* functions exist that provide the capability to convert from a symbolic image or natural image to a value. The basic underlying algorithms for both *Value* functions are shown in Figure 7-3. All symbolic image *Value* functions are implemented slightly differently. The algorithm involves first interpreting the symbolic image bytes as a character string or integer based on whether the discrete typecaster generic is character-based or bit-based. This interpreted result is then used in conjunction with *The_Map* to obtain the desired value of the symbolic image. Conversely, all natural image *Value* functions are exactly the same. The algorithm involves using the Ada *'Value* attribute as the mapping function to obtain the desired value of the natural image. Finally, the *Value* functions are implemented so that the exception *Constraint_Error* is raised by the Ada runtime when either conversion fails.

Two *Image* functions exist that provide the capability to convert from a value to a symbolic image or natural image. The basic underlying algorithms for both *Image* functions are shown in Figure 7-3. All symbolic image *Image* functions are implemented slightly differently. The algorithm involves using *The_Map* to obtain either a character string or integer symbolic image, based on whether the discrete typecaster generic is character-based or bit-based. The result of the mapping is converted to an array of bytes. All natural image *Image* functions are very similar. The algorithm involves using the Ada *'Image* attribute as the mapping function to obtain the desired character string of the value. This character string is then left padded with spaces, as necessary, to fill in the fixed-length natural image. Finally, because of the strong typing provided by Ada, the solution assumes that values are always valid, therefore, the *Image* function should never fail.

Two *Check* functions exist that provide the capability to check the validity of a symbolic image or a natural image. The basic underlying algorithms for both *Check* functions are shown in Figure 7-3. The *Check* functions are implemented exactly the same, but return different results based on whether they are checking symbolic images or natural images. The algorithm involves calling the appropriate *Value* function to see if conversion is possible. If no *Constraint_Error* is raised, then the conversion was made, implying that the image is valid. Either the *Valid_Symbolic_Image* or *Valid_Natural_Image* indicator is returned, depending upon the *Check* function. If a *Constraint_Error* is raised, then the conversion was not made, implying that the image is not valid. Either the *Invalid_Symbolic_Image* or *Invalid_Natural_Image* indicator is returned, depending upon the *Check* function.

The validity indicators returned by the *Check* functions are defined as *Symbolic_Image* or *Natural_Image* constants. Their value is based on the validity indicators defined in *Casting_Common_Types* that are named the same. The validity indicators defined in the discrete typecaster generics slice the validity indicators defined in *Casting_Common_Types* based on the size of the symbolic image and natural image. Basing the validity indicators for the discrete typecaster generics on those found in *Casting_Common_Types* promotes consistency across all typecasters. The advantages of having the validity indicators the same size as the images will become evident when the composite templates are discussed in Section 7.2.3.

The *Image* and *Value* functions are inlined for efficiency reasons.

Finally, when the generic is instantiated, the mapping is checked to ensure it is completely specified (one-to-one). That is, for every element in the domain (values), there is a corresponding element in the range (symbolic images), and vice versa. The mapping between values and natural images is assumed to be complete (one-to-one) as specified by the Ada *'Image* and *'Value* attributes.

```
with Casting_Common_Types;
generic
  -- Internal Representation
    type Type_To_Be_Cast is generic formal parameter for discrete Ada type;

    -- Mapping from symbolic images to values of type to be cast
    The_Map...;
package Generic_Typecaster is
    package CCT renames Casting_Common_Types;

  -- Universal Representation
    -- Symbolic image size and string to hold it.
    Symbolic_Image_Width : constant Integer := based on The_Map;
    subtype Symbolic_Image is CCT.Byte_Array(1..Symbolic_Image_Width);

    -- Functions for converting between a symbolic image and a value.
    -- Also a function for checking the validity of a symbolic image.
    function Value(Image_In : Symbolic_Image)  return Type_To_Be_Cast;
    function Image(Value_In : Type_To_Be_Cast) return Symbolic_Image;
    function Check(Image_In : Symbolic_Image)  return Symbolic_Image;

    -- Symbolic image validity indicators returned by Check function
    Valid_Symbolic_Image     : constant Symbolic_Image :=
        CCT.Valid_Symbolic_Image(1..Symbolic_Image_Width);
    Invalid_Symbolic_Image   : constant Symbolic_Image :=
        CCT.Invalid_Symbolic_Image(1..Symbolic_Image_Width);

  -- User Representation
    -- Natural image size and string to hold it.
    Natural_Image_Width : constant Integer := Type_To_Be_Cast'width;
    subtype Natural_Image is String(1..Natural_Image_Width);

    -- Functions for converting between a natural image and a value.
    -- Also a function for checking the validity of a natural image.
    function Value(Image_In : Natural_Image)   return Type_To_Be_Cast;
    function Image(Value_In : Type_To_Be_Cast) return Natural_Image;
    function Check(Image_In : Natural_Image)   return Natural_Image;

    -- Natural image validity indicators returned by Check function
    Valid_Natural_Image      : constant Natural_Image :=
        CCT.Valid_Natural_Image(1..Natural_Image_Width);
    Invalid_Natural_Image    : constant Natural_Image :=
        CCT.Invalid_Natural_Image(1..Natural_Image_Width);

  -- Real-Time constraint raised by Image and Value functions
    Constraint_Error : Exception;
    pragma inline(Value, Image);
end Generic_Typecaster;
```

Figure 7-2: Discrete Typecaster Generic Package Specification PDL

```
with Unchecked_Conversion;
package body Generic_Typecaster is
    -- replace * in the PDL based on the results of the condition below
    if the generic typecaster is character-based then    *=STRING
    elsif the generic typecaster is bit-based then       *=INTEGER
    Bytes_To_* is new Unchecked_Conversion (Symbolic_Image, *);}
    *_To_Bytes is new Unchecked_Conversion (*, Symbolic_Image);}
--
    function Value(Image_In : Symbolic_Image) return Type_To_Be_Cast is
        The_SI : * := Bytes_To_*(Image_In);
    begin
        return Mapping(The_Map, The_SI);
    end Value;

    function Image(Value_In : Type_To_Be_Cast) return Symbolic_Image is
        The_SI : * := Mapping(The_Map, Value_In);
    begin
        return *_To_Bytes(The_SI);
    end Image;

    function Check(Image_In : Symbolic_Image) return Symbolic_Image is
        Dummy : Type_To_Be_Cast := Value(Image_In);
    begin
        return Valid_Symbolic_Image;
    exception   when Constraint_Error => return Invalid_Symbolic_Image;
    end Check;
--
    function Value(Image_In : Natural_Image) return Type_To_Be_Cast is
    begin
        return Type_To_Be_Cast'Value(Image_In);
    end Value;

    function Image(Value_In : Type_To_Be_Cast) return Natural_Image is
    begin
        return Slice of CCT.Blank_String & Type_To_Be_Cast'Image(Value_In);
    end Value;

    function Check(Image_In : Natural_Image) return Natural_Image is
        Dummy : Type_To_Be_Cast := Value(Image_In);
    begin
        return Valid_Natural_Image;
    exception   when Constraint_Error => return Invalid_Natural_Image;
    end Check;
begin
    Ensure definition of mapping is complete
end Generic_Typecaster;
```

Figure 7-3: Discrete Typecaster Generic Package Body PDL

7.2.2. Building Blocks: Discrete Typecaster Templates

Discrete typecaster templates are the initial building blocks of the Typecaster model solution. They form the first layer of the solution (software architecture) on top of the discrete typecaster generics. This is shown abstractly in Figure 5-5 and via example in Figure 6-10. All discrete typecaster templates contain an Ada package template based upon the Typecaster model described in Chapter 4 and a test procedure for testing instances of the discrete typecaster template. The Ada code for the discrete typecaster templates is shown in Appendix Section C.3.

As mentioned in the previous section, each discrete typecaster template has a generic associated with it. The template exists for a number of reasons:

- The discrete typecaster templates contain an Ada package that has placeholders for the definition of the discrete type to be cast and any symbolic image mapping information needed to instantiate the discrete typecaster generics.[20]
- The templates provide an "interface" for instantiating the corresponding discrete typecaster generics.
- The template facilitates testing of particular instances of the discrete typecaster generics.

Figure 7-1 shows the association of discrete typecaster generics to discrete typecaster templates. Figure 7-4 lists the discrete typecaster templates and the appendix sections where the code templates can be found. Also included in Figure 7-4 are the name of an example instance of each discrete typecaster template and the appendix sections where the example code can be found.

Discrete Typecaster Template	Appendix Section	Example Instance	Appendix Section
Integer Typecaster Template	C.3.1	Hour_Typecaster	D.1.3
Math_On_Integer Typecaster Template	C.3.3	Scaled_Integer_100_500_Typecaster	D.4.1
Enumeration Typecaster Template	C.3.4	Direction_Typecaster	D.1.2
Math_On_Enumeration Typecaster Template	C.3.6	Scaled_Integer_100_1000_Typecaster	D.4.2
String_Map Typecaster Template	C.3.7	Status_Typecaster	D.1.6
Integer_Bit Typecaster Template	C.3.2	Julian_Day_Bit_Typecaster	D.1.11
Enumeration_Bit Typecaster Template	C.3.5	Direction_Bit_Typecaster	D.1.8

Figure 7-4: Discrete Typecaster Templates

Figure 7-5 shows the discrete <*Type*>*_Typecaster* package specification PDL. This is the first of two separate code modules found in all discrete typecaster templates. All <*Type*>*_Typecaster* package specifications found in the discrete typecaster templates are identical except where the PDL is ***bold, italicized, and boxed***.

[20]An alternative to this packaging approach is described in Chapter 8.

The discrete *<Type>_Typecaster* packages provide the following abstract capabilities:[21]

- Convert from a symbolic image to typed discrete Ada value.
- Convert from a typed discrete Ada value to a symbolic image.
- Check a symbolic image for validity (i.e., can it be converted to a value?).
- Convert from a natural image to typed discrete Ada value.
- Convert from a typed discrete Ada value to a natural image.
- Check a natural image for validity (i.e., can it be converted to a value?).
- An exception is raised if conversion from a symbolic image or natural image to a value fails.
- Typed discrete Ada values are always valid (enforced by the Ada runtime) and can always be converted to a symbolic image or a natural image.

These abstractions make up the Typecaster model described in Chapter 4.

Two pieces of information must be provided to instantiate a particular discrete typecaster template. The first is the type to be cast, *<Type>_Type*. This is a discrete Ada type that is the INR of the information. The second piece of information is *The_Map*. This information defines the mapping between symbolic images and values, and resides in the *<Type>_Typecaster* package. This information is expressed differently depending upon the discrete typecaster template chosen. For example, the *Hour_Typecaster* package (see Appendix Section D.1.3) has no *The_Map* because the mapping from symbolic images to values is implicit via the Ada *'Image* and *'Value* attributes. The *Direction_Typecaster* package (see Appendix Section D.1.2), on the other hand, defines *The_Map* as an Ada enumeration type where the enumeration literals are the symbolic images, and the mapping between a symbolic image and value is defined via positional correspondence of the values defined by the two discrete types. For more information on how *The_Map* is defined for individual discrete typecaster templates, see the discrete typecaster templates in Appendix Section C.3. For more information on specifying the mapping, see the directions for instantiating individual discrete typecaster templates in Appendix Section B.3.

Based upon the definitions provided for *<Type>_Type* and *The_Map*, all *<Type>_Typecaster* packages in the templates instantiate their corresponding discrete typecaster generic. These *<Type>_Typecaster* packages provide (i.e., implement) the abstract functionality described above by renaming all types, constants, and functions exported by the instantiation of the discrete typecaster generic. That is, the *<Type>_Typecaster* packages inherit the functionality provided by the instances of the discrete typecaster generics.

Figures 7-6 and 7-7 show the PDL for the exhaustive and interactive portions of the *<Type>_Typecaster_Test* procedure. This is the second of two separate code modules found in all discrete typecaster templates. The test procedures are very similar. The parts that vary most deal with the I/O portions of the test procedure. To signify those parts of the test procedures that deal with communication with the tester, the PDL is **bolded**.

[21]Note that these are the same as those that described the discrete typecaster generics in the previous section.

The test procedure, *<Type>_Typecaster_Test*, tests the instance of the *<Type>_Typecaster* package generated during template instantiation. Since each *<Type>_Typecaster* package is based on a discrete typecaster generic and these can be instantiated with discrete types only, the first portion of the test tests the functionality provided by the typecaster packages based on the finite range of values the discrete type can assume.

For each possible discrete value, the following steps are performed first for symbolic images and then for natural images, and any test case failures are reported to the tester:

1. Apply *Image* function, i.e., convert the value to an image.
2. Apply *Check* function, i.e., check the image for validity.
3. If *Check* function indicates an invalid image then the typecaster has failed.
4. Apply *Value* function, i.e., convert the image back to a value.
5. If *Value* function raises a constraint error then the typecaster has failed.
6. Also, if the discrete value from Step 1 is not the same value that results in Step 4 then the typecaster has failed.

But exhaustive testing only performs testing of typecaster functionality based on valid symbolic images, natural images, and values. There is also a need to test typecaster functionality based on invalid symbolic images, natural images, and values, and to make sure the mapping specified when the template was instantiated is correct. This testing is provided through interactive testing.

Because the typecasters perform translation and validation based on the relationship between either symbolic images and values, or natural images and values, the interactive testing is broken up into two parts. The first part tests translation and validation between tester-supplied values and symbolic images. The second part tests translation and validation between tester-supplied values and natural images.

The following steps are performed for each of the two testing parts described above:

1. Prompt tester for image.
2. Apply *Check* function, i.e., check the image for validity.
3. Notify tester of image validity.
4. Apply *Value* function, i.e., convert the image to a value.
5. Show tester value of image.
6. If *Value* function raises a constraint error then notify tester of image invalidity.
7. Prompt tester for value.
8. Apply *Image* function, i.e., convert the value to an image.
9. Show tester image of value.
10. If tester types ^Z at any prompt, exit the test.

This form of interactive testing allows the tester to enter invalid symbolic and natural images to test the functionality of the typecasters based on invalid images. The tester can ensure that the typecaster truly treats them as invalid based on the information returned by the test routine in the form of error messages and the presentation of the results of the *Check* function. The tester can also use interactive testing to ensure the mapping used to instantiate the typecaster was specified properly by entering valid values and images and examining the results.

```
with Generic _Typecaster; -- The corresponding generic typecaster
package <Type>_Typecaster is
  -- Internal Representation & Internal Representation Description
    -- Type to be cast
    type <Type>_Type is ?? ;
    -- Symbolic image (universal representation) mapping information
    <Type>_Map ?? ;
    -- Instantiation of typecaster appropriate for this template
    <Type>_TC is new Generic _Typecaster (<Type>_Type, <Type>_Map ) ;

  -- Universal Representation
    -- Symbolic image size and string to hold it.
    Symbolic_Image_Width : Integer renames <Type>_TC.Symbolic_Image_Width;
    subtype Symbolic_Image is <Type>_TC.Symbolic_Image;

    -- Functions for converting between a symbolic image and a value.
    -- Also a function for checking the validity of a symbolic image.
    function Value(Image_In : Symbolic_Image) return <Type>_Type
        renames <Type>_TC.Value;
    function Image(Value_In : <Type>_Type) return Symbolic_Image
        renames <Type>_TC.Image;
    function Check(Image_In : Symbolic_Image) return Symbolic_Image
        renames <Type>_TC.Check;

    -- Symbolic image validity indicators returned by Check function
    Valid_Symbolic_Image : renames <Type>_TC.Valid_Symbolic_Image;
    Invalid_Symbolic_Image : renames <Type>_TC.Invalid_Symbolic_Image;

  -- User Representation
    -- Natural image size and string to hold it.
    Natural_Image_Width : Integer renames <Type>_TC.Natural_Image_Width;
    subtype Natural_Image is <Type>_TC.Natural_Image;

    -- Functions for converting between a natural image and a value.
    -- Also a function for checking the validity of a natural image.
    function Value(Image_In : Natural_Image) return <Type>_Type
        renames <Type>_TC.Value;
    function Image(Value_In : <Type>_Type) return Natural_Image
        renames <Type>_TC.Image;
    function Check(Image_In : Natural_Image) return Natural_Image
        renames <Type>_TC.Check;

    -- Natural image validity indicators returned by Check function
    Valid_Natural_Image : renames <Type>_TC.Valid_Natural_Image;
    Invalid_Natural_Image : renames <Type>_TC.Invalid_Natural_Image;

  -- Real-Time constraint raised by Image and Value functions
    Constraint_Error : Exception;
end <Type>_Typecaster;
```

Figure 7-5: Discrete Typecaster Template Package Specification PDL

```
with <Type>_Typecaster;
with Unchecked_Conversion;
with Casting_Common_Types;
with Text_IO;
procedure <Type>_Typecaster_Test is
begin

    -- Exhaustive Testing Loop: loop through range of discrete values
    for Value in <Type>_Typecaster.<Type>_Type'First ..
                 <Type>_Typecaster.<Type>_Type'Last loop
    begin
      -- Symbolic Image Testing
        Test_SI  := <Type>_Typecaster.Image(Value);
        Check_SI  := <Type>_Typecaster.Check(Test_SI);
        if Test_SI /= <Type>_Typecaster.Valid_Symbolic_Image then
            Notify tester symbolic image exhaustive testing FAILED.
        end if;

        Test_Value  := <Type>_Typecaster.Value(Test_SI);
        if Test_Value   /= Value then
            Notify tester symbolic image exhaustive testing FAILED.
        end if;

      -- Natural Image Testing
        Test_NI  := <Type>_Typecaster.Image(Value);
        Check_NI  := <Type>_Typecaster.Check(Test_NI);
        if Test_NI /= <Type>_Typecaster.Valid_Symbolic_Image then
            Notify tester natural image exhaustive testing FAILED.
        end if;

        Test_Value  := <Type>_Typecaster.Value(Test_NI);
        if Test_Value   /= Value then
            Notify tester natural image exhaustive testing FAILED.
        end if;

    exception
        when Constraint_Error =>
            Errors := True;
            if raised during symbolic image testing then
                Notify tester symbolic image exhaustive testing FAILED.
            elsif raised during natural image testing then
                Notify tester natural image exhaustive testing FAILED.
            end if;
    end;

    end loop;
    if Errors occurred then
        Notify tester exhaustive testing completed unsuccessfully.
    else
        Notify tester exhaustive testing completed successfully.
    end if;
```

Figure 7-6: Discrete Typecaster Template Test PDL (Exhaustive)

```
    -- Interactive Testing Loop
    loop
    begin

      -- Symbolic Image Testing
        Prompt tester for character string or integer symbolic image, Test_SI.
        Convert Test_SI to CCT.Byte_Array.
        Check_SI := <Type>_Typecaster.Check(Test_SI);
        Show tester validity indicator, Check_SI, returned by Check function.
        Test_Value := <Type>_Typecaster.Value(Test_SI);
        Show tester value, Test_Value, returned by Value function.

        Prompt tester for a value, Test_Value.
        Test_SI := <Type>_Typecaster.Image(Test_Value);
        Interpret Test_SI as character string or integer.
        Show tester symbolic image, Test_SI, returned by Image function.

      -- Natural Image Testing
        Prompt tester for character string natural image, Test_NI.
        Check_NI := <Type>_Typecaster.Check(Test_NI);
        Show tester validity indicator, Check_NI, returned by Check function.
        Test_Value := <Type>_Typecaster.Value(Test_NI);
        Show tester value, Test_Value, returned by Value function.

        Prompt tester for a value, Test_Value.
        Test_NI := <Type>_Typecaster.Image(Test_Value);
        Show tester natural image, Test_NI, returned by Image function.

    exception
        when Constraint_Error | Data_Error =>
            Notify tester of invalid input.
        when End_Error =>
            Notify tester testing complete.
            exit;
        when Others =>
            Notify tester unknown failure has occurred.
            exit;
    end;
    end loop;

end <Type>_Typecaster_Test;
```

Figure 7-7: Discrete Typecaster Template Test PDL (Interactive)

7.2.3. Building Blocks: Composite Typecaster Templates

Composite typecaster templates are the expansion building blocks of the Typecaster model solution. They form the remaining layers of the solution (software architecture) and can be built upon instances of discrete typecaster templates or other instances of composite typecaster templates. This is shown abstractly in Figure 5-5 and via example in Figure 6-11. All composite typecaster templates contain an Ada package specification and body template based upon the Typecaster model described in Chapter 4, and a test procedure for testing instances of the composite typecaster template.

The Ada code for the composite typecaster templates is shown in Appendix Section C.4.

The composite typecaster provides a mapping between symbolic images or natural images and typed composite Ada values. To provide this mapping, composite typecasters use the functionality provided by typecasters previously created for the elements of the composite. The composite typecaster template exists for a number of reasons:

- To allow the designer to group both discrete and composite typecasters (and thus their symbolic images, natural images, values, and validity indicators) in a manner consistent with the EXR description.[22] The composite typecasters are therefore dependent upon other typecasters.
- Ada generic formal parameters cannot be of an Ada composite type. Thus, we created our own "generics" in the form of code templates. These templates both automate the production of code and perpetuate the consistencies from the discrete typecasters.
- The template facilitates testing of particular instances of the composite typecaster templates.

Figure 7-8 lists the composite typecaster templates and the appendix sections where the code templates can be found. Also included in Figure 7-8 are the name of an example instance of each composite typecaster template and the appendix sections where the example code can be found.

Composite Typecaster Template	**Appendix Section**	**Example Instance**	**Appendix Section**
Record Typecaster Template	C.4.1	Julian_Date_Time_Record_Typecaster	D.2.1
Private_Record Typecaster Template	C.4.2	FooBar_Message_Private_Record_Typecaster	D.2.2
Array Typecaster Template	C.4.3	Barrier_Segment_Array_Typecaster	D.4.3
Private_Array Typecaster Template	C.4.4	Probability_Private_Array_Typecaster	D.4.4
Wrapper Typecaster Template	C.4.5	Probability_Wrapper_Typecaster	D.4.5

Figure 7-8: Composite Typecaster Templates

[22]This grouping continues until all symbolic images are grouped and are in an order compatible with the UNR expected by the EXR TV model solution for a particular message.

Figures 7-9 and 7-10 show the composite *<Type>*_**Composite**_*Typecaster* package specification PDL. Figures 7-11 and 7-12 show the composite *<Type>*_**Composite**_*Typecaster* package body PDL. These are the first two of three separate code modules found in all composite typecaster templates. All *<Type>*_**Composite**_*Typecaster* package specifications and bodies found in the composite typecaster templates are identical except where the PDL is ***bold, italicized, and boxed***.

The composite *<Type>*_**Composite**_*Typecaster* packages provide the following abstract capabilities:[23]

- Convert from a symbolic image to typed composite Ada value.
- Convert from a typed composite Ada value to a symbolic image.
- Check a symbolic image for validity (i.e., can it be converted to a value?).
- Convert from a natural image to a typed composite Ada value.
- Convert from a typed composite Ada value to a natural image.
- Check a natural image for validity (i.e., can it be converted to a value?).
- An exception is raised if conversion from a symbolic image or natural image to a value fails.
- Typed composite Ada values are always valid (enforced by the Ada runtime) and can always be converted to a symbolic image or a natural image.
- Provide the positions of the elemental images within the composite image (for both symbolic and natural images).
- Provide the positions of the validity indicators for each elemental image within the validity indicator for the composite image (for both symbolic and natural images).

These abstractions make up the Typecaster model described in Chapter 4.

To instantiate a particular composite typecaster template, some information must be provided. The first is the name of the Ada composite type to be cast, *<Type>*_**Composite**_*Type*. This is a composite Ada type that is the INR of the information. The next piece of information is the names of the types, *<T1>_Type..<Tn>_Type* that the composite type *<Type>*_**Composite**_*Type* groups. These types (that are grouped by the composite typecaster being defined) must already be defined in the typecasters *<T1>_Typecaster..<Tn>_Typecaster* because the *<Type>*_**Composite**_*Type* inherits the types it groups from the corresponding typecasters. Finally, the names of the elements *<E1>..<En>* must be provided. These names are used to define:

- The elements of the composite type *<Type>*_**Composite**_*Type*. These represent objects of the types *<T1>_Type..<Tn>_Type* that are inherited from the typecasters *<T1>_Typecaster..<Tn>_Typecaster*.

[23]Note that these are basically the same as described in the previous two sections that described the discrete typecaster generics, and templates with the addition of some capabilities to handle elemental image positions.

- The names of the elements specified in the discrete type *<Type>_Element_Names*. These names are used to access:
  - Elements of the *Symbolic_Image*. These correspond to the *Symbolic_Image*s inherited from the typecasters *<T1>_Typecaster..<Tn>_Typecaster* via the *Image* function.
  - Elements of the *Natural_Image*. These correspond to the *Natural_Images* inherited from the typecasters *<T1>_Typecaster..<Tn>_Typecaster* via the *Image* function.
  - Elemental validity indicators corresponding to the *Symbolic_Image*. These correspond to the validity indicators inherited from the typecasters *<T1>_Typecaster..<Tn>_Typecaster* via the *Check* function.
  - Elemental validity indicators corresponding to the *Natural_Image*. These correspond to the validity indicators inherited from the typecasters *<T1>_Typecaster..<Tn>_Typecaster* via the *Check* function.

This information is expressed differently depending upon the composite typecaster template chosen. For example, creating the *Julian_Date_Time_Record_Typecaster* package (see Appendix Section D.2.1) involves supplying the names of all elements, and also the types of all elements grouped by the Ada record. On the other hand, the *Barrier_Segment_Array_Typecaster* package (see Appendix Section D.4.3) involves supplying first and last elements (i.e., an integer range) and only one type, the type of the elements grouped by the array. For more information on individual composite typecaster templates, see the composite typecaster templates in Appendix Section C.4. For more information on instantiating individual composite typecaster templates see Appendix Section B.4.

Based upon the composite type *<Type>_***<Composite>***_Type*, the names of the elements *<E1>..<En>*, the elemental typecasters *<T1>_Typecaster..<Tn>_Typecaster* that are grouped, and some information obtained from the package *Casting_Common_Types*, all composite typecaster templates provide the abstract functionality described above. The next thing to discuss is how the functionality is provided (i.e., implemented).

The *Symbolic_Image_Width* is defined as an integer constant. Its value is based on the sum of the *Symbolic_Image_Width*s inherited from the elemental typecasters, *<T1>_Typecaster..<Tn>_Typecaster*. *Symbolic_Image*, the type used to hold a symbolic image, is defined as an array of bytes of length *Symbolic_Image_Width*. The type that defines an array of bytes comes from *CCT.Byte_Array*. This promotes consistency among the typecasters.

Similarly, the *Natural_Image_Width* is defined as an integer constant. Its value is based on the sum of the *Natural_Image_Width*s inherited from the elemental typecasters *<T1>_Typecaster..<Tn>_Typecaster*. *Natural_Image*, the type used to hold a natural image, is defined and a character string of length *Natural_Image_Width*. The type that defines a character string comes from *Standard.String*. This promotes consistency among the typecasters.

The_Symbolic_Image_Positions is defined as a constant array that holds the starting and stopping positions of the elemental symbolic images and is accessed using the names of the elements specified by the discrete type *<Type>_Element_Names*. The starting and stopping positions of the elemental symbolic images are based on the progressive summing of the *Symbolic_Image_Width*s inherited from the elemental typecasters *<T1>_Typecaster..<Tn>_Typecaster*. The type that defines an array of positions comes from *CCT.Position_Array*. This promotes consistency among the typecasters.

Similarly, *The_Natural_Image_Positions* is defined as a constant array that holds the starting and stopping positions of the elemental natural images and is accessed using the names of the elements specified by the discrete type *<Type>_Element_Names*. The starting and stopping positions of the elemental natural images are based upon the progressive summing of the *Natural_Image_Width*s inherited from the elemental typecasters *<T1>_Typecaster..<Tn>_Typecaster*. The type that defines an array of positions comes from *CCT.Position_Array*. This promotes consistency among the typecasters.

Two *Value* functions exist that provide the capability to convert from a symbolic image or natural image t? a composite value. The basic underlying algorithm for the symbolic image *Value* function is shown in Figure 7-11, and the algorithm for the natural image *Value* function is shown in Figure 7-12. Both algorithms work the same and can be summarized as follows: For each element *<En>*, slice the elemental image out of the composite image using the known positions of the elemental images. Convert this image slice to a value using the *Value* function inherited from the elemental typecaster and assign the value to the element of the composite type. Finally, if any call to an elemental typecaster *Value* function results in the *Constraint_Error* exception being raised, the exception is propagated out.

Two *Image* functions exist that provide the capability to convert from a composite value to a symbolic image or natural image. The basic underlying algorithm for the symbolic image *Image* function is shown in Figure 7-11, and the algorithm for the natural image *Image* function is shown in Figure 7-12. Both algorithms work the same and can be summarized as follows: For each element *<En>* of the composite type, convert the value to an image using the *Image* function inherited from the elemental typecaster. Assign the elemental image to a slice of the composite image using the known positions of the elemental images. Finally, because of the strong typing provided by Ada, the solution assumes that values are always valid, therefore the *Image* function should never fail.

Two *Check* functions exist that provide the capability to check the validity of a symbolic image or a natural image. The basic underlying algorithm for the symbolic image *Check* function is shown in Figure 7-11, and the algorithm for the natural image *Check* function is shown in Figure 7-12. Both algorithms work the same and can be summarized as follows: Attempt to convert the image to a value using the composite *Value* function. If no *Constraint_Error* is raised, then the image is valid so return a valid image indicator. If a *Constraint_Error* is raised, handle it. This implies the image is invalid and th› invalid image indicator must be built from the elemental validity indicators. This is done via the following steps: For each element *<En>*, slice the elemental image out of the composite image using the known positions of the elemental images. Obtain a validity indicator for the elemental image by applying the *Check* function inherited from the elemental typecaster. Place the elemental validity indicator in the composite validity indicator, again using the known positions of the elemental images.

The validity indicators returned by the *Check* functions are defined as *Symbolic_Image* or *Natural_Image* constants. The composite valid image indicators *Valid_Symbolic_Image* and *Valid_Natural_Image* are defined as the concatenation of the elemental valid image indicators inherited from the elemental typecasters *<T1>_Typecaster..<Tn>_Typecaster*. Invalid image indicators are not defined, but instead they are built. They are built by concatenating the validity indicators for each elemental image. See the description of the *Check* function above for more details.

Finally, *Image* and *Value* functions are inlined for efficiency reasons.

The private composite typecasting templates provide additional capabilities to enforce inter-element dependencies imposed by the textual EXR description. Figure 7-13 shows the parts of the PDL of the package specification portion of the *<Type>_Private_***Composite_***Typecaster* template that are needed to provide the additional capabilities.

The *<Type>_Private_***Composite_***Typecaster* packages provide the following additional capabilities:

- Convert composite values from an accessible form whose data are available to the application to a protected form where inter-element dependencies are checked and preserved.
- Convert composite values from a protected form where inter-element dependencies are checked and preserved to an accessible form whose data are available to the application.

To instantiate a particular private composite typecaster template, one must provide the body of the function *Is_Consistent*. This function must provide the inter-element consistency checking based upon the textual EXR description.

The following are the additional constants, types, and functions defined in the private composite typecaster templates.

*<Type>_Public_***Composite_***Type* is the Ada composite type to be cast. *<Type>_Private_***Composite_***Type* is the private version of the *<Type>_Public_***Composite_***Type*. Its purpose is to encapsulate and hide the information from the application and maintain data integrity, i.e., ensure inter-element consistency.

Inconsistent_Symbolic_Image is defined as a *Symbolic_Image* constant. It is used to hold the indicator returned by the *Check* function when inter-element inconsistencies in the symbolic image are detected. Its value is defined by *CCT.Inconsistent_Symbolic_Image* and its width is *Symbolic_Image_Width*.

Inconsistent_Natural_Image is defined as a *Natural_Image* constant. It is used to hold the indicator returned by the *Check* function when inter-element inconsistencies in the natural image are detected. Its value is defined by *CCT.Inconsistent_Natural_Image* and its width is *Natural_Image_Width*.

The *Make_Public* function provides the capability to convert from a value of type *Private_***Composite***_Type* to a value of type *Public_***Composite***_Type*, that is, to convert from the protected form to the accessible form. The basic underlying algorithm is to do an explicit type conversion from the private type to the public type.

The *Make_Private* function provides the capability to convert from a value of type *Public_***Composite***_Type* to a value of type *Private_***Composite***_Type*, that is, to convert from the accessible form to the protected form. The basic underlying algorithm is to first do an explicit type conversion to get the *Private_***Composite***_Type* value and then check the inter-element dependencies via the *Is_Consistent* function. If the elements are not consistent then a *Constraint_Error* is raised.

The basic underlying algorithms involved in these functions are shown in Figure 7-14.

Figures 7-15 and 7-16 show the PDL for the symbolic image and natural image portions of the *<Type>_Typecaster_Test* procedure. This is the third of three separate code modules found in all composite typecaster templates. The test procedures are very similar. The parts that vary most deal with the I/O portions of the test procedure. To signify those parts of the test procedures that deal with communication with the tester, the PDL is **bolded**.

The test procedure, *<Type>_***Composite***_Typecaster_Test*, tests the instance of the *<Type>_***Composite***_Typecaster* package generated during template instantiation. It uses predefined test cases based on symbolic images specified by the detailed designer when the instance of the template was created. The detailed designer also specifies whether each symbolic image test case is valid or invalid, so the test routine can determine if the expected results were obtained. The entire range of functionality of the typecaster is tested based on the symbolic image test cases. This type of testing is considered canned testing.

Because the typecasters perform translation and validation based on the relationship between either symbolic images and values, or natural images and values, the canned testing is broken up into two parts. The first part tests translation and validation between the predefined test symbolic image and value. The second part tests translation and validation between the resulting value and natural image.

The following steps are performed for canned testing:

1. Apply *Check* function to predefined test symbolic image, i.e., check the image for validity.
2. Apply *Value* function to predefined symbolic image, i.e., convert the image to a value.
3. If *Value* function raises a constraint error and the test case was specified as valid, then notify tester of symbolic image test case failure.

4. Apply *Image* function, i.e., convert the value to a symbolic image.
5. If the resulting symbolic image is not the predefined test symbolic image started with (i.e., from Step 1), then notify tester of symbolic image test case failure.
6. Apply *Image* function to value from Step 2, i.e., convert the value to a natural image.
7. Apply *Check* function, i.e., check the natural image for validity.
8. Apply *Value* function, i.e., convert the natural image to a value.
9. If the resulting value is not the value started with (i.e., from Step 6), then notify tester of natural image test case failure.
10. If *Value* function raises a constraint error and the test case was specified as valid, then notify tester of natural image test case failure.

This form of canned testing allows testing based on valid or invalid symbolic images. It allows testing of valid natural images but does not allow testing of invalid natural images. The specification of the validity of each individual test case allows the test procedure to pass, even though the typecaster may fail because of an invalid test case.

```
with Casting_Common_Types;
--Obtain access to the appropriate discrete or composite typecasters
with <T1>_Typecaster; ... with <Tn>_Typecaster;
package <Type>_Composite_Typecaster is
    package CCT renames Casting_Common_Types;
  -- Internal Representation & Internal Representation Description
                                     Based on the elements <E1>..<En> and
    type <Type>_Composite_Type is their associated types <T1>_Type..<Tn>_Type ;
    -- Names of the elements
    type <Type>_Element_Names is <E1> ... <En> ;

  -- Universal Representation and Universal Representation Description
    -- Symbolic image size and string to hold it.
    Symbolic_Image_Width : constant Integer :=
        <T1>_Typecaster.Symbolic_Image_Width +
        <T2>_Typecaster.Symbolic_Image_Width +
        ...
        <Tn>_Typecaster.Symbolic_Image_Width;
    subtype Symbolic_Image is CCT.Byte_Array(1..Symbolic_Image_Width);

    -- Functions for converting between a symbolic image and a value.
    -- Also a function for checking the validity of a symbolic image.
    function Value(Image_In : Symbolic_Image) return <Type>_Type;
    function Image(Value_In : <Type>_Type) return Symbolic_Image;
    function Check(Image_In : Symbolic_Image) return Symbolic_Image;

    -- Symbolic image validity indicators returned by Check function
    Valid_Symbolic_Image : constant Symbolic_Image :=
        <T1>_Typecaster.Valid_Symbolic_Image &
        <T2>_Typecaster.Valid_Symbolic_Image &
        ...
        <Tn>_Typecaster.Valid_Symbolic_Image;
    Inconsistent_Symbolic_Image : constant Symbolic_Image :=
        CCT.Inconsistent_Symbolic_Image(1..Symbolic_Image_Width);

    -- Pointers to the position of each elemental symbolic image
    The_Symbolic_Image_Positions : constant
        CCT.Position_Array(<Type>_Element_Names) :=
        (<E1> => (Start => 0 + 1,
                  Stop =>  0 + <E1>_Typecaster.Symbolic_Image_Width),
         <E2> => (Start => The_Symbolic_Image_Positions(<E1>) + 1,
                  Stop => The_Symbolic_Image_Positions(<E1>) +
                          <E2>_Typecaster.Symbolic_Image_Width),
         ...
         <En> => (Start => The_Symbolic_Image_Positions(<En-1>) + 1,
                  Stop => The_Symbolic_Image_Positions(<En-1>) +
                          <En>_Typecaster.Symbolic_Image_Width));
```

Figure 7-9: Composite Typecaster Template Package Spec PDL - Part 1

```
-- User Representation and User Representation Description
  -- Natural image size and string to hold it
  Natural_Image_Width : constant Integer :=
      <T1>_Typecaster.Natural_Image_Width +
      <T2>_Typecaster.Natural_Image_Width +
      ...
      <Tn>_Typecaster.Natural_Image_Width;
  subtype Natural_Image is CCT.Byte_Array(1..Natural_Image_Width);

  -- Functions for converting between a natural image and a value.
  -- Also a function for checking the validity of a natural image.
  function Value(Image_In : Natural_Image) return <Type>_Type;
  function Image(Value_In : <Type>_Type) return Natural_Image;
  function Check(Image_In : Natural_Image) return Natural_Image;

  -- Natural image validity indicators returned by Check function
  Valid_Natural_Image : constant Natural_Image :=
      <T1>_Typecaster.Valid_Natural_Image &
      <T2>_Typecaster.Valid_Natural_Image &
      ...
      <Tn>_Typecaster.Valid_Natural_Image;
  Inconsistent_Natural_Image : constant Natural_Image :=
      CCT.Inconsistent_Natural_Image(1..Natural_Image_Width);


  -- Pointers to the position of each elemental natural image
  The_Natural_Image_Positions : constant
      CCT.Position_Array(<Type>_Element_Names) :=
      (<E1> => (Start => 0 + 1,
                Stop => 0 + <E1>_Typecaster.Natural_Image_Width),
       <E2> => (Start => The_Natural_Image_Positions(<E1>) + 1,
                Stop => The_Natural_Image_Positions(<E1>) +
                        <E2>_Typecaster.Natural_Image_Width),
       ...
       <En> => (Start => The_Natural_Image_Positions(<En-1>) + 1,
                Stop => The_Natural_Image_Positions(<En-1>) +
                        <En>_Typecaster.Natural_Image_Width));

-- Real-Time constraint raised by Value function
  Constraint_Error : Exception;
  pragma inline(Value, Image);
end <Type>_Composite_Typecaster;
```

Figure 7-10: Composite Typecaster Template Package Spec PDL - Part 2

```
package body <Type>_ Composite _Typecaster is
function Value(Image_In : Symbolic_Image) return <Type>_Type is
    Return_Value : <Type>_Type;
begin
    <E1> of Return_Value := <T1>_Typecaster.Value(
                   Image_In(The_Symbolic_Image_Positions(<E1>).Start ..
                            The_Symbolic_Image_Positions(<E1>).Stop));
    ...
    <En> of Return_Value := <Tn>_Typecaster.Value(
                   Image_In(The_Symbolic_Image_Positions(<En>).Start ..
                            The_Symbolic_Image_Positions(<En>).Stop));
    return Return_Value;
end Value;

function Image(Value_In : <Type>_Type) return Symbolic_Image is
    Return_Image : Symbolic_Image;
begin
    Return_Image(The_Symbolic_Image_Positions(<E1>).Start ..
                 The_Symbolic_Image_Positions(<E1>).Stop) :=
      <T1>_Typecaster.Image(<E1> of Value_In);
    ...
    Return_Image(The_Symbolic_Image_Positions(<En>).Start ..
                 The_Symbolic_Image_Positions(<En>).Stop ) :=
      <Tn>_Typecaster.Image(<En> of Value_In);
    return Return_Image;
end Image;

function Check(Image_In : Symbolic_Image) return Symbolic_Image is
    Return_Check : Symbolic_Image;
    Return_Value : <Type>_ Composite _Type;
begin
    Return_Value := Value(Image_In);
    return Valid_Symbolic_Image;
exception
    when Constraint_Error =>
    Return_Check(The_Symbolic_Image_Positions(<E1>).Start ..
                 The_Symbolic_Image_Positions(<E1>).Stop) :=
      <T1>_Typecaster.Check(Image_In(The_Symbolic_Image_Positions(<E1>).Start ..
                                     The_Symbolic_Image_Positions(<E1>).Stop));
    ...
    Return_Check(The_Symbolic_Image_Positions(<En>).Start ..
                 The_Symbolic_Image_Positions(<En>).Stop) :=
      <Tn>_Typecaster.Check(Image_In(The_Symbolic_Image_Positions(<En>).Start ..
                                     The_Symbolic_Image_Positions(<En>).Stop));
    return Check_Image;
end Check;
```

Figure 7-11: Composite Typecaster Template Package Body PDL - Part 1

```
function Value(Image_In : Natural_Image) return <Type>_Type is
    Return_Value : <Type>_Type;
begin
    <E1> of Return_Value := <T1>_Typecaster.Value(
                 Image_In(The_Natural_Image_Positions(<E1>).Start ..
                          The_Natural_Image_Positions(<E1>).Stop));
    ...
    <En> of Return_Value := <Tn>_Typecaster.Value(
                 Image_In(The_Natural_Image_Positions(<En>).Start ..
                          The_Natural_Image_Positions(<En>).Stop));
    return Return_Value;
end Value;

function Image(Value_In : <Type>_Type) return Natural_Image is
    Return_Image : Natural_Image;
begin
    Return_Image(The_Natural_Image_Positions(<E1>).Start ..
                 The_Natural_Image_Positions(<E1>).Stop) :=
      <T1>_Typecaster.Image(<E1> of Value_In);
    ...
    Return_Image(The_Natural_Image_Positions(<En>).Start ..
                 The_Natural_Image_Positions(<En>).Stop) :=
      <Tn>_Typecaster.Image(<En> of Value_In);
    return Return_Image;
end Image;

function Check(Image_In : Natural_Image) return Natural_Image is
    Return_Check : Natural_Image;
    Return_Value : <Type>_Composite_Type;
begin
    Return_Value := Value(Image_In);
    return Valid_Natural_Image;
exception
    when Constraint_Error =>
    Return_Check(The_Natural_Image_Positions(<E1>).Start ..
                 The_Natural_Image_Positions(<E1>).Stop) :=
      <T1>_Typecaster.Check(Image_In(The_Natural_Image_Positions(<E1>).Start ..
                                     The_Natural_Image_Positions(<E1>).Stop));
    ...
    Return_Check(The_Natural_Image_Positions(<En>).Start..
                 The_Natural_Image_Positions(<En>).Stop) :=
      <Tn>_Typecaster.Check(Image_In(The_Natural_Image_Positions(<En>).Start ..
                                     The_Natural_Image_Positions(<En>).Stop));
    return Check_Image;
end Check;
end <Type>_Composite_Typecaster;
```

Figure 7-12: Composite Typecaster Template Package Body PDL - Part 2

```
package <Type>_Private_[Composite]_Typecaster is

  -- Internal Representation & Internal Representation Description
    type <Type>_Private_[Composite]_Type is private;
                                             [Based on the elements <E1>..<En>
                                              and their associated
    type <Type>_Public_[Composite]_Type is    types <T1>_Type..<Tn>_Type      ];

          .
          .
          .

    -- Symbolic image validity indicators returned by Check function
    Inconsistent_Symbolic_Image : Constant Symbolic_Image :=
        CCT.Inconsistent_Symbolic_Image(1..Symbolic_Image_Width);
    Inconsistent_Natural_Image : Constant Natural_Image :=
        CCT.Inconsistent_Natural_Image(1..Natural_Image_Width);

    -- Functions for converting between the <Type>_Private_[Composite]_Type
    -- and the <Type>_Public_[Composite]_Type types.
    function Make_Public (Private_Value_In: <Type>_Private_[Composite]_Type);
    function Make_Private (Public_Value_In: <Type>_Public_[Composite]_Type);

private
   type <Type>_Private_[Composite]_Type is new
    <Type>_Public_[Composite]_Type;

end <Type>_Private_[Composite]_Typecaster;
```

Figure 7-13: Private Composite Specification PDL

```
package <Type>_Private_ Composite _Typecaster body is

    function Is_Consistent (Value_In: <Type>_Private_ Composite _Type)
        return Boolean is
    begin
        The inter-element dependencies of the composite type are checked
        here. If an inconsistency is found false is returned, otherwise true is
        returned.
    end Is_Consistent;

    function Make_Public (Private_Value_In : <Type>_Private_ Composite _Type)
        return <Type>_Public_ Composite _Type is
    begin
        return <Type>_Public_ Composite _Type(Private_Value_In);
    end Make_Public;

   function Make_Private (Public_Value_In : in <Type>_Public_ Composite _Type)
        return <Type>_Private_ Composite _Type is

        Private_Value : <Type>_Private_ Composite _Type;
    begin
        Private_Value := <Type>_Private_ Composite _Type(Public_Value_In);
        if Is_Consistent(Private_Value) then
            return Private_Value;
        else
            raise Constraint_Error;
        end if;
    end Make_Private;

end <Type>_Private_ Composite _Typecaster;
```

Figure 7-14: Private Composite Body PDL

```
with <Type>_ Composite _Typecaster;
with Unchecked_Conversion;
with Casting_Common_Types;
with Text_IO;
procedure <Type>_ Composite _Typecaster_Test is
    N : integer := ??;          -- number of test cases
    type Test_Case_Type is
    record
        Test_Image : <Type>_ Composite _Typecaster.Symbolic_Image;
        Valid      : Boolean;
    end record;

    Test_Cases : array(1..N) of Test_Case_Type :=
        (1=> (Test_Image => "??",
              Valid      =>  ??),
        ...
         N=> (Test_Image => "??",
              Valid      =>  ??));
begin

    -- Canned Testing Loop
    for I in Test_Cases'First .. Test_Cases'Last loop

      -- Symbolic Image Testing
      begin
        -- start symbolic image testing with the test symbolic image
        Check_SI := <Type>_Typecaster.Check(Test_Cases(I).Test_Image);
        Test_Value_SI := <Type>_Typecaster.Value(Test_Cases(I).Test_Image);
        Test_SI := <Type>_Typecaster.Image(Test_Value_SI);

        -- if symbolic image we started with is not what we have now, FAIL
        if Test_SI /= Test_Cases(I).Test_Image then
            raise Unknown_Symbolic_Failure;
        end if;

      exception
        when Constraint_Error =>
            if Test_Cases(I).Valid then
                Notify tester a valid test case has failed in symbolic image testing
                Show tester Test_Cases(I).Test_Image that failed
                Show tester validity indicator Check_SI
            end if;
        when Unknown_Symbolic_Failure =>
            Notify tester an unknown failure has occurred in symbolic image testing
            Show tester Test_Cases(I).Test_Image that failed
      end;
```

Figure 7-15: Composite Typecaster Template, Test PDL for Symbolic Images

```
    -- Natural Image Testing
    begin

      -- start natural image testing with value from symbolic image testing
      Test_NI := <Type>_Typecaster.Image(Test_Value_SI);
      Check_NI := <Type>_Typecaster.Check(Test_NI);
      Test_Value_NI := <Type>_Typecaster.Value(Test_NI);

      -- if value we started with is not what we have now, FAIL
      if Test_Value_NI /= Test_Value_SI then
          raise Unknown_Natural_Failure;
      end if;

    exception
      when Constraint_Error =>
          if Test_Cases(I).Valid then
              Notify tester a valid test case has failed in natural image testing
              Show tester Test_Cases(I).Test_Image that failed
              Show tester validity indicator Check_NI
          end if;
      when Unknown_Natural_Failure =>
          Notify tester an unknown failure has occurred in natural image testing
          Show tester Test_Cases(I).Test_Image that failed
    end;
  end loop;

  if Errors occurred then
      Notify tester canned testing completed unsuccessfully
  else
      Notify tester canned testing completed successfully
  end if;

end <Type>_Composite_Typecaster_Test;
```

Figure 7-16: Composite Typecaster Template, Test PDL for Natural Images

7.3. EXR TV Model Solution

The EXR TV model provides the functionality described in Chapter 4. In this section the acronym ICD (i.e., Interface Control Document) is used in the same context as EXR descriptions. An ICD contains textual EXR descriptions, and ICD formatted messages are EXRs.

7.3.1. Foundation Utilities: Field_Utilities Package

The Field_Utilities package provides the capability to convert between an EXR and a UNR of a field (of a message). The following functionality is provided:

- Extract a character or bit-based field from an EXR, based upon the EXR description, and insert it into a UNR.
- Extract an element from a UNR and insert it into a character or bit-based field in an EXR, based upon an EXR description.
- Convert from an EXR ordering of information to a UNR ordering of information.
- Convert from a UNR ordering of information to an EXR ordering of information.
- Check an EXR at the indicated position for a specified separator based on an EXR description.
- Check an EXR at the indicated position for an end-of-message indicator.

The procedure *Extract_Field_Image_From_ICD* provides the capability to extract character-based or bit-based fields from EXRs and insert them into UNRs. This is accomplished by slicing the field out of the appropriate positions of the EXR based upon the EXR description of the field, and inserting it into the appropriate elemental slice of the UNR based upon the UNR description.

The procedure *Insert_Field_Image_Into_ICD* provides the capability to extract character-based or bit-based elements from UNRs and insert them into EXRs. This is accomplished by slicing the elemental symbolic image out of the UNR, based on the UNR description, and inserting it into the appropriate field positions of the EXR, based upon the EXR description of the field.

The procedure *Put_Array_Image_In_Element_Order* provides the capability to rearrange selected fields in a symbolic image from a field-ordered to an element-ordered orientation. The procedure is passed the symbolic image in field-order, the number of elements in each field, and the size of the elements in each field. The reordering is accomplished in two loops. An inner loop extracts, one at a time, an elemental image from each field in the symbolic image and inserts it into an element-ordered symbolic image. The outer loop iterates until all of the field-ordered elements have been extracted and placed in the element-ordered symbolic image.

The procedure *Put_Array_Image_In_Field_Order* provides the capability to rearrange selected fields in a symbolic image from an element-ordered to a field-ordered orientation. The procedure is passed the symbolic image in elemental-order, the number of elements in each field, and the size of the elements in each field. The reordering is accomplished in two loops. An inner loop extracts one at a time an elemental image for each field in the symbolic image and inserts it into a field-ordered symbolic image. The outer loop iterates until all of the element-ordered elements have been extracted and placed in the field-ordered symbolic image.

The function *Separator_Is_In_Position* checks an EXR at the indicated position for a specified separator based on an EXR description.

The function *EO_Text_Is_Valid* checks an EXR at the indicated position for a specified separator based on an EXR description. It checks to see if the end of the EXR has been reached and also checks if the indicated Eo_Text marker for the EXR is at the indicated position.

7.3.2. Foundation Utilities: ICD_Utilities Generic Package

The ICD_Utilities generic package provides the capability to convert between an EXR and a UNR of a message. The following functionality is provided:

- Construct an EXR of a message from a UNR.
- Construct a UNR of a message from an EXR while checking the EXR for syntactic validity based on the EXR description.
- Check an EXR of a message for validity based upon the EXR description.

The function *Construct_ICD* constructs an EXR from a UNR (symbolic image). This is accomplished by making any necessary cuts to the UNR and then inserting the UNR of each field into the EXR one at a time based upon the EXR and UNR descriptions. After all of the fields have been inserted into the EXR, an end-of-text marker is added.

The function *Extract_Universal_Image* constructs the UNR (symbolic image) from an EXR. This is accomplished by extracting each field one at a time from the EXR and inserting it into a UNR based on the EXR and UNR descriptions. Once the last field has been extracted, and if the the end of the EXR has been reached, then any necessary cuts are performed. If end of text has not been reached, then a Constraint_Error is explicitly raised.

The procedure *Check_ICD* checks an EXR to see if it is syntactically valid based on the EXR description stored in *Fields*. This is accomplished by calling the Extract_Universal_Image function. If the UNR is extracted from the EXR with no problems, then it is valid. If, however, a Constraint_Error is raised, then each field of the EXR is reprocessed individually until the invalid field in the EXR is located. A pointer indicating where the bad field is located represents the validity indicator and is returned.

7.3.3. Building Block: Message_ICD Template

This is the building block for the EXR Translation and Validation model solution. The Message_ICD template contains an Ada package specification and body based upon the EXR Translation and Validation model described in Chapter 4, and a test procedure for testing instances of the Message_ICD template.

The Ada code for the message ICD template is shown in Appendix Section C.5.

The Message_ICD template provides a mapping between the EXRs and UNRs of a message. The Message_ICD template exists for a number of reasons:

- To facilitate the generation of code to convert between EXRs and UNRs.
- To aid the designers in codifying the EXR of a message in a form expected by the model solution.
- To facilitate the testing of instances of the Message_ICD template.

Two example instances of the Message_ICD template exist. These are the *FooBar_Message_ICD*, found in Appendix Section D.3.1 and the *FooBar_Bit_Message_ICD*, found in Appendix Section D.3.2.

Figure 7-17 shows the <Msg_Id>_ICD Ada package specification template PDL. Figure 7-18 shows the <Msg_Id>_ICD Ada package body template PDL. These are two of three separate code modules found in the Message_ICD template.

The <Msg_Id>_ICD packages provide the following abstract capabilities:

- Convert from an EXR to a UNR of a message.
- Convert from a UNR to an EXR of a message.
- Check an EXR for syntactic validity based upon the EXR description.
- An exception is raised if conversion from an EXR to UNR fails.
- Provide descriptions of the fields within the EXR of a message.
- Provide positions of the elements within the UNR of a message.

The abstractions make up the EXR TV model described in Chapter 4.

To instantiate the Message_ICD template, some information must be provided. The first is the name of the message. This is used to name the package, *<Msg_Id>_ICD*. The next piece of information is names of the fields, *<N1>..<Nn>*, taken from textual EXR description, followed by the width of the symbolic image, *<Symbolic_Image_Width>*. Last is the number of cuts, *Number_Of_Cuts*, that need to be performed on a message.

The *Symbolic_Image_Width* is defined as an integer constant. Its value is based on the textual EXR description of the message, with all of the separators stripped out. *Symbolic_Image* (UNR) is the type used to hold a symbolic image and is defined as an array of bytes of length *Symbolic_Image_Width*. The type that defines an array of bytes comes from *CCT.Byte_Array*.

EXRs are stored using the *CCT.Icd_Message_Type* type. *CCT.Icd_Message_Type* is a record that consists of a string for holding the EXR of the message and an integer for holding the message length.

The EXR description is captured formally in *Fields*. *Fields* is of type *Icd_Util.Description_Array* and consists of an array of records, each of which describes one field. *Fields* is indexed by the field names that are enumerated in the *Field_Names* type, given by *<N1>..<Nn>*. The symbolic image description (UNR description) is also captured formally in *Fields*. This description is based on the EXR description, where punctuation has been stripped out and varying length fields padded to their maximum width.

Eo_Text is defined as a constant of type *CCT.Little_String_Type*, and represents the end-of-text marker for an EXR of a message. Some predefined values can be found in *CCT*.

Cuts is defined as a variable of type *ICD_Util.Cut_Array*, and it holds the starting and stopping positions of the cuts that need to be made to a message. The information for defining *Cuts* is obtained by examining the textual EXR description and the INR selected.

The function *Construct_Icd* provides the capability to convert from a symbolic image to an EXR. The basic underlying algorithm is to first perform any cuts that need to be made to the element-ordered symbolic image. After the cuts have been made, the elements are then extracted from the field-ordered symbolic image and inserted into the EXR one at a time. After all the elements have been inserted into the message, the end-of-text marker is added to the end of the message and the EXR is returned.

The function *Extract_Universal_Image* provides the capability to convert from an EXR to a symbolic image. The basic underlying algorithm is to extract the fields from the EXR and insert them into a field-ordered symbolic image. When the current field being processed can be the last field and the end-of-text marker is found, then any cuts that need to be done are performed. Following the completion of the cuts the element-ordered symbolic image is returned.

The procedure *Check_Icd* provides the capability to check an EXR for syntactic validity based upon the EXR description in *Fields*. The basic underlying algorithm is to pass the EXR to the *Extract_Universal_Image* to check if a symbolic image can be extracted from it. If the EXR does not contain a valid symbolic image, a Constraint_Error is trapped and the EXR is parsed to find the position in the message where the error occurs. The status of the message and the position where the message goes bad are reported back by the procedure.

Figure 7-19 shows the PDL for the <Msg_Id>_ICD_Test procedure. This is the third of three separate code modules found in the Message_ICD template. The I/O portions of the code that deal with notifying the tester of some error with the test are **bolded**.

The test procedure, <Msg_Id>_ICD_Test, tests the instance of the <Msg_Id>_ICD package generated during template instantiation. It uses predefined test cases based on EXRs specified by the detailed designer when the instance of the template was created. The detailed designer also specifies whether each EXR test case is valid or invalid, and if invalid, also specifies the position where the message is invalid, so that the test routine can determine if the expected results were obtained. The entire range of functionality provided by the EXR TV for the message is tested based on the EXR test cases. This type of testing is considered canned testing.

The following steps are performed for canned testing:

1. Apply *Check_ICD* procedure to predefined test EXR, i.e., check the EXR for validity.
2. If *Check_ICD* procedure reports the validity as not equal to the specified validity of the test case, or if the test case is invalid and the specified bad position is not equal to the returned bad position, then notify the tester of EXR test case failure.
3. Apply the *Extract_Universal_Image* function to predefined test EXR, i.e., convert the EXR to a symbolic image.
4. If *Extract_Universal_Image* function raises a constraint error then notify the tester of EXR test case failure.
5. Apply the *Construct_ICD* function to the symbolic image from Step 3, i.e., convert the symbolic image to an EXR.
6. If the resulting EXR is not equal to the original EXR test case, then notify the tester of EXR test case failure.

This form of canned testing allows testing based on a valid or invalid EXR. The specification of the validity of each individual test case allows the test procedure to pass even though the EXR TV may fail because of an invalid test case.

These templates are used by the detailed designer to build an instance of the EXR TV model solution for a particular message. The implementation the template provides must be understood when modifications or enhancements to the model solution are necessary.

```
with Casting_Common_Types;
with ICD_Utilities;

package <Msg_Id>_ICD is

    package Cct renames Casting_Common_Types;

    Symbolic_Image_Width : constant Integer := <Symbolic_Image_Width>;

    -- Names of the fields
    type Field_Names is (<N1>, <N2>, ...);

    Number_Of_Cuts : constant Integer := <Number_Of_Cuts>;

    -- To obtain operations to convert between the UNR (symbolic image)
    -- and the EXR (ICD format) of a message.
    package ICD_Util is new ICD_Utilities(
        Symbolic_Image_Width => Symbolic_Image_Width,
        Field_Names => Field_Names,
        Number_Of_Cuts => Number_Of_Cuts);

    subtype Symbolic_Image is ICD_Util.Symbolic_Image;

    -- End of text marker.
    Eo_Text : constant Cct.Little_String_Type := CCT.<EOT>;

    Fields : ICD_Util.Description_Array := | Based on the textual EXR description |;

    -- Pointers to the cut positions for the ICD message
                                          | Based on info given in the textual EXR |
    Cuts : ICD_Util.Cut_Array :=          | description and the INR description    |;

    -- Functions for converting between a UNR (symbolic image) and
    -- an EXR (ICD formatted message).
    -- Also a procedure for checking the validity of an EXR.
    function Construct_ICD(Universal_Image : in Symbolic_Image)
                           return Cct.Icd_Message_Type;
    function Extract_Universal_Image(Icd : in Cct.Icd_Message_Type)
                           return Symbolic_Image;
    procedure Check_Icd(Icd : in Cct.Icd_Message_Type;
                        Icd_Id_Ok : out Boolean;
                        Bad_Position : out Integer);

    pragma inline (Construct_Icd, Extract_Universal_Image);

end <Msg_Id>_ICD;
```

Figure 7-17: Message ICD Template Package Spec PDL

```
package body <Msg_Id>_ICD is

    function Construct_Icd (Universal_Image : in Symbolic_Image)
        return Cct.Icd_Message_Type is
    begin
        return ICD_Util.Construct_Icd (
                   Universal_Image => Universal_Image,
                   Fields => Fields,
                   Cuts => Cuts,
                   Eo_Text => Eo_Text);
    end Construct_Icd;

    function Extract_Universal_Image (Icd : in Cct.Icd_Message_Type)
        return Symbolic_Image is
    begin
        return ICD_Util.Extract_Universal_Image(
                   Icd => Icd,
                   Fields => Fields,
                   Cuts => Cuts,
                   Eo_Text => Eo_Text);
    end Extract_Universal_Image;

    procedure Check_Icd (Icd : Cct.Icd_Message_Type;
                         Icd_Is_Ok : out Boolean;
                         Bad_Position : out Integer) is
    begin
        ICD_Util.Check_Icd (
                   Icd => Icd,
                   Fields => Fields,
                   Cuts => Cuts,
                   Eo_Text => Eo_Text,
                   Icd_Is_Ok => Icd_Is_Ok,
                   Bad_Position => Bad_Position);
    end Check_Icd;

begin

    ICD_Util.Initialize(Fields, Cuts, Symbolic_Image_Width);

end <Msg_Id>_ICD;
```

Figure 7-18: Message ICD Template Package Body PDL

```
with <Msg_Id>_Icd;
with Text_Io;
with Casting_Common_Types;
with Unchecked_Conversion;
Procedure <Msg_Id>_Icd_Test is
    package Cct renames Casting_Common_Types;
    package Byte_Io is new Text_Io.Integer_Io(Cct.Byte);

    type Test_Case_Type is record
        Test_Icd     : Cct.Icd_Message_Type;
        Bad_Position: Integer;
        Valid        : Boolean;
    end record;

    N : Integer := <Number_Of_Test_Cases>;
    Test_Cases : array (1..N) of Test_Case_Type := Test case descriptions ;
begin
    -- Canned Testing Loop: loop through test cases
    for I in Test_Cases'First .. Test_Cases'Last loop
    begin
        <Msg_Id>_Icd.Check_Icd(Test_Cases(I).Test_Icd,
                               Resulting_Check,
                               Position_Pointer);

        If Test_Case(I).Valid /= Resulting_Check then
            raise Unknown_Error;
        elsif not Test_Cases(I).Valid then
            if Test_Cases(I).Bad_Position /= Position_Pointer then
                raise Unknown_Error;
            end if;
        else
            Test_UI := <Msg_Id>_Icd.Extract_Universal_Image(
                                          Test_Cases(I).Test_Icd);
            Resulting_Icd := <Msg_Id>_Icd.Construct_Icd(Test_UI);
            if Resulting_Icd /= Test_Cases(I).Test_Icd then
                raise Unknown_Failure;
            end if;
        end if;

    exception
        when Unknown_Error =>
            if Test_Cases(I).Valid then
                Notify tester a valid test case has failed
                Show tester Test_Cases(I).Test_Icd that failed
            end if;
        when others =>
            Notify tester a test case has failed and for an unknown reason
            Show tester Test_Cases(I).Test_Icd that failed
    end;
    end loop;
end <Msg_Id>_Icd_Test;
```

Figure 7-19: Message ICD Template Test PDL

8. MTV Model Solution Adaptation Description

This chapter discusses issues related to enhancing or modifying the MTV model solution. Three classes of modifications are addressed:

- Section 8.1 addresses whole-scale replacement of either the EXR TV or the Typecaster model solution.
- Section 8.2 addresses enhancement and extension of either the EXR TV or the Typecaster model solution.
- Section 8.3 addresses alternative software architectures (i.e., reorganization of component parts) for the Typecaster model solution.

This chapter is targeted for the maintainer and model adapter. It is expected that the maintainer and adapter have read Chapters 1 through 7, thus sufficiently understanding:

- what concepts are embodied in and provided by the model
- how to apply the model solution
- how the model is implemented

8.1. Replacing Whole Parts of the MTV Model Solution

It is important to remember that the MTV model solution consists of two parts:

1. the EXR TV model
2. the Typecaster model

These two models share a common interface, the UNR. Because of this common interface, either model solution can be replaced with another solution as long as the interface stays the same.

Therefore, if a table-driven parsing approach is better suited to the EXRs found in a message set or the table-driven approach shows better performance based upon the message set, then the EXR TV model solution could be replaced with a table-driven solution provided the table-driven solution can supply UNRs to the Typecaster model solution.

8.2. Enhancement and Extension of the MTV Model Solution

The following are suggested enhancements and extensions to each of the two major parts of the MTV model solution: the EXR TV model solution and the Typecaster model solution. Extensions to the model solution are modifications made to enlarge the scope of applicability of the model solution. The modifications usually entail adding new components (generics and/or templates) in a style compatible with existing components. The components are added to increase the scope of applicability of the model solution, thus enabling it to be used to translate and validate messages that the original solution could not. Enhancements to the model solution are modifications made to improve the performance of the model solution. The modifications usually entail modifying the algorithms or the interface to access the functionality provided. The algorithms or interface are changed to increase the attractiveness of the model solution. Examples of why these changes might be made include: increased speed, decreased size, increased usability and understandability, etc.

8.2.1. EXR TV Model Solution

If new field types are found in a message set that cannot be defined using the *CCT.Field_Description_Type*, and thus the field cannot be described by the instantiator of the Msg_ICD template, modifications to the types used to define the elements in the *CCT.Field_Description_Type* may be necessary. These type definitions are also found in *CCT* and they include *Base_Type*, *Field_Types*, *Odd_Description*, etc. There may also be the need to add new elements to the *CCT.Field_Description_Type*. Any of these modifications also require modification of the *ICD_Utilities* generic package, and possibly the *Field_Utilities* package. These are packages where parsing decisions are made based on the description of the field.

Another issue pertaining to the EXR TV model solution that was mentioned in a previous chapter is the fact that the only reordering of information from the EXR format to the order defined by the INR is via cutting. If the ordering constraint is not acceptable, then another type of information movement can be specified and added to the solution. This would involve moving information from an EXR order to an INR order. The mapping could be specified by placing the field names in an array in the desired INR order, similar to the way *Cuts* is defined, and would require modifying the parsing algorithm in the *ICD_Utilities* generic package to initiate the move similar to the way cutting is initiated. The utilities for performing movement would be added to the *Field_Utilities* package, as are the utilities for performing cutting.

8.2.2. Typecaster Model Solution

The enhancements and extensions for discrete and composite typecasters are discussed separately below.

8.2.2.1. Discrete Typecasters

The two issues of extension and enhancement are discussed for the discrete typecasters. The first issue deals with why a developer would want to extend the discrete typecaster portion of the solution and some suggestions on how to extend it. The second issue deals with general enhancements to the discrete portion of the Typecaster model solution.

If conversion requirements exist between the types of fields in the message (EXR) and the desired scalar Ada types (INR) that are not handled by the solution as is shown in Figure 6-8, then new discrete typecaster generics and the corresponding discrete typecaster templates need to be developed. This should be done to allow the Typecaster model solution to be applicable to a wider variety of EXR and INR relationships. For example, if a message set has fields that represent fixed point numbers, then a *Fixed_Point_Typecaster* generic package and template must be generated. This example is used to describe the modification process.

The interface provided by all discrete typecasters is the same. How the interface is implemented is what changes from one discrete typecaster generic to the next. Section 7.2.1 (specifically Figures 7-2 and 7-3) describes the portions of the discrete typecaster generics that are different from generic to generic and need to be customized based on the type and the mapping information. The following is a high-level description of the steps that must be taken to add a new discrete typecaster generic:

1. Start by making a copy of an existing discrete typecaster generic (e.g., the *Enumeration_Typecaster*) and rename it *Fixed_Point_Typecaster*.
2. Modify the generic formal parameter to capture the possible range of fixed-point types and the types and/or objects needed to perform the mapping between symbolic images and values. That is, the INR and UNR should be defined.
3. Preserve the generic package specification interface with an exception for how the length of the symbolic image is determined.
4. Modify the bodies of the functions for converting between symbolic images and values to reflect the mapping functions chosen.
5. Preserve the bodies of the functions for converting between natural images and values since these are based upon the Ada *'Image* and *'Value* functions.

Next, the model adapter needs to create a new template that instantiates the new *Fixed_Point_Typecaster* generic package. Section 7.2.2 (specifically Figures 7-5, 7-6, and 7-7) describes the portions of the discrete typecaster templates that are different from one discrete typecaster template to the next, and need to be customized based on the type and mapping information. The following is a high-level description of the steps that must be taken to add a new discrete typecaster template:

1. Start with an existing discrete typecaster template (e.g., the *Enumeration Typecaster Template*) and rename it.

2. Define placeholders based upon the generic formal parameters needed by the generic at instantiation time. These placeholders provide an interface for instantiating the generic and thus specify the Ada type and the mapping between values and symbolic images. Placeholders are inserted into the package specification at the appropriate places and the remainder of the package specification stays the same.
3. Distribute placeholders throughout the test procedure. This mainly involves adding code to instantiate a different I/O package for performing I/O operations on the Ada type defined.
4. Test the new typecaster thoroughly before release for use.

The second issue deals with a general enhancement to all discrete typecasters that allows greater flexibility for specifying the natural images.

The Ada language supplies the *'Image* and *'Value* functions for converting between strings and Ada values. The MTV model solution uses these functions, by default, as the mapping functions for natural images. The authors assumed that the implementors of C^3I systems would specify the discrete Ada types (INR) in a more natural fashion, (i.e., using more meaningful names) than is represented in the EXR. For example, the *Direction* field of the *Foobar* message is represented in the EXR by the symbolic images *N*, *E*, *S*, and *W*, whereas it is represented in the INR by the Ada type:

type Direction_Type is (North, South, East, West).

Therefore, the solution assumes that the INR specified by the developer and mapping function chosen by the solution for natural images (i.e., USR) is appropriate.

If this is not the case, then there is a need to specify a mapping function for conversion between natural images and Ada values. The mapping can be achieved in a manner similar to the way the mapping between symbolic images and Ada values is achieved. Modifications to all discrete typecaster generic packages and templates would be necessary. This also raises alternative packaging considerations that are discussed in Section 8.3.

8.2.2.2. Composite Typecasters

The two issues of extension and enhancement are discussed for the composite typecasters. The first issue deals with why a developer would want to extend the composite typecaster portion of the solution and some suggestions on how to extend it. The second issue deals with general enhancements to the composite portion of the Typecaster model solution.

If a new logical grouping of fields in a message (EXR) is desired that is not currently available with the existing composite typecasters, then new composite typecaster templates need to be developed. This should be done to allow the Typecaster model solution to be applicable to a wider variety of logical groupings of information.

The interface provided by all composite typecasters is exactly the same. How the interface is implemented is what changes from one composite typecaster to the next. Section 7.2.3 (specifically Figures 7-9, 7-10, 7-11, 7-12, 7-15, and 7-16) describes the portions of the composite typecasters that are variable and need to be customized based on a new logical grouping.

An example of a general enhancement to the composite portion of the Typecaster model solution would be to simplify the access to elements within the symbolic images and natural images. Currently, the developer must use the position arrays, *The_Symbolic_Image_Positions* and *The_Natural_Image_Positions*, and the name of the element defined in the enumeration type, *<Type>_Element_Names*, to obtain the slice of the symbolic or natural image in which the element resides. For example, to obtain the natural image of the reporting location from the natural image of the Foobar message, the following command is issued:

```
Rep_Loc_NI :=
    FooBar_NI(The_Natural_Image_Positions(Reporting_Location).Start..
              The_Natural_Image_Positions(Reporting_Location).Stop);
```

This can be tedious if done often, so enhancing the composite portion of the Typecaster model solution to include the procedures *Get_Element*, *Set_Element*, and *Element_Length* may be desirable. This would result in the following code for accessing a natural image element:

```
Rep_Loc_NI := Get_Element(FooBar_NI, Reporting_Location);
```

Also note that the same function can be used to access the validity indicator corresponding to the natural image element:

```
Rep_Loc_VI := Get_Element(FooBar_VI, Reporting_Location);
```

If this enhancement of the Typecaster model is desirable, all composite typecasters must be modified, but the modifications are the same for all:

1. Add the functions *Get_Element*, *Set_Element*, and *Element_Length* to the composite typecaster package specifications.
2. Move the position arrays, *The_Symbolic_Image_Positions* and *The_Natural_Image_Positions* to the composite typecaster package bodies to account for the new procedural interface added above and thus to hide the slicing concept.
3. Add the implementations of the functions *Get_Element*, *Set_Element*, and *Element_Length* to the package bodies. These implementations should use the position arrays.
4. Modify the test procedures to include testing of the new functionality.

8.3. Alternative Packaging Strategies

Based on feedback from reviewers and users of the MTV model solution, some discussion on other packaging strategies is in order. These strategies affect only the Typecaster model solution. The changes focus on the best way to either package the use of the foundation utilities, or package the functionality found in the foundation utilities. That is, the generics do not change, but how the templates group instances of generics and other templates does change. Each packaging strategy is shown graphically in figures based on a portion of the FooBar message. The figures show the software architecture that will result if the new packaging strategy is applied. Figure 6-13 shows the current software architecture of the FooBar message and can be used as a point of reference.

8.3.1. Packaging Strategy #1: Removing Types from Typecasters

The first packaging strategy is based upon removing the Ada types and mapping information from the typecaster packages. There are two possible approaches:

1. Move the type declarations (INR) and mapping information (UNR <--> INR) from the typecaster packages into individual packages, one package for each type declaration and its corresponding mapping information. See Figure 8-1.
2. Move all type declarations and mapping information for a particular message from the typecaster packages into a single package. See Figure 8-2.

Both approaches allow other portions of the application to access the types without accessing the typecaster functionality. The first approach has the advantage of allowing other operations on the type to be defined and located in the package with the type, e.g., an imaginary arithmetic type may have some operations defined for imaginary arithmetic. The second approach has the advantage of having all types and mapping information about the message being present in a single place. This may make comprehension of the message specification easier.

Either approach would require modification of the typecaster templates to handle this packaging approach. The discrete typecaster generic templates would be reduced to a generic instantiation and test procedure since the only reason for the package specification in the past was to hold the type and mapping information. In both the discrete and composite typecaster templates, new placeholders must be defined that will specify the name of the package where the type and mapping reside.

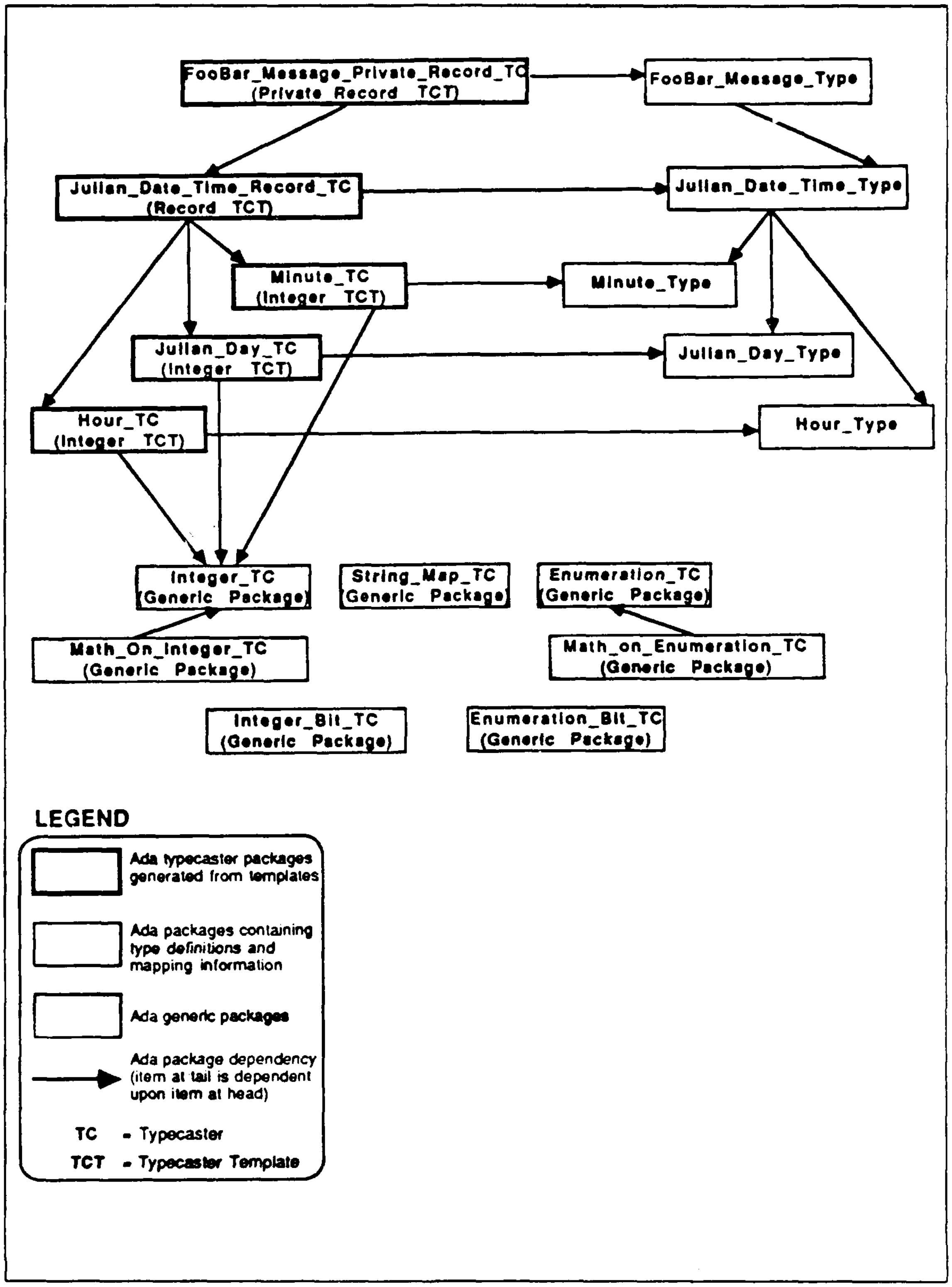
FooBar_Message_Private_Record_TC (Private Record TCT)
FooBar_Message_Type
Julian_Date_Time_Record_TC (Record TCT)
Julian_Date_Time_Type
Minute_TC (Integer TCT)
Minute_Type
Julian_Day_TC (Integer TCT)
Julian_Day_Type
Hour_TC (Integer TCT)
Hour_Type
Integer_TC (Generic Package)
String_Map_TC (Generic Package)
Enumeration_TC (Generic Package)
Math_On_Integer_TC (Generic Package)
Math_on_Enumeration_TC (Generic Package)
Integer_Bit_TC (Generic Package)
Enumeration_Bit_TC (Generic Package)
LEGEND
Ada typecaster packages generated from templates
Ada packages containing type definitions and mapping information
Ada generic packages
Ada package dependency (item at tail is dependent upon item at head)
TC - Typecaster
TCT - Typecaster Template

Figure 8-1: Typecaster Type Declarations Moved to Individual Packages

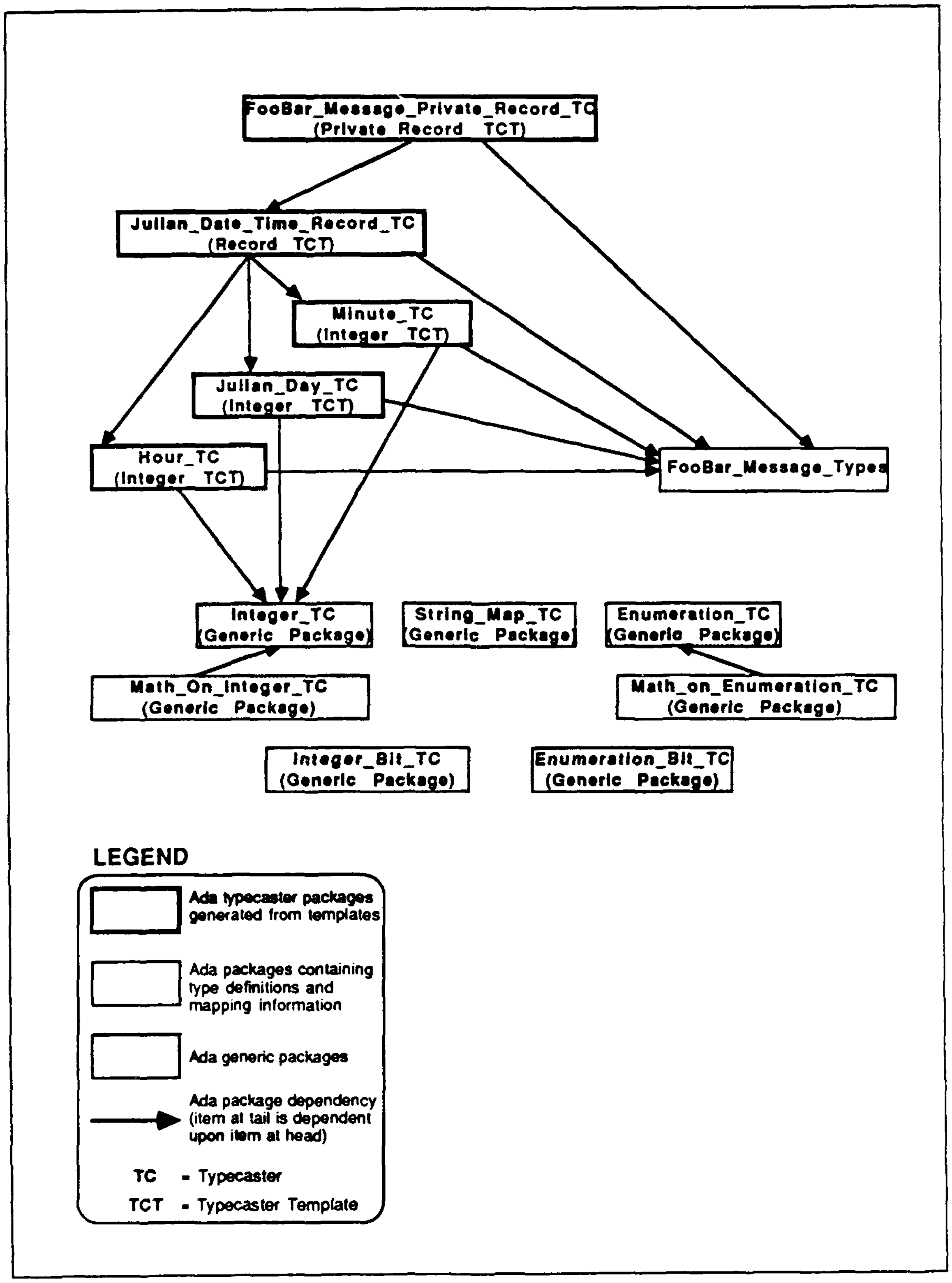
FooBar_Message_Private_Record_TC (Private Record TCT)
Julian_Date_Time_Record_TC (Record TCT)
Minute_TC (Integer TCT)
Julian_Day_TC (Integer TCT)
Hour_TC (Integer TCT)
FooBar_Message_Types
Integer_TC (Generic Package)
String_Map_TC (Generic Package)
Enumeration_TC (Generic Package)
Math_On_Integer_TC (Generic Package)
Math_on_Enumeration_TC (Generic Package)
Integer_Bit_TC (Generic Package)
Enumeration_Bit_TC (Generic Package)
LEGEND
Ada typecaster packages generated from templates
Ada packages containing type definitions and mapping information
Ada generic packages
Ada package dependency (item at tail is dependent upon item at head)
TC = Typecaster
TCT = Typecaster Template

Figure 8-2: Typecaster Type Declarations Moved to a Single Package

8.3.2. Packaging Strategy #2: Separating Typecaster Functionality

Removing the types and mappings from the typecaster packages also leads to a second packaging strategy. This strategy separates the two sets of functionality provided by the typecasters, symbolic image, and natural image into individual packages. See Figures 8-3 and 8-4. This strategy stems from the original concepts of the UNK TV model and USR TV model described in Chapter 4. These two models were merged into the Typecaster model for implementation reasons.

The advantage of separating the Typecaster model into two models is to allow the developer to instantiate only that functionality that is needed. We have seen cases where only symbolic images and Ada values for a message are needed by an application and other cases where only the natural images and Ada values are needed. Because the separation forces the types to be separate, as described above, another advantage is the availability of the types to other parts of the application.

The disadvantage of separating the Typecaster models is an increase in the number of packages needed to translate and validate a message by a factor of two to three, depending upon which type and mapping packaging strategy is selected, i.e., type and mapping in the typecasters, one package per type and mapping, or one package per message for all types and mappings).

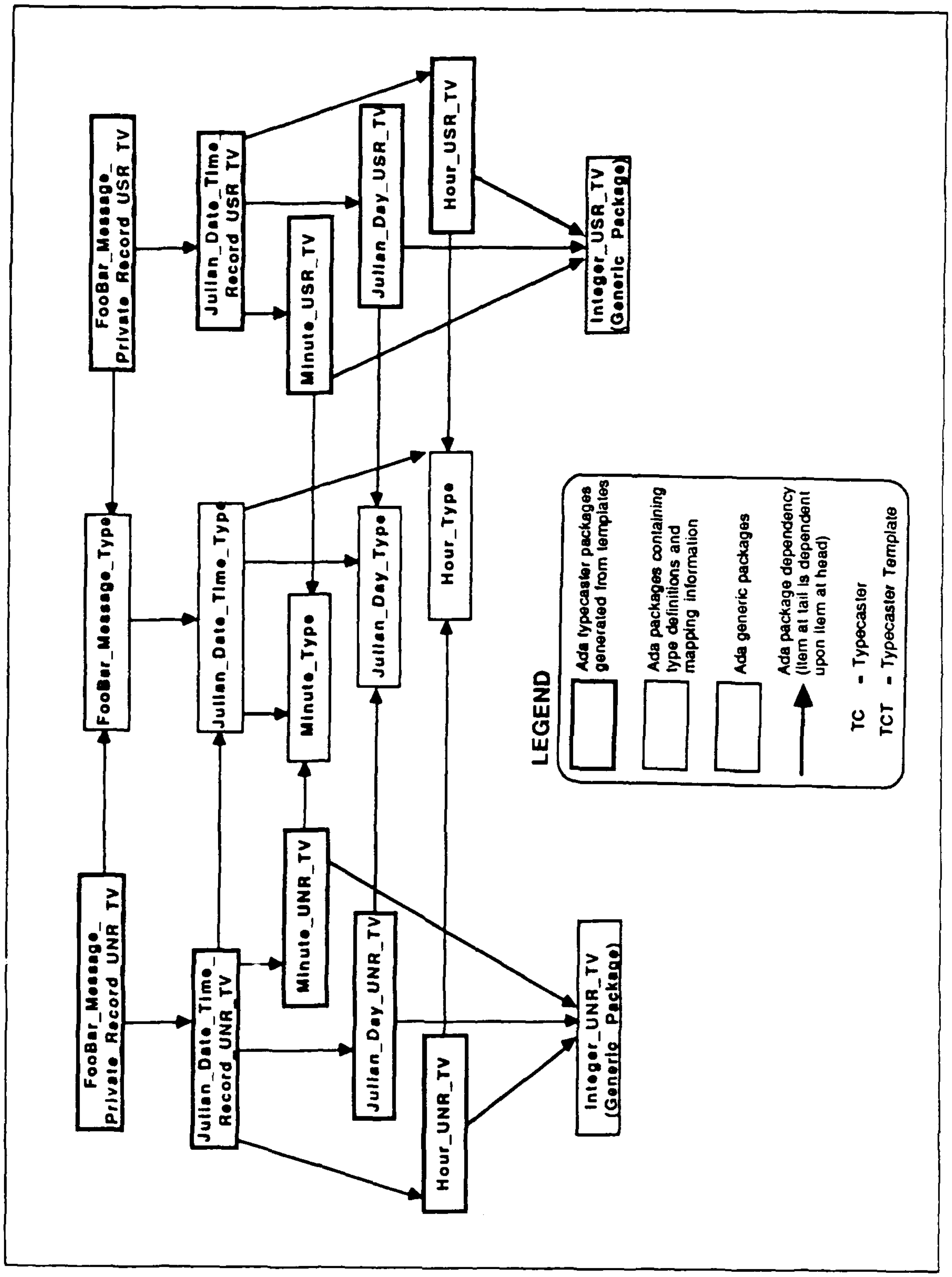
FooBar_Message_Private_Record_USR_TV
Julian_Date_Time_Record_USR_TV
Minute_USR_TV
Julian_Day_USR_TV
Hour_USR_TV
Integer_USR_TV (Generic Package)
FooBar_Message_Type
Julian_Date_Time_Type
Minute_Type
Julian_Day_Type
Hour_Type
FooBar_Message_Private_Record_UNR_TV
Julian_Date_Time_Record_UNR_TV
Minute_UNR_TV
Julian_Day_UNR_TV
Hour_UNR_TV
Integer_UNR_TV (Generic Package)
LEGEND
Ada typecaster packages generated from templates
Ada packages containing type definitions and mapping information
Ada generic packages
Ada package dependency (Item at tail is dependent upon item at head)
TC - Typecaster
TCT - Typecaster Template

Figure 8-3: Splitting Typecaster Functionality and Type Declarations Moved to Individual Packages

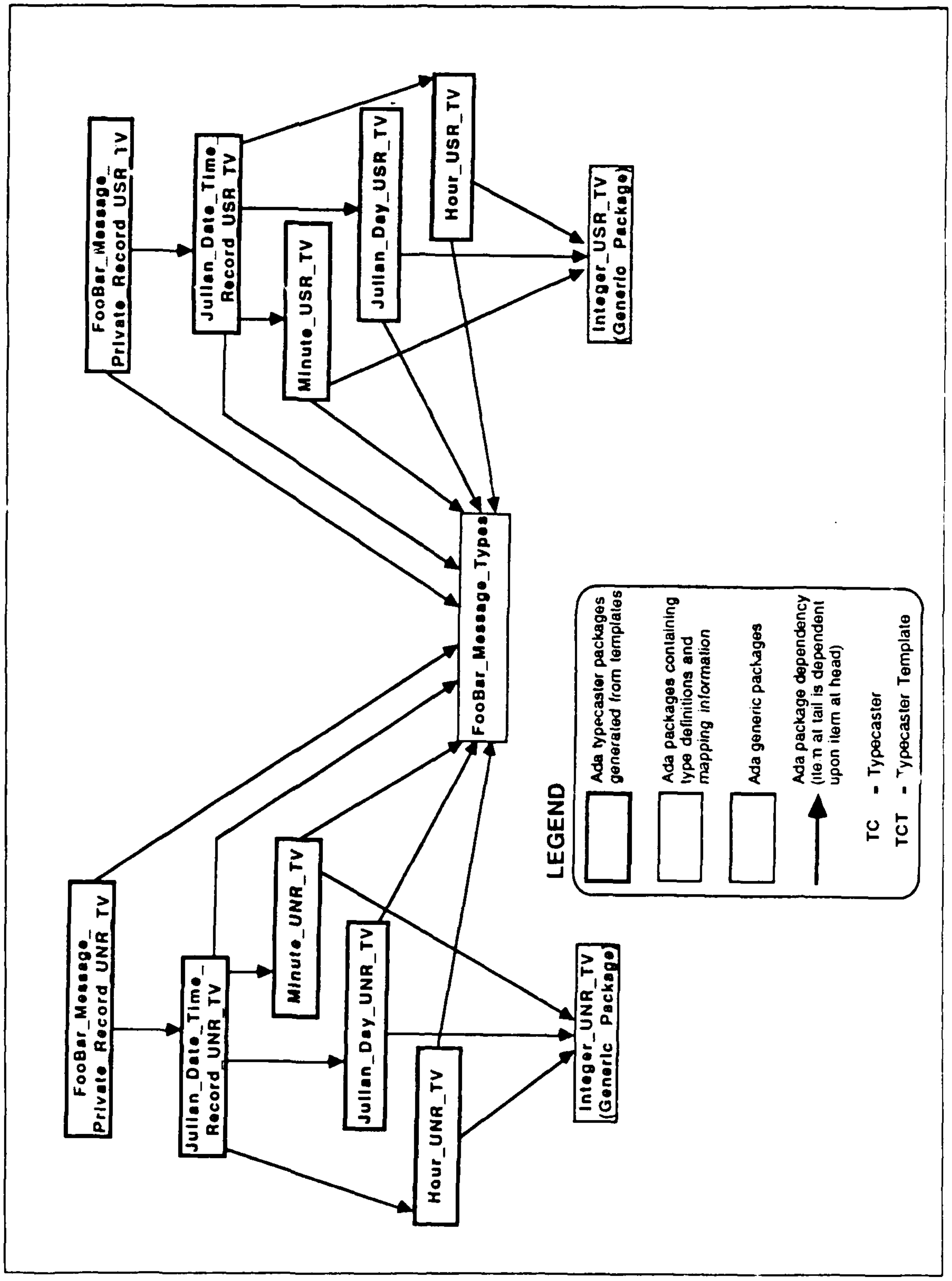
FooBar_Message_Private_Record_USR_TV
Julian_Date_Time_Record_USR_TV
Minute_USR_TV
Julian_Day_USR_TV
Hour_USR_TV
Integer_USR_TV (Generic Package)
FooBar_Message_Types
FooBar_Message_Private_Record_UNR_TV
Julian_Date_Time_Record_UNR_TV
Minute_UNR_TV
Julian_Day_UNR_TV
Hour_UNR_TV
Integer_UNR_TV (Generic Package)
LEGEND
Ada typecaster packages generated from templates
Ada packages containing type definitions and mapping information
Ada generic packages
Ada package dependency (item at tail is dependent upon item at head)
TC - Typecaster
TCT - Typecaster Template

Figure 8-4: Splitting Typecaster Functionality and Type Declarations Moved to a Single Package

8.3.3. Packaging Strategy #3: Conglomeration of Typecasters

Finally, the third packaging strategy is based upon having one composite typecaster package per message. The types and mappings for the elements of the message are declared in the composite typecaster package for the message. All discrete typecaster generics are instantiated in the package.

One advantage of this approach is that there is only one package per message, thus all information regarding the Ada types and mapping information is in one place. The understandability of this information regarding a message is probably increased for smaller messages that might not have many logical groupings, but the authors suspect that the understandability decreases as the messages become more complicated. Another advantage that stems from the absence of composite-level template nesting is that this approach provides a flatter software architecture, thus fewer procedure calls and potentially more efficient code.

But the absence of composite-level template nesting also creates a disadvantage. The message composite typecaster will have all types, mappings and generic instantiations defined in the same package, thus making the code more complicated as the message size grows since no logical groupings are possible. In summary, the abstraction mechanism provided by Ada — composite types — is not being used. Also, the discrete type, mapping information, and generic instantiation can still be made a template, but these templates must be copied into the composite template on an as needed basis. This implies that the simple process of instantiating templates has gotten more complicated and thus more prone to specifier error. Other disadvantages include the loss of reuse of discrete typecaster instances already created for fields that are common across many messages.

Finally, the other packaging strategies can also be developed using combinations of this third strategy with the first two. These strategies will not be discussed.

8.4. Model Adaptations Performed to Date

The following are enhancements and modifications to the MTV model solution that have been made by various projects based on their requirements.

The initial version of the MTV model solution did not handle bit-based messages. Both parts of the solution were extended to handle bit-based fields. This was a cooperative effort of the authors and Granite Sentry Phase II personnel. The field description data structure in the EXR TV model solution was updated to allow specification of bit-based fields and the algorithms were modified to extract the bit-based fields, and place them into a UNR form. Two new typecasters were added to the Typecaster model solution: *Integer_Bit_Typecaster* and *Enumeration_Bit_Typecaster*. Also, Granite Sentry Phase II recently extended the EXR parsing capabilities to better handle variable-length, character-based fields. These modifications have all been incorporated into the MTV model solution.

Two new discrete typecasters have been created by Granite Sentry Phase II. The first, *Fixed_Point_Typecaster*, handles conversions between strings representing fixed-point numbers and Ada values of the Fixed_Point type. The second, *String_Typecaster*, handles conversions between strings and Ada values of the String type. This typecaster allows strings to pass through the typecasters, and validation is based on the presence of printable characters.

Granite Sentry Phase II has also created extended versions of the composite record typecaster to handle 20 element records and 32 elements records. This work was given to, and performed by, technical writers, not programmers.

Finally, the Rapier program has chosen to change the packaging strategy to the third packaging strategy listed above. Their analysis and evaluation is documented in a report entitled "*Typecaster Use Prototype Technical Report*" by Djoef Woessner and Bill Schmidt [Woessner 89].

9. Open Issues

This chapter discusses unresolved issues pertaining to the performance, limitations, testing, automation, and building upon the MTV model solution.

9.1. Real-Time Performance

Performance statistics are available with all engineering models. But because of the different hardware platforms combined with the different Ada compilers available for the various hardware platforms, the general performance characteristics of the model solution cannot be defined. Instead, this document provides timing performance characteristics of the MTV model solution based upon the sample messages found in this document. These performance measurements were taken on a MicroVAX II running Version 5.1 of the VAX/VMS operating system and compiled under Version 1.5 of the VAX Ada compiler. A description of these timing measurements is in Section 5.5.

The provided performance characteristics tell little about the performance of the model solution on different hardware using a different Ada compiler, and applied to a specific message set. But one can easily measure the performance characteristics of the solution.

1. Identify a small set of messages that are characteristic of those required to be translated and validated.
2. Create the software for the reduced message set. Note that it is straightforward to instantiate the software to translate and validate a message using the templates available and following the steps described in Chapter 6.
3. Run the test procedures and use them as a basis for obtaining performance information by adding the calls to obtain the time before and after the calls to the typecaster functions.

The above steps allow measurement of the timing performance. Sizing performance (i.e., size of the resulting object code) can also be examined. Page faults, memory size, and other system parameters may also need to be examined. All results can then be extrapolated to determine the performance estimates for the rest of the system.

9.2. Limitations

The following programming constructs are used in the MTV model solution and are considered implementation-dependent features of the Ada language. This is based upon Chapter 13 of the *Ada Language Reference Manual* (Ada LRM), ANSI/MIL-STD-1815A-1983 [Ada 83]. Thus, their use in the solution will be briefly discussed.

The length clause attribute *'size* (Ada LRM, Section 13.2) is used to define a *Byte* as 8 bits whose values are natural integers that can range from 0..255. It is also used to define a *Bit*, with a size of one bit, whose values are Boolean.

```
type Byte is new Natural range 0..255;
for Byte'size use 8;

type Bit is new Boolean;
for Bit'size use 1;
```

These definitions are in the package *Casting_Common_Types*. They support the bit-based EXR TV and typecasting algorithms.

The representation attribute *'size* (Ada LRM, Section 13.7.2) is used to obtain the number of bits allocated to the bit-based objects being translated and validated. It is also used to determine how many bits are needed to hold bit-based objects. This attribute supports the bit-based EXR TV and typecasting algorithms. The *'size* attribute can be found in the following packages:

- *Casting_Common_Types* (package specification)
- *Integer_Bit_Typecaster* (generic package specification and body)
- *Enumeration_Bit_Typecaster* (generic package specification and body)
- *ICD_Utilities* (generic package body)
- *Field_Utilities* (package body)

The unchecked programming function, *Unchecked_Conversion*, (Ada LRM, Section 13.10.2) is used to convert between *Byte* and *Character* types. This was done to support the bit-based EXR TV and typecasting algorithms that comprise the MTV model solution. This function is used throughout the solution.

Running the MTV model solution on a Rational machine has been problematic because of the use of the *Unchecked_Conversion* function for converting between an array of bytes and a character string. The Rational machine does not store characters in 8-bit bytes. It optimizes and stores them in a reduced format. The MTV model solution assumes that characters are stored in 8-bit bytes.

Bytes were used instead of characters to allow the solution to support bit-based typecasting of UNRs. Characters in Ada are defined as having an ASCII value between 0 and 127. Using the *Character* type would have limited the solution to using only 7 of the 8 bits available for bit-based representations. Thus, bytes are used and character-based typecasters, initially, convert from the bytes to characters.

9.3. Testing Philosophy

Although the inclusion of test drivers in each template is a step in the right direction, the testing philosophy used leaves room for improvement. This section describes those areas where improvement is needed.

9.3.1. Discrete Typecaster Testing

The interactive portions of the tests for discrete typecasters lead the tester through the functions exported by the typecasters one at a time in a predefined order, first prompting for a symbolic image, then a value, etc.[24] If at any point the tester enters a bad test case (which is the intended use of the interactive portion of testing) the predefined order is repeated from the beginning. It would be more desirable to allow the tester to select the function to test.

All symbolic images (UNR) and natural images (USR) are, by definition, fixed-length. The interactive portions of the tests for discrete typecasters give no indication of how long the input string should be. This problem could easily be solved by informing the tester up front of how many characters are expected for symbolic images or natural images. It could also be solved graphically, by printing the proper number of dashes above the input prompt. For example, the symbolic image for an hour value is two characters, so the prompt could look something like:

```
                          --
Enter Hour Symbolic_Image =>
```

Finally, the possibility of creating a generic test driver for all discrete typecasters has been informally examined but a conclusion has not yet been reached. The advantage of having a generic test driver becomes evident when the testing philosophy changes, as described above. Currently, all test procedures in all templates would need to be changed, and test driver consistency would be a manually enforced responsibility. If a generic were developed, the test driver could be changed in one place, and the instantiation of the test driver in the discrete templates would stay the same.

9.3.2. Composite Typecaster Testing

The canned testing of composite typecasters step through the functions exported by the typecasters one at a time in a predefined order, testing each function based on the result of the previous function.[25] Testing the first function is based on a predefined (at template instantiation) symbolic image. If this symbolic image is bad, the natural image functions are not tested correctly because they depend upon the symbolic image testing to produce a valid

[24]See Figure 7-7 for the PDL describing the discrete typecaster test.

[25]See Figures 7-15 and 7-16 for the PDL describing the composite typecaster test.

Ada value. Therefore, it is impossible to test the natural image functions based upon an invalid natural image. This approach should be changed to allow testing of invalid natural images.

9.4. Building Upon the MTV Model Solution — User Scenarios

The Typecaster model solution provides the UNR and USR of a message along with descriptors that describe the format of the representation, i.e., pointers to the elements within the UNR and USR. Also provided are the validity indicators that indicate whether the UNR and USR are valid. The validity indicators are also described by the same descriptors as the USR and UNR, i.e., pointers to the validity indicators for the elements within the UNR and USR.

These descriptions *support* a user interface because the information provided by the MTV model is in a form that can be manipulated by a user interface model. Also, the interface to the MTV model is defined and stabilized.

Further user interface work (i.e., another model solution) is currently being investigated by Granite Sentry Phase II. This solution will use the information provided by the MTV model solution, along with the X-Toolkit interface, to automate the production of user interface code. This will be done in a style similar to the MTV code to ensure consistency and support automation.

9.5. Automated Code Generation

The MTV model solution is "automated." That is, all that needs to be done to create software to perform message translation and validation is select templates and do editor substitutions based upon the EXR and INR. This process is described in Chapter 6.

One can certainly envision a tool that requires the user to specify the format of the EXR and the desired INR. Based upon this information, the tool would select the appropriate templates and create the software for translating and validating the message.

Changes in message formats would require changes in specifications of the EXR and/or INR, and the code would be regenerated.

This approach is possible because a reliable, tested model solution exists. As other model solutions become available, they can be added to a library and selected based on their appropriateness to the C^3I problem specified.

References

[Ada 83] ANSI.
American National Standard Reference Manual for the Ada Programming Language.
American National Standards Institute, 1430 Broadway, New York, NY. 10018, 1983.

[D'Ippolito1 89] Richard S. D'Ippolito.
Using Models In Software Engineering.
In *Proceedings Tri-Ada '89*, pages 256-264. October, 1989.

[D'Ippolito2 89] Richard S. D'Ippolito and Charles P. Plinta.
Software Development Using Models.
In *Proceedings Fifth International Workshop on Software Specification and Design*, pages 140-142. May, 1989.

[Goyden 89] Major Mike Goyden.
The Software Lifecycle With Ada: A Command and Control Application.
In *Proceedings Tri-Ada '89*, pages 40-55. October, 1989.

[GSSDP 89] CVG-2, CVG-M and CVG-S.
Software Development Plan for the Granite Sentry Phase II Project.
Technical Report GS-SDP-02, USAF ESD/AVSG, May, 1989.

[Lee1 88] Kenneth J. Lee, Michael S. Rissman, Richard D'Ippolito, Charles Plinta, and Roger Van Scoy.
An OOD Paradigm for Flight Simulators, 2nd Edition.
Technical Report CMU/SEI-88-TR-30, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. 15213, September 1988.

[Lee2 89] Kenneth J. Lee, Michael S. Rissman.
An Object-Oriented Solution Example: A Flight Simulator Electrical System.
Technical Report CMU/SEI-89-TR-5, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. 15213, February 1989.

[Lee3 88] Kenneth Lee, Charles Plinta, and Michael Rissman.
Application of Domain Specific Software Architectures.
Software Engineering Institute Technical Review :142-162, 1988.

[Plinta 89] Charles Plinta and Kenneth Lee.
A Model Solution for the C^3I Domain.
In *Proceedings Tri-Ada '89*, pages 56-67. October, 1989.

[VanScoy 87] Roger Van Scoy.
Prototype Real-Time Monitor: Executive Summary.
Technical Report CMU/SEI-87-TR-35, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. 15213, November 1987.

[Woessner 89] Djoef Woessner and Bill Schmidt.
Typecaster Use Prototype Technical Report.
Technical Report M2SWAP55100, HA ASD/AFSC, Wright Patterson AFB, Dayton, OH., June 1989.

Appendix A: Definitions

The following terms are defined for use in this report. We are not trying to define terms gratuitously, but we need a common vocabulary for discussing the work. Terms in **boldface** within the definitions are also defined.

Bit String
A contiguous set of bytes where each byte is interpreted as a stream of bits.

Character String
A contiguous set of bytes where each byte is interpreted as an ASCII character.

Cut
The act of converting between a **field**-oriented format and an **element**-oriented format. The **field**-oriented format is described by the **external representation description** and the **element**-oriented format is that format in the **universal representation** that is converted directly into an **internal representation.**

Designer
The person responsible for analyzing the **user's** needs with the intent of generating solutions (designs) from existing solutions (**models**).

Detailed Designer
The person responsible for transforming the design (i.e., *models*) into a detailed specification that can be built by *implementors.*

Element
An Ada typed value in an **internal representation** that represents a piece of information. An element may be of a discrete or composite type. There is at least one element for each **field** in an **external representation** of a **message.**

EXR
External Representation.

External Representation
A **string** representation of a **message.** The format is described by an **external representation description.** The external representation is comprised of **fields** and **punctuation.** The external representation can be represented as either **character strings** and/or **bit strings.** External representations are received from (or sent to) systems outside the C^{3}I system being developed.

External Representation Description

A textual description of the **external representation** format of a **message**. It defines the size and location of the **fields** and **punctuation**. It also describes the information found in the **message** and how to interpret the information. The external representation description is defined outside the scope of the C^3I system being developed and becomes part of the requirements levied upon the system. For the Granite Sentry Program, the document is the ICD.

Field

A **string** in an **external representation** that stands for a piece of information. Fields may contain a single piece of data or several pieces of data. Fields that are represented as **character strings** are said to be character-based fields. Character-based fields may be variable length, including null length. Fields that are represented as **bit strings** are said to be bit-based fields.

ICD

Interface Control Document.

Image

A **string** representing a **value**. There are two types of images: **symbolic images** and **natural images**. **Values** are created from valid images by the **typecasters**. In addition, the **typecasters** create valid images from **values**.

Implementor

The person responsible for building the system (coding) as per the detailed specifications supplied by the **detailed designer**.

INR

Internal Representation.

Internal Representation

Ada typed **values** representing a **message**. The types are described by an **internal representation description**. The internal representation is comprised of **elements**. Internal representations of a **message** are used by the C^3I system being developed.

Internal Representation Description

Ada type declarations needed to define an **internal representation** format of a **message**. Each Ada type defines the type of an **element**. The internal representation description (Ada type) is defined during system design.

Maintainer

The person responsible maintaining all aspects of the system (models, detailed specification, code, etc.) after it has been delivered to the **user**.

Message

Pieces of related information.

Messages are a means of passing information from one system to another. For C^3I systems, the other systems that it must communicate with tend to be geographically separated and are heterogeneous in nature.

The external format of the information is described by an **external representation description**. The internal format of the information is described by an **internal representation description**. The information is represented as an **external representation**, a **universal representation**, an **internal representation**, and a **user representation**.

Model

To quote from Webster's:

> " ...a pattern of something to be made...
> ...an example for imitation or emulation...
> ...a description or analogy used to help visualize something (as an atom) that cannot be directly observed..."

Models are used by the **designer** to specify a solution to a problem (design).

Model Adapter

The person who enhance existing **models** that can not be used, as is, to solve a problem. Enhancements can take the form of added functionality or better resource utilization.

MTV

Message Translation and Validation.

Natural Image

A **character string** equivalent of a **value**. A natural image is the **character string** created by applying the Ada 'image (tick-image) attribute function directly to a **value**. For example, for the Ada value *North*, the natural image would be be the character string *"North"*. The **user representation** of a **message** is a **character string** containing natural images of each of the **elements**. Natural images are validated and converted into **values** by **typecasters**. **Typecasters** also convert **values** into natural images.

Non-Real-Time

A system or part of a system whose operation is not considered **real-time**.

Placeholders

Replaceable snippets of code in **templates** that stand for Ada types, names, and values. There are two types of placeholders:

1. The first is of the form <*Type*> or <*First*>, i.e., a phrase enclosed in brackets. The entire phrase (including the brackets) must be replaced. For example, <*Type*>_*Type* becomes *Hour_Type* for all instances of <*Type*>_*Type* in a file when <*Type*> is replaced by "*Hour*".
2. The second form is the double question mark, *??*. This form means that some special action must be taken by the user. For example, this form is used for those places in the code that the user needs to supply a function body, supply test cases, or remove some lines from the **template**, e.g., the instructions at the beginning of each **template**.

Punctuation

Character strings or **bit strings** that separate **fields** in an **external representation**. For character-based **fields**, punctuation may be a single character, several characters, or absent. End-of-message punctuation occurs at the end of some **external representations**.

Real-Time

Pertaining to a system or part of a system whose operation can be characterized by the following:

> "When it is done is as important as what is done."[26]

String

A contiguous set of bytes. Two types of strings exist for the message translation and validation model, **character strings** and **bit strings**.

[26]Quote from Robert Firth, Senior Member of the Technical Staff at the Software Engineering Institute.

Symbolic Image
A **string** containing a codification of information that represents a **value**. For example, for the Ada value *North*, the **symbolic image** might be the character string "*N*". The **universal representation** of a **message** is a **string** containing symbolic images of each of the **fields**. Symbolic images are validated and converted into **values** by **typecasters**. **Typecasters** also convert **values** into symbolic images.

Template
A file containing an Ada package specification, package body, and test procedure. The file contains **placeholders** for the name of the package, the Ada type used in the template, and so on. The **placeholders** must be globally replaced with their appropriate values using an editor. Global replacement of the **placeholders** affects the specification, the body, and the test procedure. The template with the **placeholders** replaced is called a **typecaster**. All templates contain instructions in the first few lines of the template. The instruction lines are to be removed when the template has become a **typecaster**.

TV Translation and Validation.

Typecaster
A compilable package specification and package body. A typecaster is generated from a **template** by substituting all **placeholders** with appropriate Ada types, names, or values. A typecaster is specific for the Ada type that was used in its generation and provides the capability to convert between **symbolic images** and **values** of that type. The typecaster also provides a diagnostic routine for checking the validity of a **symbolic image**. Any value outside the range of the Ada type causes the Ada runtime environment to raise a Constraint_Error exception. It also provides the capability to convert between **natural images** and **values** of that type, and provides a diagnostic routine for checking the validity of a **natural image**.

Typecasting
The act of converting between an **image** (**symbolic image** or **natural image**) and a **value**. The implementation of the typecasting operations mimics the Ada attribute functions 'image (tick-image) and 'value (tick-value). The functions defined in the **typecasters** for performing the typecasting operations are called *Image* and *Value*.

Universal Representation
An internal view of an **external representation**. The universal representation is a fixed length **string**, containing a **symbolic image** for each **field** in the **external representation**, with **punctuation** removed, variable length, null, and optional character-based **fields** padded with blanks to their maximum length, and bit-based **fields** expanded to multiples of byte lengths. The information is **cut** to match the **internal representation**, if necessary.

UNR
Universal Representation.

User
The person whose job can be performed more effectively using the completed C^3I system.

User Representation
A user-readable view of an **internal representation**. The user representation is a fixed-length **string**, containing a **natural image** for each **element** of the **internal representation**.

USR
User Representation.

Value
An Ada typed value representing an **image**. Values are created from valid **images** by the **typecasters**. **Typecasters** also create valid **images** from values.

Appendix B: Detailed Description of the Templates

This appendix is a reference manual for the MTV model solution, whose application was described in Chapter 6. It describes all templates individually, in detail. This appendix is for the detailed designer and implementor. For the duration of this appendix, they are addressed as users of the templates.

The Ada code templates can be found in Appendix C.

B.1. Format of the Template Descriptions

The sections of this appendix describe the discrete typecaster templates, composite typecaster templates, and ICD template. Each template description contains the following information:

- capabilities provided by an instance of a typecaster template
- when to use a particular typecaster template
- how to generate an instance of a typecaster from a typecaster template
- how to use the generated typecaster

Before looking at the discrete and composite typecaster templates individually, the following describes the above points in general for all typecaster templates.

Capabilities

Typecasters provide the capability to convert between strings representing discrete/composite values (symbolic images) and discrete/composite Ada values.

Typecasters provide the capability to convert between character string images of a discrete/composite value (natural images) and discrete/composite Ada values.

Typecasters provide the capability to check strings for symbolic image validity and natural image validity.

The Ada type checking insures the validity of discrete/composite values.

When to Use the Templates

The templates described in this appendix should be used when the detailed designers wish to generate Ada software to translate and validate externally formatted bit-based or character-based messages.

How to Generate an Instance of a Typecaster

Once the detailed designer decides which typecaster template to use, the template must then be copied into an empty file and editor substitutions must be performed. These substitutions are based on the placeholders found in the header comments at the beginning of the template. Also, any special instructions, indicated by the "??" placeholder, should be followed.

One test procedure is provided in each template.

- For discrete typecasters the test procedure first does an exhaustive test on the entire range of values and then allows interactive testing by prompting for symbolic images, natural images and values.
- For composite typecasters the test procedure does testing based on a set of canned test cases that are supplied before compilation.

All test procedures report on testing problems, e.g., unexpected results.

How to Use the Typecaster

Each typecaster package exports a set of types, constants and functions that support the capabilities listed above. Sample uses are shown.

B.2. Template Naming Conventions

It is critical that the user of these templates be aware of the naming conventions enforced by the templates.

All typecasters are named via a substitution for the <*Type*> placeholder. A suffix is hard coded into the templates. The name of the type to be cast, which is exported by the typecaster, is handled similarly.

For example, for any character-based discrete typecaster, the name of the typecaster and the type it exports will look like:

```
        suffix provided              suffix provided
       _____|_____                      __|__
       |         |                      |   |
<Type>_Typecaster               <Type>_Type
```

whereas for bit-based discrete typecasters, the name of the typecaster and the type it exports will look like:

```
        suffix provided                  suffix provided
       _____|_________                  _____|___
       |             |                  |       |
<Type>_Bit_Typecaster           <Type>_Bit_Type
```

The composite typecasters and the type they export have different suffixes depending on the composite type:

```
        suffix provided                  suffix provided
       _____|_________________          _____|__________
       |                     |          |              |
<Type>_{composite}_Typecaster   <Type>_{composite}_Type
```

where {composite} will be *"Array"* in the Array typecaster template, *"Private_Record"* in the Private_Record typecaster template, and so on.

In all cases it is only necessary to substitute for the <*Type*> placeholder, e.g., replacing <*Type*> with *Hour* results in

```
<Type>_Typecaster => Hour_Typecaster

<Type>_Type => Hour_Type
```

It is also necessary to know the names of the elemental typecasters when generating a composite typecaster. The names without the *_Typecaster* and *_Type* suffixes should be specified. The suffixes are provided throughout the composite typecaster templates for each elemental reference.

B.3. Discrete Typecaster Templates

The discrete typecaster templates are the building blocks of the typecaster model solution. The following is a list of the discrete typecaster templates:

1. Integer Typecaster Template
2. Integer_Bit Typecaster
3. Math_On_Integer Typecaster Template
4. Enumeration Typecaster Template
5. Enumeration_Bit Typecaster Template
6. Math_On_Enumeration Typecaster Template
7. String_Map Typecaster Template

All discrete typecaster templates are based upon the typecaster model described in Chapter 4. Figure 7-5 shows the incomplete Ada package specification that is representative of all discrete typecaster templates.

Each discrete typecaster template is customized based upon the incomplete Ada package specification shown in Figure 7-5. The customization entails specifying those portions of the incomplete Ada package specification that are in $\boxed{\textbf{\textit{bold, italics and boxed}}}$. These customizations are either in the form of compilable Ada code or code template placeholders that the detailed designer must specify when instantiating the template. The customizations are specific to the typecaster operations provided by individual discrete typecaster templates.

B.3.1. Integer Typecaster Template

The Integer Typecaster Template is shown in Appendix C.3.1 and resides in the file named

```
INTEGER_TEMPLATE_.ADA
```

The name of a typecaster generated from the Integer typecaster template will result from substitution of the placeholder <*Type*> as in

```
<Type>_Typecaster => Hour_Typecaster
```

The name of the Integer type, generated in the Integer typecaster, will result from substitution of the placeholder <*Type*> as in

```
<Type>_Type => Hour_Type
```

Capabilities

The generated package will provide the capability to convert between character strings representing integers (symbolic images) and integer values (values) of a specified integer range. The strings representing integers are zero padded and can be signed or unsigned. It also provides the capability to check character strings for symbolic image validity. The Ada type checking insures the validity of integer values.

The generated package will also provide the capability to convert between character strings images of integers (natural images) and integer values (values) of a specified integer range. It also provides the capability to check character strings for natural image validity.

When to Use the Template

Use this template to create a typecaster for casting between character-based symbolic images and values of a specified integer range, and also for casting between character-based natural images and values of a specified integer range.

It handles integers in the range <*First*>..<*Last*>. Type <*Type*>*_Type* will specify the integer range for instantiating the *Integer_Typecaster* generic.

An implicit mapping between both the symbolic images and natural images, and the values in the range specified by the type <*Type*>*_Type* must exist, e.g.,

```
subtype Hour_Type is Integer range 0..23;

symbolic                        natural
  image <=> value                image <=> value
----------------                ---------------
  "00"  <=>   0                 "  0" <=>   0
          .                               .
          .                               .
          .                               .
  "23"  <=>  23                 " 23" <=>  23
```

How to Generate an Instance of a Typecaster

The following lines describe how to create an instance of an Integer typecaster from the Integer typecaster template.

The following global placeholder substitutions need to be made:

- Replace the placeholder <*Type*> with the integer type, e.g., *Hour*. It is not necessary to append *_Type* to the placeholder <*Type*>, the template contains *_Type* in the correct places. See section B.2 for more details.
- Replace <*First*> with the starting value of the range of the integer type, e.g., "0".
- Replace <*Last*> with the ending value of the range of the integer type, e.g., "23".
- Replace <*Is-Signed*> with "True" if the symbolic image is signed, otherwise replace it with "False".

One test procedure is provided in the template, <*Type*>*_Typecaster_Test*. The test procedure first does an exhaustive test on the entire range of values for the type <*Type*>*_Type*. Then, the procedure allows interactive testing by prompting for symbolic images, natural images and values of integers.

Appendix D.1.3 shows an example instance (*Hour_Typecaster*) of the Integer typecaster template.

How to Use the Typecaster

The Integer typecaster can be used by providing the following statements and declarations within a compilable unit:
NOTE: prior to placing the following lines within the compilable unit, replace the <*Type*> placeholder, shown below, with the proper Ada type.

```
with <Type>_Typecaster;

An_Integer              : <Type>_Typecaster.<Type>_Type;
Integer_Symbolic_Image  : <Type>_Typecaster.Symbolic_Image;
Check_Symbolic_Image    : <Type>_Typecaster.Symbolic_Image;
Integer_Natural_Image   : <Type>_Typecaster.Natural_Image;
Check_Natural_Image     : <Type>_Typecaster.Natural_Image;
```

The functions available in the Integer typecaster are used as follows:

```
An_Integer := <Type>_Typecaster.Value(Integer_Symbolic_Image);

Integer_Symbolic_Image := <Type>_Typecaster.Image(An_Integer);

Check_Symbolic_Image :=
    <Type>_Typecaster.Check(Integer_Symbolic_Image);

An_Integer := <Type>_Typecaster.Value(Integer_Natural_Image);

Integer_Natural_Image := <Type>_Typecaster.Image(An_Integer);

Check_Natural_Image :=
    <Type>_Typecaster.Check(Integer_Natural_Image);
```

B.3.2. Integer_Bit Typecaster Template

The Integer_Bit Typecaster Template is shown in Appendix C.3.2 and resides in the file named

```
INTEGER_BIT_TEMPLATE_.ADA
```

The name of a typecaster generated from the Integer_Bit typecaster template will result from substitution of the placeholder *<Type>* as in

```
<Type>_Bit_Typecaster => Julian_Day_Bit_Typecaster
```

The name of the Integer type, generated in the Integer_Bit typecaster, will result from substitution of the placeholder *<Type>* as in

```
<Type>_Bit_Type => Julian_Day_Bit_Type
```

Capabilities

The generated package will provide the capability to convert between bit strings representing integers (symbolic images) and integer values (values) of a specified integer range. It also provides the capability to check bit strings for symbolic image validity. The Ada type checking insures the validity of integer values.

The generated package will also provide the capability to convert between character string images of integers (natural images) and integer values (values) of a specified integer range. It also provides the capability to check character strings for natural image validity.

When to Use the Template

Use this template to create a typecaster for casting between bit-based symbolic images and values of a specified integer range, and also for casting between character-based natural images and values of a specified integer range.

It handles integers in the range *<First>..<Last>*. Type *<Type>_Bit_Type* will specify the integer range for instantiating the *Integer_Bit_Typecaster* generic.

An implicit mapping between both the symbolic images and natural images, and the values in the range specified by the type *<Type>_Bit_Type* must exist, e.g.,[27].

```
subtype Julian_Day_Bit_Type is Integer range 1..366;

    symbolic                        natural
     image       <=> value           image   <=> value
    -------------------------      ------------------
    16#01#,16#00#  <=>   1          "   1" <=>   1
                    .                        .
                    .                        .
                    .                        .
    16#6E#,16#01#  <=> 366          " 366" <=>   366
```

[27]The symbolic image is two bytes because it takes two bytes to represent a number from 1..366

How to Generate an Instance of a Typecaster

The following lines describe how to create an instance of an Integer_Bit typecaster from the Integer_Bit typecaster template.

The following global placeholder substitutions need to be made:

- Replace the placeholder <*Type*> with the integer type, e.g., *Julian_Day*. It is not necessary to append *_Bit_Type* to the placeholder <*Type*>, the template contains *_Bit_Type* in the correct places. See Section B.2 for more details.
- Replace <*First*> with the starting value of the range of the integer type, e.g., "1".
- Replace <*Last*> with the ending value of the range of the integer type, e.g., "366".

One test procedure is provided in the template, <*Type*>*_Bit_Typecaster_Test*. The test procedure first does an exhaustive test on the entire range of values for the type <*Type*>*_Bit_Type*. Then, the procedure allows interactive testing by prompting for symbolic images, natural images and values of integers.

Appendix D.1.11 shows an example instance (*Julian_Day_Bit_Typecaster*) of the Integer_Bit typecaster template.

How to Use the Typecaster

The Integer_Bit typecaster can be used by providing the following statements and declarations within a compilable unit:
NOTE: prior to placing the following lines within the compilable unit, replace the <*Type*> placeholder, shown below, with the proper Ada type.

```
with <Type>_Bit_Typecaster;

An_Integer : <Type>_Bit_Typecaster.<Type>_Bit_Type;

Integer_Symbolic_Image : <Type>_Bit_Typecaster.Symbolic_Image;
Check_Symbolic_Image : <Type>_Bit_Typecaster.Symbolic_Image;

Integer_Natural_Image : <Type>_Bit_Typecaster.Natural_Image;
Check_Natural_Image : <Type>_Bit_Typecaster.Natural_Image;
```

The functions available in the Integer_Bit typecaster are used as follows:

```
An_Integer := <Type>_Bit_Typecaster.Value(Integer_Symbolic_Image);

Integer_Symbolic_Image := <Type>_Bit_Typecaster.Image(An_Integer);

Check_Symbolic_Image :=
    <Type>_Bit_Typecaster.Check(Integer_Symbolic_Image);


An_Integer := <Type>_Bit_Typecaster.Value(Integer_Natural_Image);

Integer_Natural_Image := <Type>_Bit_Typecaster.Image(An_Integer);

Check_Natural_Image :=
    <Type>_Bit_Typecaster.Check(Integer_Natural_Image);
```

B.3.3. Math_On_Integer Typecaster Template

The Math_On_Integer Typecaster Template is shown in Appendix C.3.3 and resides in the file named

```
MATH_ON_INTEGER_TEMPLATE_.ADA
```

The name of a typecaster generated from the Math_On_Integer typecaster template will result from substitution of the placeholder <*Type*> as in

```
<Type>_Typecaster => Scaled_Integer_100_500_Typecaster
```

The name of the Integer type, generated in the Math_On_Integer typecaster, will result from substitution of the placeholder <*Type*> as in

```
<Type>_Type => Scaled_Integer_100_500_Type
```

Capabilities

This generated package provides the capability to convert between character strings representing integers (symbolic images) and scaled integers (values). The strings representing integers are zero padded and can be signed or unsigned. It also provides the capability to check character strings for symbolic image validity. The Ada type checking insures the validity of integer values.

The generated package will also provide the capability to convert between character string images of the scaled integers (natural images) and scaled integers (values). It also provides the capability to check character strings for natural image validity.

When to Use the Template

Use this template to create a typecaster for casting between character-based symbolic images and values of a specified integer, where the value is a scaled integer. Also for casting between character-based natural images and values of a scaled integer.

This package handles strings representing integers whose values are in the range *<First>* / *<Factor>*..*<Last>* / *<Factor>*. Type *<Type>_Type* specifies the range of scaled values for instantiating the *Math_On_Integer_Typecaster* generic.

An implicit mapping (after applying the scale factor) between both the symbolic images and natural images, and the values (scaled integers) in the range specified by the type *<Type>_Type* must exist, e.g.,

```
factor : Integer := 100;
subtype Scaled_Integer_100_500_Type is Integer range 100..500;

symbolic                    natural
  image <=> value            image   <=>  value
-----------------           -------------------
   "1"  <=>  100             " 100"  <=>  100
         .                            .
         .                            .
         .                            .
   "5"  <=>  500             " 500"  <=>  500
```

How to Generate an Instance of a Typecaster

The following lines describe how to create an instance of a Math_On_Integer Typecaster from the Math_On_Integer typecaster template.

The following global placeholder substitutions need to be made:

- Replace the placeholder *<Type>* with the scaled integer type, e.g., *Scaled_Integer_100_500*. It is not necessary to append *_Type* to the placeholder <*Type*>, the template contains *_Type* in the correct places. See Section B.2 for more details.
- Replace *<First>* with the starting value of the range of the scaled integer type <*Type*>*_Type*, , e.g., "100".
- Replacc *<Last>* with the ending value of the range of the scaled integer type <*Type*>*_Type*, , e.g., "500".
- Replace *<Is-Signed>* with "True" if the symbolic image is signed, otherwise replace it with "False".
- Replace *<Factor>* with the scaling factor. The default factor is 100.

One test procedure is provided in the template, <*Type*>*_Typecaster_Test*. The test procedure first does an exhaustive test on the entire range of values for the type <*Type*>*_Type*. Then, the procedure allows interactive testing by prompting for symbolic images, natural images and scaled values of integers.

Appendix D.4.1 shows an example instance (*Scaled_Integer_100_500_Typecaster*) of the Math_On_Integer_typecaster template.

How to Use the Typecaster

The Math_On_Integer typecaster can be used by providing the following statements and declarations within a compilable unit:
NOTE: prior to placing the following lines within the compilable unit, replace the <*Type*> placeholder, shown below, with the proper Ada type.

```
with <Type>_Typecaster;

An_Scaled_Integer : <Type>_Typecaster.<Type>_Type;

Scaled_Integer_Symbolic_Image : <Type>_Typecaster.Symbolic_Image;
Check_Symbolic_Image : <Type>_Typecaster.Symbolic_Image;

Scaled_Integer_Natural_Image : <Type>_Typecaster.Natural_Image;
Check_Natural_Image : <Type>_Typecaster.Natural_Image;
```

The functions available in the Math_On_Integer typecaster are used as follows:

```
    A_Scaled_Integer:=
        <Type>_Typecaster.Value(Scaled_Integer_Symbolic_Image);

    Scaled_Integer_Symbolic_Image:=
        <Type>_Typecaster.Image(A_Scaled_Integer);

    Check_Symbolic_Image :=
        <Type>_Typecaster.Check(Scaled_Integer_Symbolic_Image);


    A_Scaled_Integer :=
        <Type>_Typecaster.Value(Scaled_Integer_Natural_Image);

    Scaled_Integer_Natural_Image :=
        <Type>_Typecaster.Image(A_Scaled_Integer);

    Check_Natural_Image :=
        <Type>_Typecaster.Check(Scaled_Integer_Natural_Image);
```

B.3.4. Enumeration Typecaster Template

The Enumeration Typecaster Template is shown in Appendix C.3.4 and resides in the file named

```
ENUMERATION_TEMPLATE_.ADA
```

The name of a typecaster generated from the Enumeration typecaster template will result from substitution of the placeholder *<Type>* as in

```
<Type>_Typecaster => Direction_Typecaster
```

The name of the discrete type generated in the Enumeration typecaster will result from substitution of the placeholder *<Type>* as in

```
<Type>_Type => Direction_Type
```

Capabilities

This generated package provides the capability to convert between character strings representing discrete values (symbolic images) and discrete values (values). It also provides the capability to check character strings for symbolic image validity. The Ada type checking insures the validity of the discrete values.

The generated package also provides the capability to convert between character string images of discrete values (natural images) and discrete values (values). It also provides the capability to check character strings for natural image validity.

When to Use the Template

Use this template to create a typecaster for casting between character-based symbolic images, which can be represented as an enumerated type, and discrete values of the specified discrete type, *<Type>_Type*. Also for casting between character-based natural images and and discrete values of the specified discrete type.

Type *<Type>_Type* will specify the discrete values, and the type *<Type>_Map* specifies the symbolic images and the mapping between the symbolic images and the values. These are for instantiating the *Enumeration_Typecaster* generic.

There must be an explicit one-to-one positional mapping between the enumerated values specified by *<Type>_Map*, which are the enumerated values representing valid symbolic images, and the discrete values specified. There is an implicit mapping between natural images and the discrete values, e.g.,

```
type Direction_Type is (North, South, East, West);
type Direction_Map  is (N,     S,     E,    W);

symbolic                         natural
  image <=> value                 image   <=>  value
-----------------                -------------------
   "N"   <=>  North                "NORTH" <=>  North
   "S"   <=>  South                "SOUTH" <=>  South
   "E"   <=>  East                 " EAST" <=>  East
   "W"   <=>  West                 " WEST" <=>  West
```

How to Generate an Instance of a Typecaster

The following lines describe how to create an instance of a Enumeration Typecaster from the Enumeration typecaster template.

The following global placeholder substitution needs to be made:

- Replace the placeholder <*Type*> with the Enumeration type, e.g., *Direction*. It is not necessary to append _*Type* to the placeholder <*Type*>, the template contains _*Type* in the correct places. See Section B.2 for more details.
- Do a search for *??* and fill in the necessary information:
    - the definition of <*Type*>_*Type*, the meaning of the symbolic images. remember that the natural images are derived from the values, e.g., ("North", ...).
    - the definition of <*Type*>_*Map*, the symbolic images expressed as an enumeration type, e.g., (N, ...).

One test procedure is provided in the template, <*Type*>_*Typecaster_Test*. The test procedure first does an exhaustive test on the entire range of values for the type <*Type*>_*Type*. Then, the procedure allows interactive testing by prompting for symbolic images, natural images and values.

Appendix D.1.2 shows an example instance (*Direction_Typecaster*) of the Enumeration typecaster template.

How to Use the Typecaster

The Enumeration typecaster can be used by providing the following statements and declarations within a compilable unit:
NOTE: Prior to placing the following lines within the compilable unit, replace the <*Type*> placeholder, shown below, with the proper Ada type.

```
with <Type>_Typecaster;

    A_Discrete : <Type>_Typecaster.<Type>_Type;

    Discrete_Symbolic_Image : <Type>_Typecaster.Symbolic_Image;
    Check_Symbolic_Image : <Type>_Typecaster.Symbolic_Image;

    Discrete_Natural_Image : <Type>_Typecaster.Natural_Image;
    Check_Natural_Image : <Type>_Typecaster.Natural_Image;
```

The functions available in the Enumeration typecaster are used as follows:

```
A_Discrete := <Type>_Typecaster.Value(Discrete_Symbolic_Image);

Discrete_Symbolic_Image := <Type>_Typecaster Image(A_Discrete);

Check_Symbolic_Image :=
    <Type>_Typecaster.Check(Discrete_Symbolic_Image);


A_Discrete := <Type>_Typecaster.Value(Discrete_Natural_Image);

Discrete_Natural_Image := <Type>_Typecaster.Image(A_Discrete);

Check_Natural_Image :=
    <Type>_Typecaster.Check(Discrete_Natural_Image);
```

B.3.5. Enumeration_Bit Typecaster Template

The Enumeration_Bit Typecaster Template is shown in Appendix C.3.5 and resides in the file named

```
ENUMERATION_BIT_TEMPLATE_.ADA
```

The name of a typecaster generated from the Enumeration_Bit typecaster template will result from substitution of the placeholder <*Type*> as in

```
<Type>_Bit_Typecaster => Direction_Bit_Typecaster
```

The name of the discrete type generated in the Enumeration_Bit typecaster will result from substitution of the placeholder <*Type*> as in

```
<Type>_Type => Direction_Bit_Type
```

Capabilities

This generated package provides the capability to convert between bit strings representing enumerated values (symbolic images) and discrete values (values). It also provides the capability to check bit strings for symbolic image validity. The Ada type checking insures the validity of the discrete values.

The generated package also provides the capability to convert between character string images of enumerated values (natural images) and discrete values (values). It also provides the capability to check character strings for natural image validity.

When to Use the Template

Use this template to create a typecaster for casting between bit-based symbolic images and enumeration values of the specified enumeration type, <*Type*>*_Bit_Type*, and also for casting between character-based natural images and enumeration values of the specified enumeration type.

Type <*Type*>*_Bit_Type* will specify the values and *The_Map* will specify the mapping between bit-based symbolic images and the values. These are for instantiating the *Enumeration_Bit_Typecaster* generic.

There must be an explicit one-to-one mapping between the bit-based symbolic images and enumerated values. Type <*Type*>*_Bit_Type* will specify the values and *Symbolic_Image_Range* will specify the range of integer values possible for the bit-based symbolic images. *The_Map* explicitly defines the mapping between bit-based symbolic images and the values. There is an implicit mapping between natural images and the discrete values, e.g.,[28]

[28]The symbolic image is one byte because it takes one byte to represent a number from 0..3.

```
type Direction_Bit_Type is (North, South, East, West);
subtype Symbolic_Image_Range is Integer range 0..3;
The_Map : The_Map_Type :=(
      North => 0,
      South => 1,
       East => 2,
       West => 3);

symbolic                          natural
  image <=> value                  image   <=>  value
---------------------             ------------------
   16#00#  <=>  North             "NORTH"  <=>  North
   16#01#  <=>  South             "SOUTH"  <=>  South
   16#02#  <=>  East              " EAST"  <=>  East
   16#03#  <=>  West              " WEST"  <=>  West
```

How to Generate an Instance of a Typecaster

The following lines describe how to create an instance of a Enumeration_Bit Typecaster from the Enumeration_Bit typecaster template.

The following global placeholder substitution needs to be made:

- Replace the placeholder <*Type*> with the Enumeration_Bit type, e.g., *Direction*. It is not necessary to append *_Bit_Type* to the placeholder <*Type*>, the template contains *_Bit_Type* in the correct places. See Section B.2 for more details.
- Do a search for *??* and fill in the necessary information:
    - the definition of <*Type*>*_Bit_Type*, the meaning of the symbolic images, remember that the natural images are derived from the values, e.g., ("North", ...)
    - the definition of <*Symbolic_Image_Range*>, the range of integer values expected in the bit-based symbolic images, e.g., "0..3"
    - the definition of <*The_Map*>, the mapping between enumeration values and the bit-based symbolic images, e.g., "North => 0"

One test procedure is provided in the template, <*Type*>*_Bit_Typecaster_Test*. The test procedure first does an exhaustive test on the entire range of values for the type <*Type*>*_Bit_Type*. Then, the procedure allows interactive testing by prompting for symbolic images, natural images and values.

Appendix D.1.8 shows an example instance (*Direction_Bit_Typecaster*) of the Enumeration_Bit typecaster template.

How to Use the Typecaster

The Enumeration_Bit typecaster can be used by p iding the following statements and declarations within a compilable unit:
NOTE: prior to placing the following lines within the compilable unit, replace the <*Type*> placeholder, shown below, with the proper Ada type.

```
with <Type>_Bit_Typecaster;

    An_Enumeration : <Type>_Bit_Typecaster.<Type>_Bit_Type;

    Enumeration_Symbolic_Image : <Type>_Bit_Typecaster.Symbolic_Image;
    Check_Symbolic_Image : <Type>_Bit_Typecaster.Symbolic_Image;

    Enumeration_Natural_Image : <Type>_Bit_Typecaster.Natural_Image;
    Check_Natural_Image : <Type>_Bit_Typecaster.Natural_Image;
```

The functions available in the Enumeration_Bit typecaster are used as follows:

```
    An_Enumeration :=
        <Type>_Bit_Typecaster.Value(Enumeration_Symbolic_Image);

    Enumeration_Symbolic_Image :=
        <Type>_Bit_Typecaster.Image(An_Enumeration);

    Check_Symbolic_Image :=
        <Type>_Bit_Typecaster.Check(Enumeration_Symbolic_Image);


    An_Enumeration :=
        <Type>_Bit_Typecaster.Value(Enumeration_Natural_Image);

    Enumeration_Natural_Image :=
        <Type>_Bit_Typecaster.Image(An_Enumeration);

    Check_Natural_Image :=
        <Type>_Bit_Typecaster.Check(Enumeration_Natural_Image);
```

B.3.6. Math_On_Enumeration Typecaster Template

The Math_On_Enumeration Typecaster Template is shown in Appendix C.3.6 and resides in the file named

```
MATH_ON_ENUMERATION_TEMPLATE_.ADA
```

The name of a typecaster generated from the Math_On_Enumeration typecaster template will result from substitution of the placeholder <*Type*> as in

```
<Type>_Typecaster => Scaled_Integer_100_1000_Typecaster
```

The name of the scaled integer type, generated in the Math_On_Enumeration typecaster, will result from substitution of the placeholder <*Type*> as in

```
<Type>_Type => Scaled_Integer_100_1000_Type
```

Capabilities

This generated package provides the capability to convert between character strings representing enumerated values (symbolic images) and scaled integers (values). It also provides the capability to check character strings for symbolic image validity. The Ada type checking insures the validity of the scaled integer values.

The generated package will also provide the capability to convert between character string images of the scaled integers (natural images) and scaled integers (values). It also provides the capability to check character strings for natural image validity.

When to Use the Template

Use this template to create a package for casting between character-based symbolic images, which can be represented as an enumerated type, and values of a specified integer range, where the value is a scaled integer. Also for casting between character-based natural images and values of a scaled integer.

Type *<Type>_Type* will specify the integer range of the scaled integer values. Type *<Type>_Map* will specify the enumerated values corresponding to the symbolic images. These are for instantiating the *Math_On_Enumeration_Typecaster* generic.

There must be an explicit one-to-one positional mapping between the character-based symbolic images and scaled integer values after the factor is applied. Valid symbolic images are expressed by the type *<Type>_Map* and are mapped positionally to the to the range of scaled integer values specified by *<Type>_Type* after the factor is applied. There is an implicit mapping between natural images and the discrete values, e.g.,

```
subtype Scaled_Integer_100_1000 is Integer range 100..1000;
type The_Map is (A,B,C,D,E,F,G,H,I,J);

symbolic                          natural
  image <=> value                  image    <=>  value
-----------------                --------------------
   "A"  <=>  100                 "  100"  <=>  100
            .                                .
            .                                .
            .                                .
   "J"  <=> 1000                 " 1000"  <=> 1000
```

How to Generate an Instance of a Typecaster

The following lines describe how to create an instance of a Math_On_Enumeration Typecaster from the Math_On_Enumeration typecaster template.

The following global placeholder substitutions need to be made:

- Replace the placeholder *<Type>* with the scaled integer type, e.g., *Scaled_Integer_100_1000*. It is not necessary to append *_Type* to the placeholder *<Type>*, the template contains *_Type* in the correct places. See Section B.2 for more details.
- Replace *<First>* with the starting value of the range of the scaled integer type *<Type>_Type*, e.g., "100".
- Replace *<Last>* with the ending value of the range of the scaled integer type *<Type>_Type*, e.g., "1000".
- Replace *<Factor>* with the scaling factor. The default factor is 100.
- Do a search for *??* and fill in the necessary information:
    - the definition of *<Type>_Map*, the symbolic images expressed as an enumeration type, e.g., (A, ...).

One test procedure is provided in the template, *<Type>_Typecaster_Test*. The test procedure first does an exhaustive test on the entire range of values for the type *<Type>_Type*. Then, the procedure allows interactive testing by prompting for symbolic images, natural images and scaled values of integers.

Appendix D.4.2 shows an example instance *(Scaled_Integer_100_1000_Typecaster)* of the Math_On_Enumeration typecaster template.

How to Use the Typecaster

The Math_On_Enumeration typecaster can be used by providing the following statements and declarations within a compilable unit:
NOTE: prior to placing the following lines within the compilable unit, replace the *<Type>* placeholder, shown below, with the proper Ada type.

```
with <Type>_Typecaster;

A_Scaled_Integer: <Type>_Typecaster.<Type>_Type;

Scaled_Integer_Symbolic_Image: <Type>_Typecaster.Symbolic_Image;
Check_Symbolic_Image: <Type>_Typecaster.Symbolic_Image;

Scaled_Integer_Natural_Image: <Type>_Typecaster.Natural_Image;
Check_Natural_Image: <Type>_Typecaster.Natural_Image;
```

The functions available in the Math_On_Enumeration typecaster are used as follows:

```
A_Scaled_Integer:=
    <Type>_Typecaster.Value (Scaled_Integer_Symbolic_Image);

Scaled_Integer_Symbolic_Image:=
    <Type>_Typecaster.Image (A_Scaled_Integer);

Check_Symbolic_Image :=
    <Type>_Typecaster.Check (Scaled_Integer_Symbolic_Image);


A_Scaled_Integer:=
    <Type>_Typecaster.Value (Scaled_Integer_Natural_Image);
Scaled_Integer_Natural_Image:=
    <Type>_Typecaster.Image (A_Scaled_Integer);

Check_Natural_Image :=
    <Type>_Typecaster.Check (Scaled_Integer_Natural_Image);
```

B.3.7. String_Map Typecaster Template

The String_Map Typecaster Template is shown in Appendix C.3.7 and resides in the file named

```
    STRING_MAP_TEMPLATE_.ADA
```

The name of a typecaster generated frc m the String_Map typecaster template will result from substitution of the placeholder *<Type>* as in

```
    <Type>_Typecaster => Status_Typecaster
```

The name of the discrete type generated in the String_Map typecaster will result from substitution of the placeholder *<Type>* as in

```
    <Type>_Type => Status_Type
```

Capabilities

This generated package provides the capability to convert between character strings representing discrete values (symbolic images), as defined in *<Type>_Map*, and discrete values (values). It also provides the capability to check character strings for symbolic image validity. The Ada type checking insures the validity of discrete values.

The generated package will also provides the capability to convert between character string images of discrete values (natural images), as defined in *<Type>_Type*, and discrete values (values). It also provides the capability to check character strings for natural image validity.

When to Use the Template

This template is primarily used when:

1. The user wishes to map from character-based integer symbolic images to enumeration values.
2. One of the possible character-based symbolic images is an Ada key word, e.g., "do" or "if".
3. The possible character-based symbolic images are case sensitive, i.e., "AA" and "aa" are valid symbolic images with different meanings, or one is legal and the other is not.

It is also important to note that this template can be used to replace all other character-based discrete typecasters templates. The reason the other generic typecasters exist is because less is involved to define the mapping. Performance studies may indicate that this typecaster should be used in preference to the other character-based typecasters.

This typecaster handles discrete types with a range specified by the type *<Type>_Type*. The symbolic image portion of the mapping information is defined in *<Type>_Map*. These are for instantiating the *Sting_Map_Typecaster* generic.

To provide the above mentioned capability the user must explicitly define the map between the symbolic images and the discrete values. There is an implicit mapping between natural images and the discrete values, e.g.,

```
type Status_Type is (Operational, Non_Operational);
Status_Map : Status_Map_Type := (
     Operational     => "0"
     Non_Operational => "1");

symbolic                          natural
  image <=>    value                image          <=>   value
----------------------------   -----------------------------------
   "0"  <=>  Operational       "    Operational" <=> Operational
   "1"  <=>  Non_Operational   "Non_Operational" <=> Non_Operational
```

How to Generate an Instance of a Typecaster

The following lines describe how to create an instance of a String_Map Typecaster from the String_Map typecaster template.

The following global placeholder substitutions need to be made:

- Replace the placeholder *<Type>* with the String_Map type, e.g., *Status*. It is not necessary to append *_Type* to the placeholder *<Type>*, the template contains *_Type* in the correct places. See Section B.2 for more details.
- Replace *<Length>* with the length of the string images, e.g., "1".
- Do a search for *??* and fill in the necessary information:
  - the definition of *<Type>_Type*, range of discrete values, e.g., "(Operational, Non_Operational)"
  - the definition of *<Type>_Map*, .e.g, Operational => "0"...

One test procedure is provided in the template, *<Type>_Typecaster_Test*. The test procedure first does an exhaustive test on the entire range of values for the type *<Type>_Type*. Then, the procedure allows interactive testing by prompting for symbolic images, natural images and discrete values.

Appendix D.1.6 shows an example instance (*Status_Typecaster*) of the String_Map typecaster template.

How to Use the Typecaster

The String_Map typecaster can be used by providing the following statements and declarations within a compilable unit:
NOTE: prior to placing the following lines within the compilable unit, replace the *<Type>* placeholder, shown below, with the proper Ada type.

```
with <Type>_Typecaster;

A_Discrete : <Type>_Typecaster.<Type>_Type;

Discrete_Symbolic_Image : <Type>_Typecaster.Symbolic_Image;
Check_Symbolic_Image : <Type>_Typecaster.Symbolic_Image;

Discrete_Natural_Image : <Type>_Typecaster.Natural_Image;
Check_Natural_Image : <Type>_Typecaster.Natural_Image;
```

The functions available in the String_Map typecaster are used as follows:

```
A_Discrete := <Type>_Typecaster.Value(Discrete_Symbolic_Image);

Discrete_Symbolic_Image := <Type>_Typecaster.Image(A_Discrete);

Check_Symbolic_Image :=
    <Type>_Typecaster.Check(Discrete_Symbolic_Image);


A_Discrete := <Type>_Typecaster.Value(Discrete_Natural_Image);

Discrete_Natural_Image := <Type>_Typecaster.Image(A_Discrete);

Check_Natural_Image :=
    <Type>_Typecaster.Check(Discrete_Natural_Image);
```

B.4. Composite Typecaster Templates

The composite typecaster templates are the building blocks of the typecaster model solution. The following is a list of the composite typecaster templates:

1. Array Typecaster Template
2. Private_Array Typecaster Template
3. Record Typecaster Template
4. Private_Record Typecaster Template
5. Wrapper Typecaster Template

All composite typecaster templates are based upon the typecaster model described in Chapter 4. Figures 7-9 and 7-10 show the incomplete Ada package specification that is representative of all composite typecaster templates.

Each composite typecaster template is customized based upon the incomplete Ada package specification shown in Figures 7-9 and 7-10. The customization entails specifying those portions of the incomplete Ada package specification that are in ***bold, italics and boxed***. These customizations are either in the form of compilable Ada code or code template placeholders that the detailed designer must specify when instantiating the template. The customizations are specific to the typecaster operations provided by individual composite typecaster templates.

All composite typecasters are capable of grouping one or any combination of the following:

- an instance of a character-based discrete typecaster
- an instance of a bit-based discrete typecaster
- an instance of a composite typecaster

Therefore, in this section, when the authors refer to *strings*, the reader may substitute bit string and/or character string.

Another item to note is that symbolic images and natural images for composites are defined as the concatenation of the symbolic and natural images of the elements that they group.

B.4.1. Array Typecaster Template

The Array Typecaster Template is shown in Appendix C.4.3 and resides in the file named

```
ARRAY_TEMPLATE.ADA
```

The name of a typecaster generated from the Array typecaster template will result from substitution of the placeholder <*Type*> as in

```
<Type>_Array_Typecaster => Barrier_Segment_Array_Typecaster
```

The name of the Array type generated in the Array typecaster will result from substitution of the placeholder <*Type*> as in

```
<Type>_Array_Type => Barrier_Segment_Array_Type
```

Capabilities

This package provides the capability to convert between strings representing values of the type <*Type*>*_Array_Type* (symbolic images) and values of the type <*Type*>*_Array_Type* (values). It also provides the capability to check strings for symbolic image validity. The Ada type checking insures the validity of the values.

The package also provides the capability to convert between character string images of values of the type <*Type*>*_Array_Type* (natural images) and values of the type <*Type*>*_Array_Type* (values). It also provides the capability to check character strings for natural image validity.

When to Use the Template

Use when the symbolic image is a string representing an array of elements of type <*Type*>*_Array_Type*.

This package depends on a typecaster being defined for the element type, <*Tc*>*_Typecaster*.<*Tc*>*_Type*. This is necessary to access the type of the array elements, Symbolic_Image and Natural_Image for the elements, and the Value, Image and Check functions.

How to Generate an Instance of a Typecaster

The following lines describe how to create an instance of an Array Typecaster from the Array typecaster template.

The following global placeholder substitutions need to be made:

- Replace the placeholder <*Type*> with the Array type, e.g., *Barrier_Segment*. It is not necessary to append *_Array_Type* to the placeholder <*Type*>, the template contains *_Array_Type* in the correct places. See Section B.2 for more details.
- Replace the placeholder <*First*> with the number of the first element of the array, e.g., 1.
- Replace the placeholder <*Last*> with the number of the last element of the array, e.g., 8.

- Replace the placeholder <*Tc*> with the type of the elements of the array, e.g., *Barrier_Segment_Record*. It is not necessary to append *_Type* or *_Typecaster* to the placeholder <*Tc*>, the template contains *_Type* and *_Typecaster* in the correct places. See Section B.2 for more details.
- Do a search for *??* and fill in the necessary information:
    - If the number of elements in the array is less than 8, some lines must be removed. Conversely, if the array has more than 8 elements, equivalent lines must be added to the typecaster template.
    - the canned test cases
    - selection of proper output format (bit or character based) to support test case failure processing

One test procedure is provided in the template, <*Type*>*_Array_Typecaster_Test*. The test procedure does testing based on a set of canned test cases that the user supplies before compilation. The test procedure reports on testing problems, e.g., unexpected results.

Appendix D.4.3 shows an example instance (*Barrier_Segment_Array_Typecaster*) of the Array typecaster template.

How to Use the Typecaster

The Array typecaster can be used by providing the following statements and declarations within a compilable unit:
NOTE: prior to placing the following lines within the compilable unit, replace the <*Type*> placeholder, shown below, with the proper Ada type.

```
with <Type>_Array_Typecaster;

An_Array : <Type>_Array_Typecaster.<Type>_Array_Type;

Array_Symbolic_Image : <Type>_Array_Typecaster.Symbolic_Image;
Check_Symbolic_Image : <Type>_Array_Typecaster.Symbolic_Image;

Array_Natural_Image : <Type>_Array_Typecaster.Natural_Image;
Check_Natural_Image : <Type>_Array_Typecaster.Natural_Image;
```

The functions available in the Array typecaster are used as follows:

```
An_Array := <Type>_Array_Typecaster.Value(Array_Symbolic_Image);

Record_Symbolic_Image := <Type>_Array_Typecaster.Image(An_Array);

Check_Symbolic_Image :=
    <Type>_Array_Typecaster.Check(Array_Symbolic_Image);


An_Array := <Type>_Array_Typecaster.Value(Array_Natural_Image);

Record_Natural_Image := <Type>_Array_Typecaster.Image(An_Array);

Check_Natural_Image :=
    <Type>_Array_Typecaster.Check(Array_Natural_Image);
```

B.4.2. Private_Array Typecaster Template

The Private_Array Typecaster Template is shown in Appendix C.4.4 and resides in the file named

```
PRIVATE_ARRAY_TEMPLATE.ADA
```

The name of a typecaster generated from the Private_Array typecaster template will result from substitution of the placeholder <*Type*> as in

```
<Type>_Private_Array_Typecaster =>
    Probability_Private_Array_Typecaster
```

The name of the Private_Array type generated in the Private_Array typecaster will result from substitution of the placeholder <*Type*> as in

```
<Type>_Private_Array_Type => Probability_Private_Array_Type
```

Capabilities

This package provides the capability to convert between strings representing values of the type <*Type*>*_Private_Array_Type* (symbolic images) and values of the private type <*Type*>*_Private_Array_Type* (values). It provides the capability to check strings for symbolic image validity. The Ada type checking insures the validity of the values. It also provides the capability to convert between the public type, <*Type*>*_Public_Array_Type*, and the private type, <*Type*>*_Private_Array_Type*.

The package also provides the capability to convert between character string images of values of the type <*Type*>*_Private_Array_Type* (natural images) and values of the type <*Type*>*_Private_Array_Type* (values). It also provides the capability to check character strings for natural image validity.

When to Use the Template

Use when the symbolic image is a string representing an array of elements of type <*Type*>*_Public_Array_Type* and inter-element dependencies exist. The private type <*Type*>*_Private_Array_Type* is a private instance of the public type <*Type*>*_Public_Array_Type*.

This package depends on a typecaster being defined for the elements type <*Tc*>*_Typecaster.*<*Tc*>*_Type*. This is necessary to access the type of the array elements, Symbolic_Image and Natural_Image for the elements, and the Value, Image and Check functions.

The type <*Type*>*_Private_Array_Type* is private because there are dependencies among the elements of the array. The data being private allows only this package to monitor and enforce the inter-element dependencies of the array. The public type, <*Type*>*_Public_Array_Type*, provides access to a readable and writable copy of the data.

How to Generate an Instance of a Typecaster

The following lines describe how to create an instance of a Private_Array Typecaster from the Private_Array typecaster template.

The following global placeholder substitutions need to be made:

- Replace the placeholder <*Type*> with the Private_Array type, e.g., *Probability_Wrapper*. It is not necessary to append *_Private_Array_Type* to the placeholder <*Type*>, the template contains *_Private_Array_Type* in the correct places. See Section B.2 for more details.
- Replace the placeholder <*First*> with the number of the first element of the private array, e.g., 1.
- Replace the placeholder <*Last*> with the number of the last element of the private array, e.g., 8.
- Replace the placeholder <*Tc*> with the type of the elements of the private array, e.g., *Probability_Wrapper*. It is not necessary to append *_Type* or *_Typecaster* to the placeholder <*Tc*>, the template contains *_Type* and *_Typecaster* in the correct places. See Section B.2 for more details.
- Do a search for *??* and fill in the necessary information:
  - If the number of elements in the array is less than 8, some lines must be removed. Conversely, if the array has more than 8 elements, equivalent lines must be added to the typecaster template.
  - the **Is_Consistent** function body for maintaining inter-element dependencies
  - the canned test cases
  - selection of proper output format (bit or character based) to support test case failure processing

One test procedure is provided in the template, <*Type*>*_Private_Array_Typecaster_Test*. The test procedure does testing based on a set of canned test cases that the user supplies before compilation. The test procedure reports on testing problems, e.g., unexpected results.

Appendix D.4.4 shows an example instance (*Probability_Private_Array_Typecaster*) of the Private_Array typecaster template.

How to Use the Typecaster

The Private Array typecaster can be used by providing the following statements and declarations within a compilable unit:

NOTE: prior to placing the following lines within the compilable unit, replace the <*Type*> placeholder, shown below, with the proper Ada type. Also note that the Ada type that specifies the value in all typecasting operations is of the type <*Type*>*_Private_Array_Type* and not <*Type*>*_Public_Array_Type*. The reason for this is that the typecaster must maintain the validity of the object (i.e., enforce inter-element dependencies).

```
with <Type>_Private_Array_Typecaster;

An_Array, Private_Array :
    <Type>_Private_Array_Typecaster.<Type>_Private_Array_Type;
Public_Array :
    <Type>_Private_Array_Typecaster.<Type>_Public_Array_Type;

Array_Symbolic_Image:<Type>_Private_Array_Typecaster.Symbolic_Image;
Check_Symbolic_Image:<Type>_Private_Array_Typecaster.Symbolic_Image;

Array_Natural_Image : <Type>_Private_Array_Typecaster.Natural_Image;
Check_Natural_Image : <Type>_Private_Array_Typecaster.Natural_Image;
```

The functions available in the Private Array typecaster are used as follows:

```
An_Array :=
    <Type>_Private_Array_Typecaster.Value(Array_Symbolic_Image);

Array_Symbolic_Image :=
    <Type>_Private_Array_Typecaster.Image(An_Array);

Check_Symbolic_Image :=
    <Type>_Private_Array_Typecaster.Check(Array_Symbolic_Image);

Private_Array := <Type>_Private_Array_Typecaster.
    Make_Private(Public_Array);

Public_Array := <Type>_Private_Array_Typecaster.
    Make_Public(Private_Array);


An_Array :=
    <Type>_Private_Array_Typecaster.Value(Array_Natural_Image);

Array_Natural_Image :=
    <Type>_Private_Array_Typecaster.Image(An_Array);

Check_Natural_Image :=
    <Type>_Private_Array_Typecaster.Check(Array_Natural_Image);
```

B.4.3. Record Typecaster Template

The Record Typecaster Template is shown in Appendix C.4.1 and resides in the file named

```
RECORD_TEMPLATE.ADA
```

The name of a typecaster generated from the Record typecaster template will result from substitution of the placeholder *<Type>* as in

```
<Type>_Record_Typecaster => Julian_Date_Time_Record_Typecaster
```

The name of the Record type generated in the Record typecaster will result from substitution of the placeholder *<Type>* as in

```
<Type>_Record_Type => Julian_Date_Time_Record_Type
```

Capabilities

This package provides the capability to convert between strings representing values of the type *<Type>_Record_Type* (symbolic images) and values of the type *<Type>_Record_Type* (values). It also provides the capability to check strings for symbolic image validity. The Ada type checking insures the validity of record values.

The package also provides the capability to convert between character string images of values of the type *<Type>_Record_Type* (natural images) and values of the type *<Type>_Record_Type* (values). It also provides the capability to check character strings for natural image validity.

When to Use the Template

Use when the symbolic image is a string representing a record of type *<Type>_Record_Type.*

<Type>_Record_Type is a record containing the following elements:

```
<E1>
<E2>
<E3>
<E4>
<E5>
<E6>
<E7>
<E8>
```

This package depends on typecasters being defined for all elements, *<En>*, of the Ada record *<Type>_Record_Type.* This is necessary to access the types of the elements, Symbolic_Image and Natural_Image for the elements, and Value, Image and Check functions.

The record elements are defined by the following Ada types:

```
<T1>_Typecaster.<T1>_Type
<T2>_Typecaster.<T2>_Type
<T3>_Typecaster.<T3>_Type
<T4>_Typecaster.<T4>_Type
<T5>_Typecaster.<T5>_Type
<T6>_Typecaster.<T6>_Type
```

```
<T7>_Typecaster.<T7>_Type
<T8>_Typecaster.<T8>_Type
```

How to Generate an Instance of a Typecaster

The following lines describe how to create an instance of a Record Typecaster from the Record typecaster template.

The following global placeholder substitutions need to be made:

- Replace the placeholder *<Type>* with the Record type, e.g., *Julian_Date_Time*. It is not necessary to append *_Record_Type* to the placeholder *<Type>*, the template contains *_Record_Type* in the correct places. See Section B.2 for more details.
- Do the following loop until all record elements are defined:
  - replace *<En>* with the name of the nTH element of the record, e.g., replace *<E1>* with *Julian_Day*
  - replace *<Tn>* with the type of the nTH element of the record, e.g., replace *<T1>* with *Julian_Day*
- Search for and remove lines and code blocks with *<Tn>*'s and *<En>*'s remaining, e.g., if the record has 5 elements, then lines containing *<T6>..<T8>* and *<E6>..<E8>* must be removed. Conversely, if the record has more than 8 elements, equivalent lines must be added to the typecaster template.
- Do a search for *??* and fill in the necessary information:
  - the canned test cases
  - selection of proper output format (bit or character based) to support test case failure processing

One test procedure is provided in the template, *<Type>_Record_Typecaster_Test*. The test procedure does testing based on a set of canned test cases that the user supplies before compilation. The test procedure reports on testing problems, e.g., unexpected results.

Appendix D.2.1 shows an example instance (*Julian_Date_Time_Record_Typecaster*) of the Record typecaster template.

How to Use the Typecaster

The Record typecaster can be used by providing the following statements and declarations within a compilable unit:
NOTE: prior to placing the following lines within the compilable unit, replace the *<Type>* placeholder, shown below, with the proper Ada type.

```
with <Type>_Record_Typecaster;

A_Record : <Type>_Record_Typecaster.<Type>_Record_Type;

Record_Symbolic_Image : <Type>_Record_Typecaster.Symbolic_Image;
Check_Symbolic_Image : <Type>_Record_Typecaster.Symbolic_Image;

Record_Natural_Image : <Type>_Record_Typecaster.Natural_Image;
Check_Natural_Image : <Type>_Record_Typecaster.Natural_Image;
```

The functions available in the Record typecaster are used as follows:

```
A_Record := <Type>_Record_Typecaster.Value(Record_Symbolic_Image);

Record_Symbolic_Image := <Type>_Record_Typecaster.Image(A_Record);

Check_Symbolic_Image :=
    <Type>_Record_Typecaster.Check(Record_Symbolic_Image);


A_Record := <Type>_Record_Typecaster.Value(Record_Natural_Image);

Record_Natural_Image := <Type>_Record_Typecaster.Image(A_Record);

Check_Natural_Image :=
    <Type>_Record_Typecaster.Check(Record_Natural_Image);
```

B.4.4. Private_Record Typecaster Template

The Private_Record Typecaster Template is shown in Appendix C.4.2 and resides in the file named

```
PRIVATE_RECORD_TEMPLATE.ADA
```

The name of a typecaster generated from the Private_Record typecaster template will result from substitution of the placeholder <*Type*> as in

```
<Type>_Private_Record_Typecaster =>
    FooBar_Message_Private_Record_Typecaster
```

The name of the private record type generated in the Private_Record typecaster will result from substitution of the placeholder <*Type*> as in

```
<Type>_Private_Record_Type =>
    FooBar_Message_Private_Record_Type
```

Capabilities

This package provides the capability to convert between strings representing values of the private type *<Type>_Private_Record_Type* (symbolic images) and values of the private type *<Type>_Private_Record_Type* (values). It provides the capability to check strings for symbolic image validity. The Ada type checking insures the validity of record values. It also provides the capability to convert between the public type *<Type>_Public_Record_Type* and the private type *<Type>_Private_Record_Type.*

The package also provides the capability to convert between character string images of values of the type *<Type>_Private_Record_Type* (natural images) and values of the type *<Type>_Private_Record_Type* (values). It also provides the capability to check character strings for natural image validity.

When to Use the Template

Use when the symbolic image is a string representing a record of type <Type>_Public_Record_Type and inter-element dependencies exist.

The private type *<Type>_Private_Record_Type* is a private instance of the public type *<Type>_Public_Record_Type.* *<Type>_Public_Record_Type* is a record containing the following elements:

```
<E1>
<E2>
<E3>
<E4>
<E5>
<E6>
<E7>
<E8>
```

This package depends on typecasters being defined for all elements of the Ada record *<Type>_Public_Record_Type.* This is necessary to access the types of the elements, Symbolic_Image and Natural_Image for the elements, and Value, Image and Check functions.

The public record elements are defined by the following Ada types:

```
<T1>_Typecaster.<T1>_Type
<T2>_Typecaster.<T2>_Type
<T3>_Typecaster.<T3>_Type
<T4>_Typecaster.<T4>_Type
<T5>_Typecaster.<T5>_Type
<T6>_Typecaster.<T6>_Type
<T7>_Typecaster.<T7>_Type
<T8>_Typecaster.<T8>_Type
```

How to Generate an Instance of a Typecaster

The following lines describe how to create an instance of a Private_Record Typecaster from the Private Record typecaster template.

The following global placeholder substitutions need to be made:

- Replace the placeholder *<Type>* with the Private record type, e.g., *FooBar_Message*. It is not necessary to append *_Private_Record_Type* to the placeholder *<Type>*, the template contains *_Private_Record_Type* in the correct places. See Section B.2 for more details.
- Do the following loop until all record elements are defined
    - Replace *<En>* with the name of the nTH element of the record, e.g., replace *<E1>* with *Detection_Confidence*.
    - Replace *<Tn>* with the type of the nTH element of the record, e.g., replace *<T1>* with *Detection_Confidence*.
- Search for and remove lines and code blocks with *<Tn>*'s and *<En>*'s remaining, e.g., if the record has 5 elements, then lines containing *<T6>*..*<T8>* and *<E6>*..*<E8>* must be removed. Conversely, if the record has more than 8 elements, equivalent lines must be added to the typecaster template.
- Do a search for *??* and fill in the necessary information:
    - the **Is_Consistent** function body for maintaining inter-element dependencies
    - the canned test cases
    - selection of proper output format (bit or character based) to support test case failure processing

One test procedure is provided in the template, *<Type>_Private_Record_Typecaster_Test*. The test procedure does testing based on a set of canned test cases that the user supplies before compilation. The test procedure reports on testing problems, e.g., unexpected results.

Appendix D.2.2 shows an example instance (*FooBar_Message_Private_Record_Typecaster*) of the Private_Record typecaster template.

How to Use the Typecaster

The Private Record typecaster can be used by providing the following statements and declarations within a compilable unit:

NOTE: prior to placing the following lines within the compilable unit, replace the *<Type>* placeholder, shown below, with the proper Ada type. Also note that the Ada type that specifies the value in all typecasting operations is of the type *<Type>_Private_Record_Type* and not *<Type>_Public_Record_Type*. The reason for this is that the typecaster must maintain the validity of the object (i.e., enforce inter-element dependencies).

```
with <Type>_Private_Record_Typecaster;

A_Record, Private_Record : <Type>_Private_Record_Typecaster.
    <Type>_Private_Record_Type;
Public_Record : <Type>_Private_Record_Typecaster.
    <Type>_Public_Record_Type;

Record_Symbolic_Image:
    <Type>_Private_Record_Typecaster.Symbolic_Image;
Check_Symbolic_Image:
    <Type>_Private_Record_Typecaster.Symbolic_Image;

Record_Natural_Image:<Type>_Private_Record_Typecaster.Natural_Image;
Check_Natural_Image:<Type>_Private_Record_Typecaster.Natural_Image;
```

The functions available in the Private Record typecaster are used as follows:

```
A_Record :=
    <Type>_Private_Record_Typecaster.Value(Record_Symbolic_Image);

Record_Symbolic_Image :=
    <Type>_Private_Record_Typecaster.Image(A_Record);

Check_Symbolic_Image := <Type>_Private_Record_Typecaster.
    Check(Record_Symbolic_Image);

Public_Record := <Type>_Private_Record_Typecaster.
    Make_Public(Private_Record);

Private_Record := <Type>_Private_Record_Typecaster.
    Make_Private(Public_Record);


A_Record :=
    <Type>_Private_Record_Typecaster.Value(Record_Natural_Image);

Record_Natural_Image :=
    <Type>_Private_Record_Typecaster.Image(A_Record);

Check_Natural_Image := <Type>_Private_Record_Typecaster.
    Check(Record_Natural_Image);
```

B.4.5. Wrapper Typecaster Template

The Wrapper Typecaster Template is shown in Appendix C.4.5 and resides in the file named

```
WRAPPER_TEMPLATE.ADA
```

The name of a typecaster generated from the Wrapper typecaster template will result from substitution of the placeholder *<Type>* as in

```
<Type>_Wrapper_Typecaster => Probability_Wrapper_Typecaster
```

The name of the Wrapper type generated in the Wrapper typecaster will result from substitution of the placeholder *<Type>* as in

```
<Type>_Wrapper_Type => Probability_Wrapper_Type
```

Capabilities

This package provides the capability to convert between strings representing values of the discriminant record type *<Type>_Wrapper_Type* (symbolic images) and values of the discriminant record type *<Type>_Wrapper_Type* (values). It also provides the capability to check strings for symbolic image validity. The Ada type checking insures the validity of record values.

The package also provides the capability to convert between character string images of values of the type *<Type>_Wrapper_Type* (natural images) and values of the type *<Type>_Wrapper_Type* (values). It also provides the capability to check character strings for natural image validity.

When to Use the Template

Use when the symbolic image is a string representing a record of type *<Type>_Record_Type* and the presence of the symbolic image is optional.

The discriminant portion of the record is an enumerated type, *Availability*, whose possible values are *Available* or *Unavailable*. If the information is *Available*, then the type *<Type>_Record_Type* is used for casting. If the information is *Unavailable*, then the type *Unavailability_Reason* is used for casting.

The *Unavailability_Reason* portion of the discriminated record *<Type>_Wrapper_Type* is an enumerated type whose possible values are *Unreported*, i.e., no element existed in the report, or *Reported_Unknown*, i.e., there was an explicit indication that nothing is known about the element.

The *<Type>_Record_Type* portion of the discriminated record *<Type>_Wrapper_Type* is a record containing the following elements:

```
<E1>
<E2>
<E3>
<E4>
<E5>
```

```
<E6>
<E7>
<E8>
```

This package depends on typecasters being defined for all elements of the Ada record *<Type>_Record_Type*. This is necessary to access the types of the elements, Symbolic_Image and Natural_Image for the elements, and Value, Image and Check functions.

The record elements are defined by the following Ada types:

```
<T1>_Typecaster.<T1>_Type
<T2>_Typecaster.<T2>_Type
<T3>_Typecaster.<T3>_Type
<T4>_Typecaster.<T4>_Type
<T5>_Typecaster.<T5>_Type
<T6>_Typecaster.<T6>_Type
<T7>_Typecaster.<T7>_Type
<T8>_Typecaster.<T8>_Type
```

How to Generate an Instance of a Typecaster

The following lines describe how to create an instance of a Wrapper Typecaster from the Wrapper typecaster template.

The following global placeholder substitutions need to be made:

- Replace the placeholder *<Type>* with the Wrapper type, e.g., *Probability*. It is not necessary to append *_Wrapper_Type* to the placeholder *<Type>*, the template contains *_Wrapper_Type* in the correct places. See Section B.2 for more details.
- Do the following loop until all record elements are defined:
  - replace *<En>* with the name of the nTH element of the record, e.g., replace *<E1>* with *Beam_Number*
  - replace *<Tn>* with the type of the nTH element of the record, e.g., replace *<T1>* with *Hour*
- Search for and remove lines and code blocks with *<Tn>*'s and *<En>*'s remaining, e.g., if the record has 5 elements, then lines containing *<T6>*..*<T8>* and *<E6>*..*<E8>* must be removed. Conversely, if the record has more than 8 elements, equivalent lines must be added to the typecaster template.
- Do a search for *??* and fill in the necessary information:
  - The value of the boolean, *Unreported_NoImage_Possible*. If true then the EXR of the message can have a field where no symbolic image is present. When nothing in the EXR is present for the field, the symbolic image will contain *Unreported_Symbolic_NoImage* defined below.
  - The value of the boolean, *Unreported_Image_Possible*. If true then the EXR of the message can have a symbolic image present that indicates that no information regarding the field is available. The symbolic image that indicates this is defined by *Unreported_Symbolic_Image*.
  - The value of an *Unreported_Symbolic_NoImage*. When it is possible for a field in the EXR to not be present, this is the what will be placed in the symbolic image to signify that nothing was present.

- The value of an *Unreported_Symbolic_Image*. When it is possible for a field in the EXR to have a value that signifies that no information regarding the field is available, this is the what will be found in the symbolic image.
- the canned test cases
- selection of proper output format (bit or character based) to support test case failure processing

At least one of the two booleans must be true, otherwise there would be no need to use the wrapper typecaster.

One test procedure is provided in the template, *<Type>_Wrapper_Typecaster_Test*. The test procedure does testing based on a set of canned test cases that the user supplies before compilation. The test procedure reports on testing problems, e.g., unexpected results.

Appendix D.4.5 shows an example instance (*Probability_Wrapper_Typecaster*) of the Wrapper typecaster template.

How to Use the Typecaster

The Wrapper typecaster can be used by providing the following statements and declarations within a compilable unit:
NOTE: prior to placing the following lines within the compilable unit, replace the *<Type>* placeholder, shown below, with the proper Ada type.

```
with <Type>_Wrapper_Typecaster;

A_Record : <Type>_Wrapper_Typecaster.<Type>_Wrapper_Type;

Record_Symbolic_Image : <Type>_Wrapper_Typecaster.Symbolic_Image;
Check_Symbolic_Image : <Type>_Wrapper_Typecaster.Symbolic_Image;

Record_Natural_Image : <Type>_Wrapper_Typecaster.Natural_Image;
Check_Natural_Image : <Type>_Wrapper_Typecaster.Natural_Image;
```

The functions available in the Wrapper typecaster are used as follows:

```
A_Record := <Type>_Wrapper_Typecaster.Value(Record_Symbolic_Image);

Record_Symbolic_Image := <Type>_Wrapper_Typecaster.Image(A_Record);

Check_Symbolic_Image :=
    <Type>_Wrapper_Typecaster.Check(Record_Symbolic_Image);


A_Record := <Type>_Wrapper_Typecaster.Value(Record_Natural_Image);

Record_Natural_Image := <Type>_Wrapper_Typecaster.Image(A_Record);

Check_Natural_Image :=
    <Type>_Wrapper_Typecaster.Check(Record_Natural_Image);
```

B.5. External Representation TV Template

The EXR TV model solution is captured in one template, the Msg_ICD Template. Figure 7-17 shows the Ada package specification template for the Msg_ICD Template.

B.5.1. ICD_Message Template

The ICD_Message template is shown in Appendix C.5 and resides in the file named

```
MSG_ICD_TEMPLATE.ADA
```

The name of the package generated from the ICD_Message template will result from substitution of the placeholder *<Msg_Id>* as in

```
<Msg_Id>_Icd => FooBar_Message_Icd
```

Capabilities

This package provides the capability to convert between strings containing fields and punctuation (an EXR whose format is described by an EXR description) and a UNR. It also provides the capability to check an EXR for validity (i.e., can the information be parsed, field by field, without any problems).

When to Use the Template

Use when there is a need to convert between an EXR, and a UNR.

How to Generate an Instance of a Typecaster

The following lines describe how to create an instance of an EXR TV model solution from the ICD_Message typecaster template.

The following global placeholder substitutions need to be made:

- Replace the placeholder *<Msg_Id>* with the ICD type, e.g., *FooBar_Message*. It is not necessary to append *_Icd* to the placeholder *<Msg_Id>*, the template contains *_Icd* in the correct places. See Section B.2 for more details.
- Replace *<En>* with the name of the nTH field, e.g., replace *<E1>* with *Reporting_Location*.
- Replace *<Number_Of_Cuts>* with the number of cuts that need to be made for the Icd, e.g., replace *<Number_Of_Cuts>* with "0".
- Replace *<Symbolic_Image_Width>* with the width of the symbolic image (UNR). Note that this should be the same as the Symbolic_Image_Width on the message typecaster side. The Symbolic Image is a fixed length image.
- Do a search for *??* and fill in the necessary information:
    - eo_text character — the end-of-text character appearing at the end of the EXR.
    - Fields — defines the field attributes, either bit-based or character-based, for the EXR. It also contains information about the symbolic image, but this is derived from the EXR description and the user need not concern himself. See Section 6.2.1 for an explanation of how to supply this information.

- Cuts — describes which fields to cut in the UNR. See Section 6.2.1 for an explanation of how to supply this information.
- canned test cases (either bit-based or character based)
- selection of the proper output format (bit-based or character-based) to support test case failure processing

See Section 6.2.1 for a more detailed description of how to specify Fields and Cuts.

One test procedure is provided in the template, *<Msg_Id>_Icd_Test*. The test procedure performs canned tests generated by the user on the *<Msg_Id>_Icd* ICD Message Typecaster.

Appendix D.3.1 shows an example instance (*FooBar_Message_ICD*) of the ICD_Message template.

How to Use the Typecaster

The ICD Message typecaster can be used by providing the following statements and declarations within a compilable unit:
NOTE: prior to placing the following lines within the compilable unit, replace the <*Type*> placeholder, shown below, with the proper Ada type.

```
with <Msg_Id>_Icd;
with Casting_Common_Types;

An_Icd : Casting_Common_Types.Icd_Message_Type;
Universal_Image : <Msg_Id>_Icd.Symbolic_Image;
Icd_Check : Boolean;
Bad_Position : Integer;
```

The functions available in the ICD Message typecaster are used as follows:

```
<Msg_Id>_Icd.Check_Icd(An_Icd, Icd_Check, Bad_Position);

Universal_Image :=
    <Msg_Id>_Icd.Extract_Universal_Image(An_Icd);

An_Icd := <Msg_Id>_Icd.Construct_Icd(Universal_Image);
```

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU-SEI-89-TR-12	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-89-TR-20
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST. 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE HANSCOM, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL (If applicable) ESD/XRS1	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS.

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
63752F	N/A	N/A	N/A

11. TITLE (Include Security Classification)
A Model Solution for C^3I Message Translation and Validation

12. PERSONAL AUTHOR(S)
Charles Plinta, Kenneth Lee, Michael Rissman

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
FINAL	FROM ______ TO ______	December 1989	176

16. SUPPLEMENTARY NOTATION
APPENDIX C: MTV Model Solution Ada Code, and APPENDIX D: FooBar Message Ada Code are bound separately.

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This document describes an artifact, the Message Translation and Validation (MTV) model solution. The MTV model solution is a general solution, written in Ada, that can be used in a system required to convert between message representations. These message representations can be character-based, bit-based, and internal (i.e., Ada values).

This document provides designers with enough information to determine whether this solution is applicable to their particular problem. It gives detailed designers the information needed to specify solutions to their particular problem using the MTV model solution. Finally, it describes the MTV model solution in enough detail to enable a maintainer or adapter to understand the solution.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL H. SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630 22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR   EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE