2

Technical Report

CMU/SEI-88-TR-26
ESD-TR-88-027

Carnegie-Mellon University
Software Engineering Institute

Using the Vienna Development Method (VDM) To Formalize a Communication Protocol

Jan Storbank Pedersen
Mark H. Klein

November 1988

This technical report was prepared for the

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

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler
SEI Joint Program Office

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

Table of Contents

List of Tables

Using the Vienna Development Method (VDM) To Formalize a Communication Protocol

Abstract: The Vienna Development Method (VDM) is based upon iterative refinement of formal specifications written in the model-oriented specification language, Meta-IV. VDM is also an informal collection of experiences in formal specification within several application domains. This paper provides an example of how VDM might be used in the area of communications, a new domain for VDM.

1. Introduction

The purpose of this document is to serve as an introduction to certain specification techniques and the specification computer language (Meta-IV) of the Vienna Development Method (VDM) and to further extend the use of VDM by applying it within a new domain. VDM has been applied to the specification of a communication protocol. The protocol is one used for the communication between an inertial navigation system (INS) and an external computer (EC). ~~as defined in~~ [14]. This protocol was chosen because it is a new application area for VDM, and it is an integral part of an existing project at the SEI. The Real-Time Embedded Systems Testbed (REST) Project is implementing the protocol as part of an INS simulator system that is being developed to investigate the use of Ada for embedded systems.

The formal specification is expressed using the specification language Meta-IV of VDM [2]. Knowledge of the specification language is not a prerequisite for reading this document, as we will introduce all the necessary parts of the language in a subsequent section.

Chapter 2 provides an introduction to communication protocols and VDM. It contains a brief overview of the protocol defined in [14] and introduces the central ideas of VDM. The formalization of key concepts in the protocol is discussed. The formal specification is introduced by presenting the necessary parts of Meta-IV, providing examples of the use of Meta-IV, and describing particular techniques and conventions used in the specification.

Chapters 3 through 7 contain the formal specification itself. The model consists of type equations that define the objects to be used in describing the communication protocol, and functions that formalize the protocol by operating on those objects.

Chapter 8 summarizes the area of formal specification of communication protocols and relates our work to previous work in the area. The document closes with Chapter 9, suggesting ideas for future work.

[illegible] INERTIAL NAVIGAT[illegible] AL[illegible]ITHMS SOFTWARE. [illegible]

2. Communication Protocols and VDM

2.1. Communication Protocols

A communication protocol is defined in [15] as:

1. **Data communication.** A formal set of conventions governing the format and relative timing of message exchange between two communications terminals.
2. **Software.** (A) A set of conventions or rules that govern the interaction of processes or applications within a computer system or network. (B) A set of rules that govern the operation of functional units to achieve communication.
3. **Station control and data acquisition.** A strict procedure required to initiate and maintain communication.

The specific communication protocol described in [14] defines rules for communication between an inertial navigation system (INS) and an external computer (EC). The INS receives information from a number of sensors that record data about a ship and, using that data, determines its current velocity and position. The protocol dictates the procedures, periodicity, and format for sending data to an EC. It defines how to establish communication when requested by the EC and how messages are transferred. It defines what each of the two computers must do in case of detected errors (receiving "wrong" input from the other computer). Finally, it defines how the EC can terminate communication.

As specified in [14], communications are performed over a 16-bit interface. In addition, each sixteen bit quantity is either an external function (EF) or data. The EF codes are used to control the communication protocol and to delimit messages. The code identifiers and their functions are listed in Table 2-1. Note that not all the EF codes defined in [14] are used in the INS simulator system. Consequently, the specification limits itself to those in Table 2-1.

The full range of message types and formats is defined in [14]. The INS simulator application uses only some of these message types. The message types that may be transmitted to the EC are listed in Table 2-2. The message types that may be received from the EC are listed in Table 2-3.

Code	Function
ACK	Acknowledge (i.e., received a valid message)
ATTN1	Indicate a time-out condition
ATTN2	Enable communications
ATTN4	Disable communications (sent by EC only)
EOM	End of message
NAK	Not-Acknowledge (i.e., received an incomplete or invalid message)
NRTR	Not ready to receive
RTR	Ready to receive
SOTM	Start of test message
SOM	Start of message

Table 2-1: External Function (EF) Codes

Message Type	Message Contents
Test Message	Contains a fixed pattern to allow checking of communications.
Time and Status Data Message	Contains fields for the time-of-day and various status codes.
Attitude Data Periodic Message	Contains various fields of numerical data pertaining to the (simulated) ship motion.
Navigation Data Periodic Message	Contains fields of numerical data pertaining to the (simulated) ship motion.

Table 2-2: Messages to EC

Message Type	Message Contents
Test Message	Contains a fixed pattern to allow checking of communications.
Select Data Message	Contains fields to select/deselect the periodic messages that may be sent from the INS.

Table 2-3: Messages from EC

2.2. Formal Specification Using VDM

VDM is a formal, mathematically oriented method for specification of systems and development of software. In general, formal specification languages fall into two classes: algebraic languages and model-oriented languages. VDM is a model-based method. Its main idea is that of giving descriptions of software systems and other systems as models. Models are specified as objects and operations on objects, where the objects represent input, output, and internal state of the system. Classes of objects are explicitly defined as so-called "domains," which are like types in a programming language. Model-based methods differ from algebraic and other formal methods in that they explicitly define the types of objects of concern and utilize primitive, predefined operations in defining higher-level operations.

VDM encourages layered, top-down development of systems, based on use of abstraction at the uppermost levels of system description (see [16]).

In this document, we do not explore the stepwise refinement aspects of VDM. Our model constitutes only a requirements specification that may be used as a starting point for implementation.

At the highest level, a specification is typically given as a rather abstract model. The objects do not capture details of representation; they are restricted to capturing only properties necessary for expressing the essential concepts of the operation of the intended software system.

Within a number of specific application areas, such as programming language semantics [3], compiler construction [20], and data bases [4], standard VDM models and guidelines for development exist.

Meta-IV, which is the specification language of VDM, is used for expressing the models [2]. The models are defined using a number of type definitions (for the objects) and function definitions (for the operations). This is different from the algebraic approach to specification, where the models (algebras) are implicitly defined by the properties captured in the axioms of the algebraic specifications.

Meta-IV is aimed at supporting abstraction in writing specifications. Abstraction is obtained through mathematical concepts, such as sets and functions, rather than through the mechanisms offered by any particular implementation language. The abstraction provided by Meta-IV is not oriented toward any particular application area, but rather offers a set of mathematically based primitives that allow the construction of application-specific models.

When using VDM, an abstract model traditionally contains the following components:

- **Semantic domains**. Types that define the objects to be operated on.
- **Invariants**. Functions that limit the set of objects defined by the semantic domains by defining a set of conditions (Boolean functions).
- **Syntactic domains**. Types that define a "language" in which to express commands for manipulating the objects defined by the semantic domains.
- **Well-formedness conditions**. Functions that define when the commands (defined by the syntactic domains) have a well-defined effect.
- **Semantic functions**. Functions that define the effect of commands on the objects defined by the semantic domains.

2.3. Formalization of the Communication Protocol

2.3.1. An Outline of the Basic Approach Used

Communication protocols are not one of the areas for which well-established VDM models or guidelines exist. Mechanisms for specifying communication protocols using other techniques exist. Probably the most well-established technique is the use of some form of a state machine. Obviously, one could "mimic" such a state machine using a VDM model, but we would like to represent the protocol at a level that is closer to the original informal documentation, i.e., without identifying "states" of the communicating components but only specifying "the communication itself."

Previously combinations of Meta-IV and Hoare's CSP [13] have been used to describe systems that involve communication. Our specification uses Meta-IV in its "pure" form without any extensions.

The definition of communication protocols given in Section 2.1 says that a communication protocol is a set of rules or conventions governing information exchange, including the relative timing of information exchanges. We will refer to exchange of information as an "event," and rules related to event ordering and their relative timing can be seen as defining proper sequences (or lists) of such events. The order in which the events occur in the list reflects the order in which they happen. Hence, one way of formalizing a communication protocol is to define all possible event sequences that are defined by the protocol. This is similar to formalizing the static semantics of a programming language by defining all legal programs that can be written in the language [21]. Obviously, the set of event sequences defined by the protocol (just like the set of all legal programs) is infinite and cannot be defined by simple enumeration. Instead the set is defined by first characterizing the set of all possible event sequences, while ignoring the ordering and timing constraints defined by the rules of the protocol (like defining a context free grammar for a programming language), and then restricting that set through Boolean functions (predicates) that define whether a given list of events is in accordance with the protocol (like defining the context conditions of a programming language).

The conditions expressed by the Boolean functions are to be obeyed by any event sequence of arbitrary length. When looking at each such individual sequence, the rules of the protocol can be divided into two groups: one expressing whether each event in the sequence occurs in the right context (defined by events preceding the one in question); and a second expressing whether the sequence is complete, meaning that no additional events are required to happen.

Some of the specific problems associated with formalizing the communication protocol are that a communication channel is generally not error-free, and the formal specification must be able to capture that the information sent from one component in the system is not necessarily identical to that received by the other component. The communication protocol describes how to detect and handle certain errors. The communication protocol also includes time constraints on the behavior of the components. Hence, the notion of "time" must be part of the model. Moreover, the system defined by the communication protocol does not exist in isolation. It has an environment that affects and is affected by the communication. For this particular protocol, the environment includes (at least) the operators of the INS and the EC.

2.3.2. Modeling Communication Errors

The protocol as defined in [14] prescribes the correct communication behavior of the INS and the EC, but this includes specifying the behavior of the INS in cases where it receives messages that are "out of sequence," for example, due to errors on the communication line or incorrect behavior of the EC.

The fact that the messages received are not necessarily those sent (due to possible transmission errors) means that the transfer of a message may be seen differently by the INS and the EC. These two "perspectives" on the events lead to the introduction of two sequences of events: one for events as seen at the INS (e.g., sending to or receiving from the EC), and a similar one for events as seen at the EC. Such a model is appealing because the correctness of the behavior of the INS and EC is decided by how each sees the world (not by what the other one actually did; transmission errors may introduce differences). This approach also captures situations where a message is lost by the channel or spontaneously created (seen by the receiver) due to noise on the channel.

From the point of view of the process issuing an event, such an event is either *correct* or *incorrect* according to the protocol, given the history (sequence) of previous events (issued as well as received). Received events, however, are either *expected* or *unexpected* (but never *incorrect*, since out-of-sequence events must be reacted to in a manner prescribed by the protocol). Hence, unexpected events capture messages that have been distorted due to errors on the communication line or an incorrect behavior of the issuing component. The relation between *correct* and *expected* events is that events that are *correct* for the issuer (given a particular history) are *expected* by the receiver (under the same circumstances).

2.3.3. Modeling "Time"

The concept of a process being willing to wait only a certain amount of time (and then "timing-out") for a response (event) is mentioned several times in [14]. One way of formalizing "time-outs" is to associate a time-stamp with each event, and then let conditions that relate to time-outs interrogate that information and let conditions related to other parts of the protocol ignore the timing information.

Time-stamps allow for the specification of correct behavior based upon the relation between time-stamps associated with certain events. A possible alternative is to introduce a timer and perhaps a clock into the formal specification. Explicitly including timers and clocks is a technique sometimes used in formal descriptions of concurrent systems for defining such events [17]. Using a timer leads to the introduction of explicit timer events such as a "start timer" and a "time-out" event. But these events are not described by the protocol; instead, the protocol describes events that are *caused by* a time-out. Hence, in the context of this communication protocol, introducing such events seems to move the formal description from the problem domain in the direction of a "solution." This led us to use time-stamps in the model.

2.3.4. The Environment and Its Impact on the Model

According to [14], the INS alerts the operator in case certain types of errors are detected. In our model, events such as alerting the operator are called *non-communication events*, i.e., they do not reflect the transfer of a message between the INS and the EC.

Informally speaking, the initiative to communicate (enable communication, initiate the sending of periodic messages, etc.) belongs to the EC, and [14] does not define what causes the EC to take such initiatives. In reality, however, this does not mean that the EC arbitrarily makes such decisions; rather the EC responds to requests from an operator or potentially another environmental influence. Hence, specific EC-operator non-communications events are introduced into the model to reflect the operator's requests (like "Enable Communication"). Thus the "arbitrary" decisions become a part of the environment of the system rather than a part of the system itself.

2.3.5. Periodic Message Transfers

The attitude and navigation messages sent from the INS to the EC are "periodic" messages. In [14] this is described by phrases like "It is transmitted every 61.44 ms" (for attitude messages, [14] pg. 8-36), and "This message is transmitted every 983.04 ms" (for navigation messages, [14] pg. 8-24). There are at least three possible interpretations of periodicity:

1. Within each designated time interval, a message of the relevant type must be transmitted exactly once, and the transmission must be completed before the start of the next interval.
2. The messages are transmitted with a fixed time interval elapsed from the start of one such message transfer sequence to the next.
3. Fixed interval for the data component of the transfer.

The first interpretation is the one used in describing certain real-time scheduling algorithms like the rate-monotonic scheduling algorithm [18]. On the other hand, the second or third interpretation should apply if the receiving component (here the EC) in its use of the data relies on receiving data with fixed inter-arrival times. This is, however, not specified in [14].

Moreover, since the transmission of a periodic message consists of a number of lower-level transmissions, namely SOM_{INS} RTR_{EC} <the data>$_{INS}$ EOM_{INS} (subscripts indicating the issuing component) and a first failed transmission will lead to a retransmission, it is not clear from [14] where the periodicity requirement applies.

In our model, interpretation 1 is used with the additional decision that the time requirement applies to the EOM_{INS} that marks the end of the message.[1]

2.4. Introduction to the Formal Model

In the first part of this section the basics of Meta-IV are introduced, which include primitive and composite types, type constructors, functions, and expressions. The following section broaches the topic of constructing Meta-IV functions by combining the aforementioned constructs. In addition, several techniques that were employed in this specification that capitalize on domain knowledge to reduce specification complexity will be discussed. Finally a list of the conventions used in this specification will be presented.

2.4.1. The Specification Language

This section defines those parts of Meta-IV that are used in the formal model; for a complete definition of Meta-IV, see [2]. The use of each relevant language construct is illustrated by examples drawn from the formal model presented in Sections 3 to 7.

[1] It could be interesting at a later time to see how easily the formal model can be changed to reflect interpretation 2 or 3 instead.

2.4.1.1. Types

Meta-IV offers a number of primitive as well as composite types.

The primitive types used are: natural numbers (N0 and N1), Booleans (BOOL), arbitrary non-decomposable text strings (QUOT), and special simple values (TOKEN). Each of these is characterized in the following.

- **N0** and **N1**. The objects are the natural numbers (including and excluding 0). The numeric literals as well as the traditional arithmetic and relational operators are predefined operations.
- **BOOL**. The objects are the Boolean values. The Boolean literals (true and false), and the logical operators $\wedge$ (and), $\vee$ (or), $\neg$ (not), and $\supset$ (implication), as well as comparison (= and $\neq$), are predefined. The logical operators are *not* commutative. This means that, for example, a $\wedge$ b is defined as: if a then b else false. The non-commutivity of $\wedge$ is utilized in our model, whereas the non-commutivity of $\vee$ is not.
- **QUOT**. The objects are arbitrary non-decomposable text strings. Being non-decomposable means that no operation exists for extracting parts of such a string; as a matter of fact the only available operations are the literals, which are underlined text strings, and equality (=) and inequality ($\neq$). The type QUOT is similar to set types in Pascal or enumeration types in Ada.
- **TOKEN**. The objects are simple values whose representation is not defined. No literals exist for this type. The only predefined operations are equality (=) and inequality ($\neq$).

Based on the above primitive types, other (composite) types can be defined. The following composite types are used in the model: sets, tuples, and trees. Each of these is described below.

- Sets of values of another type. The (postfix) type constructor is -set, and it defines the new type to consist of all finite sets of elements of the argument type. All elements in a set are different and they are not ordered. The predefined operators for sets include:
  1. Equality and inequality.
  2. Set object constructors: $\{e_1,e_2, ... ,e_n\}$ defines a set with *n* elements e_1 to e_n (provided that all the e's are different, otherwise the resulting set contains each (different) element exactly once). {} is the empty set with no elements. {F(i) | P(i)} defines a set whose elements are defined by F(i) for all *i* satisfying the predicate P(i) (a Boolean function); for example, $\{i^{**}2 \mid 2 \leq i \leq 6\}$ is {4,9,16,25,36}.
  3. Set membership: The operator $\in$ is defined for each set type. e $\in$ s expresses that the element e belongs to the set s (the value of the expression e $\in$ s is true if e belongs to s, and false otherwise); for example 5 $\in$ {5,2,1}. Similarly, e $\notin$ s is defined as: e does not belong to s. Hence, for example, 5 $\notin$ {5,2,1} is false.
  4. Subset relation: s1 $\subset$ s2 says that s1 is a proper subset of s2, i.e., all elements of s1 are also elements of s2, but there is at least one element of s2 that is not an element of s1; for example, {2,1} $\subset$ {1,2,5}.

- Tuples (or sequences) of values of another type. The (postfix) type constructor is *, and it defines the new type to consist of all finite sequences of values of the argument type including the empty sequence containing no values.[2] The important characteristic of a tuple is that the elements are ordered, so that one can refer to the i'th element. The following operations are predefined for tuples:

    1. Equality and inequality.
    2. Tuple object constructors: $<e_1,e_2, \ldots ,e_n>$ defines a tuple of length *n* with elements e_1 to e_n. <> is the empty tuple with no elements. $<F(i) \mid n \le i \le m \land P(i)>$ defines a tuple whose elements are defined by F(i) for *i* between *n* and *m* inclusive and satisfying the predicate P(i) (a Boolean function). For example, $<i^{**}2 \mid 2 \le i \le 6 \land$ Is_Even(i)>, where Is_Even is a user defined predicate with the obvious definition, is <4,16,36>. Note that *i* does not define the actual index values in the resulting tuple; the index values are always from 1 the the length of the tuple.
    3. Concatenation: t1 ^ t2 is the tuple resulting from concatenating two tuples t1 and t2; for example <1,5> ^ <2,1> is <1,5,2,1>.
    4. Extraction operations: hd t yields the first element in (or head of) the tuple t (hd <5,2,1> is 5), and tl t yields the remaining part (or tail) of the tuple t (tl <5,2,1> is <2,1>). Both these operations are *partial* in that they are not defined for empty tuples. len t is the length of a tuple t (len <5,2,1> is 3). ind t yields the set of indices for t (the values from 1 to len t). For example ind <5,2,1> is {1,2,3.}

- Named trees[3] (or cartesian products) containing values of other types. A tree is characterized by the types of its component values. All values of a particular tree type have the same number of components (as opposed to tuples where the number of components may vary). Trees are similar to record types in programming languages. The following operations are predefined for trees:

    1. Equality and inequality. Two named trees are equal only if they have the same name and the values of corresponding components are identical.
    2. *Tree object constructors*: Given a tree type defined by A :: B C, one can construct objects of type A by mk-A(b,c), where b and c are values of the types B and C, respectively. For example, a value of type D :: N0 BOOL is mk-D(5,true).
    3. Selectors: Given the above definition of A and a value of that type, one can select the individual components by s- followed by the type name of the component. For example s-B(a) yields the B-component of a value "a" of type A. This means that s-B(mk-A(b,c)) is b.

[2] A similar type constructor exists for defining non-empty tuples. The constructor is "+", but it is not used in this model.

[3] There are also unnamed trees, but we do not use those.

2.4.1.2. Type Equations and Abstract Syntaxes

The types of objects of concern in the model are defined by a number of *type equations*. A type equation defines a type in terms of (other) related types by using some of the type constructors defined in the previous section.

A number of mutually dependent type equations are often called an *abstract syntax*, and the complete set of type equations for our model is an example of such an abstract syntax. A part of the abstract syntax is shown below.

```
Event_List      = Event*

Event           = INS_Event | EC_Event

INS_Event      :: INS_Event_Info Time_Stamp

EC_Event       :: EC_Event_Info  Time_Stamp

INS_Event_Info = Comms_Event | ...

EC_Event_Info  = Comms_Event | ...

Comms_Event    = EF_Event | ...

EF_Event       = ATTN2 | RTR | SOTM | ...

Time_Stamp     = N0
```

The first type equation defines event lists to be tuples of events.

The second type equation defines an event as either an INS event or an EC event. The type constructor "|" defines the new type as consisting of values from either of the involved types. The predefined operations are those of the constituent types and apply only to the types for which they are defined.

The third type equation is an example of a named tree definition which defines an INS event as having two components: INS event information and a time stamp (both defined by other type equations).

The type equation for EF_Event defines the type as consisting of the quotation literals (of type QUOT) present on the right-hand side.

The last type equation defines time stamps as being natural numbers (where a number represents a multiple of .01 milliseconds).

An example of a value of type Event_List is:

```
< mk-EC_Event(ATTN2,2), mk-INS_Event(ATTN2,10),
  mk-EC_Event(SOTM,15), mk-INS_Event(RTR,30) >
```

This describes part of a communication where:

1. EC sends an ATTN2 (at time 2, i.e., at time 0.02 ms)
2. INS responds (at time 10)
3. EC initiates the test message sending sequence (with an SOTM)
4. the INS says that it is ready to receive (RTR) the test message

Each type equation in an abstract syntax implicitly defines a predicate that expresses whether a given value belongs to the type. Its name is is- followed by the type name. In the above example, predicates such as is-INS_Event and is-EC_Event are implicitly defined. These predicates are particularly useful for types such as INS_Event and EC_Event that are used in constructing union types (using |), because the predicates allow one to know the type of such a value; in the example, one can tell whether an event (value of type Event) is an INS_Event or an EC_Event.

At this point the reader should be able to understand the full set of type equations of the model as defined in Chapter 3.

2.4.1.3. Functions

A function is characterized by its name, the types of its parameters (if any) and the type of its result.

The general scheme for defining functions is illustrated by the following example:

```
Time_Stamps_Non_Decreasing(event_list) =

   -- function body

type: Event_List → BOOL
```

The definition gives the name of the function, here Time_Stamps_Non_Decreasing. It names the formal parameter(s), here event_list, and provides a body that defines the effect of the function. It defines the type of function, in this case a function from values of type Event_List to values of type BOOL. "→" is a type constructor that defines the type of all (total) functions from the provided parameter type(s) to the result type. Being "total" means that the function returns a well-defined value for all possible values of its parameters. Partial functions may have undefined result values for some subset of parameter values. Partial function types are constructed by using "$\overset{\sim}{\to}$ " instead of "→". For some of our partial functions we have provided a so-called "pre-condition," which is a Boolean expression defining the conditions under which the partial function is guaranteed to yield a well-defined result. It is written as: "pre: some_boolean_expression" immediately following the type of the partial function.

In some cases where the nature or role of one (or several) parameter(s) of a function is different from the rest, the function may be defined as a "curried" function. This means that instead of defining the type of function as one of, for example, two parameters like "A B → C," its type may be defined as "A → (B → C)," which says that when one applies the function to its first parameter (of type A), one gets a (new) function from the remaining parameter (here of type B) as the result. As an example consider:

```
Event_Is_Correct_at_INS(f_history)(ins_event) =

   let mk-INS_Event(event_info,) = ins_event in
   (is-Comms_Event(event_info) -->
         Comms_Event_Is_Correct_at_INS(f_history)(ins_event),
    T                           -->
         Non_Comms_Event_Is_Correct_at_INS(f_history))

type: Event_List → (INS_Event → BOOL)
```

Here the first parameter, f_history, provides the context in which the correctness of the second parameter, ins_event, is expressed. Hence, Event_Is_Correct_at_INS, when applied to a history, yields the function that expresses the correctness of INS events.

2.4.1.4. Meta-IV Expressions

In addition to the predefined operations and user-defined functions described in the previous sections, some Meta-IV language constructs are available for defining the body of functions. The constructs are: let constructs, if then else constructs, McCarthy conditionals, cases constructs, and quantified expressions. They are introduced below.

- let constructs: A let construct is used to (locally) introduce an identifier within a function definition and to bind the identifier to the value of an expression. A very simple example of its use is:

  ```
  let a = {1,3,8} in
  -- some expression using "a"
  ```

  Note that "a" is not like a variable in a procedural programming language in that one cannot update "a" once it has been bound to a particular value. It is similar to identifiers used in purely functional programming languages.

  A let construct can also be used to decompose composite values, such as trees, and give names to their components. An example of such a use is:

  ```
  let mk-INS_Event(event_info,event_time) = ins_event in
  -- some expression using event_info and event_time
  ```

  where ins_event is a name of a value of type INS_Event (maybe a parameter of a function), and event_info and event_time are two new names for the components of the event. This decomposition can be seen as an alternative to the use of (several) selectors (see Section 2.4.1.1.). In case a particular component is not used in the following expression, it need not be named on the left-hand side of the let construct (leaving the space blank where it would have appeared).

  Another form of let construct defines the selection of a value that has certain properties (defined by a predicate) and binds it to a name. Its general form is:

  ```
  let id ∈ some_set_or_type be s.t. P(id) in
  -- some expression using id
  ```

which is read: "let id belonging to some_set_or_type be such that P(id) holds in the following expression," where some_set_or_type is an arbitrary set or type and P is a predicate whose value (presumably) depends on id.

- if then else constructs: Conditional expressions of the form:

```
if boolean_expression then
   expression_1
else
   expression_2
```

can be used with the obvious semantics. Note that since the whole construct is an expression, both expression_1 and expression_2 (the else-part) must be present.

- McCarthy conditionals: A form of conditional expression that allows more than two alternatives is the McCarthy conditional. Its general form is:

```
(boolean_expression_1 --> expression_1,
(boolean_expression_2 --> expression_2,
  .
  .
  .
(boolean_expression_n --> expression_n)
```

The value of such an expression is the value of the first expression_i from the top whose $\text{boolean\_expression}_i$ has the value true. If none of them are true the value is undefined, but it is up to the user of Meta-IV to ensure that this does not happen. A special symbol "T" may be used as the $\text{boolean\_expression}_n$ to catch all remaining conditions.

An example of its use in the model is:

```
Data_Event_Is_Correct_at_INS(f_history)(ins_event) =

   let mk-INS_Event(event_info, ) = ins_event in
   ( is-Test_Msg(event_info)              -->
        Test_Msg_Is_Correct_at_INS(f_history)(event_info),

     is-Time_and_Status_Msg(event_info) -->
        Time_and_Status_Msg_Is_Correct_at_INS(f_history),

     is-Attitude_Msg(event_info)          -->
        Attitude_Msg_Is_Correct_at_INS(f_history)
                                          (event_info),

     is-Navigation_Msg(event_info)        -->
        Navigation_Msg_Is_Correct_at_INS(f_history)
                                            (event_info),

     T                                      --> false )

type: Event_List → (INS_Event → BOOL)
```

The McCarthy conditional uses the predefined is- operation to distinguish the four kinds of event information that correspond to (potentially) correct data events from the INS. The "T" covers situations where the event information is not one of the four messages explicitly mentioned; it could be a Select_Data_Msg or an Error Data Message: see Chapter 3.

- cases constructs: One of the most used forms of conditional expressions in the model is the cases construct. Its general form is:

```
cases select_expression:
   (expression_or_pattern$_1$ --> expression$_1$,
   (expression_or_pattern$_2$ --> expression$_2$,
    .
    .
    .
   (expression_or_pattern$_n$ --> expression$_n$)
```

The value of such an expression is the value of the first expression$_i$ from the top whose expression_or_pattern$_i$ is either an expression whose value is equal to that of the select_expression or a pattern that matches the value of the select_expression. Patterns are like expressions (typically of composite types). The difference is that they may contain unbound identifiers or simply empty spaces. Unbound identifiers are bound by the pattern so that they may be used in expression$_i$ for referring to the corresponding components. Empty spaces are used for components that are of no importance in expression$_i$. If none of the expressions or patterns matches the select_expression, the value of the whole cases expression is undefined, but it is up to the user of Meta-IV to ensure that this does not happen. A special symbol "T" may be used as expression_or_pattern$_n$ to catch the remaining possible values of select_expression.

An example of its use in the model is:

```
SOM_Is_Correct_When_Periodic_Msgs_Are_Activated_at_INS
                                     (f_history)(som_time) =

   cases Number_Of_Outstanding_SOMs_Sent_at_INS(f_history):
      (0 -->
          cases Last_n(2,f_history):                     --6.3.2.2.a
             ( < mk-EC_Event( , ),
                 mk-INS_Event(ACK, ) >        -->
                    true,

               < mk-INS_Event(ATTN1, ),
                 mk-INS_Event(event_info, ) > -->
                    is-Signal_to_INS_Operator(event_info),

               < mk-INS_Event( , ),
                 mk-EC_Event(ACK, ) >         -->
                    true,

               < mk-EC_Event(ATTN1, ),
                 mk-EC_Event(event_info, ) >  -->
                    is-Signal_to_EC_Operator(event_info),

               T                              -->
                    false ),

       1 -->
          cases Last(f_history):
             ( mk-INS_Event(ef, )             -->
                  ef = ATTN1,

               mk-EC_Event(ef, ef_time)       -->
                  ef = ATTN1
                  ∨
                  ef = NAK
                  ∨
                  ( ef = NRTR ∧
                    som_time - ef_time > Sleep_Period )),

       T --> false )

type: Event_List → (Time_Stamp →  BOOL)
```

The function has an outer cases construct governing a choice that depends on the number of currently outstanding SOMs (0, 1 or more). If the number is 0, a cases construct utilizing pattern matching is used. It "looks" at the last two events; if they match one of the four explicit patterns, each being a two-component tuple, the value of the corresponding right-hand side is the function result. Note that in two of the patterns the identifier "event_info" occurs. This is an unbound identifier that is being bound by the match and is used in the right-hand side expression. If the number of outstanding SOMs is 1, only the last event is of interest, and the condition depends on whether it is an INS event or an EC event.

- Quantified expressions: Both existential and universal quantifications are used. Their general form is:

```
(∃ id ∈ some_set_or_type)(P(id))

(∀ id ∈ some_set_or_type)(P(id))
```

which are read: "there exists a value (id) belonging to some_set_or_type for which P holds" and "for all values (id) belonging to some_set_or_type P holds," where P is a predicate that depends on id. In both expressions more than one identifier may be used in place of "id."

An example of its use in the model is:

```
Time_Stamps_Non_Decreasing(event_list) =

    (∀ i,j ∈ ind event_list)
       ((i < j) ⊃
          (s-Time_Stamp(event_list[i]) ≤
                              s-Time_Stamp(event_list[j])))

type: Event_List → BOOL
```

The function uses the universal quantifier to express that for all possible pairs of positions in an event list (indices in a tuple), it must be the case that if one is lower than the other, then the time stamp associated with the event in the first position is less than or equal to the time stamp of the event in the second position.

2.4.2. Example Meta-IV Function Descriptions

Three functions from the formal specification are discussed in detail. Each function is introduced with a textual description of its purpose. Each of its constituent parts is then discussed to illustrate the use of Meta-IV in formalizing the original textual description.

2.4.2.1. ACK_Is_Correct_at_INS

The correctness of a number of events depends only on a few of the immediately preceding events and not on the whole history of earlier events. Moreover, in many cases the actual time at which those events occurred is unimportant. The following function illustrates both aspects. It expresses the conditions under which an $\mathrm{ACK}_{\mathrm{INS}}$ may occur. [14] states that an $\mathrm{ACK}_{\mathrm{INS}}$ is issued by the INS after it has received a valid test message or select data message (both are of the type Data_Message in our model).

```
0. ACK_Is_Correct_at_INS(f_history) =

1.     cases Last_n(2,f_history):
2.        ( < mk-EC_Event(data_msg, ),        --6.3.2.2.c[4]
3.            mk-EC_Event(EOM, ) > -->
4.              Valid_Data_Message(data_msg),
5.          T                            -->
6.              false )

type: Event_List → BOOL
```

Annotations:

- The correctness of an ACK at the INS is expressed as a function of event history only, since there are no timing constraints (line 0).
- Only the last two elements in the history are considered (line 1). (Note that the function Last_n returns the entire history if the history has less than two elements.)
- The last two events must have been EC events (lines 2-3), and the last one must have been an EOM that terminates the message (line 2). The identifier "data_msg" is bound to the event information present in the second to the last event (line 2).
- Note that since the time stamps are of no concern, no identifiers are provided for those components in the pattern. Pattern matching allows for comparing event sequences while ignoring time stamps. This scheme is used throughout the specification.
- The event information in the second to the last event must be a valid data message (line 4). Also notice that the presence of an EOM_{EC} implies that the preceding event must have been a Data_Message from the EC.
- If the last two events did not match the pattern, an ACK is not correct, i.e., false is returned (lines 5-6). This includes the case where the history contains fewer than two elements.

2.4.2.2. Protocol_Obeyed_at_INS

The INS obeys the protocol if all events initiated at the INS are correct in the context of history (the tuple of previously occurring events).

[4] This Ada-like commment is a reference to a paragraph in [14]. References of this form are used throughout the formal model.

```
0. Protocol_Obeyed_at_INS(history)(events_at_ins) =

1.     (events_at_ins ≠ <>) ⊃
2.        (let first_event = hd events_at_ins in
3.         (is-INS_Event(first_event) ⊃
4.             Event_Is_Correct_at_INS(Filter_Event_List_at_INS(history))
                                       (first_event))
5.        ∧
6.        Protocol_Obeyed_at_INS(history ^ <first_event>)
                                (tl events_at_ins))

type: Event_List → (Event_List → BOOL)
```

Annotations:

- This function uses two event lists: one is a list of events as seen at the INS; the other is a list of events occurring prior to the first list as seen at the INS (line 0). When the function is first called, the history is empty, <>, and the "events_at_ins" is the complete list of events as seen at the INS.
- The function is defined using an implication starting on line 1. Recall that if the antecedent of the implication is false, then the entire implication is true. In fact the implication can be false only if the antecedent is true and the consequent is false. The antecedent states that the event list at the INS is not empty. This means that if the event list *is* empty, the protocol is obeyed by the INS. The consequent is the rest of the function.
- Given that the event list is not empty (line 1), the first event is the head of the list and the identifier "first_event" is introduced as a name for the first element (line 2).
- Since at the INS only the correctness of INS events is of interest, another implication is used to express the following: if the first event is an INS event (line 3), it must be correct in the context of the history when ignorable EC events have been removed (line 4). (See Section 2.4.3.1 for further details on the removal of ignorable events.)
- In addition to the first event being correct if it is an INS event, the remaining list of events must be correct (line 6). This is expressed by calling the function recursively with an extended history (adding the "first_event" to the history tuple) and the remaining events at the INS (tl events_at_ins). The recursion leads to the "events_at_ins" parameter eventually becoming the empty tuple, which will terminate the recursion by the condition of line 1.

2.4.2.3. Initiating_EF_Not_Too_Soon_at_EC

There are several EC events that we call initiating events. Specifically these are EF events issued by the EC that initiate an interaction with the INS. The IDS states that no more than two of these initiating EF events may occur within one second. These initiating events are SOM_{EC}, SOTM_{EC} and $\mathrm{ATTN2}_{EC}$.

```
0. Initiating_EF_Not_Too_Soon_at_EC(f_history)(current_time) =

1.    (∀ i,j ∈ ind f_history)
2.        ((is-EC_Event(f_history[i]) ∧ is-EC_Event(f_history[j])
3.          ∧ i < j
4.          ∧ current_time - s-Time_Stamp(f_history[i]) ≤
                      Min_Initiating_Pair_Separation_Time ) ⊃

5.              (let mk-EC_Event(event_info_i, ) = f_history[i] in
6.               let mk-EC_Event(event_info_j, ) = f_history[j] in
7.               ¬ {event_info_i,event_info_j} ⊂
                                          {SOM,SOTM,ATTN2}))

type: Event_List → (Time_Stamp → BOOL)
```

Annotations:

- The aforementioned EF initiation rate is one criterion for the correctness of these initiating EFs. This function is a predicate that returns true if the condition is satisfied and false otherwise. It is called with an event history and the time associated with the initiating event under scrutiny (line 0).
- The condition can be restated in terms of events in an event list as follows: given any two events, e1 and e2, such that e1 occurred before e2 and e1 occurred within one second of the event under scrutiny, then they should not both be initiating events.
- Line 1 expresses this in Meta-IV using universal quantification over the indices of the event history. It states that for all i and j that are elements of the index set of the f_history, some condition must be satisfied.
- The condition is an implication that begins on line 2. Recall that if the antecedent of the implication is false, then the entire implication is true. In fact the implication can be false only if the antecedent is true and the consequent is false.
- The antecedent states that the i'th and j'th events should be EC events (line 2), that the i'th event occurs before the j'th event (line 3), and that the i'th event is within a specified amount of time (one second in this case) from the current_time (line 4). This amount of time is specified by the constant function Min_Initiating_Pair_Separation_Time.
- The consequent states that the set comprised of the i'th and j'th event should not be a subset of the set of initiating EFs (lines 5-7).

2.4.3. Basic Techniques Applied in the Model

2.4.3.1. Filtering of Event Lists

A general property of the protocol is that EFs that are received out of sequence at the INS (for example, due to a transmission error) are to be ignored by the INS. Out of sequence EFs received at the EC may either be ignored or responded to by an ATTN1. To reflect the effect of ignoring EFs that are out of sequence and at the same time to simplify the functions in the model, we decided to explicitly remove ignorable EFs from the history before expressing the correctness of the following event. We refer to this process as "filtering." The filtering functions for the INS and the EC are defined in Sections 5.3 and 6.3, respectively.

2.4.3.2. Completeness of Event Lists

The conditions defined by the correctness functions of the model basically express whether an event is *allowed* to occur in a given context, and not whether it is *required* to happen. This approach takes care of certain forms of non-determinism, i.e., contexts where one of several next events is possible, for example, in the case that the EC receives an EF that is out of sequence and may respond in either of two ways. The approach, however, means that even if all events in the two sequences are correct, additional events may be required by the protocol. This is not captured by the correctness functions. An example is the required sending of periodic messages. The concept of such "required events" is expressed in the model by a completeness criterion that defines for the two correct event lists (at the INS and at the EC) whether more events are required to happen. It is formalized by the function Is_Complete, which is defined in Section 4.2.

2.4.3.3. Use of Constant Functions

In [14] several specific numbers are used to define time intervals for required response times, periodicity, etc. To symbolically represent the numbers, we have defined a set of parameterless, constant functions. The names of these functions have been chosen to best reflect the role of the number, for example, the function "Attitude_Period" is defined as part of describing the periodicity of attitude messages (its value is 6144, measured in 0.01 millisecond units). The constant functions are defined in Section 7.2.

2.4.4. Structure of the Formal Model

The formal specification of the INS communication protocol is presented in the remaining chapters. Chapter 3 contains the type equations. Chapter 4 defines functions that formalize the protocol at the systems level by considering both the INS and the EC. Chapters 5 and 6 define functions that are specific to the INS and the EC, respectively. Chapter 7 contains general functions, some that express important constants defined by the protocol, some that provide elementary operations on one of the types defined in Chapter 3, and finally some that are auxiliary functions used by several functions in Chapters 4 to 6. An index including all functions is also provided.

The type equations are supplemented with annotations, and each function is accompanied by a rationale section. Also, when appropriate, functions are augmented with references of the form -- 6.3.2.1.c. These are references to sections in [14]. A number of conventions that are not dictated by VDM have been used to increase the readability of the model. They are:

- Type names in our model always begin with a capital letter.
- Function names always begin with a capital letter.

- Formal parameters and objects are always in lowercase letters.
- All functions specific to the INS have a name ending in "_at_INS". Similarly, names of EC-specific functions end in "_at_EC".
- A history that has been filtered is always referred to as "f_history".

It may be helpful to read the functions using the following strategy:

- Read the rationale for a function before trying to understand the function itself. Functions tend to be decomposable into logical segments. The rationale mirrors this decomposition.
- Read the top level functions in detail (Chapter 4). Then read the first three sections of INS functions and EC functions (Chapters 5 and 6, respectively). Become familiar with the function hierarchy. Read selected portions in detail.
- Keep in mind that the specification is basically three predicates (the top level functions) that are being applied to two sequences of events. The idea is that an arbitrarily long communication session between the EC and the INS is being examined to determine whether it adheres to the protocol. This is expressed by looking at each event (one by one) in the context of all previous events (the event history) and examining the communication event sequence as viewed from the INS and as viewed from the EC. Note that all events must obey the protocol. Thus, as soon as one event fails to satisfy the conditions, the predicate returns false and no further examination is necessary. This generates the following assumptions that are used throughout the specification. If the event sequence represents the INS perspective, then all events generated by the INS that are in the event history satisfy the protocol. If the event sequence represents the EC perspective, then all events generated by the EC that are in the event history satisfy the protocol.

3. Formal Model Type Equations

```
Event_List      =   Event*

Event           =   INS_Event | EC_Event

INS_Event      ::   INS_Event_Info  Time_Stamp

EC_Event       ::   EC_Event_Info   Time_Stamp

INS_Event_Info =   Comms_Event | INS_Non_Comms_Event

EC_Event_Info   =   Comms_Event | EC_Non_Comms_Event

Comms_Event     =   EF_Event | Data_Message

EF_Event        =   ACK | ATTN1 | ATTN2 | ATTN4 | EOM |
                    NAK | NRTR | RTR | SOM | SOTM | Error EF

Data_Message    =   Periodic_Data_Message | Non_Periodic_Data_Message |
                    Error Data Message

Periodic_Data_Message     = Attitude_Msg | Navigation_Msg

Non_Periodic_Data_Message = Test_Msg | Time_and_Status_Msg | Select_Data_Msg

Attitude_Msg          :: Data

Navigation_Msg        :: Data

Test_Msg              :: Data

Time_and_Status_Msg :: Data

Select_Data_Msg       :: Data Selected_Msg_Type

Selected_Msg_Type = Attitude | Navigation | None | Both

Data          = Data_Element*

Data_Element = TOKEN

INS_Non_Comms_Event = Signal_to_INS_Operator

Signal_to_INS_Operator :: QUOT

EC_Non_Comms_Event        = Signal_from_EC_Operator | Signal_to_EC_Operator

Signal_from_EC_Operator = Enable Comms | Disable Comms | Send Test
                          Select Attitude | Select Navigation |
                          Select None | Select Both

Signal_to_EC_Operator :: QUOT

Time_Stamp = N0
```

Annotations:

- Communication between the two computers is being modeled as a time-ordered sequence of communications-related events. Thus the highest level and most pervasive object type in the specification is an Event_List. An Event_List is defined as a tuple of Events.
- Each Event is associated with one of the two computers, the INS or the EC. Consequently an event is either an INS_Event or an EC_Event.
- In addition to being able to model sequential ordering in time, there are circumstances that necessitate modeling time, i.e., the time at which an Event occurred. Thus Events (both INS_Events and EC_Events) are modeled as two-component trees. The first component captures information that allows further characterization the of event itself: INS_Event_Info and EC_Event_Info. The second component, Time_Stamp, represents the time at which the event occurred.
- Both INS_Events and EC_Events may be communications events or non-communications events. Comms_Events model those events that are issued by one computer and may directly affect the other computer. These are either EF_Events or Data_Messages. The valid EFs are enumerated. An extra EF_Event event is included to capture incorrect EFs.

  Non-communications events (INS_Non_Comms_Event and EC_Non_Comms_ Event for the INS and EC, respectively) represent interaction with a console operator or some other external influences on one computer that are not from the other computer.
- Data_Messages are further subdivided into periodic and non-periodic messages, which are enumerated.

  Each message type is defined as a tree. Attitude, navigation, test, and time and status messages are single component trees, the component being the data communicated in each message, respectively. Data are later defined as a tuple of Data_Elements, which are TOKENs. Recall that the representation of TOKENs is undefined. Therefore the exact nature of the data being transmitted is not being specified.

  The fifth message type is the select data message. It is modeled as a two component tree. The first component, Data, is exactly like the previous messages. The second component, Selected_Msg_Type, conveys information concerning the content of the Data. Note that its representation is not specified, but the information content of the message is. The Selected_Msg_Type is further defined as an enumeration of the four commands that the EC can issue to the INS through a select data message; namely, to commence sending periodic attitude messages, to commence sending periodic navigation messages, to commence sending both periodic messages, or to stop sending periodic messages.
- The next several type equations deal with INS and EC non-communications events, specifically interactions with each computer's console operator. The INS_Non_Comms_Events are simply signals to the INS operator—no signals from the INS operator are being modeled. They are not specified in any further detail, since they are by-products of communications but have no direct impact upon communications. The EC_Non_Comms_Events are either signals from or

to the EC operator. Signal_from_EC_Operator is further defined as an enumeration of the commands that may be issued by the EC operator and impact communications between the two computers.

- The final type equation simply states that Times_Stamps are being modeled as natural numbers. Time is expressed in multiples of .01 milliseconds, which is the accuracy to which [14] specifies time.

4. Formal Model Top Level Functions

4.1. Time Stamps Are Non-Decreasing in Event Sequences

```
Time_Stamps_Non_Decreasing(event_list) ≡

    (∀ i,j ∈ ind event_list)
       ((i < j) ⊃
           (s-Time_Stamp(event_list[i]) ≤ s-Time_Stamp(event_list[j])))

type: Event_List → BOOL
```

Rationale:

- Since the ordering of the elements in a tuple of events captures the order in which these events occur, the time stamps of the events must be non-decreasing in the sequence. The reason for not requiring strictly increasing time stamps is that the model allows consecutive events to happen "at the same time" (within the accuracy of the time measurement units).

4.2. Determine If the Protocol Is Obeyed

```
Protocol_Obeyed(events_at_ins, events_at_ec) ≡

    Protocol_Obeyed_at_INS(<>)( events_at_ins )
    ∧
    Protocol_Obeyed_at_EC (<>)( events_at_ec  )

type: Event_List Event_List → BOOL
```

Rationale:

- We model communications between the INS and EC as a sequence of events. This sequence is viewed from two perspectives: 1) at the INS communications ports, and 2) at the EC communications ports.
- Under "normal" situations the two views should yield identical sequences (excepting non-communications events). However, if transmission errors occur, then the differing perspectives may yield different sequences.

4.3. Determine If the Event Sequences Are Complete

```
Is_Complete(events_at_ins, events_at_ec) =

    ¬ ( ∃ event ∈ {x | is-Event(x) ∧ ¬ is-Signal_from_EC_Operator(x) } )
          ( Protocol_Obeyed( events_at_ins ^ <event>,
                             events_at_ec  ^ <event> ) )


type: Event_List Event_List → BOOL

pre: Protocol_Obeyed(events_at_ins, events_at_ec)
```

Rationale:

- The Protocol_Obeyed function determines if all events in the event history are correct in the context of each events history. All events that have transpired may indeed be correct, but the communications protocol may dictate that a future event must occur.
- The idea captured by the above predicate is that, if an event could happen, it should happen (e.g., if ACK_{INS} can occur but doesn't, then we define the event sequence as incomplete). Excepting signals from the EC operator, it is incorrect for no event to occur when there is an event that can occur and obey the protocol. The only type of event that can occur at any time is a signal from the EC operator.
- This function is only applied to the event history if the history obeys the protocol, and thus Protocol_Obeyed is a precondition to this function.

5. Formal Model INS Functions

5.1. Determine If Protocol Is Obeyed at the INS

```
Protocol_Obeyed_at_INS(history)(events_at_ins) =

    (events_at_ins ≠ <>) ⊃
       (let first_event = hd events_at_ins in
        (is-INS_Event(first_event) ⊃
            Event_Is_Correct_at_INS(Filter_Event_List_at_INS(history))
                                    (first_event))
        ∧
        Protocol_Obeyed_at_INS(history ^ <first_event>)(tl events_at_ins))

type: Event_List → (Event_List → BOOL)
```

Rationale:

- This function determines if the protocol is obeyed by the INS as seen at the INS; i.e., ensures that the INS generates **correct** events. Note that from the INS perspective, its own events are correct or incorrect and EC events are expected or unexpected.
- This function examines each event in a sequence and determines if it is correct in the context of a **filtered** history.
- Filter_Event_List_at_INS removes unexpected EC events (caused by transmission errors), thus simplifying the event stream that needs to be examined for completeness and correctness. In essence this process removes EC events that the INS may ignore (see Sections 2.4.2.2 and 5.3).

5.2. Look at Each INS Event in Context of History

```
Event_Is_Correct_at_INS(f_history)(ins_event) =

   let mk-INS_Event(event_info,) = ins_event in
   (is-Comms_Event(event_info) -->
         Comms_Event_Is_Correct_at_INS(f_history)(ins_event),
    T                          -->
         Non_Comms_Event_Is_Correct_at_INS(f_history))

type: Event_List → (INS_Event → BOOL)
```

Rationale:

- This function separates the treatment of communications events and non-communications events.
- Notice that the name of the history parameter is **f_history**. This signifies that this function is operating on a previously filtered event list.

5.2.1. Look at Each INS Communications Event in Context of History

```
Comms_Event_Is_Correct_at_INS(f_history)(ins_event) =

   let mk-INS_Event(event_info,) = ins_event in
   ( is-EF_Event(event_info)      -->
        EF_Event_Is_Correct_at_INS(f_history)(ins_event),

     is-Data_Message(event_info) -->
        Data_Event_Is_Correct_at_INS(f_history)(ins_event) )

type: Event_List → (INS_Event ⥲ BOOL)
```

Rationale:

- This function separates the treatment of EF events and data events.
- Also, notice that this is a partial function, since it prescribes no result for non-communications INS_Events.

5.2.1.1. Look at Each INS Communications EF Event in Context of History

```
EF_Event_Is_Correct_at_INS(f_history)(ins_event) =

    let mk-INS_Event(event_info,event_time) = ins_event in
    cases event_info:
        ( ACK        -->
              ACK_Is_Correct_at_INS   (f_history),
          ATTN1      -->
              ATTN1_Is_Correct_at_INS(f_history)(event_time),
          ATTN2      -->
              ATTN2_Is_Correct_at_INS(f_history),
          ATTN4      -->
              ATTN4_Is_Correct_at_INS,
          EOM        -->
              EOM_Is_Correct_at_INS   (f_history)(event_time),
          NAK        -->
              NAK_Is_Correct_at_INS   (f_history),
          NRTR       -->
              NRTR_Is_Correct_at_INS  (f_history)(event_time),
          RTR        -->
              RTR_Is_Correct_at_INS   (f_history)(event_time),
          SOM        -->
              SOM_Is_Correct_at_INS   (f_history)(event_time),
          SOTM       -->
              SOTM_Is_Correct_at_INS  (f_history)(event_time),
          Error_EF   -->
              false
        )

type: Event_List → (INS_Event ⥲ BOOL)
```

Rationale:

- This function divides EF events into the individual EFs and invokes functions to express the conditions for the individual EFs.
- Also, notice that this function is partial, since it does not prescribe a result for non-communications events or for Data_Message events.

5.2.1.2. Look at Each INS Communications Data Event in Context of History

```
Data_Event_Is_Correct_at_INS(f_history)(ins_event) =

   let mk-INS_Event(event_info, ) = ins_event in
   ( is-Test_Msg(event_info)              -->
        Test_Msg_Is_Correct_at_INS(f_history)(event_info),

     is-Time_and_Status_Msg(event_info) -->
        Time_and_Status_Msg_Is_Correct_at_INS(f_history),

     is-Attitude_Msg(event_info)          -->
        Attitude_Msg_Is_Correct_at_INS(f_history)(event_info),

     is-Navigation_Msg(event_info)        -->
        Navigation_Msg_Is_Correct_at_INS(f_history)(event_info),

     T                                    --> false )

type: Event_List → (INS_Event → BOOL)
```

Rationale:

- This function divides the data events into the four data messages that are sent from the INS to the EC (see [14], p. 8-19, Table 8-2).

5.2.2. Look at Each INS Non-Communications Event in Context of History

```
Non_Comms_Event_Is_Correct_at_INS(f_history) =

   Signal_to_Operator_Is_Correct_at_INS(f_history)

type: Event_List → BOOL
```

Rationale:

- The only INS non-communications event is a signal to the INS operator.

5.2.2.1. Look at Each INS Signal to Operator in the Context of History

```
Signal_to_INS_Operator_Is_Correct_at_INS(f_history) =

   Time_Out_After_EC_Initiated_SOM_at_INS(f_history)                -- 6.3.2.2.b
   v
   Time_Out_After_EC_Initiated_SOTM_at_INS(f_history)               -- 6.3.2.3.a
   v
   Error_After_Second_SOM_at_INS(f_history)                         -- 6.3.2.1
   v
   Error_After_Second_SOTM_at_INS(f_history)                        -- 6.3.2.3.c
   v
   Invalid_Message_Or_Test_Message_Received_at_INS(f_history)       -- 6.3.2.2.c
   v
   ATTN1_Received_at_INS(f_history)                                 -- 6.3.6.2.d

type: Event_List → BOOL
```

Rationale:

- Signals to the INS operator indicate error situations detected by the INS. Such errors can be divided into:
  - An INS time out after EC-initiated communication (after SOM or SOTM)
  - Failing a second attempt to communicate (after SOMs or SOTMs)
  - Receiving invalid messages or test messages
  - Receiving an ATTN1 from the EC during communication

5.3. Filter Event List at INS

```
Filter_Event_List_at_INS(events_at_ins) =

   Remove_Unexpected_EC_Events( <> )( events_at_ins )

type: Event_List → Event_List
```

Rationale:

- Out of sequence EFs are ignored ([14], 6.3.6.2.e).

5.3.1. Remove Unexpected EC Events from Event List

```
Remove_Unexpected_EC_Events( history )( events_at_ins ) =

   ( events_at_ins ≠ <> -->
        ( let first_event = hd events_at_ins in
        ( is-EC_Event(first_event) ∧
          is-Comms_Event( s-EC_Event_Info(first_event)) -->
             ( Comms_Event_Is_Correct_at_EC( history )( first_event ) -->
                  Remove_Unexpected_EC_Events(history ^ <first_event>)
                                           (tl events_at_ins)

               T                                                      -->
                  Remove_Unexpected_EC_Events(history )
                                          (tl events_at_ins) ),

          T                                             -->
             Remove_Unexpected_EC_Events(history ^ <first_event>)
                                      (tl events_at_ins)),

     T                   -->  history )
```

type: Event_List → (Event_List → Event_List)

Rationale:

- Out of sequence EFs are defined as those that do not satisfy "EC correctness."

5.4. Determine Correctness of Individual INS EF Events

5.4.1. ACK

```
ACK_Is_Correct_at_INS(f_history) =

   cases Last_n(2,f_history):
      ( < mk-EC_Event(data_msg, ),            --6.3.2.2.c
          mk-EC_Event(EOM, ) > -->
            Valid_Data_Message(data_msg),
        T                      -->
            false )
```

type: Event_List → BOOL

Rationale:

- For the INS to issue an ACK, it must have received a Data_Msg followed by an EOM and determined that the message is valid.
- Notice that there are no timing requirements between receiving an EOM_{EC} and issuing the ACK_{INS}.

- Also notice that the presence of an EOM_{EC} implies that the preceding event must have been a Data_Message from the EC.

5.4.2. ATTN1

```
ATTN1_Is_Correct_at_INS(f_history)(attn1_time) =

   f_history ≠ <>
   ∧
   is-INS_Event(Last(f_history))
   ∧
   ( let mk-INS_Event(last_event,last_event_time) = Last(f_history) in
     last_event  ∈ { SOM, SOTM, EOM, ATTN2 }
     ∧
     attn1_time - last_event_time > Time_Out_Period_at_INS )  -- 6.2.1.g-h

type: Event_List → (Time_Stamp → BOOL)
```

Rationale:

- The last event in the history must be an INS event.
- Moreover, the last event must be one the following events: SOM, SOTM, EOM, or ATTN2 ([14], 6.3.2.1.a, 6.3.2.1.c, 6.3.2.3.c).
- When awaiting a response from the EC, an ATTN1 may only be issued upon the expiry of the timeout period.

5.4.3. ATTN2

```
ATTN2_Is_Correct_at_INS(f_history) =

   f_history ≠ <>
   ∧
   cases Last(f_history):
      ( mk-EC_Event(ATTN2, )  --> true,   -- 6.2.1.b-c
        T                     --> false )

type: Event_List → BOOL
```

Rationale:

- EC is responsible for enabling communications ([14], 6.2.1).
- EC sends an ATTN2 to start enabling communications ([14], 6.2.1.b).
- INS sends an ATTN2 only in response to ECs ATTN2 ([14], 6.2.1.c).

5.4.4. ATTN4

```
ATTN4_Is_Correct_at_INS =

    false                          -- 6.2.2

type: → BOOL
```

Rationale:

- ATTN4 is only sent by the EC.

5.4.5. EOM

```
EOM_Is_Correct_at_INS(f_history)(eom_time) =

   f_history ≠ <>
   ^
   cases Last(f_history):
        ( mk-INS_Event(mk-Test_Msg( ), )            -->
               true, --6.3.2.1.c

          mk-INS_Event(mk-Time_and_Status_Msg( ), ) -->
               true,

          mk-INS_Event(mk-Attitude_Msg( ), )        -->
               Periodic_Attitude_Deadline_is_Satisfied(f_history)(eom_time),

          mk-INS_Event(mk-Navigation_Msg( ), )      -->
               Periodic_Navigation_Deadline_is_Satisfied(f_history)(eom_time),

          T                                         --> false )

type: Event_List → (Time_Stamp →  BOOL)
```

Rationale:

- An EOM can only follow the sending of a data message. The INS only sends the four data messages that are specified.
- If the message is an attitude or navigation message, the EOM must meet the periodic timing requirement (see Section 2.3.4).

5.4.5.1. Determine If the Attitude Message's Deadline has Been Met

```
Periodic_Attitude_Deadline_is_Satisfied(f_history)(eom_time) =

    let end_of_prev_interval ∈ N0 be s.t.
            end_of_prev_interval*Attitude_Period < eom_time
            ∧
            (end_of_prev_interval+1)*Attitude_Period ≥ eom_time
    in
    Select_Data_Request_for_Attitude_in_Interval(f_history)
                  (end_of_prev_interval-1, end_of_prev_interval)
    ∨
    EOM_for_Attitude_in_Interval(f_history)
                  (end_of_prev_interval-1, end_of_prev_interval)

type: Event_List → (Time_Stamp → BOOL)
```

Rationale:

- The model of periodicity is one that assumes no phase shift from the beginning of execution. The periodic intervals are therefore set at the beginning of execution.
- This function first determines the interval number of the interval containing the EOM in question and then determines if the previous interval contains either a select data message or an EOM.
- If the previous interval does not contain one of the two aforementioned events, then the EOM in question is in the wrong interval (i.e., should have been in a previous interval).
- Note that Attitude_Period is a constant function defined in Section 7.1.4.

5.4.5.2. Determine If Select Data Message for the Attitude Message Is in Designated Interval

```
Select_Data_Request_for_Attitude_in_Interval(f_history)(start, end) =

   f_history ≠ <>
   ∧
   cases Last(f_history):

      ( mk-EC_Event(mk-Select_Data_Msg( ,Attitude),sd_msg_time )) -->
           sd_msg_time > start*Attitude_Period
           ∧
           sd_msg_time ≤  end*Attitude_Period,

        mk-EC_Event(mk-Select_Data_Msg( ,Both),sd_msg_time ))      -->
           sd_msg_time > start*Attitude_Period
           ∧
           sd_msg_time ≤  end*Attitude_Period,

        T                                -->
           Select_Data_Request_for_Attitude_in_Interval
                                       ( Front(f_history) ) (start, end) )

type: Event_List → (N0 N0 → BOOL )
```

Rationale:

- This function recursively searches for a select data message that selects attitude messages (by explicitly requesting attitude messages or by selecting both periodic messages). It then determines if the time associated with the message is in the designated interval.

5.4.5.3. Determine If EOM Following an Attitude Message Is in the Designated Interval

```
EOM_for_Attitude_in_Interval(f_history)(start, end) =

   cases Last_n(2,f_history):

      ( <mk-INS_Event(mk-Attitude_Msg( ), ),
        mk-INS_Event(EOM,eom_time)> -->
          start*Attitude_Period  <  eom_time
          ^
          end*Attitude_Period  ≥  eom_time,

        <>                                 -->
          false,

        T                                  -->
          EOM_for_Attitude_in_Interval( Front(f_history) ) (start, end) )

type: Event_List → (N0 N0 → BOOL )
```

Rationale:

- This function recursively searches for an attitude data message followed by an EOM and then determines if the time associated with the EOM is in the designated interval.

5.4.5.4. Determine If the Navigation Message's Deadline Has Been Bet

```
Periodic_Navigation_Deadline_is_Satisfied(f_history)(eom_time) =

   let end_of_prev_interval ∈ N0 be s.t.
          end_of_prev_interval*Navigation_Period < eom_time
          ^
          (end_of_prev_interval+1)*Navigation_Period ≥ eom_time
   in
   Select_Data_Request_for_Navigation_in_Interval(f_history)
                (end_of_prev_interval-1, end_of_prev_interval)
   v
   EOM_for_Navigation_in_Interval(f_history)
                (end_of_prev_interval-1, end_of_prev_interval)

type: Event_List → (Time_Stamp →  BOOL)
```

Rationale:

- See rationale for attitude message meeting its deadline (5.4.5.1).

5.4.5.5. Determine If Select Data Message for the Navigation Message Is in Designated Interval

```
Select_Data_Request_for_Navigation_in_Interval(f_history)(start, end) =

   f_history ≠ <>
   ^
   cases Last(f_history):

      ( mk-INS_Event(mk-Select_Data_Msg( ,Navigation),sd_msg_time )) -->
           sd_msg_time > start*Navigation_Period
           ^
           sd_msg_time ≤  end*Navigation_Period,

        mk-INS_Event(mk-Select_Data_Msg( ,Both),sd_msg_time ))        -->
           sd_msg_time > start*Navigation_Period
           ^
           sd_msg_time ≤  end*Navigation_Period,

        T                              -->
           Select_Data_Request_for_Navigation_in_Interval
                                              ( Front(f_history) ) (start, end) )

type: Event_List → (NO NO → BOOL )
```

Rationale:

- This function recursively searches for a Select Data message that selects Navigation messages (by explicitly requesting Navigation messages or by selecting both periodic messages). It then determines if the time associated with the message is in the designated interval.

5.4.5.6. Determine If EOM Following Navigation Message Is in Designated Interval

```
EOM_for_Navigation_in_Interval(f_history)(start, end) =

   f_history ≠ <>
   ∧
   cases Last_n(2,f_history):

      ( <mk-INS_Event(mk-Navigation_Msg( ), ),
        mk-INS_Event(EOM,eom_time)> -->
          eom_time > start*Navigation_Period
          ∧
          eom_time ≤  end*Navigation_Period,

        T                                 -->
          EOM_for_Navigation_in_Interval( Front(f_history) ) (start, end) )

type: Event_List → (N0 N0 → BOOL )
```

Rationale:

- This function recursively searches for a navigation data message followed by an EOM and then determines if the time associated with the EOM is in the designated interval.

5.4.6. NAK

```
NAK_Is_Correct_at_INS(f_history) =

   cases Last_n(2,f_history):             --6.3.2.2.c
     ( < mk-EC_Event(data_msg, ), mk-EC_Event(EOM, ) > -->
          ¬ Valid_Data_Message(data_msg),
       T                                               -->
          false )


type: Event_List → BOOL)
```

Rationale:

- For the INS to issue a NAK, it must have received a Data_Msg followed by an EOM and determined that the message is invalid.
- Notice that there are no timing requirements between receiving an EOM_{EC} and issuing the NAK_{INS}.
- Also notice that the presence of an EOM_{EC} implies that the preceding event must have been a Data_Message from the EC.

5.4.7. NRTR

```
NRTR_Is_Correct_at_INS =

   false

type: → BOOL
```

Rationale:

- P. 4-9 of [14] states that NRTR is not transmitted by the INS.

5.4.8. RTR

```
RTR_Is_Correct_at_INS(f_history)(rtr_time) =

   f_history ≠ <>
   ∧
   cases Last(f_history):                          --6.3.2.2.a-b
      ( mk-EC_Event(SOM, som_time)     -->
           rtr_time - som_time ≤ EC_Time_Out_Period,

        mk-EC_Event(SOTM, sotm_time) -->
           rtr_time - sotm_time ≤ EC_Time_Out_Period,

        T                              -->
           false )

type: Event_List → (Time_Stamp →  BOOL)
```

Rationale:

- SOM_{EC} or SOTM_{EC} must immediately precede an RTR.
- Moreover, the RTR must be issued within the EC allowed time-out period.

5.4.9. SOM

```
SOM_Is_Correct_at_INS(f_history)(som_time) =

   ( Periodic_Messages_Are_Activated(f_history) ∧
     SOM_Is_Correct_When_Periodic_Msgs_Are_Activated_at_INS(f_history) )

   ∨                                          --6.2.1.f
   SOM_Is_Correct_When_Periodic_Msgs_Are_Not_Activated_at_INS(f_history)


type: Event_List → (Time_Stamp →  BOOL)
```

Rationale:

- An SOM$_{INS}$ is correct if periodic messages from the INS to the EC are enabled, or
- An SOM$_{INS}$ is also correct after the enabling sequence at the beginning of the message exchange for sending a time and status message.

5.4.9.1. Determine If Periodic Messages Are Activated

```
Periodic_Messages_Are_Activated(f_history) =

   f_history ≠ <>
   ∧
   if is-EC_Event(Last(f_history)) then
      let mk-EC_Event(last_event,) = Last(f_history) in
      ( last_event = ATTN2 --> false,                --6.2.1.c
        last_event = ATTN4 --> false,                --6.2.1.a
        last_event = mk-Select_Data_Msg( ,None )    --p.8-10
                                --> false,
        last_event = mk-Select_Data_Msg( ,Attitude )
                                --> true,
        last_event = mk-Select_Data_Msg( ,Navigation )
                                --> true,
        last_event = mk-Select_Data_Msg( ,Both )
                                --> true,
        T                       -->
           Periodic_Messages_Are_Activated( Front(f_history) )
   else
      Periodic_Messages_Are_Activated( Front(f_history) )

type: Event_List → BOOL
```

Rationale:

- Periodic messages are activated only if a select data message (for attitude data, navigation data, or both) exists in the history and an ATTN2 or ATTN4 or select data for no data does not occur later.

5.4.9.2. Given that Periodic Messages are Activated, Is SOM Correct?

```
SOM_Is_Correct_When_Periodic_Msgs_Are_Activated_at_INS(f_history)(som_time) =

   cases Number_Of_Outstanding_SOMs_Sent_at_INS(f_history):
      (0 -->
          cases Last_n(2,f_history):                      --6.3.2.2.a
             ( < mk-EC_Event( , ),
                 mk-INS_Event(ACK, ) >                  -->
                    true,

               < mk-INS_Event( , ),
                 mk-INS_Event(event_info, ) > -->
                    is-Signal_to_INS_Operator(event_info),

               < mk-INS_Event( , ),
                 mk-EC_Event(ACK, ) >                   -->
                    true,

               < mk-EC_Event( , ),
                 mk-EC_Event(event_info, ) >  -->
                    is-Signal_to_EC_Operator(event_info),

               T                                        -->
                    false ),

       1 -->
          cases Last(f_history):
              ( mk-INS_Event(ef, )                      -->
                   ef = ATTN1,

               mk-EC_Event(ef, ef_time)                 -->
                   ef = ATTN1
                   ∨
                   ef = NAK
                   ∨
                   ( ef = NRTR ∧
                     som_time - ef_time > Sleep_Period )),

       T --> false )

type: Event_List → (Time_Stamp →  BOOL)
```

Rationale:

- If there are no outstanding SOMs (i.e. no incomplete message transfer sequences from the INS to the EC), then the SOM must follow one of the indicated EF events or a signal to the operator.
- If there is one outstanding SOM, then the SOM in question must follow an ATTN1, NAK, or an NRTR.
- If it follows an NRTR, there must be at least a Sleep_Period separation in time.

5.4.9.3. Given that Communications Have Just Been Enabled, Is SOM Correct?

```
SOM_Is_Correct_When_Periodic_Msgs_Are_Not_Activated_at_INS(f_history) =
                                                   --6.2.1.e & 6.3.2.1.d
   cases Number_Of_Outstanding_SOMs_Sent_at_INS(f_history):
      (0 -->
         Ends_in_Enabling_Sequence(f_history),         --6.3.2.2.a

       1 -->
         let i ∈ ind f_history be s.t.
            Number_of_Outstanding_SOMs_Sent_at_INS
               ( < f_history[j] | 1 ≤ j < i > ) = 0
            ∧
            (∀ k ∈ ind f_history)
               (( k > i)  ⊃

                  ( cases f_history[k]:
                        ( mk-INS_Event(SOM, ) --> false,
                          T                   --> true ))
         in
            Ends_in_Enabling_Sequence( < f_history[j] | 1 ≤ j < i > )

            ∧
            cases Last(f_history):
               ( mk-INS_Event(ef, ) -->
                     ef = ATTN1,

                 mk-EC_Event(ef,ef_time)  -->
                     ef = ATTN1
                     ∨
                     ef = NAK
                     ∨
                     ( ef = NRTR ∧
                       som_time - ef_time > Sleep_Period ),

       T --> false )

type: Event_List → BOOL
```

Rationale:

- If the SOM is the beginning of a message transfer of the time and status message and this is the time and status message that is due immediately after communications have been enabled, then the SOM must be preceded immediately by the enabling sequence.
- The SOM could also be the start of a retry for this message transfer.

5.4.10. SOTM

```
SOTM_Is_Correct_at_INS(f_history)(sotm_time) =

   cases Number_of_Outstanding_SOTMs_Sent_at_INS(f_history):  --6.3.2.2.a-c
      (0 -->
          cases Last_n(3, f_history):
              ( < mk-EC_Event(mk-Test_Msg(), ),
                  mk-EC_Event(EOM, ),
                  mk-INS_Event(ACK, ) >   --> true,

                < mk-EC_Event(mk-Test_Msg(), ),
                  mk-EC_Event(EOM, ),
                  mk-INS_Event(NAK, ) >   --> true,

                T                         --> false )


       1 -->
          cases Last(f_history):
              ( mk-INS_Event(ef, )        -->
                   ef = ATTN1,                   --6.3.2.1.b & 6.3.6.2.a

                mk-EC_Event(ef,ef_time ) -->
                   ef = ATTN1                    --6.3.6.1
                   ∨
                   ef = NAK
                   ∨
                   ( ef = NRTR ∧
                     sotm_time - ef_time > Sleep_Period ) ),
                                                 --6.3.2.1.b
       T --> false )

type: Event_List → (Time_Stamp →  BOOL)
```

Rationale:

- If there are no outstanding SOTMs (i.e., no incomplete test message transfers from the INS to the EC), then an SOTM must follow one of the event triples.
- If there is one outstanding SOTM, then the SOTM in question must follow an ATTN1, NAK, or an NRTR.
- If it follows an NRTR, there must be at least a Sleep_Period separation in time.

5.5. Determine Correctness of Individual INS Data Message Events

5.5.1. Test Message

```
Test_Msg_Is_Correct_at_INS(f_history)(event_info) =

   cases Last_n(2,f_history)      --6.3.2.1.a & 6.3.2.3.c
        ( < mk-INS_Event(SOTM, ),
            mk-EC_Event(RTR, )  > --> true,

          T                        --> false )

   ∧
   let i ∈ ind f_history be s.t.    --6.3.2.3.c
       cases < f_history[j] | i ≤ j ≤ i+2 >:
            ( < mk-EC_Event(mk-Test_Msg( ), ),
                mk-EC_Event(EOM, ),
                mk-INS_Event(ef, ) > --> ef ∈ {ACK, NAK},
              T                      --> false )

       ∧
       ( ∀ j ∈ ind f_history )
            ( ( j > i+2 ) ⊃

                ( cases < f_history[m] | j-2 ≤ m ≤ j >:
                        ( < mk-EC_Event(mk-Test_Msg( ), ),
                            mk-EC_Event(EOM, ),
                            mk-INS_Event(ef, ) > --> ef ∉ {ACK, NAK},
                          T                      --> true ) ) )

   in
       event_info = s-EC_Event_Info( f_history[i] )

type: Event_List → (INS_Event_Info → BOOL)
```

Rationale:

- SOTM and RTR must immediately precede a Test_Msg.
- Also, the test message data must be the same data received in the initiating test message from the EC.

5.5.2. Time and Status Message

```
Time_and_Status_Msg_Is_Correct_at_INS(f_history) =

   TSM_Is_Correct_After_Enabling_Sequence(f_history)  --6.2.1.f & p.8-32(a)
   ∨
   TSM_Is_Correct_After_Select_Data_Msg(f_history)    --p.8-32(b)

type: Event_List → BOOL
```

Rationale:

- The time and status message can be sent on two occasions: 1) immediately after the enabling sequence, and 2) immediately after the INS receives a select data message from the EC.
- Note that [14], p. 8-32c, specifies an additional condition for sending a time and status message. This condition is not being modeled.

5.5.2.1. Determine If Time and Status Message Is Correct After Enabling Sequence

```
TSM_Is_Correct_After_Enabling_Sequence(f_history) =

      cases Last_n(2,f_history):                          --6.3.2.1.a
         ( < mk-INS_Event(SOM, ),
             mk-EC_Event(RTR, )  >  -->
                SOM_Is_Correct_When_Periodic_Msgs_Are_Not_Activated_at_INS
                   (Front(Front(f_history))),

           T                            --> false )

type: Event_List → BOOL
```

Rationale:

- The message must be preceded by an SOM and an RTR.
- In addition, the enabling sequence must precede those initial EFs. This is ensured by stripping off the SOM and the RTR and applying the function that ensures that an SOM is correct when periodic messages are not activated. The only place that an SOM_{INS} can appear when periodic messages are not activated is immediately after an enabling sequence.

5.5.2.2. Determine if Time and Status Message is Correct after Select Data Message

```
TSM_Is_Correct_After_Select_Data_Msg(f_history) =

   cases Last_n(2,f_history):                   --6.3.2.1.a
       ( < mk-INS_Event(SOM, ),
           mk-EC_Event(RTR, )  >  --> true,
         T                        --> false )

   ^
   cases Number_Of_Outstanding_SOMs_Sent_at_INS(f_history):
      (1 -->
         let trunc_f_history = Front(Front(f_history)) in  --p.8-32(b)
         cases Last_n(3,trunc_f_history):
             ( < mk-EC_Event( mk-Select_Data_Msg( , Attitude) ),
                 mk-EC_Event(EOM, ),
                 mk-INS_Event(ACK, ) > --> true,

               < mk-EC_Event( mk-Select_Data_Msg( , Navigation) ),
                 mk-EC_Event(EOM, ),
                 mk-INS_Event(ACK, ) > --> true,

               < mk-EC_Event( mk-Select_Data_Msg( , Both) ),
                 mk-EC_Event(EOM, ),
                 mk-INS_Event(ACK, ) > --> true,

               T                       --> false ),

       2 -->
         TSM_Is_Correct_After_Select_Data_Msg(
           Front(Front(Front(f_history))) ^ Last_n(2,f_history) )

       T --> false )

type: Event_List → BOOL
```

Rationale:

- The message must be preceded by an SOM and an RTR.
- A select data message must precede the SOM that initiated the message transfer. Moreover, the select data message indeed must opt for data to be sent.

5.5.3. Attitude Message

```
Attitude_Msg_Is_Correct_at_INS(f_history)(event_info) =

   cases Last_n(2,f_history):
       ( < mk-INS_Event(SOM, ),
           mk-EC_Event(RTR, )  > --> Valid_Data_Message(event_info),

         T                       --> false )
```

```
type: Event_List → (INS_Event_Info → BOOL)
```

Rationale:

- Given the correctness of SOM at the EC (which has already been established, see Section 2.4.4.), the event sequences above are the only ones that may immediately precede an Attitude_Msg, or any non-test message for that matter.
- Correctness of the appearance of a periodic message is actually expressed when EOM is checked. At that point periodicity and timeliness are checked.

5.5.4. Navigation Message

```
Navigation_Msg_Is_Correct_at_INS(f_history)(event_info) =

   cases Last_n(2,f_history):
       ( < mk-INS_Event(SOM, ),
           mk-EC_Event(RTR, )  > --> Valid_Data_Msg(event_info),

         T                       --> false )
```

```
type: Event_List → (INS_Event_Info → BOOL)
```

Rationale:

- Given the correctness of SOM at the EC (which has already been established, see Section 2.4.4), the event sequences above are the only ones that may immediately precede a Navigation_Msg, or any non-test message for that matter.
- Correctness of the appearance of a periodic message is actually expressed when EOM is checked. At that point periodicity and timeliness are checked.

5.6. Determine the Correctness of Each INS Signal to Operator

5.6.1. Time-Out After EC Initiated SOM at INS

```
Time_Out_After_EC_Initiated_SOM_at_INS(f_history) =

   cases Last_n(3,f_history):
      (<mk-EC_Event(SOM, ),
        mk-INS_Event(RTR, ),
        mk-INS_Event(ATTN1, )> --> true,
       T                        --> false)
```

```
type: Event-_List → BOOL
```

Rationale:

- The INS will time out the EC (causing an ATTN1 to be sent) and alert the operator if the EC does not send its message within 10.24 ms after the INS is ready ([14], 6.3.2.2.b). The time-out period is not used by this function but is used when expressing the correctness of the ATTN1 being sent before the signal to the operator.
- [14] 6.3.2.1.a is read to imply that ATTN1 is issued before the operator is alerted.

5.6.2. Time-Out After EC Initiated SOTM at INS

```
Time_Out_After_EC_Initiated_SOTM_at_INS(f_history) =

   cases Last_n(3,f_history):
      (<mk-EC_Event(SOTM, ),
        mk-INS_Event(RTR, ),
        mk-INS_Event(ATTN1, )> --> true,
       T                        --> false)
```

```
type: Event_List → BOOL
```

Rationale:

- The INS will time out the EC (causing an ATTN1 to be sent) and alert the operator if the EC does not send its test message within 10.24 ms after the INS is ready ([14], 6.3.2.2.b, 6.3.2.3.a). The actual time-out period is used when expressing the correctness of the ATTN1 being sent before the signal to the operator.
- [14] 6.3.2.1.a is read to imply that ATTN1 is issued before the operator is alerted.

5.6.3. Error After Second SOM at INS

```
Error_After_Second_SOM_at_INS(f_history) =

   Time_Out_After_Second_SOM_at_INS(f_history)                -- 6.3.2.1.a(2)
   ∨
   NRTR_Received_After_Second_SOM_at_INS(f_history)           -- 6.3.2.1.b(2)
   ∨
   NAK_Received_After_Second_SOM_at_INS(f_history)            -- 6.3.2.1.e

type: Event_List → BOOL
```

Rationale:

- Failing a second attempt to communicate after a SOM is due to any of the following:
    - An INS time out occurs
    - The EC is not ready (NRTR)
    - The message is not accepted by the EC (NAK)

5.6.3.1. Time-out After Second SOM at INS

```
Time_Out_After_Second_SOM_at_INS(f_history) =

   Number_Of_Outstanding_SOMs_Sent_at_INS(f_history) = 2
   ∧
   cases Last(f_history):
      (mk-INS_Event(ATTN1, ) --> true,
       T                     --> false)

type: Event_List → BOOL
```

Rationale:

- An INS time out of the EC always results in an ATTN1 being sent to the EC ([14], 6.3.2.1.a(2), 6.3.6.2.a, 6.3.2.1.c.). This covers time outs directly following the second SOM as well as those following the EOM.
- [14] 6.3.2.1.a is read to imply that ATTN1 is issued before the operator is alerted.

5.6.3.2. Received an NRTR After Second SOM at INS

```
NRTR_Received_After_Second_SOM_at_INS(f_history) =

   Number_Of_Outstanding_SOMs_Sent_at_INS(f_history) = 2
   ^
   cases Last(f_history):
      (mk-EC_Event(NRTR, ) --> true,
       T                   --> false)

type: Event_List → BOOL
```

Rationale:

- If, by the second attempt, the EC is not ready to receive a message, the INS operator is alerted ([14], 6.3.2.1.b(2)).

5.6.3.3. Received a NAK After Second SOM at INS

```
NAK_Received_After_Second_SOM_at_INS(f_history) =

   Number_Of_Outstanding_SOMs_Sent_at_INS(f_history) = 2
   ^
   cases Last(f_history):
      (mk-EC_Event(NAK, ) --> true,
       T                  --> false)

type: Event_List → BOOL
```

Rationale:

- If, by the second attempt, the EC does not acknowledge the receipt of a message, the INS operator is alerted ([14], 6.3.2.1.e).

5.6.4. Error After Second SOTM at INS

```
Error_After_Second_SOTM_at_INS(f_history) =

   Time_Out_After_Second_SOTM_at_INS(f_history)
   v
   NRTR_Received_After_Second_SOTM_at_INS(f_history)
   v
   NAK_Received_After_Second_SOTM_at_INS(f_history)

type: Event_List → BOOL
```

Rationale:

- Failing a second attempt to communicate after a SOTM is due to any of the following:
  - An INS time out occurs
  - The EC is not ready (NRTR)
  - The test message is not accepted by the EC (NAK)

5.6.4.1. Time-out After Second SOTM at INS

```
Time_Out_After_Second_SOTM_at_INS(f_history) =

    Number_Of_Outstanding_SOTMs_Sent_at_INS(f_history) = 2
    ^
    cases Last(f_history):
        (mk-INS_Event(ATTN1, ) --> true,
         T                     --> false)

type: Event_List → BOOL
```

Rationale:

- An INS time out of the EC always results in an ATTN1 being sent to the EC ([14], 6.3.2.3.c, 6.3.2.1.a(2), 6.3.6.2.a, 6.3.2.1.c.). This covers time outs directly following the second SOTM as well as those following the EOM.
- [14] 6.3.2.1.a is read to imply that ATTN1 is issued before the operator is alerted.

5.6.4.2. Received an NRTR After Second SOTM at INS

```
NRTR_Received_After_Second_SOTM_at_INS(f_history) =

    Number_Of_Outstanding_SOTMs_Sent_at_INS(f_history) = 2
    ^
    cases Last(f_history):
        (mk-EC_Event(NRTR, ) --> true,
         T                   --> false)

type: Event_List → BOOL
```

Rationale:

- If, by the second attempt, the EC is not ready to receive a test message, the INS operator is alerted ([14], 6.3.2.3.c, 6.3.2.1.b(2)).

5.6.4.3. Received a NAK After Second SOTM at INS

```
NAK_Received_After_Second_SOTM_at_INS(f_history) =

   Number_Of_Outstanding_SOTMs_Sent_at_INS(f_history) = 2
   ^
   cases Last(f_history):
      (mk-EC_Event(NAK, )  --> true,
       T                   --> false)

type: Event_List → BOOL
```

Rationale:

- If, by the second attempt, the EC does not acknowledge the receipt of a test message, the INS operator is alerted ([14], 6.3.2.3.c, 6.3.2.1.e).

5.6.5. An Invalid Message or Test Message Was Received at INS

```
Invalid_Message_Or_Test_Message_Received_at_INS(f_history) =

   f_history ≠ <>
   ^
   cases Last(f_history):
      (mk-INS_Event(NAK, )  --> true,
       T                    --> false)

type: Event_List → BOOL
```

Rationale:

- Whenever an invalid message or test message is received by the INS, a NAK is sent to the EC and the operator is alerted ([14], 6.3.2.2.c, 6.3.2.3.b). NAK is only issued by the INS in the case of invalid messages.

5.6.6. Received an ATTN1 at INS

```
ATTN1_Received_at_INS(f_history) =

   f_history ≠<>
   ^
   cases Last(f_history):
      (mk-EC_Event(ATTN1, )  --> true,
       T                     --> false)

type: Event_List → BOOL
```

Rationale:

- When the INS receives an ATTN1, it alerts the INS operator ([14], 6.3.6.2.d).

6. Formal Model EC Functions

6.1. Determine If Protocol Is Obeyed at the EC

```
Protocol_Obeyed_at_EC(history)(events_at_ec) =

   (events_at_ec ≠ <>) ⊃
        (let first_event = hd events_at_ec in
         (is-EC_Event(first_event) ⊃
               Event_Is_Correct_at_EC(history)(first_event))
         ∧
         Protocol_Obeyed_at_EC(history ^ <first_event>)(tl events_at_ec)))
```

```
type: Event_List → (Event_List → BOOL)
```

Rationale:

- This function determines if the protocol is obeyed by the EC, i.e., ensures that the EC generates **correct** events. Note that from the EC perspective, its own events are correct or incorrect and INS events are expected or unexpected.
- This function examines each event in a sequence and determines if it is correct in the context of a history.
- Note that event filtering does not take place within this function as it does in the equivalent INS function.

6.2. Look at Each EC Event in Context of History

```
Event_Is_Correct_at_EC(history)(ec_event) =

   let mk-EC_Event(event_info,) = ec_event in
   (is-Comms_Event(event_info)        -->
         Comms_Event_Is_Correct_at_EC(history)(ec_event),

    T                                  -->
         let f_history = Filter_Event_List_at_EC(history) in
         Non_Comms_Event_Is_Correct_at_EC(f_history)(ec_event))
```

```
type: Event_List → (EC_Event → BOOL)
```

Rationale:

- This function separates the treatment of communications events and noncommunications events.
- Filter_Event_List_at_EC removes unexpected INS events (caused by transmission errors or incorrect INS behavior), thus simplifying the event stream that needs to be examined. In essence this process removes INS events that the EC may ignore. It is only done when the current event is an EC noncommunication event, since out-of-sequence INS EFs may cause the EC to respond with an ATTN1.

6.2.1. Look at Each EC Communications Event in Context of History

```
Comms_Event_Is_Correct_at_EC(history)(ec_event) =

    let mk-EC_Event(event_info,) = ec_event in
    (is-EF_Event(event_info)      -->
        EF_Event_Is_Correct_at_EC(history)(ec_event),

     is-Data_Message(event_info) -->
        let f_history = Filter_Event_List_at_EC(history) in
        Data_Event_Is_Correct_at_EC(f_history)(ec_event))

type: Event_List → (EC_Event ⥲ BOOL)
```

Rationale:

- This function separates the treatment of EF events and data events.
- Filter_Event_List_at_EC removes unexpected INS events (caused by transmission errors or incorrect INS behavior), thus simplifying the event stream that needs to be examined. In essence this process removes INS events that the EC may ignore. It is only done for data messages, since out-of-sequence INS EFs **may** cause the EC to respond with an ATTN1.

6.2.1.1. Look at Each EC Communications EF Event in Context of History

```
EF_Event_Is_Correct_at_EC(history)(ec_event) =

   let mk-EC_Event(event_info,event_time) = ec_event in
   if event_info = ATTN1 then
      ATTN1_Is_Correct_at_EC(history)(event_time)
   else
      let f_history = Filter_Event_List_at_EC(history) in
      cases event_info:
         (ACK      -->
             ACK_Is_Correct_at_EC(f_history)(event_time),
          ATTN2    -->
             ATTN2_Is_Correct_at_EC(f_history)(event_time),
          ATTN4    -->
             ATTN4_Is_Correct_at_EC(f_history),
          EOM      -->
             EOM_Is_Correct_at_EC(f_history)(event_time),
          NAK      -->
             NAK_Is_Correct_at_EC(f_history)(event_time),
          NRTR     -->
             NRTR_Is_Correct_at_EC(f_history)(event_time),
          RTR      -->
             RTR_Is_Correct_at_EC(f_history)(event_time),
          SOM      -->
             SOM_Is_Correct_at_EC(f_history)(event_time),
          SOTM     -->
             SOTM_Is_Correct_at_EC(f_history)(event_time),
          Error_EF -->
             false)

type: Event_List → (INS_Event ⥲ BOOL)
```

Rationale:

- This function divides EF events into the individual EFs and invokes functions to express the conditions for the individual EFs.
- Filter_Event_List_at_EC removes unexpected INS events (caused by transmission errors or incorrect INS behavior), thus simplifying the event stream that needs to be examined. In essence this process removes INS events that the EC may ignore. It is done for all EC EFs except ATTN1, since out-of-sequence INS EFs may cause the EC to respond with an ATTN1.

6.2.1.2. Look at Each EC Data Event in Context of History

```
Data_Event_Is_Correct_at_EC(f_history)(ec_event) =

    let mk-EC_Event(event_info, ) = ec_event in
    (is-Test_Msg(event_info)              -->
        Test_Msg_Is_Correct_at_EC(f_history)(ec_event),

     is-Select_Data_Msg(event_info)     -->
        Select_Data_Msg_Is_Correct_at_EC(f_history)(ec_event),

     T                                  -->
        false)

type: Event_List → (EC_Event → BOOL)
```

Rationale:

- This function separates the treatment of test messages, select data messages, and other messages (the latter are never correct data events from the EC).

6.2.2. Look at Each EC Non-Communications Event in Context of History

```
Non_Comms_Event_Is_Correct_at_EC(f_history)(ec_event) =

    let mk-EC_Event(event_info, ) = ec_event in
    (is-Signal_from_EC_Operator(event_info) -->
        Signal_from_EC_Operator_Is_Correct_at_EC,
     T                                      -->
        Signal_to_EC_Operator_Is_Correct_at_EC(f_history))

type: Event_List → (EC_Event ⥲ BOOL)
```

Rationale:

- This function separates the treatment of signals *from* and *to* the EC operator.

6.2.2.1. Look at Each EC Signal from the Operator in Context of History

```
Signal_from_EC_Operator_Is_Correct_at_EC =

    true

type: → BOOL
```

Rationale:

- The behavior of the EC operator is part of the environment of the system and outside its control.

6.2.2.2. Look at Each EC Signal to the Operator in Context of History

```
Signal_to_EC_Operator_Is_Correct_at_EC(f_history) =

   ...

type: Event_List → BOOL
```

Rationale:

- The EC may send signals to its operator. The specification of when such signals occur is not part of the communication protocol.

6.3. Filter Event List at EC

```
Filter_Event_List_at_EC(events_at_ec) =

   Remove_Unexpected_INS_Events(<>)(events_at_ec)   -- 6.3.6.1.c

type: Event_List → Event_List
```

Rationale:

- Out-of-sequence INS EFs can be ignored, except when considering the legality of an EC ATTN1.

6.3.1. Remove Unexpected INS Events at EC

```
Remove_Unexpected_EC_Events(history)(events_at_ec) =

   (events_at_ec ≠ <> -->
        (let first_event = hd events_at_ec in
         (is-INS_Event(first_event) ∧
          is-Comms_Event(s-INS_Event_Info(first_event))       -->
              (Comms_Event_Is_Correct_at_INS(history)(first_event) -->
                   Remove_Unexpected_INS_Events(history ^ <first_event>)
                                                    (tl events_at_EC)
               T                                                   -->
                   Remove_Unexpected_INS_Events(history)(tl events_at_ec)),

          T                                               -->
               Remove_Unexpected_EC_Events(history ^ <first_event>)),
                                         (tl events_at_ec)

    T                 --> history)

type: Event_List → (Event_List → Event_List)
```

Rationale:

- Out-of-sequence INS EFs are defined as those not satisfying the "INS correctness."

6.4. Determine Correctness of Individual EC EF Events

6.4.1. ACK

```
ACK_Is_Correct_at_EC(f_history)(ack_time) =

   cases Last_n(2,f_history):
      (<mk-INS_Event(data_msg, ),
        mk-INS_Event(EOM,eom_time)> -->
          Valid_Data_Message(data_msg)                 -- 6.3.2.1.d
          ∧
          ack_time - eom_time ≤ Time_Out_Period_at_INS,   -- 6.2.3.2.a
       T                                 -->
          false)

type: Event_List → (Time_Stamp → BOOL)
```

Rationale:

- The EC acknowledges the receipt of a message (responds with an ACK) after the corresponding EOM if the message is valid.
- The ACK response must occur before the INS time-out period associated with the EOM.

6.4.2. ATTN1

```
ATTN1_Is_Correct_at_EC(history)(attn1_time) =

   history ≠ <>
   ∧
   (is-INS_Event(Last(history)) -->                        -- 6.3.6.1.c
      INS_EF_Is_Out_Of_Sequence_at_EC(Front(history))(Last(history)),
    T                         -->                          -- 6.3.6.1.a
      let mk-EC_Event(last_event,last_event_time) = Last(history) in
      last_event ∈ {RTR, SOM, SOTM}
      ∧
      attn1_time - last_event_time > Time_Out_Period_at_EC)

type: Event_List → (Time_Stamp → BOOL)
```

Rationale:

- An ATTN1 from the EC is either the response to an INS EF that is received out of sequence, or to the EC timing out the INS in cases where a timely response is required (after RTR, SOM or SOTM, but not after EOM as at the INS; [14] 6.3.2.1.b, 6.3.2.2.a, 6.3.2.3.a).
- Note that the INS_EF_Is_Out_Of_Sequence_at_EC function is true for histories that include periodic messages that do not satisfy the required periodicity.

6.4.2.1. Detect out of Sequence INS EF Events at EC

```
INS_EF_Is_Out_Of_Sequence_at_EC(history)(ins_event) =

   let f_history = Filter_Event_List_at_EC(history) in
   let mk-INS_Event(last_event, ) = ins_event in
   is-EF_Event(last_event)
   ∧
   ¬ EF_Event_Is_Correct_at_INS(f_history)(ins_event)

type: Event_List → (INS_Event → BOOL)
```

Rationale:

- For an INS event to be an out-of-sequence EF (at the EC), it must (obviously) be an EF event, and secondly it must be **incorrect** when seen as an event at the INS, given the history as seen by the EC [14] (6.3.6.1.c).

6.4.3. ATTN2

```
ATTN2_Is_Correct_at_EC(f_history)(attn2_time) =

   Enabling_Requested_at_EC(f_history)
   ∧
   Initiating_EF_Not_Too_Soon_at_EC(f_history)(attn2_time)     -- 6.2.3.2.b

type: Event_List → (Time_Stamp → BOOL)
```

Rationale:

- ATTN2 is issued by the EC as the first EF in an enabling sequence ([14], 6.2.1.b and p. 5-2). Hence, enabling must have been requested by the EC operator.
- ATTN2 falls into the category of initiating EFs (which includes SOM and SOTM as well), and no more than two of such EFs are allowed per second.

6.4.3.1. No More Than Two Initiating EFs per Second from the EC

```
Initiating_EF_Not_Too_Soon_at_EC(f_history)(current_time) =

    (∀ i,j ∈ ind f_history)
        ((is-EC_Event(f_history[i]) ∧ is-EC_Event(f_history[j])
          ∧ i < j
          ∧ current_time - s-Time_Stamp(f_history[i]) ≤
                     Min_Initiating_Pair_Separation_Time ) ⊃

            (let mk-EC_Event(event_info_i, ) = f_history[i] in
             let mk-EC_Event(event_info_j, ) = f_history[j] in
             ¬ {event_info_i,event_info_j} ⊂
                                              {SOM,SOTM,ATTN2}))

type: Event_List → (Time_Stamp → BOOL)
```

Rationale:

- No more than two initiating EFs (i.e., SOM, SOTM or ATTN2) may occur within one second [14] (6.2.3.2.b).
- This means that for any two previous EC events (from the history) that happened within that period, they cannot both fall into this category of initiating EFs.

6.4.3.2. Enabling has been Requested by the EC Operator

```
Enabling_Requested_at_EC(f_history) =

    f_history ≠ <>
    ∧
    (if is-EC_Event(Last(f_history)) then
          s-EC_Event_Info(Last(f_history)) = Enable_Comms
     else Enabling_Requested_at_EC(Front(f_history)))

type: Event_List → BOOL
```

Rationale:

- The last EC event in the history must be an operator request for enabling communications.

6.4.4. ATTN4

```
ATTN4_Is_Correct_at_EC(f_history) =

   Disabling_Requested_at_EC(f_history)

type: Event_List → BOOL
```

Rationale:

- ATTN4 from the EC is the EF used to disable communication. ATTN4 can be issued whenever the EC operator requests it [14] (6.2.2.a).

6.4.4.1. Disabling has been Requested by the EC Operator

```
Disabling_Requested_at_EC(f_history) =

   f_history ≠ <>
   ∧
   (if is-EC_Event(Last(f_history)) then
       s-EC_Event_Info(Last(f_history)) = Disable_Comms
    else Disabling_Requested_at_EC(Front(f_history)))

type: Event_List → BOOL
```

Rationale:

- The last EC event in the history must be an operator request for disabling communications.

6.4.5. EOM

```
EOM_Is_Correct_at_EC(f_history)(eom_time) =

   cases Last_n(2,f_history):
      (<mk-INS_Event(RTR,rtr_time),
        mk-EC_Event(ec_info, )> -->
          is-Data_Message(ec_info)
          ∧
          eom_time - rtr_time ≤ Time_Out_Period_at_INS, -- 6.3.2.2.b
      T                         -->
          false)

type: Event_List → (Time_Stamp → BOOL)
```

Rationale:

- An EOM must follow a data message from the EC [14] (6.2.3.2).
- It must be issued before the INS times out the EC.

6.4.6. NAK

```
NAK_Is_Correct_at_EC(f_history)(nak_time) =

   cases Last_n(2,f_history):
      (<mk-INS_Event(data_msg, ),
        mk-INS_Event(EOM,eom_time)> -->
          ¬ Valid_Data_Message(data_msg)                 -- 6.3.2.1.d
          ∧
          nak_time - eom_time ≤ Time_Out_Period_at_INS, -- 6.2.3.2.a
       T                            -->
          false)
```

```
type: Event_List → (Time_Stamp → BOOL)
```

Rationale:

- When a message has been received from the INS (terminated by an EOM) and the validity check of the message fails, the EC responds with a NAK.
- The EC must issue this response before it is being timed out by the INS.

6.4.7. NRTR

```
NRTR_Is_Correct_at_EC(f_history)(nrtr_time) =

   f_history ≠ <>
   ∧
   ¬ Data_Buffer_Is_Ready_at_EC(f_history)                  -- 6.3.2.1.b
   ∧
   cases Last(f_history):
      (mk-INS_Event(SOM,som_time)   -->
          nrtr_time - som_time ≤ Time_Out_Period_at_INS,   -- 6.2.3.2.a
       mk-INS_Event(SOTM,sotm_time) -->
          nrtr_time - sotm_time ≤ Time_Out_Period_at_INS,  -- 6.2.3.2.a
       T                            -->
          false)
```

```
type: Event_List → (Time_Stamp → BOOL)
```

Rationale:

- An NRTR signals that the data buffer at the EC is not ready.
- An NRTR must follow either an SOM or an SOTM from the INS, and do so before the EC is timed out by the INS.

6.4.8. RTR

```
RTR_Is_Correct_at_EC(f_history)(rtr_time) =

   f_history ≠ <>
   ∧
   Data_Buffer_Is_Ready_at_EC(f_history)                    -- 6.3.2.1.b
   ∧
   cases Last(f_history):
      (mk-INS_Event(SOM,som_time)     -->
          rtr_time - som_time ≤ Time_Out_Period_at_INS,    -- 6.2.3.2.a
       mk-INS_Event(SOTM,sotm_time) -->
          rtr_time - sotm_time ≤ Time_Out_Period_at_INS,   -- 6.2.3.2.a
       T                              -->
          false)

type: Event_List → (Time_Stamp → BOOL)
```

Rationale:

- An RTR signals that the data buffer at the EC is ready.
- An RTR must follow either an SOM or an SOTM from the INS, and do so before the EC is timed out by the INS.

6.4.8.1. Determine whether the EC Data Buffer is Ready

```
Data_Buffer_Is_Ready_at_EC(f_history) =

   . . .

type: Event_List → BOOL
```

Rationale:

- [14] does not define when the EC data buffer is ready. The function is included in the model because the availability of a buffer affects certain EC responses (RTR and NRTR).

6.4.9. SOM

```
SOM_Is_Correct_at_EC(f_history)(som_time) =

   Communications_are_Enabled(f_history)
   ∧
   Initiating_EF_Not_Too_Soon_at_EC(f_history)(som_time)
   ∧
   Select_Data_Message_Requested(f_history)

type: Event_List → (Time_Stamp → BOOL)
```

Rationale:

- An SOM can only be sent after communications have been enabled.

- SOM falls into the category of initiating EFs (which includes SOTM and ATTN2 as well), and no more than two such EFs are allowed per second.
- SOMs are only used by the EC to initiate the sending of a select data message. Therefore, a select data message must have been requested by the operator before the EC issues an SOM.

6.4.9.1. Communications are Enabled

```
Communications_are_Enabled(f_history) =

   Ends_in_Enabling_Sequence(f_history)      -- 6.2.1.a-e
   ∨
   (f_history ≠ <>
    ∧
    cases Last(f_history):
       (mk-EC_Event(ATTN2, ) --> false,
        mk-EC_Event(ATTN4, ) --> false,
        T                    -->
           Communications_are_Enabled(Front(f_history))))

type: Event_List → BOOL
```

Rationale:

- Communications are enabled if the event sequence contains an enabling sequence, and there are no later ATTN2s or ATTN4s from the EC after the last enabling sequence.

6.4.9.2. A Select Data Message must have been Requested

```
Select_Data_Message_Requested(f_history) =

   f_history ≠ <>
   ∧
   (if is-EC_Event(Last(f_history)) then
       let mk-EC_Event(ec_info, ) = Last(f_history) in
        ec_info ∈ {Select_Attitude, Select_Navigation,
                   Select_None, Select_Both}
    else Select_Data_Message_Requested(Front(f_history))

type: Event_List → BOOL
```

Rationale:

- The last EC event in the history must be an operator request for sending a select data message.

6.4.10. SOTM

```
SOTM_Is_Correct_at_EC(f_history)(sotm_time) =

   f_history ≠ <>
   ∧
   Initiating_EF_Not_Too_Soon_at_EC(f_history)(sotm_time)
   ∧
   cases Last(f_history):
      (mk-INS_Event(ATTN2,attn2_time)    -->
          sotm_time - attn2_time ≤ Time_Out_Period_at_INS, -- 6.2.1.c-d
       T                                 -->
          Communications_are_Enabled(f_history)
          ∧
          Test_Message_Requested(f_history))

type: Event_List → (Time_Stamp → BOOL)
```

Rationale:

- SOTM falls into the category of initiating EFs (which includes SOM and ATTN2 as well), and no more than two of such EFs are allowed per second.
- Moreover, the SOTM must either:
  - follow an ATTN2 from the INS (as part of enabling the communications), or
  - have been explicitly requested by the EC operator (in which case communications must have been enabled).

6.4.10.1. Determine If a Test Message has been Requested by the EC Operator

```
Test_Message_Requested(f_history) =

   f_history ≠ <>
   ∧
   (if is-EC_Event(Last(f_history)) then
       s-EC_Event_Info(Last(f_history)) = Send Test
    else Test_Message_Requested(Front(f_history)))

type: Event_List → BOOL
```

Rationale:

- The last EC event in the history must be an operator request for sending a test message.

6.5. Determine Correctness of Individual EC Data Message Events

6.5.1. Test Message

```
Test_Msg_Is_Correct_at_EC(f_history)(ec_event) =

   let mk-EC_Event(test_msg, ) = ec_event in
   cases Last_n(2,f_history):
      (<mk-EC_Event(SOTM, ),
        mk-INS_Event(RTR, )> --> Valid_Data_Message(test_msg),
       T                     --> false)

type: Event_List → (EC_Event ⥲ BOOL)
```

Rationale:

- A test message can only be sent to the INS if the INS is ready to receive a test message [14] (6.3.2.3.a and 6.3.2.2.b).
- The test message must be valid when sent.

6.5.2. Select Data Message

```
Select_Data_Msg_Is_Correct_at_EC(f_history)(ec_event) =

   cases f_history:
      (f_history_prefix ^ <mk-EC_Event(SOM, ),
                            mk-INS_Event(RTR, )> -->
          let mk-EC_Event(select_data_msg,event_time) = ec_event in
          Valid_Data_Message(select_data_msg)
          ^
          Corresponding_Select_Data_Msg_Requested(f_history_prefix)
                                                  (select_data_msg)
          ^
          Select_Data_Msg_Not_Too_Often_at_EC(f_history)(event_time),
       T                                         -->
          false)

type: Event_List → (EC_Event ⥲ BOOL)
```

Rationale:

- A select data message can only be sent to the INS if the INS is ready to receive such a message ([14], 6.3.2.2.a-b).
- The select data message must be valid.
- The select data message must specify the selection of data requested by the EC operator.
- Select data messages must not be sent too often (more than once per second).

6.5.2.1. The Corresponding Select Data Message Must have been Requested

```
Corresponding_Select_Data_Msg_Requested(f_history)(select_data_msg) =

   f_history ≠ <>
   ∧
   (if is-EC_Event(Last(f_history)) then
       let mk-EC_Event(ec_info, ) = Last(f_history) in
       let selected_msg_type = s-Selected_Msg_Type(select_data_msg) in
       cases selected_msg_type:
          (Attitude   --> ec_info = Select Attitude,
           Navigation --> ec_info = Select Navigation,
           None       --> ec_info = Select None,
           Both       --> ec_info = Select Both)
    else Corresponding_Select_Data_Msg_Requested(Front(f_history))
                                                   (select_data_msg))

type: Event_List → (Select_Data_Msg → BOOL)
```

Rationale:

- The select data message sent to the INS must reflect the selection specified by the EC operator. The last EC event must be an operator request for sending a select data message.

6.5.2.2. No More than One Select Data Message from the EC per Second

```
Select_Data_Msg_Not_Too_Often_at_EC(f_history)(time_stamp) =

   ¬ (∃ i ∈ ind f_history)
        (is-EC_Event(f_history[i])
         ∧
         is-Select_Data_Msg(s-EC_Event_Info(f_history[i]))
         ∧
         time_stamp - s-Time_Stamp(f_history[i]) <
                                       Min_Select_Data_Separation_time)

type: Event_List → (Time_Stamp → BOOL)
```

Rationale:

- No more than one select data message is allowed per second, i.e., no (other) select data message can have occurred within the last second ([14], pg. 8-10).

7. Formal Model General Functions

7.1. Auxiliary Functions

7.1.1. Determine If a Data Message is Valid

```
Valid_Data_Message(data_msg) =

    (data_msg = Error Data Message --> false,
     T                                 --> ...     )
type: Data_Message → BOOL
```

Rationale:

- This function validates the correctness of data messages at the INS and at the EC.
- However, its detailed specification is not part of the communication protocol and hence is considered to be outside the scope of this effort.

7.1.2. Determine If the History Ends in an Enabling Sequence

```
Ends_in_Enabling_Sequence(f_history) =

   cases Last_n(3,f_history):                   --6.2.1.d-e
      ( < mk-INS_Event(mk-Test_Msg( ), ),
          mk-INS_Event(EOM, ),
          mk-EC_Event(ACK, ) >   --> true,

        T                        --> false )

   ∧
   (∃ i ∈ ind f_history)                             --6.2.1.c-e
      (  cases f_history[i]:
             ( mk-INS_Event(ATTN2, ) --> true,
               T                     --> false )

          ∧
          (∀ j ∈ {k| k ∈ ind f_history ∧ k > i } )
             (  cases f_history[j]:
                    ( mk-INS_Event(ATTN2, ) --> false,
                      T                     --> true ) )

          ∧
          (∀ j ∈ {k| k ∈ ind f_history ∧ k > i ∧ k ≤ (len f_history)-2-3} )
             (  cases < f_history[m] | j ≤ m ≤ j+2 >:
                    ( < mk-INS_Event(mk-Test_msg( ), ),
                        mk-INS_Event(EOM, ),
                        mk-EC_Event(ACK, ) > --> false,
                      T                      --> true ) )

type: Event_List → BOOL
```

Rationale:

- Regardless of retries, a successful enabling sequence must end in the indicated sequence of EFs.
- Moreover, there are no ATTN2s or ATTN4s after the ATTN2$_{INS}$ that was issued in response to the EC initiation of communications via ATTN2$_{EC}$. Note that this function employs the knowledge that the only correct EF to follow an ATTN4 is an ATTN2$_{EC}$ and an ATTN2$_{INS}$ can only follow an ATTN2$_{EC}$.
- Moreover, the only subsequence after the ATTN2$_{INS}$ consisting of Test-Msg$_{INS}$, EOM$_{INS}$, ACK$_{EC}$ is the last three events.

7.1.3. Number of SOMs Sent from the INS and Still Outstanding

```
Number_Of_Outstanding_SOMs_Sent_at_INS(f_history) =

   if f_history ≠ <> then
      cases Last(f_history):
         (mk-EC_Event(last_event, )  -->
             (last_event ∈ {ACK, ATTN2, ATTN4, SOM, SOTM} --> 0, -- 6.3.2.1.e
              T                                           -->
                 Number_Of_Outstanding_SOMs_Sent_at_INS(Front(f_history))),
          mk-INS_Event(last_event, ) -->
             (last_event = SOM                      -->
                 1 + Number_Of_Outstanding_SOMs_Sent_at_INS(Front(f_history)),
              is-Signal_to_INS_Operator(last_event) --> 0,
              T                                     -->
                 Number_Of_Outstanding_SOMs_Sent_at_INS(Front(f_history))))
   else 0

type: Event_List → N0
```

Rationale:

- Yields the number of SOMs issued by the INS for non-terminated messages.

7.1.4. Number of SOTMs Sent from the INS and Still Outstanding

```
Number_Of_Outstanding_SOTMs_Sent_at_INS(f_history) =

   if f_history ≠ <> then
      cases Last(f_history):
         (mk-EC_Event(last_event, )  -->
             (last_event ∈ {ACK, ATTN2, ATTN4, SOM, SOTM} --> 0, -- 6.3.2.1.e
              T                                           -->
                 Number_Of_Outstanding_SOTMs_Sent_at_INS(Front(f_history))),
          mk-INS_Event(last_event, ) -->
             (last_event = SOTM                     -->
                1 + Number_Of_Outstanding_SOTMs_Sent_at_INS(Front(f_history)),
              is-Signal_to_INS_Operator(last_event) --> 0,
              T                                     -->
                Number_Of_Outstanding_SOTMs_Sent_at_INS(Front(f_history))))
   else 0

type: Event_List → N0
```

Rationale:

- Yields the number of SOTMs issued by the INS for non-terminated test messages.

7.2. Constant Functions

7.2.1. Constant Function for INS Time-Out Period

```
Time_Out_Period_at_INS =

   1024            --6.2.3.2.a

type: → N1
```

7.2.2. Constant Function for EC Time-Out Period

```
Time_Out_Period_at_EC =

   800            --6.3.2.1.b

type: → N1
```

7.2.3. Constant Function for Sleep Period

```
Sleep_Period =

   512            --6.3.2.1.b.1

type: → N1
```

7.2.4. Constant Function for Attitude Period

```
Attitude_Period =

   6144          --pg. 8-36

type: → N1
```

7.2.5. Constant Function for Navigation Period

```
Navigation_Period =

   98304       --pg. 8-24

type: → N1
```

7.2.6. Constant Function for the Period Within Which No More Than One Select Data Message May Occur

```
Min_Select_Data_Separation_Time =

   100000              -- pg. 8-10

type:  → N1
```

7.2.7. Constant Function for the Period Within Which No More Than Two Initiating EC EFs May Occur

```
Min_Initiating_Pair_Separation_Time =

   100000              -- 6.2.3.2.b

type:  → N1
```

7.3. Tuple Manipulation

7.3.1. Get Last Event in Event List

```
Last(event_list) =

   event_list[ len event_list]

type: Event_List ⥲ Event

pre: event_list ≠ <>
```

7.3.2. Get Last n Events in Event List

```
Last_n(n, event_list) =

   ( n < len event_list          -->
        < event_list[i] | (len event_list - n + 1) ≤ i
                           ^
                           i ≤ len event_list >,
     n ≥ len event_list)          --> event_list )

type: N0 Event_List →  Event_List
```

7.3.3. Get All Events Except Last One in Event List

```
Front(event_list) =

   < event_list[i] | 1 ≤ i ≤ len event_list - 1>

type: Event_List → Event_List
```

8. Issues in Formal Specification of Communication Protocols

This chapter discusses several issues pertinent to formalizing communication protocols. Its purpose is to illustrate how formal specification allows certain issues to be addressed, and how different specification techniques within our approach are used to address these issues. It is not intended to be a comprehensive survey of formal approaches to defining communication protocols; such a survey can be found in [7].

Specification methods that have been applied to communication protocols can be divided into three major categories [10, 5]:

1. Transition models
2. Programming language models
3. Hybrid models

Transition models define protocols by specifying the behavior of each communicating entity in response to events. Examples of transition models are: finite state machines [8, 6]; formal languages/grammars [11]; and Petri nets [8, 1]. Programming language models describe a protocol as an algorithm and define the algorithm in a high-level language; an example is [9]. Hybrid models extend transition models, typically finite state machines or Petri nets, by introducing variables to limit the number of different states, and thereby avoid the problem of getting unworkably large state machines (a problem known as state explosion); see [8, 10] for examples.

Our model can be characterized as a hybrid model. However, it differs from the traditional hybrid models in that it is not based on a state machine approach but on a formal language approach. The type equations for event sequences, events, etc., define a formal language or grammar describing part of the systems behavior, and the set of Boolean functions further restrict the set of event sequences defined by the type equations. The event sequences that obey the protocol constitute the language.

A formal definition of a communication protocol can, due to its unambiguous interpretation, serve several purposes. It can be used to validate the protocol [5, 10] and as a basis for an implementation [5, 8]. These two roles of a formal protocol specification are further discussed below.

Protocol Validation

The activities involved in ensuring that a system satisfies its design specification and that it operates according to what the user expects is normally referred to as system validation. Verification is a part of validation that can be achieved by formal reasoning about interesting properties of the system. In the case of communication protocols, the properties that one might be interested in verifying include [5, 10]:

- the absence of deadlocks

- completeness (all possible inputs accepted)

In state-based protocol definitions, the absence of deadlock is normally shown by performing an analysis of all reachable global states, where a global state is the combination of all states of the component state machines augmented by the current contents of message buffers, if any. A global state with no exits is either a deadlock or a desired termination state. The process of generating and analyzing all reachable global states can, due to the decidability of finite state machines, be automated [5, 10].

Our approach does not define states. Hence, the concept of "reachable states" does not apply. However, the issue of detecting and avoiding deadlocks is still relevant. In the context of event lists, a deadlock is characterized by an incomplete event list, i.e., " ¬ Is_Complete" in our model, that obeys the protocol, but for which no correct extension exists except for commands from the operator, meaning that there is no correct next communication event.

A communication protocol is said to be complete if all possible inputs are treated. When defining a communication protocol using a state machine, the inputs refer to information received by one of the communicating components from either another communicating component or from the external world. In our model, the communicating components are the INS and the EC. Protocol completeness in the context of our model means that any arbitrary input events at the EC (e.g., INS_Event or Signal_from_EC_Operator) can be appended to the event list at the EC without Protocol_Obeyed_at_EC being undefined and similarly for EC_Events and Protocol_Obeyed_at_INS. In that respect our protocol specification is complete. Allowing all possible inputs is a convenient way of capturing the behavior of a communication line that is not error-free, which is the case for [14], and it automatically ensures completeness. Moreover, due to the fact that unexpected input is generally ignored by the receiver according to [14], our technique of removing such events before considering event histories provides a clean separation of the treatment of normal and abnormal cases. Moreover, separation of the normal from the abnormal cases is a well-accepted software engineering principle. Requiring completeness in a finite state model that allows communication errors generally leads to complex models. This appears to be why such models often assume error-free communication [6].

Protocol Implementation

Finite-state machines and programming language models are fairly easy to turn into implementations. A model such as ours, without states, is more abstract [1] and hence not directly implementable. One way of implementing a protocol that has been specified using our approach is to observe that some of the Boolean functions that describe the correct context of an event capture state-like information about the sequence of events leading up to the event in question. That information can be used to build an extended finite-state model, which can then be implemented in a traditional manner. The general reason for building an extended finite-state machine instead of a pure one is to avoid the problem of state explosion; more specifically, the time-related parts of the protocol are better captured by having time-variables and updating those than by introducing additional states.

9. Suggested Future Work

This chapter briefly discusses three ideas for continued work within the area of formal methods. Each idea involves using the formal specification presented in this report as a starting point in examining different aspects of formal software development. The ideas focus on the practical side of using formal methods. Hence, gaining experience with available tools plays a central role. The ideas involve examining:

1. The stepwise development aspects of VDM
2. Anna [19] and its related tools
3. Statemate [12, 22] as a specification and implementation tool

9.1. VDM and Stepwise Development

The purpose of this work is to further examine VDM. The project has so far only explored formally specifying an existing natural language specification. This prυposed activity will examine the refinement method aspects of VDM. The idea is to develop a system based on the formal specification. The development will involve the design as well as the final implementation of the system. Since the protocol has already been implemented within the REST Project, the suggestion is to develop a testing tool from the formal specification. The formal specification lends itself to this use since it consists of predicates that are true for event histories that obey the communication protocol. One would think of a testing tool in this context as a tool that would evaluate the data transmitted between the two computers to determine whether they are communicating properly. This would involve the following:

- developing a formal specification in Meta-IV for the design of the testing tool using stepwise refinement
- showing that the formal design is a refinement of the formal requirements specification
- implementing the testing tool in Ada from the formal design
- applying the testing tool to data acquired from the current INS implementation

9.2. Anna and Its Tools

Anna is a language that extends Ada by providing means for annotating Ada programs with assertions. Anna is supported by a set of tools including a pre-processor to an Ada compiler that translate assertions into run-time checks and reports any violation of the assertions. The Anna assertions are directly associated with the Ada code and not any preceding specifications. However, being able to trace the assertions back to an abstract specification would increase the value of the assertions significantly. The purpose of the proposed work is to examine Anna and its tool set, and investigate the possibilities of systematically identifying relevant Anna assertions from an abstract formal specification. The idea is to annotate the current implementation of the INS communication protocol developed by the REST Project with Anna assertions that are drawn from the formal specification presented in this document. This would involve:

- drawing pertinent assertions from the formal specification
- annotating the INS implementation of the communications protocol with Anna assertions
- using the Anna tools to perform run-time checks of the implementation of the protocol to demonstrate its correctness

9.3. Statemate

The purpose of this work is to examine state-oriented tools including Statemate [12, 22], which is a state-oriented graphical formalism, by looking at the relationship between our event-list-based VDM formal specification and a corresponding state-based specification. A state machine is a well-established way of specifying and implementing communication protocols. Statemate appears to have an advanced set of tools supporting graphical specification and subsequent implementation in Ada. Given both our model and the Statemate tool set, the opportunity exists to address two kinds of questions: Is using Statemate a good way of implementing our protocol specification? And, assuming that one wants to use Statemate, is our event list based specification helpful in constructing a Statemate specification? The idea is therefore to write a Statemate specification of the communication protocol based upon our current specification. This would include:

- examining our predicates and determining an appropriate corresponding set of states
- creating the formal graphical representation using Statemate
- simulating and generating the resulting implementation

References

[1] Billington, J., G.R. Wheeler, and M.C. Wilbur-Ham.
PROTEAN: A High-Level Petri Net Tool for the Specification and Verification of Communication Protocols.
IEEE Transactions on Software Engineering 14(3):301-316, March, 1988.

[2] Bjørner, D. and C.B. Jones (editors).
Lecture Notes in Computer Science. Volume 61: *The Vienna Development Method: The Meta-Language.*
Springer-Verlag, Berlin, W. Germany, 1978.

[3] Bjørner, D. and O.N. Oest (editors).
Lecture Notes in Computer Science. Volume 98: *Towards a Formal Description of Ada.*
Springer-Verlag, Berlin, W. Germany, 1980.

[4] Bjørner, D.
Rigorous Development of Interpreters and Compilers.
Formal Specification and Software Development.
Prentice/Hall International, Englewood Cliffs, 1982, pages 271-320.

[5] Bochmann, G.V. and C.A. Sunshine.
Formal Methods in Communication Protocol Design.
IEEE Transactions on Communications Com-28(4):624-631, April, 1980.

[6] Brand, D. and P. Zafiropulo.
On Communicating Finite-State Machines.
Journal of the ACM 30(2):323-342, April, 1983.

[7] British Telecommunications et al.
Formal Methods Applied to Protocols (FORMAP).
A Literature Survey, Report No. 1 to the Alvey Directorate , July, 1985.

[8] Danthine, A.S.
Protocol Representation with Finite-State Models.
IEEE Transactions on Communications Com-28(4):632-643, April, 1980.

[9] Good, D.I. and R.M. Cohen.
Verifiable Communications Processing in Gypsy.
In *Compcon78*, pages 28-35. New York, 1978.

[10] Guzman, M.E.
Automatic Validation of Communication Protocol Specifications in Horn Clause Logic.
Technical Report TRITA-CS-8307, The Royal Institute of Technology, Sweden, September, 1983.

[11] Harangozo, J.
An Approach to Describing a Data Link Level Protocol with a Formal Language.
In *Proceedings of 5th Data Communications Symposium.* IEEE, New York, 1977.

[12] Harel, D.
On Visual Formalisms.
Technical Report CMU-CS-87-126, Carnegie Mellon University, Pittsburgh, PA 15213, 1987.

[13] Hoare, C.A.R.
Communicating Sequential Processes.
Comm. of the ACM, August, 1978.

[14] NAVSEA.
Interface Design Specification for the Inertial Navigation Set AN/WSN-5 to External Computer.
Technical Report, NAVSEA T9427-AA-IDS-010/WSN-4, August, 1987.

[15] IEEE Standards Coordinating Committee on Definitions (SCC 10).
IEEE Standard Dictionary of Electrical and Electronic Terms, ANSI/IEEE Std 100-1984.
The Institute of Electrical and Electronics Engineers, Inc., New York, 1984.

[16] Jones, C.B.
Software Development: A Rigorous Approach.
Prentice/Hall International, Englewood Cliffs, N.J., 1980.

[17] Løvengreen, H.H.
Parallelism in Ada.
Lecture Notes in Computer Science. Volume 98. *Towards a Formal Description of Ada.*
Springer-Verlag, Berlin, W. Germany, 1980.

[18] Liu, C.L. and J.W. Layland.
Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.
Journal of the ACM 20(1):46-61, January, 1973.

[19] Luckham, D. C., F. V. von Henke, B. Krieg-Brückner, O. Owe.
Lecture Notes in Computer Science. Volume 260: *ANNA, A Language for Annotating Ada Programs, Reference Manual.*
Springer Verlag, Berlin, W. Germany, 1987.

[20] Oest, O.N.
VDM from Research to Practice.
In *Information Processing 86, Proceedings of the 10th World Computer Congress,* pages 527-533. IFIP, Dublin, Ireland, September, 1986.

[21] Pedersen, J. Storbank.
VDM in Three Generations of Ada Formal Descriptions.
Lecture Notes in Computer Science. Volume 252. *VDM'87. VDM–A Formal Method at Work.*
Springer-Verlag, Berlin, W. Germany, 1987, pages 33-48.

[22] *Documentation for the Statemate1 System.*
AD CAD, Inc., Cambridge, MA 02138, 1987.

Index

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS
NONE

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-88-TR-26

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESD-TR-88-027

6a. NAME OF PERFORMING ORGANIZATION
SOFTWARE ENGINEERING INSTITUTE

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, MA 01731

8a. NAME OF FUNDING/SPONSORING ORGANIZATION
SEI JOINT PROGRAM OFFICE

8b. OFFICE SYMBOL *(If applicable)*
SEI JPO

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F1962885C0003

8c. ADDRESS *(City, State and ZIP Code)*
CARNEGIE MELLON UNIVERSITY
SOFTWARE ENGINEERING INSTITUTE JPO
PITTSBURGH, PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
	N/A	N/A	N/A

11. TITLE *(Include Security Classification)*
Using the Vienna Development Method (VDM) to Formalize a Communications Protocol

12. PERSONAL AUTHOR(S)
Jan Storbank Pedersen and Mark H. Klein

13a. TYPE OF REPORT
FINAL

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Yr., Mo., Day)*
August 1988

15. PAGE COUNT
93 pp.

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Vienna Development Method (VDM)
formal specification
development methods
Meta-IV
communications
SEI

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The Vienna Development Method (VDM) is based upon iterative refinement of formal specifications written in the model oriented specification language, Meta-IV. VDM is also an informal collection of experience in formal specification within several application domains. This paper provides an example of how VDM might be used in the area communications, a new domain for VDM.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. □ DTIC USERS ☒

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED, UNLIMITED

22a. NAME OF RESPONSIBLE INDIVIDUAL
KARL SHINGLER

22b. TELEPHONE NUMBER *(Include Area Code)*
(412) 268-7630

22c. OFFICE SYMBOL
SEI JPO