Computer Science and Systems Analysis

Computer Science and Systems Analysis Technical Reports

Miami University *Year* 1996

Applying an Operational Formal Method to Safety-Critical Systems

Ann Sobel
Miami University, commons-admin@lib.muohio.edu

This paper is posted at Scholarly Commons at Miami University.
http://sc.lib.muohio.edu/csa_techreports/14

MIAMI UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE
& SYSTEMS ANALYSIS

TECHNICAL REPORT: MU-SEAS-CSA-1996-003

Applying an Operational Formal Method to Safety-Critical Systems

Ann E. Kelley Sobel

School of Engineering & Applied Science | Oxford, Ohio 45056 | 513-529-5928

Applying an Operational Formal Method to Safety-Critical Systems

by

Ann E. Kelley Sobel
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #96-003 11/96

Applying an Operational Formal Method to Safety–Critical Systems

Ann E. Kelley Sobel
Systems Analysis
Miami University
Oxford, OH 45056
(513) 529-7541

Abstract

Despite *thirty* years of study by the academic community, industry has not embraced the systematic usage of formal methods. To address this concern, a formal method is proposed which possesses many of the qualities that practitioners have listed as lacking from current formal methods: inclusion of both a specification and verification model, a tabular notation that only requires knowledge of first-order logic, support for both composition and decomposition, application throughout the software life-cycle, and tool support. The presentation includes several applications to safety-critical software systems.

Keywords and Phrases Formal methods, specification, trace-based systems, software development, concurrency, verification.

1. Introduction

Despite *thirty* years of study in the academic community, formal specification and verification models of software systems have been slow in making a large impact on industrial software development. Of the instances of applying formal methods during large-scale software development, most of these systems have involved safety-critical applications [1,7,9,17,18,23]. The safety-critical environment was a natural candidate because of the generally accepted advantages of applying formal method analysis: increased assurance of reliability, predictibility of system behavior, and the ability to identify potential faults and subsequent recovery measures.

Recently, the academic community has focused on why industry hasn't embraced the systematic usage of formal methods [4,24,25,30]. Roundtable discussions from academicians, practitioners, and engineers [25] have identified a number of potential causes: inadequate tools, inadequate examples, the mathematics required is beyond the standard engineering curriculum, lack of support for the entire software life-cycle, scalability, and cost-effectiveness. Some practitioners have felt distanced by the academic community due to the academician's lack of understanding of the industrial problem domain which may, in part, explain why a majority of formalisms aren't readily or directly applicable to industrial problems. This distancing can be so extreme that some practitioners encourage the avoidance of methods that are overloaded with formalisms [30].

Having identified the qualities that are lacking from most formal methods, academicians should now focus on creating formal methods with qualities that industry both needs and desires. Namely, a method that

1. Has a mathematical foundation
2. Abstracts the state of a system to a level that neither leads to omission nor biases an implementation
3. Supports composition
4. Isolates failures
5. Supports modularity
6. Allows developers to use more formal techniques in the specification and design phases, support refinement to executable code, and proof-of properties

7. Is applicable to the industrial problem domain
8. Has tool support
9. Is easily used by engineers

In this paper, a formal method is proposed which meets all the characteristics and qualities outlined above. It is based upon the modular verification model defined in [26] to determine the externally visible behavior of concurrent systems. The majority of the presentation centers on the introduction of a corresponding specification model which is also modular and fully abstract. The specification model was also influenced by the engineering notations used in the variety of specification requirements documents created by NASA contractors for the International Space Station Alpha. The combination of the specification and verification models comprise a formal method that can be applied throughout the software life-cycle, has been used in industrial software development, and appeals to engineers due to its operational nature. The tool PVS [19] can be used to predict behavior and establish proof-of properties of these specifications. The specification model is easy to teach as it is currently taught at the undergraduate level and is used in other core curriculum courses of a software engineering undergraduate degree.

The method used to specify a concurrent program is presented in section II. Examples of the application of this specification model to the Alternating Bit Protocol, Byzantine Agreement, and a safety-critical application are elaborated in section III. The integration of the proposed formal method into the software life-cycle is outlined in section IV and includes an example of refinement into executable code of the Alternating Bit Protocol. A comparison to related works is made in section V.

2. Specification Model

The use of a formal specification notation to outline the functional characteristics of the intended behavior of a concurrent program allows the programmer to be concise and unambiguous. Specifications also aid in understanding the complex interactions between processes and provide a basis for the verification of the resulting program. When producing specifications, it is important to avoid biasing the choice of implementation by including details which suggest or imply a particular implementation strategy. It is also desirable to use the same formalism to specify the characteristics of the system components created by gradually refining the high–level specification of the concurrent program into individual process specifications. Clearly, a formal specification model that provides the ability to express high-level modular specifications without implementation details and that supports hierarchical system decomposition will aid in the creation and comprehension of concurrent programs.

A *process* is specified as the collection of specifications of the externally visible "actions" of the process which will constitute its externally visible behavior. An *action* represents an interaction between this process and the other processes of the concurrent program. The specification of an action describes when that action may occur in terms of the current value of the process trace and the effect of the occurrence of this action on the process trace by the addition of another element to the sequence. The externally visible behavior of a process does not reference any internal state changes nor does it permit assumptions concerning the behavior of the other processes in the program.

A *concurrent program* is specified as the collection of specifications of the externally visible actions of the program. The set of externally visible actions of the processes that constitute the concurrent program is larger than the set of externally visible actions of the program itself. This fact is due to individual process actions caused by internal program interactions exclude them from membership in the set of externally visible actions of the program. It is important to note that this model of specifying concurrency is independent of any particular concurrent programming language.

The specification of the externally visible behavior of each concurrent program captures the global behavior of that program without the need to examine the internal structure of the individual processes. Therefore, our specifications will not bias toward a particular implementation nor a particular concurrent programming language. They are also written at an appropriate level of abstraction since they capture the intended behavior of a concurrent program in sufficient detail to verify its correctness without cluttering the specification with the details of how the program achieves this behavior. Since the specification of a process is defined in terms of its externally visible actions without assuming any possible behavior of the other processes in the program, our specifications are modular: a modification of the implementation of any one process would not cause a change in the specifications of the other processes in the program.

2.1 Trace Notation

The variable h is used to represent the process trace sequence. A subscript, h_i, is used to identify the process to which this trace sequence is associated. Generally, process trace sequences are initialized to the empty sequence, ε. Trace operations are defined as follows.

Notation	*Definition*
$\#h$	the length of h
$h' \sqsubseteq h$	h' is a prefix of h
$h^\frown h'$	h is concatenated to h'
h^r	the reverse of h
h/i	the restriction of the elements of h only to those elements involving process P_i
h^{ext}	the restriction of the elements of h only to those elements involving programs external to the program P

2.2 Process Specifications

A concurrent program P is a *correct implementation of a specification* if the following conditions are satisfied:

1. If the execution of P begins in a state satisfying the precondition of P, then the execution of P terminates in a state in which the values of the process traces, $h_1, \ldots, h_n$, and their initial values satisfy the relation defined by the specification.
2. If the execution of P begins in a state satisfying the precondition of P and the execution of P is not expected to terminate, then the execution of P maintains a state invariant in which the values of the process traces, $h_1, \ldots, h_n$, and their initial values satisfy the relation defined by the specification.
3. If the precondition of P is not satisfied by the initial state, then the specification does not determine the behavior of P.

Thus, a specification determines the precondition on the initial state and the relationship between the initial and final (or intermediate) state(s) of P.

Formally, an individual process execution is represented by a sequence of the form

$$h_i^0 \xrightarrow{\alpha_k} h_i^1 \xrightarrow{\alpha_l} h_i^2 \xrightarrow{\alpha_m} \ldots$$

where h_i^j is the prefix of length j of the process trace sequence h_i and α_k is an element from the set of externally visible actions of the process P_i. All possible process trace sequences for the process P_i defines a "behavioral" model of the process. This sequence of externally visible actions of a process was chosen to coincide with the definition of externally observable behavior in [26] which defined the modular semantics of concurrent systems.

Since actions coincide with the observable behavior of a process, they typically are some form of communication: (a)synchronous message passing or shared variable communication. An action is defined in terms of a change of value of the corresponding process trace. A specification of an action, which includes the current and extended process trace, will be written in two parts: an *enabling* and an *effect*. The effect specifies the change in value of the process trace by concatenating another element to the trace. The enabling part specifies when this action may occur as a guard to the trace update.

To write the specification of a process, the actions which comprise the behavior of that process are determined. For each action, all possible enabling conditions and their effect on the process trace are listed. Tabular notation is used to represent the process specification where each table is an appealing visual representation of the potential changes of a process trace. Specification tables have the following form.

P_i	action
enable	effect

A sequence element has the form, $\langle CA, i, j, \overline{X} \rangle$, where C represents the type of communication (either input or output), A represents the externally visible action, and i names the process to which this element is associated. If C represents output, then a j component is included to name the process that receives the output. Lastly, $\overline{X}$ represents the data sent or received.

The enable condition is typically written in terms of the last element of the process trace sequence at any point in time, namely $h_i^r(1)$. Therefore, an individual element as the enable condition means that the current last element of the sequence is asserted to be of this form. Similarly, the effect lists the element which is added to the process trace sequence for a particular action. To conserve table space, the i

and j component is omitted from elements when their values are readily apparent. Additionally, the action *initial* will be omitted from the table when the process trace sequence is initialized to ε.

The notation, P_i **sat** t_i, indicates that the process trace sequence values satisfy the predicate composed of the specification table entries at any point during the execution of P_i. The predicate t_i is constructed as follows: $\bigwedge_j \ enable_j \ \Rightarrow \ effect_j$ where j ranges over the rows of the table t_i.

2.3 (De)Compositionality

The individual process traces, h_i, are combined to determine a concurrent program trace, h, representing the externally observable behavior of the concurrent system consisting of the composition of processes. The combining of the process traces must obey compatibility, e.g. if P_i sends a particular value to P_j, then this value must be recorded on both sequences in an appropriate location. Essentially, compatibility (or mutual consistency) ensures that the set of process trace sequences under consideration could arise during the execution of the corresponding set of processes. In fact, any program trace sequence can be used to define individual process traces by projecting out and concatenating each element of h which involves that particular process.

It is this observation which forms the basis of the rule for composing the individual process specifications into a concurrent program specification, where P is composed of processes $[\![P_1 \| \ldots \| P_n]\!]$.

$$\frac{\begin{array}{c} P_i \ \mathbf{sat} \ t_i, \quad i = 1, \ldots, n \\ \exists h. \ [\![\ [\ h/1 = h_1 \ \wedge \ t_1 \ \wedge \ h/2 = h_2 \ \wedge \ t_2 \ \wedge \ldots \wedge \ h/n = h_n \ \wedge \ t_n] \ \Rightarrow \ t \ ]\!] \end{array}}{P \ \mathbf{sat} \ t}$$

The following inference rule can be used to weaken the specification, t_i.

$$\frac{P_i \ \mathbf{sat} \ t_i, \ t_i \ \Rightarrow \ t_k}{P_i \ \mathbf{sat} \ t_k}$$

This concurrent program may also be a subsystem of a larger concurrent program. If so, the program trace sequence can be modified to represent the externally visible actions of the program by eliminating all actions involving pairs of processes that are components of this program. The program trace sequence must only contain actions involving an internal process and a process which is external to this program.

Program trace sequences can be decomposed into partial process trace sequences. The elements of the original sequences are partitioned and associated with individual process traces on the basis of the externally visible actions of a process. It is necessary to preserve the original ordering of the elements in the program trace in the process trace. During the process of iterative refinement stages which continually adds implementation details, additional processes will be added to the concurrent program. Each additional process will have a set of externally visible actions which are new to the concurrent program. Those processes which inherit externally visible actions from the concurrent program may also increase their set of actions with additional externally visible actions involving this process and one of the newly created processes.

$$\frac{\begin{array}{c} P \ \mathbf{sat} \ t \\ \forall i \in ProcID(h).\ [\![\ \exists h_i.\ [h/i = h_i^{ext} \ \wedge \ t^i \ \Rightarrow \ t_i] \ ]\!] \end{array}}{P_i \ \mathbf{sat} \ t_i}$$

The function, $ProcID(h)$, returns the set of processes identified in the elements of the program trace sequence h. The notation t^i defines a specification table consisting of those actions which are also externally visible actions of the process P_i. Any references to actions in the remaining enabling conditions which are **not** shared by P and P_i must be eliminated. Finally, we can not assume that the enabling condition as denoted in t refers to the last element of the process trace sequence, h_i. However, it *does* refer to some prior element; so the enabling condition, e, must be modified as follows: $Max\{j : j \leq \#h_i \text{ and } h_i(j) = e\}$.

This inference rule ensures that the order of elements in the program trace sequence h is preserved in the individual process trace sequences h_i for those externally visible actions shared by both the process P_i and the program P. The rule also ensures that the assertion corresponding to the portion of the specification table t of the shared externally visible actions implies the assertion corresponding to the new specification table t_i.

3. Examples

3.1 Alternating Bit Protocol

The Alternating Bit Protocol is a classic network protocol [2]. It requires only one bit of control information to guarantee reliability. The original protocol consists of two processes which can send and receive data from outside users and a potentially faulty communication medium. To simplify the specification, we will only permit process *Receive* to receive data from users and only permit process *Send* to send data to users.

The process *Receive* accepts data from an external unbounded source of data, adds a sequence bit to the data to create a message, and sends this message to process *Send* by calling the method *Forward*. The process *Send* receives the message, strips off the sequence bit, and outputs the data to an unbounded sink. Acknowledgments, defined as a sequence bit, are sent back to the process *Receive*. The communication medium is modeled as single element buffers that can change a sequence number to the constant "error" (any integer other than 0 or 1) to represent a corruption of the data. The specification of this protocol must demonstrate the delivery of the messages in the correct order despite possible corruption by the medium.

Receive	Action		
Enable	Accept	Forward	Reply
$\{\langle ?\text{Reply,ack}\rangle \vee \varepsilon\}$	$\langle ?\text{Accept,data}\rangle$		
$\{\langle ?\text{Accept,data}\rangle \vee \langle ?\text{Reply, ack}\rangle\}$		$\langle !\text{Forward,r\_ack,data}\rangle$	
$\langle !\text{Forward,r\_ack,data}\rangle$			$\langle ?\text{Reply,ack}\rangle$

Send	Action		
Enable	Forward	Write	Reply
$\{\langle !\text{Reply,s\_ack}\rangle \vee \varepsilon\}$	$\langle ?\text{Forward,num,info}\rangle$		
$\langle ?\text{Forward,num,info}\rangle$		$\langle !\text{Write,info}\rangle$	
$\{\langle ?\text{Forward,num,info}\rangle \vee \langle !\text{Write, info}\rangle\}$			$\langle !\text{Reply,s\_ack}\rangle$

The specification of the process *Receive* includes the datum to be sent and *r_ack* which is used to determine the receipt of the acknowledgement from the process *Send*. The specification of the process *Send* defines *info* to represent the message received and *num* which is used to determine whether *info* contains an uncorrupted new message.

The specification of the Alternating Bit Protocol must ensure the delivery of these messages in the correct order. From examination of the process trace sequence h_{Res} for the process *Receive*, a sequence of data items, $\overline{r}$, can be constructed representing all data received from the outside source. Correspondingly, the sequence $\overline{s}$, containing all data sent to the users, can be constructed from the process trace sequence h_{Send} for the process *Send*. Therefore, the specification for the protocol will ensure that $\overline{r} \sqsubseteq \overline{s}$ for the protocol trace h.

3.2 Byzantine Agreement

The Byzantine Agreement problem, first introduced by Pease et al.[22], illustrates fault tolerance in distributed systems. A system which communicates by message passing has two different kinds of processes: *reliable* and *unreliable*. If there are m $(m \geq 0)$ unreliable processes, then there must be at least $3 * m + 1$ reliable processes for a solution to the Byzantine Agreement problem to exist. There is a designated process, called *commander*, that may or may not be reliable. Each process i has a local variable byz_i. Agreement is reached when every reliable process sets its local variable, byz_i, to a common value. If the *commander* process is reliable, then this common value is d_c, the initial value of the *commander's* local variable; otherwise, the common value is NIL. Since *reliable* processes are indistinguishable from *unreliable* ones, it is unknown at the time of receipt whether an individual message is arbitrary or not.

The specification of the Byzantine Agreement algorithm states that

$$byz_i \ = \ byz_j, \text{where } i, j \text{ are } \textit{reliable} \text{ processes}$$

$$\text{and}$$

$$\textit{commander} \text{ is reliable} \implies byz_i \ = \ d_c$$

The algorithm assumes a function *majority* with the property that if a majority of the values v_i equal d_c then *majority*$(v_1, \ldots, v_{n-1})$ equals d_c. If no majority value among the v_i exists, then *majority*$(v_1, \ldots, v_{n-1})$ equals NIL. For simplicity, we will initially assume that $m = 1$. The specifications for the *commander* and an arbitrary process P_i from the $n-1$ processes are given.

Commander	Action
Enable	RecC
$\{\varepsilon \vee \langle$!RecC,k,v$_k\rangle\}$	$\langle$!RecC,1,v$_1\rangle$
...	...
$\{\varepsilon \vee \langle$!RecC,k,v$_k\rangle\}$	$\langle$!RecC,n-1,v$_{n-1}\rangle$

P_i	Action		
Enable	RecC	Forwd	RecP
ε	$\langle$?RecC,v$_i\rangle$		
$\{\langle$?RecC,v$_i\rangle \vee \langle$!Forwd,k,v$_i\rangle\}$ $\exists k\ [1 \leq k \leq n-2 \wedge k \neq i]$		$\langle$!Forwd,j,v$_i\rangle$ $\forall j\ [1 \leq j \leq n-2 \wedge j \neq i]$	
$\langle$!Forwd,k,v$_i\rangle$ $\exists k\ [1 \leq k \leq n-2 \wedge k \neq i]$			$\langle$?RecP,v$_j\rangle$ $\forall j\ [1 \leq j \leq n-2 \wedge j \neq i]$

In the first step, the *commander* sends a value v_i to all $n-1$ processes. If the *commander* is reliable, the v_i are equal to d_c, the value of the *commander's* local variable. If the *commander* is unreliable, the values of the v_i are arbitrary. Once the process P_i receives the *commander's* message, the value received is forwarded to the other $n-2$ processes. P_i either receives a copy of the value that each other process received from the *commander* or an arbitrary value. The *majority* function is then used to determine the value of byz_i.

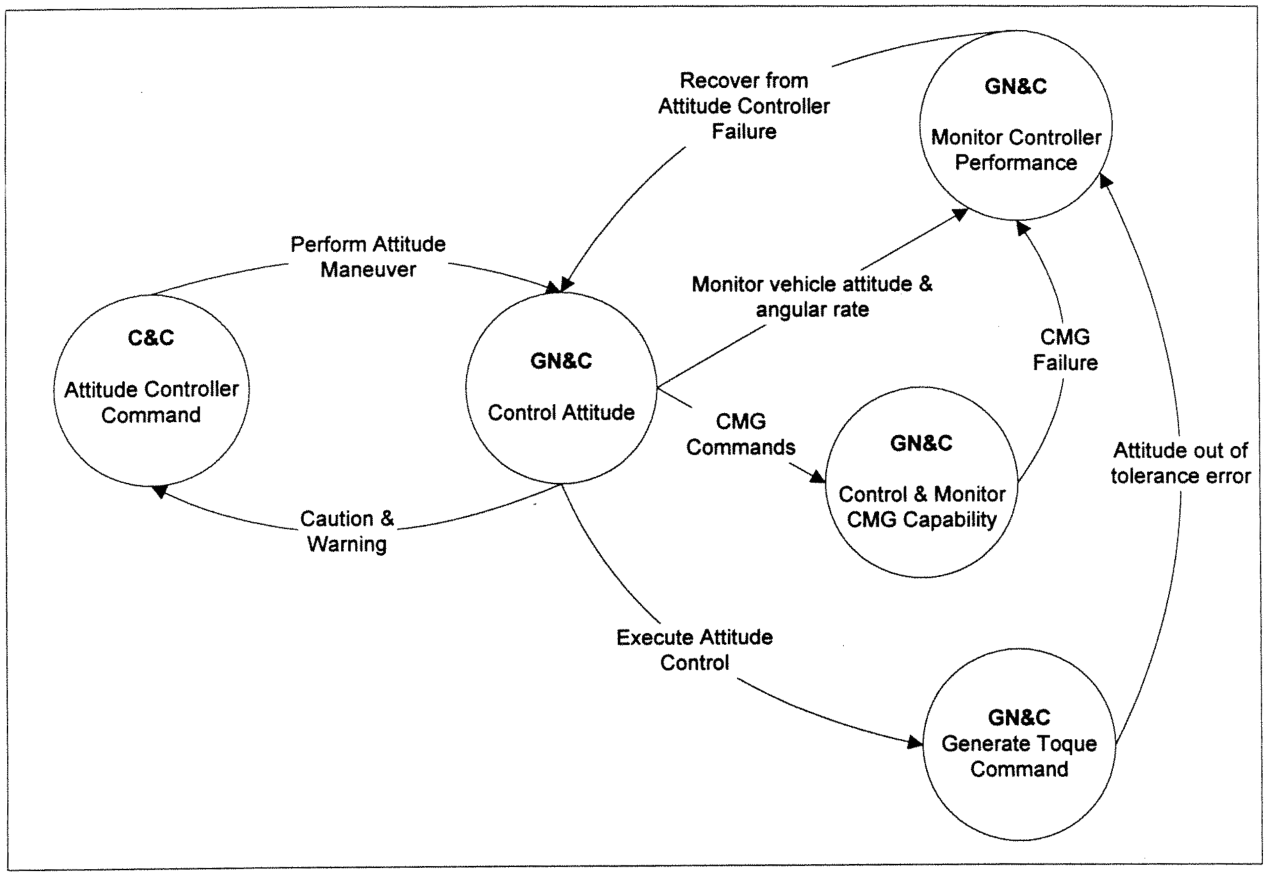
The complexity of the Byzantine Agreement algorithm grows significantly with larger numbers of faulty processes. By just increasing m to 2, the specification table must be duplicated $n-1$ times.

3.3 ISSA Command & Control Example

As part of an ongoing project by NASA to study the effectiveness of formal methods in improving the quality of software requirements, formal analysis of the software requirements of the Failure, Detection, Isolation, and Recovery (FDIR) system of the International Space Station Alpha (ISSA) has been performed [6]. This analysis focused on whether the dynamic interactions of the FDIR system were both consistent and complete and whether the most catastrophic failures had appropriate recovery measures. The difficulty of this analysis is compounded by the fact that multiple subsystems are responsible for identifying, isolating, and recovering from just *one* type of failure.

The software subsystem, Command & Control (C&C), of ISSA is mainly responsible for station level control, command and data handling, and communications functions. It is this system which issues the command to perform nonpropulsive attitude maneuvers; a maneuver which is performed as part of the docking of the Space Shuttle with ISSA. The ISSA subsystem, Guidance, Navigation and Control (GN&C), is responsible for navigation, attitude determination, and attitude control. The following diagram illustrates the actions taken by these two subsystems when an attitude maneuver command is issued and a Caution & Warning (e.g. failure) event occurs.

In the following diagram, nodes represent modular components of either the C&C or the GN&C subsystem and the arcs represent a subset of possible actions of the two subsystems.


Recover from Attitude Controller Failure
GN&C
Monitor Controller Performance
Perform Attitude Maneuver
Monitor vehicle attitude & angular rate
C&C
Attitude Controller Command
GN&C
Control Attitude
CMG Failure
CMG Commands
GN&C
Control & Monitor CMG Capability
Attitude out of tolerance error
Caution & Warning
Execute Attitude Control
GN&C
Generate Toque Command

The tabular specification of the C&C attitude controller and the GN&C control attitude subsystem follow.

C&C AC	Action	
Enable	AttMan	ComRes
$\{\varepsilon \vee \langle ?\text{ComRes}, status\rangle\}$	$\langle !\text{AttMan}, \overline{X}\rangle$	
$\langle !\text{AttMan}, \overline{X}\rangle$		$\langle ?\text{ComRes}, status\rangle$

GN&C CA	Action	
Enable	AttMan	ComRes
$\{\varepsilon \vee \langle !\text{ComRes},\mathit{status}\rangle\}$	$\langle ?\text{AttMan},\overline{X}\rangle$	
$\langle ?\text{AttMan},\overline{X}\rangle$		$\langle !\text{ComRes},\mathit{status}\rangle$

In the diagram, the GN&C control attitude subsystem is composed of four separate components: control attitude manager, torque command generator, CMG (control moment gyros) control & monitor, and the controller performance monitor. To illustrate the compositionality of individual component specification tables, the separate GN&C component specifications are given. Those actions which involve only GN&C control attitude subsystem components are *internal* to this subsystem and are not represented in the GN&C CA specification table.

GN&C CA_Mgmt	Action			
Enable	AttMan	AttCntl	CMG_C	...
$\{\varepsilon \vee \langle !\text{ComRes},\mathit{status}\rangle\}$	$\langle ?\text{AttMan},\overline{X}\rangle$			
$\{\langle ?\text{AttMan},\overline{X}\rangle \vee \langle !\text{CMG\_C},\overline{Y}\rangle \vee \langle !\text{MonAtt},\overline{Z}\rangle\}$		$\langle !\text{AttCntl},\overline{W}\rangle$		
$\{\langle ?\text{AttMan},\overline{X}\rangle \vee \langle !\text{AttCntl},\overline{W}\rangle \vee \langle !\text{MonAtt},\overline{Z}\rangle\}$			$\langle !\text{CMG\_C},\overline{Y}\rangle$	

GN&C CA_Mgmt (cont)	Action		
Enable	MonAtt	CntlRes	ComRes
$\{\langle ?\text{AttMan},\overline{X}\rangle \vee \langle !\text{AttCntl},\overline{W}\rangle \vee \langle !\text{CMG\_C},\overline{Y}\rangle\}$	$\langle !\text{MonAtt}, \overline{Z}\rangle$		
$\{\langle !\text{AttCntl},\overline{W}\rangle \vee \langle !\text{CMG\_C},\overline{Y}\rangle \vee \langle !\text{MonAtt},\overline{Z}\rangle\}$		$\langle ?\text{CntlRes}, \mathit{status}\rangle$	
$\langle ?\text{CntlRes},\mathit{status}\rangle$			$\langle !\text{ComRes},\mathit{status}\rangle$

GN&C CntlPer	Action			
Enable	MonAtt	AttRes	CMGRes	CntlRes
$\{\langle \varepsilon \vee \langle ?\text{CMGRes},GS\rangle \vee \langle ?\text{AttRes},AttS\rangle\}$	$\langle ?\text{MonAtt},\overline{Z}\rangle$			
$\{\varepsilon \vee \langle ?\text{MonAtt},\overline{Z}\rangle \vee \langle ?\text{CMGRes},GS\rangle\}$		$\langle ?\text{AttRes},AttS\rangle$		
$\{\varepsilon \vee \langle ?\text{MonAtt},\overline{Z}\rangle \vee \langle ?\text{AttRes},AttS\rangle\}$			$\langle ?\text{CMGRes},GS\rangle$	
$\{\langle ?\text{MonAtt},\overline{Z}\rangle \vee \langle ?\text{AttRes},AttS\rangle \vee \langle ?\text{CMGRes},GS\rangle\}$				$\langle !\text{CntlRes},status\rangle$

GN&C CMG_Cap	Action	
Enable	CMG_C	CMGRes
$\{\varepsilon \vee \langle !\text{CMGRes},GS\rangle\}$	$\langle ?\text{CMG\_C},\overline{Y}\rangle$	
$\langle ?\text{CMG\_C},\overline{Y}\rangle$		$\langle !\text{CMGRes},GS\rangle$

GN&C GenTorq	Action	
Enable	AttCntl	AttRes
$\{\varepsilon \vee \langle !\text{AttRes},AttS\rangle\}$	$\langle ?\text{AttCntl},\overline{W}\rangle$	
$\langle ?\text{AttCntl},\overline{W}\rangle$		$\langle !\text{AttRes},AttS\rangle$

4. Application in Software Development

The requirements specification of a software system is only one phase of the software life cycle. First, the specifications are written, analyzed for consistency and completeness, and tested. The specifications are then refined by iterative design stages into an implementation in a particular programming language. Next, the implementation is verified and processes are tested independently. Processes are combined into independent modules and tested. Modules are integrated into a system, which is tested and validated. Finally, the system is maintained.

The proposed formal method is all encompassing in that it can be applied to any phase of the software life cycle. This formal method contains a specification model that is both modular and compositional. A specification model possessing these characteristics supports the development process by driving the creation of smaller, disjoint system components of the concurrent program. One of the most important issues concerning specifications is the assurance that a particular implementation satisfies its specification. The verification model included in the proposed formal method provides this assurance by establishing either a postcondition or invariant of the implementation which can be shown to imply the specification.

The productivity of the development process is improved by locating errors at the earliest moment; ideally, when creating and analyzing the specification but certainly when ensuring that a particular implementation satisfies a specification. The specifications can be analyzed using the tool PVS [19] and the tabular specification notation is a natural origin for test case generation. Lastly, the maintainability of the system is simplified by providing a formal statement of the individual system component's behavior.

To demonstrate the applicability of the formal method to other phases of the software life cycle, an example of establishing that an implementation satisfies its specification follows.

3.1 Refinement of the Alternating Bit Protocol

The implementation language chosen for the Alternating Bit Protocal is Java which supports thread synchronization through the use of monitors. In general, a monitor encapsulates data along with a set of access functions which support single thread access to a data item. Critical sections are identified with the keyword **synchronized** and thread (de)activation is controlled by **wait** and **notify** commands.

An implementation of the specification of the Alternating Bit Protocol presented in section III follows.

```
class Receive
{
        private item data;
        private int ackno, WaitAck = 1, LastSent = 0;
        private boolean progress = false;
        public synchronized void Accept(item data) {
            while (LastSent == WaitAck && progress == false)
                wait();
            LastSent = (LastSent + 1) MOD 2;
            Forward(LastSent, data);
            while (progress == false);
            progress = false;
            while (LastSent == WaitAck) {
                Forward(LastSent, data);
                while (progress == false);
                progress = false;
            }
        }
        protected synchronized void Reply(int ackno) {
            if (ackno == WaitAck) {
                WaitAck = (WaitAck + 1) MOD 2;
                progress = true;
                notify();
            }
            else
                progress = true;
        }
    }

    class Send {
        private item info;
        private int messno, NextRequired = 1;
        protected void Forward(int messno, item info) {
            if (messno == NextRequired) {
                Write(info);
                NextRequired = (NextRequired + 1) MOD 2;
            };
            Reply((NextRequired - 1) MOD 2);
        };
    };
```

The class *Receive* includes the suspension of the execution of the method *Accept* when a new data item is received but notification by the class object *Send* confirms that the previous data item forwarded to *Send* has been corrupted. The current data item must be repeatedly forwarded to *Send* until notification by *Send* confirms that the data item has been received. At this time, any one of the suspended threads of execution can be resumed.

To prove this Java implementation satisfies the specification of the Alternating Bit Protocol presented in section III, one must verify that the postcondition of each call and of each method called defines a process trace for the class that satisfies the corresponding row of the class' specification table. A full set of axioms and rules of inference for monitors are provided in [28] in order to perform this task; however, highlights of this process follow.

A process trace will be associated with each declared object of a class; for this example, h_{Rec} and h_{Send}. The elements comprising the process trace sequence h_{Send} are very similar to those listed in the specification table for the process *Send* when including the information omitted when preserving table space. The major difference is in the elements representing either a call to a method or that a method is called. Instead of using a one action element, two trace elements are used to record the call being made and the subsequent return from the method called as well as the start and end of the execution of a method.

On the other hand, the elements included in the process trace sequence h_{Rec} include several new trace elements, two of which are the new elements listed for h_{Send}. Two more elements will be introduced to represent the suspensions and resumptions of the threads of execution which occur due to the **wait** and **notify** commands. Therefore, the execution of a **wait** command causes the addition of a suspension element to the class object trace sequence h_{Rec} and the execution of a **notify** command causes the addition of a resume element. These new elements contain additional information due to the need to introduce incarnation numbers which represent particular incarnations of the method *Accept* [28].

For the method *Forward* in the class object *Send*, the postcondition is as follows:

Postcondition

$$\begin{aligned}
&\{\ h^{r}_{Send}(1) = \langle \text{!Forward} \rangle \ \wedge\ h^{r}_{Send}(2) = \langle \text{?Reply} \rangle \ \wedge \\
&\quad h^{r}_{Send}(3) = \langle \text{!Reply}, \mathit{NextRequired} - 1\ \mathit{Mod}\ 2 \rangle \ \wedge \\
&\quad \wedge\ \mathit{NextRequired} = \mathit{messno} \ \Rightarrow \\
&\qquad \{\mathit{NextRequired} = \mathit{NextRequired} + 1\ \mathit{Mod}\ 2\ \wedge \\
&\qquad\quad h^{r}_{Send}(4) = \langle \text{?Write} \rangle \ \wedge\ h^{r}_{Send}(5) = \langle \text{!Write}, \mathit{info} \rangle \ \wedge \\
&\qquad\quad h^{r}_{Send}(6) = \langle \text{?Forward}, \mathit{messno}, \mathit{info} \rangle \} \\
&\quad \wedge\ \mathit{NextRequired} \neq \mathit{messno} \ \Rightarrow \\
&\qquad \{h^{r}_{Send}(3) = \langle \text{!Reply}, \mathit{NextRequired} - 1\ \mathit{Mod}\ 2 \rangle \ \wedge \\
&\qquad\quad h^{r}_{Send}(4) = \langle \text{?Forward}, \mathit{messno}, \mathit{info} \rangle \} \ \}
\end{aligned}$$

After making the substitution of the specification variable names with the corresponding implementation variable names, removing from h_{Rec} those elements which were added for the purpose of applying the verification model to this implementation, and transforming the postcondition of the method *Forward* into an invariant for the class object *Send*, will allow this invariant to satisfy the specification table in section III.

For the method *Reply* defined in the class object *Receive*:

Postcondition $\{\ h^{r}_{Rec}(1) = \langle !\text{Reply}\rangle \ \wedge\ \{ackno \neq WaitAct\ \Rightarrow$
$h^{r}_{Rec}(2) = \langle ?\text{Reply}, ackno\rangle\} \ \wedge\ \{\ ackno = WaitAck\ \Rightarrow$
$\{WaitAck = WaitAck + 1\ Mod\ 2\ \wedge$
$h^{r}_{Rec}(2) = \langle ?\text{Resume}\rangle\}\ \}\ \wedge\ progress = true\ \}$

For the method *Accept* defined in the class object *Receive*:

Postcondition $\{\ h^{r}_{Rec}(1) = \langle !\text{Accept}\rangle\ \wedge\ progress = false\ \wedge$
$\exists k.\ \{\forall j.\ 2 \leq j \leq 2*k.\ [\ Even(j) \wedge (LastSent + 1Mod2 = WaitAck)\ \Rightarrow$
$h^{r}_{Rec}(j) = \langle ?\text{Forward}\rangle\ \wedge$
$h^{r}_{Rec}(j+1) = \langle !\text{Forward}, LastSent, data\rangle\ ]\ \wedge$
$h^{r}_{Rec}(2*k+2) = \langle ?\text{Forward}\rangle\ \wedge$
$h^{r}_{Rec}(2*k+3) = \langle !\text{Forward}, LastSent, data\rangle\ \wedge$
$[LastSent \neq WaitAck\ \Rightarrow\ h^{r}_{Rec}(2*k+4) = \langle ?\text{Accept}, data\rangle]\ \wedge$
$[LastSent = WaitAck\ \Rightarrow\ h^{r}_{Rec}(2*k+4) = \langle !\text{Suspend}\rangle\ \wedge$
$h^{r}_{Rec}(2*k+5) = \langle ?\text{Accept}, data\rangle]\ \}\}$

Combining the postconditions of the two methods and handling the multiple incarnations of the method *Accept* make the determination of the class object invariant for *Receive* difficult. The construction of the invariant requires the application of several proof rules found in [28] which are not presented here. However, the pattern of sequence elements imposed by both the *Accept* and *Reply* method is consistent with the specification table for *Receive*.

The individual class object invariants will be combined with a compatibility requirement on the class object trace sequences to ensure that any action in which two objects participate is recorded on the two trace sequences in a mutually consistent fashion. The definition of $Compat(h_1, \ldots, h_n)$ is as follows.

$$\exists\ \text{h}.\ [\forall \text{i} \leq \text{n}\ [\text{h}/\text{i}=\text{h}_i]]$$

Therefore, the values for those elements recorded in the trace sequence h_{Rec} when issuing a call to the method *Forward* will indeed be recorded in the trace sequence elements for receiving the method call in h_{Send}. The proposed two sequences of data items, $\overline{r}$ and $\overline{s}$, can be constructed from the $?Accept$ elements of h_{Rec} and from the $!Write$ elements of h_{Send} to ensure that this implementation satisfies the protocol requirement, $\overline{r} \sqsubseteq \overline{s}$.

5. Comparison to Related Works

Trace-based semantics for networks are based on either individual channel or process traces. The majority of these proof systems are of the channel trace variety and unfortunately suffer from being incomplete [14]. In order to overcome incompleteness, some authors have resorted to abandoning first-order logic in favor of temporal logic [31]. The process trace-based model presented here is (relatively) complete [5], relies only on first-order logic, and is modular [26]. It has been applied to all forms of concurrency and in a production-quality, large-scale software development system [27]. This process trace semantic model is a full abstraction of the operational model (contains sufficient information to prove essential properties of the model without containing too much information, e.g. the internals of a process).

Lamport[15,16] introduces specifications which consist of a collection of state functions that map program states into sets of values; a collection of initial values for these functions which define the set of states in which the system may begin computation; and a collection of properties, written using temporal logic, describing both the safety and liveness conditions required of the system. Lamport's specification model significantly differs from the proposed specification model in that only externally visible behavior of the concurrent program is specified. Another major departure lies in the use of control predicates and program counters in the specification of a process. Conversely, the proposed specification model captures a process' behavior at a higher level of abstraction. In the proposed model, a process can be viewed as an abstract data object where the externally visible actions change the value (i.e. state) of the object. Therefore, the specification of a process defines the semantics of this abstract data object without including information concerning its implementation. In order to verify a process, Lamport must introduce auxiliary state functions and use temporal logic whereas in the proposed model the corresponding verification method in [26] can be directly applied to the original specifications.

Leveson's AND/OR Tables [11] are a tabular representation of the disjunctive normal form of a Boolean expression. Specifications are written in RSML, Requirements State Machine Language, which uses state transitions to capture the functional behavior of the software. Transition conditions are translated into AND/OR tables. Functions were chosen to guarantee completeness and consistency and tools are provided to support this analysis. However, this choice forces specifications to be deterministic. AND/OR tables can become difficult to use when transition conditions are written using such first-order logic operators as implication and equivalence. Lastly, the application of this specification model, as well as the remaining specification models in this section, focuses only on the requirements specification stage of the software life-cycle.

Parnas proposes the use of tabular representations of relations [20,21] for creating program specifications which describe a set of state sequences in a finite state machine. A variety of table formats are used to define the mathematical functions and relations that capture the state transitions of the software system. The table format is proported to aid in the understanding of multidimensional expressions and simplify the inspection of requirements specification documents. Numerous rules are provided for changing table formats in order to provide the most readable function definition. The main emphasis for providing a tabular notation is to support the construction of readable systems requirements documents.

The SCR, Software Cost Reduction, method [12] describes the functional requirements of software and is applied during the requirements specification of a software system. This method includes Parnas' tabular representations to describe system functions, timing, and precision; however, the SCR method attempts to provide a formal basis through the use of a (deterministic) state automaton, monitored and controlled variables, conditions, and events. Table functions are used to define output variables, terms, and mode classes for condition, event, and mode transition tables. Tools are available for consistency and completeness checks. The main application of this method is to embedded process control systems.

6. Summary

A formal method is proposed that meets many of the needs and desires of the industrial community: the method can be applied throughout the software life-cycle, has been used during industrial software development, and uses a notation that appeals to engineers. The specification model proposed records the external behavior of a process in a tabular manner using process trace sequences which record the externally visible actions of this process. Using this model, the specification of the externally visible behavior of each concurrent program captures the program's behavior without examining the internal structure of the individual processes. The modular proof systems presented in [26] can be used to verify that a particular implementation meets its specification and current tools exist to support this activity.

Acknowledgements

Sincerest thanks are extended to Brent Auernheimer for his careful reading and suggestions on the improvement of the presentation of this work. The International Space Station Alpha work was started while the author was supported as a NASA/ASEE Summer Faculty Fellow at NASA's Jet Propulsion Laboratory, California Institute of Technology. The teaching of this formal method as part of the undergraduate core curriculum is supported by an NSF Educational Innovation Grant, CDA-9522257.

References

1. Barroca, L.M. and J.A. McDermid, "Formal Methods: Use and Relevance for the Development of Safety–Critical Systems", *The Computer Journal*, Vol. **35**, No. **6**, 1992, pp. 579–599.

2. Bartlett, K.A., R.A. Scantlebury, and P.T. Wilkinson, "A Note on Reliable Full–Duplex Transmission Over Half–Duplex Links". *Comm. ACM*, Vol **12**, No. **5**, pp. 260–261, May 1969.

3. Bowen, J.P. and M.G. Hinchey, "Seven More Myths of Formal Methods", *IEEE Software*, July 1995, pp. 34–41.

4. Butler, R.W., J.L. Caldwell, V.A. Carreno, C.M. Holloway, P.S. Miner, B.L. DiVito, "NASA Langley's Research and Technology-Transfer Program in Formal Methods", *COMPASS'95*: Proc. of the 10th Annual Conference on Computer Assurance, June 25-29, 1995, pp. 135–149.

5. Cook, S.A., "Soundness and Completeness of an Axiom System for Program Verification", *SIAM Journal on Computing*, Vol. **7**, February 1978, pp. 70–90.

6. Covington, R., A. Sobel, and K. Freise, "Applications of Formal Methods to International Space Station Alpha FDIR: Phase II Summary", Internal Report, NASA Johnson Space Center, April 1996.

7. Gerhart, S., D. Craigen, and T. Ralson, "Experience with Formal Methods in Critical Systems", *IEEE Software*, Vol. **11**, No. **1**, January 1994, pp. 21-28.

8. Hall, A., "Seven Myths of Formal Methods", *IEEE Software*, September 1990, pp. 11–19.

9. Hall, A., "Using Formal Methods to Develop an ATC Information System", *IEEE Software*, Vol. **13**, No. **2**, March 1996, pp. 66–76.

10. Hatley, D.J. and I.A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House Publishing Co., Inc., 1987.

11. Heimdahl, M. P.E. and N.G. Leveson, "Completeness and Consistency in Hierarchical State-Based Requirements", *IEEE Trans. on Software Engineering*, May 1996.

12. Heitmeyer, C., B. Labaw, and D. Kiskis, "Consistency Checking of SCR-Style Requirements Specifications", *Intern. Symp. on Requirements Engineering*, York, England, March 1995.

13. Hoare, C.A.R., "An Overview of Some Formal Methods for Program Design", *IEEE Computer*, September 1987, pp. 85–91.

14. Jonsson, B., "A Fully Abstract Trace Model for Dataflow Networks", *16th ACM POPL*, Austin, Texas, 1989, pp. 155-165.

15. Lamport, L., "A Simple Approach to Specifying Concurrent Systems", *Comm. ACM*, Vol. **32**, No. **1**, January 1989, pp. 32–45.

16. Lamport, L., "The Temporal Logic of Actions", *ACM Trans. Program. Lang. Syst.*, Vol. **16**, No. **3**, May 1994, pp. 872–923.

17. Larsen, P.G., J. Fitzgerald, T. Brookes, "Applying Formal Specification in Industry", *IEEE Software*, Vol. **13**, No. **3**, May 1996, pp. 48–56.

18. Miller, S.P. and M. Srivas, "Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial use of Formal Methods", *Proc. of the Workshop on Industrial-Strength Formal Specification Techniques*, April 5-8, 1995, pp. 2–16.

19. Owre, S., N. Shankar, and J. Rushby, "User Guide for the PVS Specification and Verification System", Computer Science Laboratory, SRI International, Menlo Park, CA, March 1993.

20. Parnas, D.L., "Mathematical Description and Specification of Software", *IFIP 94*, Vol. **1**, pp. 354–359.

21. Parnas, D.L., "Tabular Representation of Relations", CRL Report No. **260**, October 1992.

22. Pease, M., R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults", *Journ. ACM*, Vol. **27**, No. **2**, April 1980, pp. 228–234.

23. Rushby, R., "Formal Methods and the Certification of Critical Systems", Technical Report *SRI-CSL-93-07*, November 1993.

24. Saiedian, H. and M.G. Hinchey, "Challenges in the Successful Transfer of Formal Methods Technology into Industrial Applications", *Information and Software Technology*, Vol. **38**, No. **5**, May 1996.

25. Saiedian, H., "An Invitation to Formal Methods", *Computer*, Vol. **29**, No. **4**, April 1996, pp. 16–30.

26. Sobel, A.E.K. Modular Verification of Concurrent Systems. Ph.D. Dissertation, The Ohio State University, August 1986.

27. Sobel, A.E.K. "Proposed SEDL Specification of Concurrency", IBM Research Advanced Abstract, RC **13336**, December 1987.

28. Sobel, A.E.K. and N. Soundararajan, "A Proof System for Distributed Processes", *Acta Informatica*, Vol. **25**, No. **3**, 1988, pp. 305-332. The extended abstract appears in *LNCS* **193**: Proceedings of the Logics of Programs, pp. 343–358.

29. Soundararajan, N., "Tracing the Missing Information", Technical Report **OSU-CISRC-8/94-TR46**, The Ohio State University, 1994.

30. Weber–Wulff, D., "Selling Formal Methods to Industry", *LNCS* **670**, FME'93: Industrial-Strength Formal Methods, April 1993, pp. 671–677.