Using SPARK-Ada to Model and Verify a MILS Message Router*

Bryan Rossebo, Paul Oman, Jim Alves-Foss, Ryan Blue, and Paul Jaszkowiak
Center for Secure and Dependable Systems
University of Idaho
Moscow, ID 83844-1008

Abstract

The concept of information classification is used by all nations to control information distribution and access. In the United States this is referred to as Multiple Levels of Security (MLS), which includes designations for unclassified, confidential, secret, and top secret information. The U.S. Department of Defense has traditionally implemented MLS separation via discrete physical devices, but with the transformation of military doctrine to net-centric warfare, the desire to have a single device capable of Multiple Independent Levels of Security (MILS) emerged. In this paper we present a formal model of a MILS message router using SPARK-ADA. The model is presented as a case study for the design and verification of high assurance computing systems in the presence of an underlying separation kernel. We utilized the correctness-by-design approach to secure system development and discuss the limitations of that approach for the type of system we model.

1. The need for certifiably secure systems

One of the largest problems facing the field of computer science is that of computer and network security. With the increased connectivity of Information Technology (IT) systems and process control systems, security is needed to defend against malicious persons intent on abusing or attacking network resources. This is especially true for unbounded networks like the Global Information Grid (GIG) [1].

Every year, billions of dollars are lost due to cyber intrusions and computer viruses that threaten corporate and government systems. The "ILOVEYOU" virus alone caused over $10 billion in losses [2]. While monetary losses call attention to the importance of secure systems, there are other, more important, reasons for computer security. Critical systems that protect human life, such as avionics, weapons controls, reactor controls and life support systems, for example, require a higher level of secure and safe operation.

The DO-178B security standard is the specification for minimum safety in any airplane computer system [3]. Critical systems of this nature need a greater degree of safety and security than normal IT systems (like banking, for instance). Problematically, there exist many such standards for different application areas. What is safe for DO-178B may or may not be secure for other applications. This is why the Common Criteria Project [4, 5], jointly sponsored by several nations, defined seven Evaluation Assurance Levels (EAL), ranging from the lowest security and safety, EAL1, to the highest, EAL7. Standards and products are mapped to a specific EAL (or better, created to an EAL specification), enabling developers to satisfy security and safety policies by fulfilling different level-specific requirements based. EAL5 through EAL7 apply to critical systems.

Certification at the upper levels of security and safety calls for design and implementation rigor involving software engineering process control and mathematical formal methods that verify algorithm correctness. Using a set of formal proofs of an abstract model of the system, designers are expected to mathematically show the system will operate in the way that it was designed. EAL6 and EAL7, for example, require the use of formal methods, mathematical models and proofs. For large complex systems this is a most daunting task, but the separation of the overall system into certified, reusable components simplifies the verification process by enabling a proof that demonstrates the correct interaction of those previously proven components.

* This material is based on research sponsored by AFRL and DARPA under agreement number F30602-02-1-0178 and NSF under grant number DUE-0114016. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFLR, DARPA, or the U.S. Government.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **2006**	2. REPORT TYPE	3. DATES COVERED **00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE
Using SPARK-Ada to Model and Verify a MILS Message Router

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Idaho,Center for Secure and Dependable Systems,PO Box 441008,Moscow,ID,83844-1008

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR’S ACRONYM(S)

11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**8**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

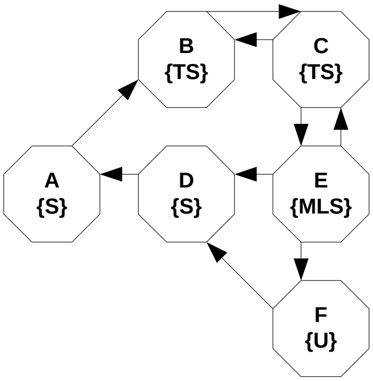


Figure 1. A security policy digraph for communication paths

2. Multiple independent levels of security

In the early 1980's, Rushby presented the concept of *separability* for the design and implementation of secure computer systems [6, 7]. The basic premise of his research was that computer systems that are completely separate from each other are by definition secure from one another. Rushby argued that this separation need not be physical, as long as at some level of abstraction we have separate devices, and a well-defined security policy between them. He proposed that a specialized real-time operating system, called the separation kernel, could provide an environment where multiple processes could run on the same processor and yet be totally separate from one another. Rushby showed that if these systems are connected through a well-defined communication policy, they can be proven secure because of the level of control exerted over that well-defined policy. Recently, Greve et al. created a formal security policy for a separation kernel [8]. That is, they defined a formal security policy that the separation kernel must enforce to ensure that processes running on the system are truly separate.

Original research by Alves-Foss [9, 10] led to the idea of a software/firmware architecture that would support Multiple Independent Levels of Security (MILS). The concept of Multiple Levels of Security (MLS) has traditionally been implemented by keeping multiple classifications of data (e.g., top-secret, secret, classified, unclassified) on physically separate devices. The MILS architecture uses a separation kernel to keep multiple data classifications on the same machine but completely independent of each other [11, 12]. Under the MILS implementation, there is no shared data whatsoever unless that data travels through the well-defined communication path. The separation kernel does this by creating *partitions* for each data classification type and keeping them separate by allowing no shared memory access, processor use, or hard disk storage. At the core of this architecture are the well-defined communication paths and a proven formal security policy that facilitates a proof of the whole system as an interaction of the certified components. The MILS architecture facilitates the composition of MLS components and single-level components via the well-defined and proven secure communication policy. Fig. 1 shows an example communications digraph to enforce a secure policy of communication paths (arrows) between Unclassified (U), Secret (S), Top Secret (TS), and MLS partitions. Once this policy is proven, the system is secure as long as individual components are secure. In other words, proving the communication policy is done once, and from then on we only have to prove that individual components adhere to the security policy for the system.

In the MILS architecture, secure communication is implemented in two subsystems: the MILS Message Router (MMR) and a collection of middleware *guards*. The guards enforce object-level method access rights and are not the focus of this paper (see [13] for implementation details of the MILS guards). As shown in Fig. 2, the MMR enforces the digraph of allowed typed communication, as defined in the security policy (Fig. 1). Communication classifications are enforced by the MMR as a mechanism to stratify messages (such as Unclassified/Classified/Secret/Top Secret). The MMR verifies that the sending partition is allowed to send a message to the receiving partition at the requested level. If the request is authorized by the security policy, then the MMR passes the message to the receiving partition. Thus, the MMR imposes a policy of "who can talk to whom and at what classification level." In previous work, it has been shown that once the MMR is proven to satisfy the security policy in conjunction with any service running in the partitions of which it governs communications, the entire system satisfies the security policy, provided that the services themselves satisfy the security policy individually [10]. This means that meeting EAL7 requirements on the MMR would make possible and

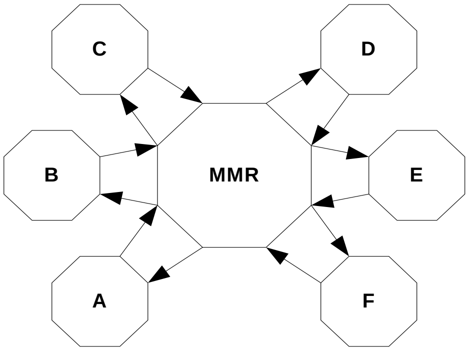


Figure 2. The MMR contains the security policy digraph from Figure 1

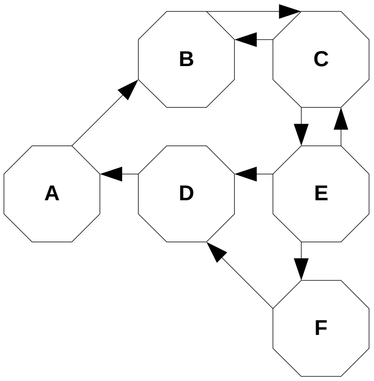


Figure 3. An MMR security policy digraph

greatly simplify the work of building and certifying MILS-based EAL7 systems for critical uses.

3. A SPARK-Ada MMR model

The main goal of the research described in this paper is to model the MMR and then verify the model's correct operation via formal methods. One of the principle objectives of the MILS initiative is the creation of EAL7-certified components. EAL7 requires the entire system to be mathematically proven using formal methods [4]. Specifically, we must create a high-level MMR design that is (a) proven in a formal modeling language, and (b) traceable to the code implementation [5]. SPARK-Ada is a formal methods tool that facilitates both criteria; specifically, it incorporates formal Hoare logic operations with executable Ada code, enabling the creation of verifiably correct executable models. Specifically, Hoare logic pre-, post-, and assert conditions are incorporated into the Ada source code (delineated with a "--#" prefix) and are used by the SPARK-Ada formal logic proof checkers during the verification process. The verification process and proof tools are described Section 4.

We used SPARK-Ada to implement a formal model of the MMR consistent with the MILS architecture targeted for an unspecified separation kernel. Our constraints were to build a MMR that: (1) used finite (not virtual) working memory, (2) had "zero-copy" message buffering modeling high performance implementations, (3) was consistent with EAL7 certification methodologies, (4) served as an example of how client processes should actually be implemented by developers, and (5) was easily updated for testing and removing assumptions in the model.

The model is composed of several "processes," each typically in a separate partition, but not necessarily so, along with the MMR component that resides in a partition by itself. Every process is allowed to run in an *a priori* static scheduling order that includes the execution of the MMR. The execution cycle continues indefinitely. If a process wishes to send a message to another process, it places the message in its message buffer and waits until the MMR is invoked. When the MMR processes the message, it checks to see if the message is authorized and valid and proceeds with the transmission by swapping ownership of that message buffer (hence, it's a zero-copy process). If the message is not valid or not permitted, it is expunged (zeroed) with no notice to either process. No process is allowed to read a section of memory that it does not own and each piece of memory is owned by exactly one process at a time.

Fig. 3 depicts the six processes from Fig. 1, denoted A through F, that we modeled in our SPARK-Ada MMR. The MMR is only aware of the authorized communication channels depicted in this Fig.. A comparison of this figure with Fig. 1 will indicate two interesting data flows. First, we can see that there is authorized flow from the MLS device E to C (top-secret), D (secret) and F (unclassified). This is a depiction of a trusted downgrader (such as a cryptographic device) that takes in high level information and processes it in a way to ensure no unauthorized information leakage. Second is the flow from F (unclassified) to D (secret), which depicts flow from an untrusted process (say a network device) into the system through a guard D. Although this is not an unauthorized flow, we have to be sure that there is no possible back channel, i.e., that D cannot block the sending of data from F. In addition, D will have to protect itself from attacks launched through F.

The MILS kernel and the MMR are not aware of the downgrading effort or the guarding, and *they do not care*. The purpose of the lower layer is to provide a set of security services to the higher levels. The kernel and the MMR restrict information flow so that the only points of concern are the flow out of E and from F to D. The MILS system will provide a verified separation kernel and MMR that can be reused by system developers. The developers will have to verify that the software running in partition E is multi-level secure, and that the single-level code in partition D is self-protecting and does not provide back channels. The other components can simply be verified as "black boxes" because the formally proven secure communication policy already verifies the security of the interactions. Thus, the level of proof rigor is significantly reduced when compared to traditional MLS solutions.

3.1. MMR model assumptions

Our model has a few assumptions to make verification simpler; while these assumptions could be removed, doing so would greatly increase the complexity of the verification proof. The main reason for the simplification of the model is to demonstrate that (a) the MMR does in fact operate correctly to facilitate secure messaging, and (b) SPARK-Ada is a viable proof-verification tool for executable models. The assumptions our SPARK-Ada MMR model are:

1. All pointers are unique. Each pointer to the working memory must be unique for our model; otherwise, communication could pass between the two or more owning processes of that pointer. This assumption aids our verification because the pointers are used as array indices in memory, and swaps between the memory spaces are simplified by knowing this information. We can easily verify this for the code-level implementations since we do not use dynamic pointers or aliasing.
2. Each process reads its incoming messages on each execution, and afterward, all of those memory spaces are zeroed. The zeroing of the receive buffers is needed because of the swaps on a successful send. If the receive buffer is not zeroed out, then the message in that buffer could be read by the sending process, creating an unchecked communication channel. This problem could also be resolved by zeroing out the receiving buffer before the swap occurs. Both approaches are possible solutions to closing the covert channel, but our model uses the first approach.
3. The security policy is static. The security policy for our system is based on a simple directed graph security policy (Fig. 3). If process A is allowed to send a message to process B, then there is a directed edge connecting process A to process B. Each edge is one-way, so for process B to send to process A there would need to be a separate directed edge from B to A. In our model the digraph is static for simplicity, but the policy can be set at design time or at execution time, as long as the policy has been proven correct prior to instantiation in the model.
4. No process is allowed to send a message to itself. Self-directed messaging could be allowed but is unnecessary and therefore removed for simplification.
5. Each process is allowed to send at most one message per activation. This assumption was used for simplification. Enabling multiple sends per activation can be implemented but requires checks to make sure that memory is not overrun, which complicates verification.

3.2. SPARK-Ada MMR implementation

Our MMR model is implemented in six Ada packages: *Lbl_t*, *Msg_t*, *Mem_t*, *MMR*, *System*, and *Main*. We use italics to differentiate the *MMR* packages from the generic MMR components discussed previously.

1. The *Lbl_t* (label) package consists of type definitions for labels that enumerate the complete list of the processes in the system, including the null process. The package also contains a constant defining the size of memory space. It is a very simple package that all other packages use as a naming scheme.
2. The *Msg_t* (message) package defines the *Msg* data type, which is the template for individual communications in the system. This *Msg* data type is a basic model of messages from one process to another. It consists of a process identifier to denote origin of the message, a process identifier to denote destination, and the message data (a simple integer in our model). Also defined in this package are default values for the *Msg* data type, a null origin and destination, and procedures to access and modify the *Msg* data type.
3. The *Mem_t* (memory) package defines a simple data type that creates another data type, *Mem_Row*, that is an array of *Msg*'s. The *Mem_Row* data type is what a process receives when it reads its messages.
4. The *MMR* package acts as a virtual memory manager in the system. The *MMR* package has two internal packages that it uses in its operation, a *Memory* package and a *Policy* package. The *MMR* keeps a set of pointers to *Memory* that it uses to distinguish which process owns which section of *Memory*. The *MMR* package has a procedure, *Route*, that scans each outgoing *Msg* to see if it is valid and allowed by the *Policy*. If it is valid and allowed, the *MMR* delivers the message by swapping pointers (in effect, ownership); otherwise it expunges the message from *Memory*.
   - *Memory* acts as a large memory space. It has two procedures, *Read* and *Write*, that the *MMR* package uses. *Read* takes a pointer *P* and an *Msg M* and writes the *Msg* that is at the location pointed to by the pointer *P* to the *Msg M*. *Write* takes a pointer *P* and an *Msg M* and writes the *Msg M* to the memory space pointed to by the pointer *P*.
   - *Policy* contains a static multidimensional array that serves as the communication policy for the *MMR*. It has one function, *Is_Allowed*, that takes in two process identifiers (e.g., A and B) and returns a binary value indicating whether or not messages are allowed (e.g., from A to B).
5. The *System* package implements the executable portion of the model. It executes each of the processes in order and acts as a go-between for the processes and the *MMR* package. It guarantees that a process cannot spoof its identity when it is sending a message. It does not guarantee that the message is addressed correctly but only that the message goes into the appropriate send buffer in the *MMR*. Once each process has executed, it then runs the *MMR_Route* procedure so that the messages are sent according to the specified communication policy.

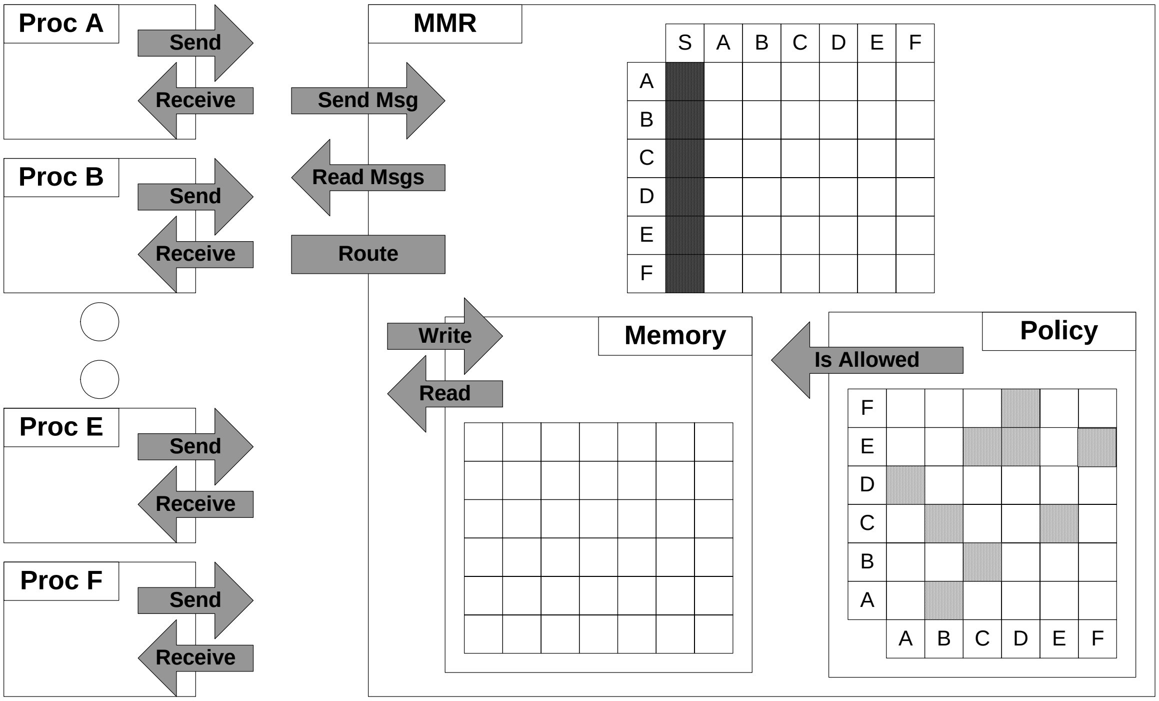


Figure 4. MMR interactions with processes A-F

```
--# inherit Lbl_t, Msg_t;
package Memory
  --# own Mem_Space : Mem_Space_T;
  --# initializes Mem_Space;
  is
  type Mem_Space_T is array
      ( Lbl_t.Pointer ) of Msg_t.Msg;
  procedure Write(
      M: in Msg_t.Msg;
      S: in Lbl_t.Pointer );
  --# global in out Mem_Space;
  --# derives Mem_Space from *,
  --#                         M,
  --#                         S;
  --# post Mem_Space = Mem_Space~[ S => M ];
  procedure Read(
      M: out Msg_t.Msg;
      S: in  Lbl_t.Pointer );
  --# global in Mem_Space;
  --# derives M from Mem_Space,
  --#                S;
  --# post M = Mem_Space( S );
end Memory;
```

(a) *Memory* package specification

```
package body Memory is
  Mem_Space: Mem_Space_T;
  procedure Write(
      M: in Msg_t.Msg;
      S: in Lbl_t.Pointer ) is
  begin
    Mem_Space( S ) := M;
  end Write;
  procedure Read(
      M: out Msg_t.Msg;
      S: in  Lbl_t.Pointer ) is
  begin
    M := Mem_Space( S );
  end Read;
begin
  Mem_Space := Mem_Space_T'(
   Lbl_t.Pointer => Msg_t.Def_Msg );
end Memory;
```

(b) *Memory* package body

Figure 5. SPARK-Ada code for *Memory* package

6. The *Main* package is just a wrapper program that executes the *System* package indefinitely.

Fig. 4 depicts the relationship between the *MMR* package, with its subordinate *Memory* and *Policy* packages, and the simulated processes denoted A through F. The *Policy* package contains an adjacency matrix representing the security policy diagraph shown in Fig. 3. The figure shows how the *MMR* is designed to interact in the model, where the partition in each row is allowed to talk to the partition in the column if the cell is shaded. The *MMR* has only three publicly available procedures: *Send_Msg*, *Read_Msgs*, and *Route*. There are two internal packages within the *MMR*: *Memory* that has two procedures, *Write* and *Read*, and *Policy* that has one

```
procedure Read_Msgs(
   P : in Lbl_t.Proc_ID;
   A : out Mem_t.Mem_Row )
 --# global in      Pointers;
 --#           in out Flags;
 --#           in out Memory.Mem_Space;
 --# derives Memory.Mem_Space,
 --#             A     from P,
 --#                          Memory.Mem_Space,
 --#                          Pointers &
 --#             Flags from *,
 --#                           P;
 --# pre
 --# for all X1 in Lbl_t.Proc_ID => (
 --#  for all Y1 in Lbl_t.Mem_Cols => (
 --#   for all X2 in Lbl_t.Proc_ID => (
 --#    for all Y2 in Lbl_t.Mem_Cols =>(
 --#     (
 --#      ( X1 /= X2
 --#        or
 --#        Y1 /= Y2 )
 --#      ->
 --#      ( Pointers(X1)(Y1) /=  Pointers(X2)(Y2))
 --#  )))));
```

```
 --# post
 --# ( for all Y in Lbl_t.Proc_ID => (
 --#    Flags(P)(Y) = FALSE
 --#    and
 --#    A(Y) =
 --#    Memory.Mem_Space~(Pointers(P)(Y))
 --#    and
 --#    Memory.Mem_Space(Pointers( P )(Y)) =
 --#     Msg_t.Def_Msg
 --#  ));
is
 Temp_Mem_Row : Mem_t.Mem_Row;
begin
 Fill_Mem_Row( P, Temp_Mem_Row );
 Zero_Flags( P );
 Zero_Mem_Row( P );
 A := Temp_Mem_Row;
end Read_Msgs;
```

Figure 6. SPARK-Ada code for *Read_Msgs* procedure body

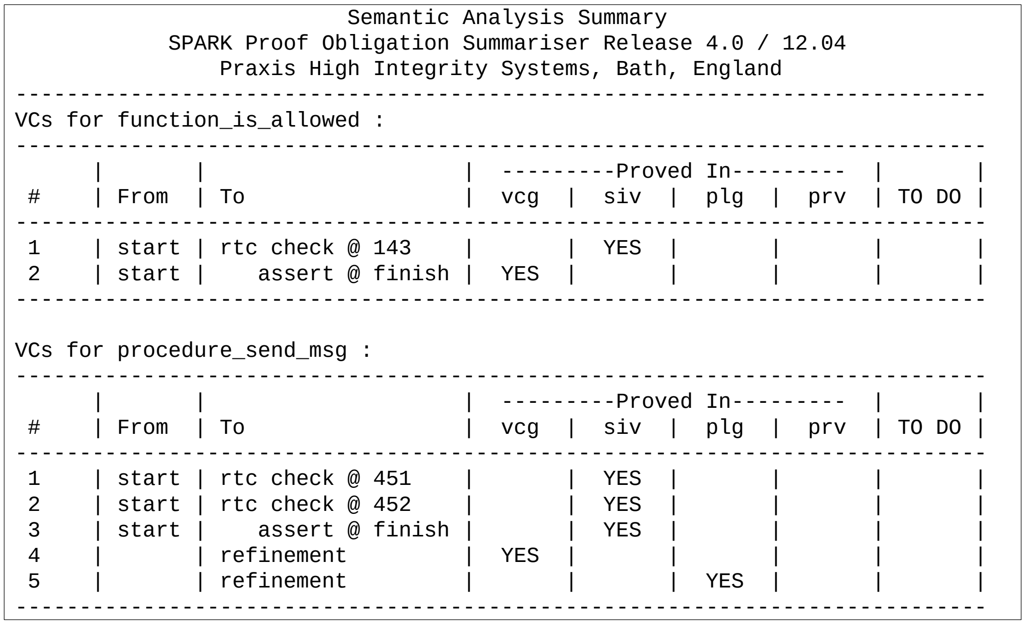
function, *Is_Allowed*. There is also an internal table within the *MMR* that keeps track of the pointers for each process and whether that memory location currently contains a message or not.

4. Model verification

Verification of the MMR model was done using the tools available with SPARK. This entailed first creating the code and proof annotations in the packages described above according to the SPARK-Ada language. There are two types of source files: <name>.ads files for package specification files and <name>.adb files for package body files. Fig. 5(a) shows the package specification for *Memory*. It states that the *Memory* package inherits from the *Lbl_t* and *Msg_t* packages; it contains its own global variable *Mem_Space*, an array made up of the *Msg* type from the *Msg_t* package of size *Pointer* from the *Lbl_t* package; and also contains two procedures, *Write* and *Read*, which manipulate the *Mem_Space* abstract variable. Write takes in two variables as input: *M* of the type *Msg* from the *Msg_t* package and *S* of the *Pointer* type from the *Lbl_t* package; it derives the *Mem_Space* from itself, *M* and *S*. When the procedure is finished, the post-condition states that *Mem_Space* will be the same, except the array location pointed to by *S* now contains the value of *M*. *Read* uses the same two inputs but derives the value of *M* from *Mem_Space* and *S*; when the procedure is finished, the post-condition states that *M* contains the value from *Mem_Space* pointed to by *S*. Fig. 5(b) is the package body for the *Memory* package. The body simply implements in code what the specifications describe from the package specification.

Fig. 6 illustrates the use of pre- and post-conditions in the *Read_Msgs* package body. The pre-condition states that all values in the *Pointer* array are different from one another – it is a uniqueness check. The post-condition states that for all of the *Proc_IDs* (the full list of process names) that the *Flags* variable in the row *P* have been set to *False*, the output array *A* has been filed with the old values from the *Memory* for the process *P*, and the spaces in *Memory* for the process *P* have now been zeroed out with the *Def_Msg* constant. In laymen's terms, the *Memory* has been read into the array *A* and then been zeroed out.

The first step in the verification process is to invoke the *SPARK Examiner* that does syntax checking and information flow checking. The *Examiner* generates verification condition files for each procedure and function within the packages (i.e., <name>.vcg). If the execution of the code matches the pre-conditions and post-conditions as described by the proof annotations, then the verification process continues. We then run the *SPARK Simplifier* which takes the <name>.vcg files and attempts to verify them using the Southampton Program Analysis Development Environment (*SPADE*) *Proof Checker*. It is usually capable of processing only the simpler verification conditions. Any verification conditions that the *Simplifie*r cannot handle are stored into <name>.siv files corresponding to the <name>.vcg files. At that point, any conditions that have not been verified by the Examiner or the Simplifier must then be manually verified by a review team or by using the



```
                    Semantic Analysis Summary
       SPARK Proof Obligation Summariser Release 4.0 / 12.04
             Praxis High Integrity Systems, Bath, England
-----------------------------------------------------------------------------
VCs for function_is_allowed :
-----------------------------------------------------------------------------
      |       |                    |  ---------Proved In---------  |        |
 #    | From  | To                 |  vcg  |  siv  |  plg  |  prv  | TO DO |
-----------------------------------------------------------------------------
 1    | start | rtc check @ 143    |       |  YES  |       |       |        |
 2    | start |    assert @ finish |  YES  |       |       |       |        |
-----------------------------------------------------------------------------

VCs for procedure_send_msg :
-----------------------------------------------------------------------------
      |       |                    |  ---------Proved In---------  |        |
 #    | From  | To                 |  vcg  |  siv  |  plg  |  prv  | TO DO |
-----------------------------------------------------------------------------
 1    | start | rtc check @ 451    |       |  YES  |       |       |        |
 2    | start | rtc check @ 452    |       |  YES  |       |       |        |
 3    | start |    assert @ finish |       |  YES  |       |       |        |
 4    |       | refinement         |  YES  |       |       |       |        |
 5    |       | refinement         |       |       |  YES  |       |        |
-----------------------------------------------------------------------------
```

Figure 7. POGS procedure and function verification summary examples

SPADE Proof Checker in manually guide mode. The proof checker program takes in <name>.vcg or <name>.siv files containing unverified conditions and outputs the manually guided verifications into <name>.plg files. In a similar manner, the review team can create verifications in a <name>.prv file containing verification conditions that have been manually verified by a review committee.

When all verification conditions have been proven, either automatically or manually, the Proof Obligation Summarizer (POGS) checks for the existence of all files (.vcg, .siv, .plg, and .prv) and produces a summary of all of the verification conditions that were generated. Fig. 7 shows the POGS summary for function *Is_Allowed* and procedure *Send_Msg*.

As shown in Fig. 7, the *Is_Allowed* function from the *Policy* package has been verified by the *Examiner* and the *Simplifier*. It has two verifications associated with it. The first is a run-time check to see that the variables are within their appropriate ranges. The second is an assertion made in the SPARK annotations that has been verified by the SPARK *Examiner*. As also shown in Fig. 7, the *Send_Msg* procedure was verified by the Examiner (vcg), Simplifier (siv), and SPADE proof checker (plg). Not all verifications are this simple, however. For example, the verification for the *Route* procedure (not shown) contains 29 verification conditions.

POGS also provides a summary of the entire proof. Fig. 8 shows the final proof summarization for the MMR model. Note that most proofs were handled by the *Simplifier* or guided to a successful formal proof through the *Proof Checker* – which is not unusual for formal methods tools. Also note that proof by committee or review team was not employed. We felt strongly that formal logic should prevail and not be overridden by informal proof methods.

5. Summary and conclusions

We have modeled a MILS Message Router capable of securely handling multiple levels of message classifications, assuming that processes handling each classification are properly contained within partitions segregated by a separation kernel. Although our model was simplified to enable a formal proof of correctness, it is easily extensible to a variety of MLS applications by modifying the digraph representation in the *Policy* adjacency matrix (or adjacency list).

The main goal of this project was to test the use of executable proof verification tools to design and create MILS components that could be certified at upper levels of the Common Criteria (e.g., EAL5 to EAL7). Exploring MMR design alternatives was also a motive behind the study; the solution presented here was the third iteration of our MMR design. What we learned from the process was enlightening but, in hindsight, not unexpected:

- The modeling process allows exploration of the solution space, albeit not as efficiently as prototyping.
- Generating formal proofs of correctness, even for simple operations contained in small components, is an arduous task that requires automated tools.

```
POGS Summary:

Total subprograms fully proved by examiner:             0
Total subprograms fully proved by simplifier:          11
Total subprograms fully proved by checker:              8
Total subprograms fully proved by review:               0
Total subprograms with at least one undischarged VC:    0
                                                    -----
Total subprograms for which VCs have been generated:   19

Total VCs by type:
                          -----------Proved By------------
                     Total  Examiner   Simp  Checker  Review  Undischarged
Assert or Post:         33         1     18       14       0             0
Precondition check:      4         0      4        0       0             0
Check statement:         0         0      0        0       0             0
Runtime check:          43         0     43        0       0             0
Refinement VCs:          6         1      0        5       0             0
Inheritance VCs:         0         0      0        0       0             0
=======================================================================
Totals:                 86         2     65       19       0             0
% Totals:                         2%    75%      22%      0%            0%
==================== End of Semantic Analysis Summary ====================
```

Figure 8. POGS summary for MMR proof

- Tying the executable code to the formal proof assertions (*a la* SPARK-Ada) enables a more rigorous proof model than can be attained through non-executable formal methods proof environments in which we have worked (e.g., ACL2).
- The lack of commercial grade on-call assistance on the use and nuances of the SPARK-Ada verification toolset was a significant hindrance to our task.

The MILS initiative and the development of a formally verified MMR embedded within a separation kernel machine are ongoing projects. We have also developed coded versions of the MMR in C/C++ and models of its operation in ACL2. Although we are uncertain if the SPARK-Ada model will eventually end up as the final definition for the MMR, we are convinced that modeling it in SPARK-Ada was a worthwhile task that better defined and refined our understanding of the MMR's processes.

6. References

[1] Global Information Grid Capstone Requirements Document, JROCM 134-01, August 30, 2001

[2] N. Nikitovic, "Building a Security Infrastructure - Maximizing Return of Security Investments" [Online document], 2003 Dec 1, [cited 2004 Mar 3], Available HTTP: http://www.dataprotection2003.info/speakers/Nemanja_Nikitovic/abstract.html

[3] RTCA, Software Considerations in Airborne Systems and Equipment Certification. RTCA DO178b, 1993.

[4] Common Criteria Project Sponsoring Organizations, Common Criteria for Information Technology Security Evaluation: Part 1: Introduction and General Model, ver 2.1, Washington, DC: RTCA, 1999.

[5] Common Criteria Project Sponsoring Organizations, Common Criteria for Information Technology Security Evaluation: Part 3: Security Assurance Requirements, ver 2.1, Washington, DC: RTCA, 1999.

[6] J.M. Rushby, "Proof of separability: A verification technique for a class of security kernels," In *Proc. International Symposium on Programming, Lecture Notes in Computer Science '82*, 1982, vol. 137, pp. 352-367.

[7] J.M. Rushby, "Design and verification of secure systems," In *Proc. ACM Symposium on Operating System Principles '81*, 1981, vol. 15, pp. 12-21.

[8] D. Greve, M. Wilding, and M. Vanfleet. A separation kernel formal security policy. *ACL2 Workshop 2003*. Boulder, CO, July. 2003.

[9] J. Alves-Foss. "Specifying Trusted Distributed System Components", *Journal of Computer and Information Science*, 2(1), 1996, pp. 238-257.

[10] J. Alves-Foss, "The architecture of secure systems," In *Proc. Hawai'i International Conference on System Sciences: Emerging Technologies Track* '98, 1998, pp. 317-307.

[11] J. Alves-Foss, W. S. Harrison, P. Oman and C. Taylor, "The MILS Architecture for High-Assurance Embedded Systems," *International Journal of Embedded Systems*, Vol. 2(1), January 2006, to appear.

[12] S. Harrison, N. Hanebutte, P. Oman, & J. Alves-Foss, "The MILS Architecture for a Secure Global Information Grid," *Crosstalk – The Journal of Defense Software Engineering*, 18(10), 2005, 20-24.

[13] N. Hanebutte, P. Oman, M. Loosbrock, A. Holland, S. Harrison, & J. Alves-Foss, "Software Mediators for Transparent Channel Control in Unbounded Environments," *Proceedings of the 6th IEEE Information Assurance Workshop*, (June 17-19, West Point, NY), IEEE Press, 2005.