Development of Security Software: A High Assurance Methodology

David Hardin[1], T. Douglas Hiratzka[1], D. Randolph Johnson[2], Lucas Wagner[1], and Michael Whalen[1]

[1] Rockwell Collins, Inc.
[2] National Security Agency

Abstract. This paper reports on a project to exercise, evaluate and enhance a methodology for developing high assurance software for an embedded system controller. In this approach, researchers at the National Security Agency capture system requirements precisely and unambiguously through functional specifications in Z. Rockwell Collins then implements these requirements using an integrated, model-based software development approach. The development effort is supported by a tool chain that provides automated code generation and support for formal verification. The specific system is a prototype high speed encryption system, although the controller could be adapted for use in a variety of critical systems in which very high assurance of correctness, reliability, and security or safety properties is essential.

1 Introduction

In order to study advanced high speed electronics technology, hardware research engineers at the National Security Agency started a project to build a prototype high speed encryption system. The system architecture they arrived at is shown in Fig. 1.

In this design, the Data Accelerators handle input/output functions, data formatting, and enforcement of some security policy rules. The encrypt core and decrypt core perform the actual encryption and decryption. These six subsystem blocks are in the high speed data paths. The control block manages the subsystem blocks but lies outside the high speed data path. An important consequence of this architecture is that the High Speed Crypto Controller (HSCC) does not need to be implemented using any exotic high speed electronics technology. The critical HSCC design goals are high reliability and achieving very high assurance of functional correctness and essential security properties. As a result, project responsibility for implementing the data accelerators and the crypto cores remained with the hardware engineering organization while responsibility for the HSCC was passed to the High Confidence Software and Systems (HCSS) Division.

Because of the general research mission of the HCSS division, the project now had two main goals. The first goal was to deliver a working controller. The second goal was to exercise, evaluate, and try to enhance a strong software development methodology. Since this is a security system, the methodology has to support a full range of development aspects from requirements through very rigorous evaluation by independent evaluators. And, in addition to being rigorous, it should also be cost-effective

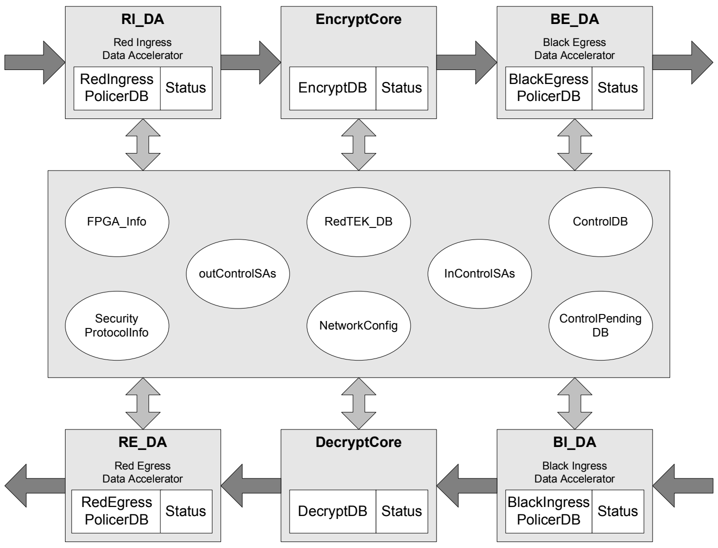


Fig. 1. High-Speed Crypto Functional Block Diagram

in time and money. Given these goals, and the limited resources of a research organization, we in the HCSS division needed an industrial partner. We found the ideal partner in Rockwell Collins.

One reason for teaming with Rockwell Collins was their capability with the AAMP7G microprocessor and high-assurance FPGA development. The AAMP7G supports strict time and space partitioning in hardware, and has received an NSA MILS certificate based in part on a formal proof of correctness of its separation kernel microcode, as specified by the EAL-7 level of the Common Criteria [5]. The formal verification of the AAMP7G partitioning system was conducted using the ACL2 theorem prover, and culminated in the proof of a theorem that the AAMP7G partitioning microcode implements a high-level security policy [4].

Perhaps more important than their hardware capabilities, Rockwell Collins has a very solid approach to software development. It features an integrated, model-based development toolchain with a focus on providing a domain-specific modeling environment that abstracts the implementation details, promotes architectural level design, and provides automated transformations between the problem domain formalisms and the target platform. The tools simplify code development and facilitate the application of automated formal analysis tools. In addition, the toolchain is capable of interfacing directly to a simulation environment, providing another level of assurance of design correctness.

For its part, HCSS researchers have experience in the Z specification language [10]. They have written Z functional specifications and design descriptions for several internal development projects. In the Tokeneer project [1,2], HCSS researchers played the role of customers and read and commented on draft specifications and designs in Z written by Praxis High Integrity Systems. In addition to experience in the requirements stage of development, HCSS people are familiar with the security evaluation work done by other NSA personnel.

The approach we chose for this project was for HCSS to take the lead in writing control software requirements in the form of functional specifications in Z. Rockwell Collins would take these specifications as input into their established development process. They would look for opportunities to strengthen the process, including the support for evaluation, or save time and money by taking advantage of the formal specifications. Everyone would see where the tool support was good or in need of improvement. This is a report on our experience. It is only incidentally about high-speed electronics technology or cryptography. Furthermore, because of limited space, we do not discuss in this paper data separation or any other properties of the hardware, for example, of the sort described by McComb and Wildman [6].

2 Z Specification Work

Over the last ten years, HCSS researchers have worked with other organizations using Z in support of a variety of development projects. We use the Z/EVES [9] support tool and have found it quite suitable for our needs. Based on our experience, we chose to use Z to write functional specifications on this high assurance controller project.

Most of the people involved in using Z over the years had some relevant technical background. An author of this paper is a trained mathematical logician who was involved in writing the ISO standard for Z and participated actively in all the projects. Most others had backgrounds in electrical engineering, mathematics or computer science, but began with no formal methods experience at all. Some had no technical background. Since there was only one occasion in which a number of people wanted to learn Z at about the same time, the usual pattern was individual learning. The newcomer started reading a Z textbook and asked more experienced people questions whenever something was not clear. They joined the small working group (two to four people) and quickly progressed from observers to active participants. When new features of the language came up, they were explained and discussed to make sure that everyone had the correct understanding. This informal on-the-job training approach has worked quite well. We have never had anyone decide that Z or formal methods were too difficult. Quite the opposite, in fact. Those who have expressed an opinion have said that learning the notation was not as hard as they thought it would be.

Our experience is that writing a good functional specification or system design document in Z is hard work. That is because writing a good functional specification or system design document is hard work. The clarity and organization obtained through using an expressive mathematically based notation such as Z makes the work easier, not harder.

On this project we tried to follow good habits acquired over the years. We think carefully about names and try to use clear helpful names and well chosen abbreviations. We have a house style for notational details such as capitalization. The important point is that both writers and readers of Z benefit greatly from a consistent style. The specific details of the style are not nearly as important as the fact that there is a set of standard conventions. In our finished documents, we adhered strictly to the principle that every Z paragraph was immediately preceded by an accurate English translation. No naked Z allowed.

The order in which we specify different aspects of the system matters. We usually start in the middle. That is, think first about the primary activity of the system when everything is working correctly. Define important data structures, introducing given types and other basic data types as needed. Define operations in the normal, successful case. Usually the inputs, outputs, preconditions, postconditions, and invariants are fairly clear. Then consider all the possible error cases. These are usually associated with the ways in which preconditions on the inputs fail or assumed invariants fail. Finally, combine all the possible cases, correct and faulty.

Since this project was to produce the controller for a crypto system, we had to describe, at a suitable level of abstraction, the main work of the system. On the outbound data path, this includes accepting, filtering and formatting data in the Red Ingress data accelerator, encryption in the encrypt core, and formatting and sending data out in the Black Egress data accelerator. The inbound data path is a mirror image with a decrypt core.

From this basic system analysis, we could see what control data structures had to be provided by the controller to properly manage the system. Basically, the system had to match each incoming piece of user data with the right cryptographic algorithm and key material. Secondary functions such as managing and updating key material were handled next. We had to define a system control protocol to convey system management messages back and forth between the controller and the other subsystems. After specifying this basic functionality of the system and the controller, we worked on the functional description of the subsystems.

This was when we realized that the Ingress data accelerators would also receive data such as new key material addressed to the system itself. This should not be encrypted or decrypted but instead passed to the controller. Similarly, the Egress data accelerators would have to send out data from the controller addressed to external recipients. After finishing all six subsystem specifications, we revisited the combined system level-controller specification to incorporate all the added functionality not previously considered. At this point we have complete and final Z specifications for the six subordinate subsystems and are doing the final revised specification for this version of the controller.

There were two aspects of the Z work on this project which were new to us. Because the encrypt and decrypt core subsystems were being developed by one team, the data accelerators by another team, and only the controller by our team with Rockwell Collins, we decided to write the specifications as separate documents. Actually, the top level system specification and the controller specification were so intertwined, they were in one document. The need to integrate a number of separate specifications was one we had not encountered before. The challenge of keeping all seven specification documents consistent was addressed in two ways. On the conceptual level, the fact that the same people (with one change) wrote all the documents and we were all conscious of the need for consistency was sufficient for us to keep each other honest. We could and did refer to previously written documents to check on details.

Where careful thinking was not sufficient was on the level of small details. Our goal was the possibility of merging (the Z content in) all seven documents into a unified system specification which was syntactically correct, type correct, and semantically correct. This goal could be met only if all the tiny details of spelling, capitalization, presence of underscores, etc. were coordinated to make names intended to have the same meaning in the different documents appear exactly the same and names intended to have different meanings differ in some detail. This is a task for which a computer is far more effective than a whole team of human beings. We took advantage of the capability of Z/EVES to export and import Z between its internal representation and LaTeX mark up. We devised an approach and wrote some Python scripts to automate the process and use Z/EVES to help with the required checking. The new tool support did not eliminate the need for careful thinking and attention to detail, but gave us much greater confidence that we have done things right.

The work described in this paper is all part of an ongoing research program. An early version of a system specification was written over a period of about eighteen months. It consisted of 185 pages of Z and English. Using that document, specifications for the six subordinate subsystems and a lower level communication protocol, totaling 290 pages, were written in about eight months. Finally, a revised controller specification estimated to contain 255 pages of Z and English is being written in about four months. In each case two or three people were involved.

3 Model Based Development

Model-based development (MBD) refers to the use of domain-specific, graphical modeling languages that can be executed and analyzed before the actual system is built. The use of such modeling languages allows the developers to create a model of the system, execute it on their desktop, analyze it with automated tools, and use it to automatically generate code and test cases.

The next section discusses the use of MBD in the HSCC software development process.

3.1 HSCC Software Development Using MBD

Software for the HSCC system was developed in two parts. System software (drivers and interrupt/trap handling) and portions of the high-level application code (message formatting and control processing) were implemented in hand-coded SPARK. This code includes information flow annotations to enable use of the Praxis toolchain and provide assurance of correctness.

Database transactions were designed and developed using the Rockwell Collins MBD tool-chain, Gryphon [12]. Simulink/Stateflow models were created for each database transaction. Each model was then tested via simulation in the Reactis tool to discover and correct obvious errors. When complete, the Gryphon framework is used to translate the model into the Prover tool. Gryphon supports several back-end formal analysis tools, including Prover, NuSMV and ACL2; for this project Prover was deemed to have the best combination of performance and automation. Prover is used to exhaustively verify each transaction preserves properties (derived from Z specifications) about the database it is acting upon. The proven correct Simulink model was then used to generate SPARK-compliant Ada95 for use on the target. Fig. 2 below illustrates the process flow.

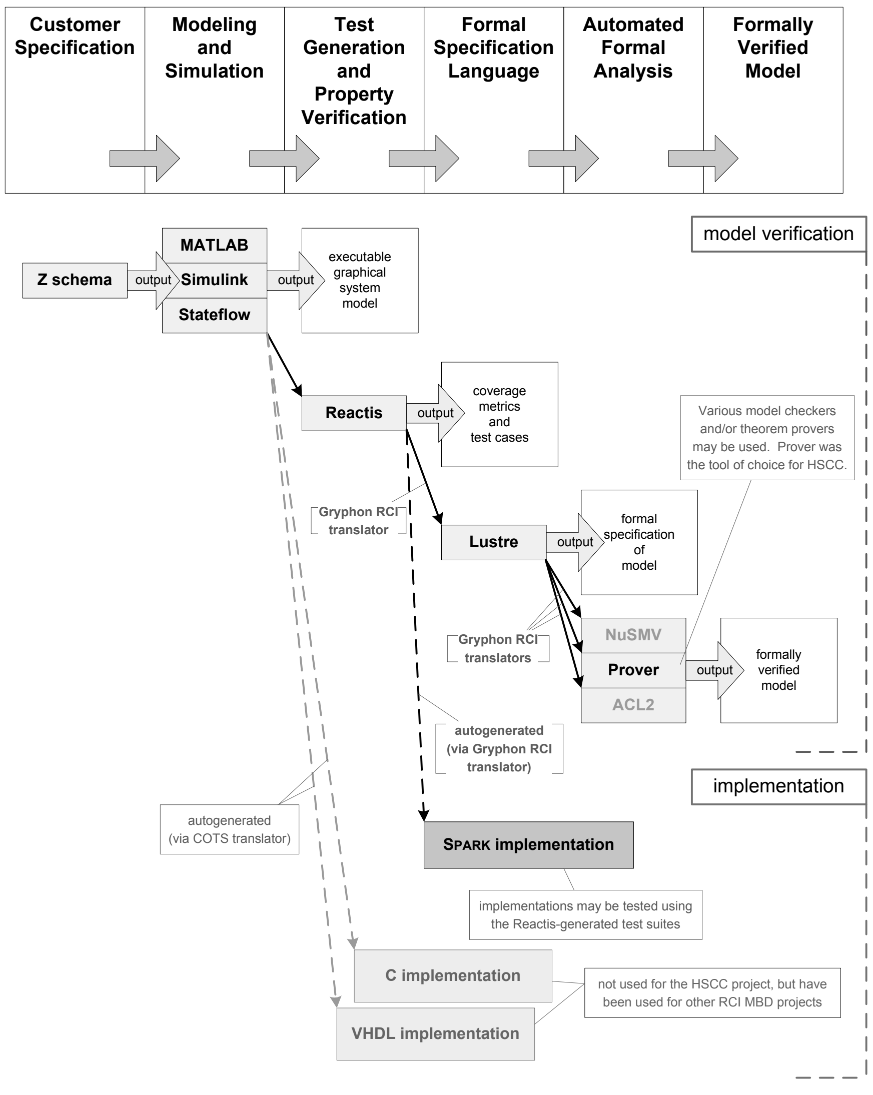

Customer Specification
Modeling and Simulation
Test Generation and Property Verification
Formal Specification Language
Automated Formal Analysis
Formally Verified Model
model verification
Z schema
output
MATLAB
Simulink
Stateflow
output
executable graphical system model
Reactis
output
coverage metrics and test cases
Various model checkers and/or theorem provers may be used. Prover was the tool of choice for HSCC.
Gryphon RCI translator
Lustre
output
formal specification of model
Gryphon RCI translators
NuSMV
Prover
ACL2
output
formally verified model
autogenerated (via Gryphon RCI translator)
implementation
autogenerated (via COTS translator)
SPARK implementation
implementations may be tested using the Reactis-generated test suites
C implementation
VHDL implementation
not used for the HSCC project, but have been used for other RCI MBD projects

Fig. 2. Model-Based Development Process Flow

The following sections briefly describe each tool involved in the HSCC software development process.

3.1.1 Simulink®, Stateflow®, MATLAB®

Simulink®, Stateflow®, and MATLAB® are products of The MathWorks, Inc. [11] Simulink is an interactive graphical environment for use in the design, simulation, implementation, and testing of dynamic systems. The environment provides a customizable set of block libraries from which the user assembles a system model by selecting and connecting blocks. Blocks may be hierarchically composed from predefined blocks. Simulink was chosen for development because it is the standard model-based development environment at Rockwell Collins and has extensive existing tool support, including support for formal analysis.

3.1.2 Reactis

Reactis® [8], a product of Reactive Systems, Inc., is an automated test generation tool that uses a Simulink/Stateflow model as input and auto-generates test code for the verification of the model. The generated test suites target specific levels of coverage, including state, condition, branch, boundary, and modified condition/decision coverage (MC/DC). Each test case in the generated test suite consists of a sequence of inputs to the model and the generated outputs from the model. Hence, the test suites may be used in testing of the implementation for behavioral conformance to the model, as well as for model testing and debugging.

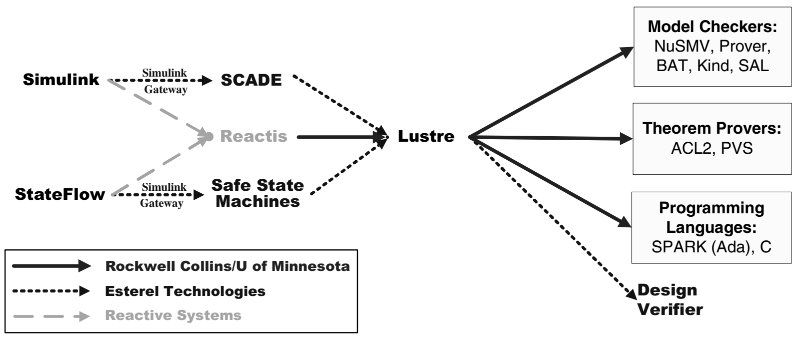

Simulink
Simulink Gateway
SCADE
Reactis
StateFlow
Simulink Gateway
Safe State Machines
Lustre
Model Checkers: NuSMV, Prover, BAT, Kind, SAL
Theorem Provers: ACL2, PVS
Programming Languages: SPARK (Ada), C
Design Verifier
Rockwell Collins/U of Minnesota
Esterel Technologies
Reactive Systems

Fig. 3. Rockwell Collins Gryphon Translator Framework

3.1.3 Gryphon

Gryphon [12] refers to the Rockwell Collins tool suite that automatically translates from two popular commercial modeling languages, Simulink/Stateflow® and SCADE™ [3], into several back-end analysis tools, including model-checkers and theorem provers Gryphon also supports code generation into Spark/Ada and C. An overview of the Gryphon framework is shown in Fig. 3. Gryphon uses the Lustre formal specification language (the kernel language of SCADE) as its internal representation. Gryphon has been in development at Rockwell Collins for the last 6 years, and has been used on several significant formal verification efforts involving Simulink models.

3.1.4 Prover

Prover [7] is a best-of-breed commercial model checking tool for analysis of the behavior of software and hardware models. Prover can analyze both finite state models and infinite-state models, that is, models with unbounded integers and real numbers, through the use of integrated decision procedures for real and integer arithmetic.

Prover supports several proof strategies that offer high performance for a number of different analysis tasks including functional verification, test-case generation, and bounded model checking (exhaustive verification to a certain maximum number of execution steps).

3.1.5 Custom Code Generation

By leveraging its existing Gryphon translator framework, Rockwell Collins designed and implemented a tool-chain capable of automatically generating SPARK-compliant Ada95 source code from Simulink/Stateflow models.

3.2 Transaction Development

Simulink/Stateflow models are used as the common starting point for both the implementation and analysis. Each model corresponds to a single database transaction. Model inputs correspond to SPARK procedure “in” parameters and outputs correspond to “out” parameters. Note the database object used by each transaction model may appear as both an input and an output if the database is modified by the transaction. In this case, the database object access appears as an “in out” parameter in the generated code. For each database, one model must be created to initialize the data object, as well as models to perform necessary transactions (add, delete, lookup) on the database. Additional models are required for the formal analysis to model invariants on the database object. This topic will be covered in more detail in subsequent sections.

The screenshot in Fig. 4 below shows a sample Simulink model which contains the “Dest_Encr_Addr_Found” lookup function performed on the Routing Table. This function performs a lookup in the Routing Table to determine if the specified destination encryptor address is found in the table. The inputs (at left) are the routing table (“Rt_Tbl”) and the destination encryptor address (“Dest_Encr_Addr”) for which to search. The output (at right) is the boolean value (“Found”) resulting from the search. The rectangular block in the center is a Simulink subsystem block which implements the database lookup.

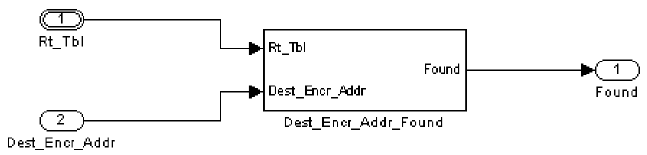


Fig. 4. Destination Encryptor Address Found model

Typically, a transaction model will contain a Stateflow chart inside the Simulink model. Stateflow is well-suited to the implementation of the database operations. The screenshot in Fig. 5 below shows the contents of the of the Simulink subsystem block. The heavy vertical bar at the left is a Simulink Bus Selector. Simulink Bus Objects are roughly analogous to a record in Ada or SPARK. (The Reactis tool does not allow Bus Objects as inputs to Stateflow charts, so a Bus Selector is used to separate the

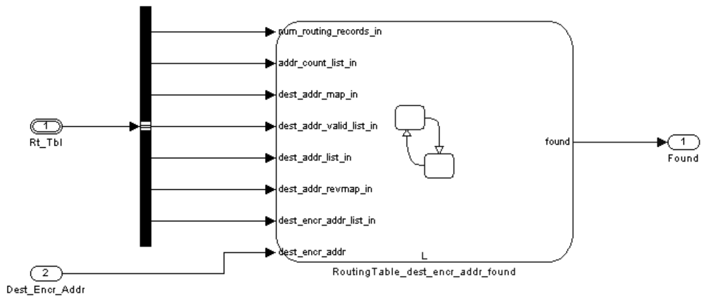
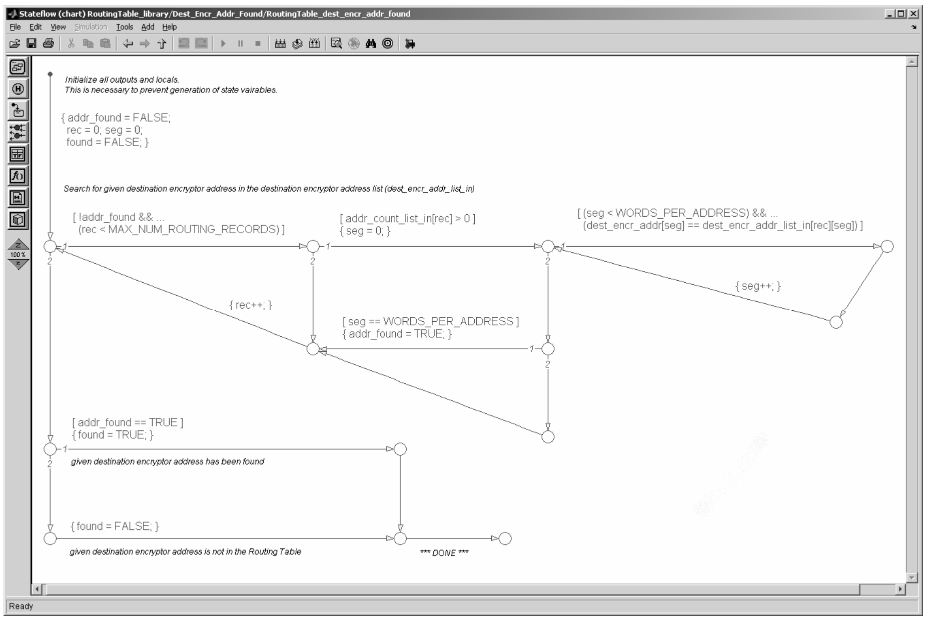


Fig. 5. Sample Simulink Subsystem


Stateflow (chart) RoutingTable_library/Dest_Encr_Addr_Found/RoutingTable_dest_encr_addr_found
File Edit View Simulation Tools Add Help
100%
Ready

Initialize all outputs and locals.
This is necessary to prevent generation of state vairables.

```
{ addr_found = FALSE;
  rec = 0; seg = 0;
  found = FALSE; }
```

Search for given destination encryptor address in the destination encryptor address list (dest_encr_addr_list_in)

```
[ !addr_found && ...
  (rec < MAX_NUM_ROUTING_RECORDS) ]

[ addr_count_list_in[rec] > 0 ]
{ seg = 0; }

[ (seg < WORDS_PER_ADDRESS) && ...
  (dest_encr_addr[seg] == dest_encr_addr_list_in[rec][seg]) ]

{ seg++; }

{ rec++; }

[ seg == WORDS_PER_ADDRESS ]
{ addr_found = TRUE; }

[ addr_found == TRUE ]
{ found = TRUE; }
```

given destination encryptor address has been found

```
{ found = FALSE; }
```

given destination encryptor address is not in the Routing Table

**** DONE ****

Fig. 6. Sample Stateflow Chart.

component parts of the Bus Object into separate inputs to the Stateflow chart.) The large rounded rectangle block is a Stateflow chart.

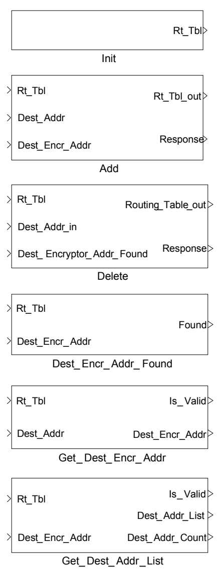
Fig. 6 shows the expanded Stateflow chart of the Dest_Encr_Addr_Found model, which implements the database search. Statements are attached to various transitions. Those in curly brackets ("{ }") represent assignment statements. Statements in square brackets ("[ ]") represent conditional expressions. Conditional transitions are executed only if the expression evaluates to "true".

As stated earlier, a model must be built for each transaction in each database. In the case of the Routing Table, these are:

- Init – procedure to initialize the routing table data structure (called upon reset)
- Add – database transaction to add a routing record to the routing table

- Delete – database transaction to remove a routing record from the routing table
- Dest_Encr_Addr_Found – database query to determine existence of destination encryptor address
- Get_Dest_Addr_List – database lookup to return list of addresses mapped to an encryptor address
- Get_Dest_Encr_Addr – database lookup to return encryptor address mapped to a destination address

Fig. 7 below show the interfaces provided by each model, alongside the generated SPARK procedure signature.



```
procedure Init (
     Rt_Tbl :out    Routing_Table_Types.Routing_Table_Type)
;

procedure Add (
     Rt_Tbl :in out    Routing_Table_Types.Routing_Table_Type ;
      Dest_Addr :in    Routing_Table_Types.Address_Type ;
       Dest_Encryptor_Addr in Rout  ing_Table_Types.Address_Type ;
      Response :out  Unsigned.Word)
;

procedure Delete (
     Rt_Tbl :in out    Routing_Table_Types.Routing_Table_Type ;
      Dest_Addr :in    Routing_Table_Types.Address_Type ;
       Dest_Encryptor_Addr_Found in Boolean ;
      Response :out  Unsigned.Word)
;

procedure  Dest_Encr_Addr_Found (
     Rt_Tbl :in    Routing_Table_Types.Routing_Table_Type ;
   Des  t_Encr_Addr :in    Routing_Table_Types.Address_Type ;
     Found :out  Boolean )
;
procedure  Get_Dest_Encr_Addr (
     Rt_Tbl :in   Routing_Table_Types.Routing_Table_Type ;
     Dest_Addr :in    Routing_Table_Types.Address_Type ;
     Is_Valid :out  Boolean ;
      Dest_Encr_Addr out    Routing_Table_Types.Address_Type )
;
procedure  Get_Dest_Addr_List (
        Rt_Tbl :  in Routing_Table_Types.Routing_Table_Type ;
    Dest_Addr :  in R outing_Table_Types.Address_Type ;
     Is_Valid :  out Boolean ;
         Dest_Addr_List :  out Routing_Table_Types.Dest_Addr_List_Type ;
      Dest_Addr_Count :  out Unsigned.Word)
;
```

Fig. 7. Transaction Models and associated SPARK Signatures

3.3 Invariant Modeling

To perform formal analysis on the transaction models, it is first necessary to model any invariants on the data structures. These invariants are taken directly from the Z specification. As an example, the following invariants appear in the Z specification for the Routing Table:

$$\forall\ kda: knownDestAddresses;\ rr_1,\ rr_2: routingRecords$$
$$\quad \bullet\ kda \in rr_1.destinationAddresses \wedge kda \in rr_2.destinationAddresses$$
$$\quad \Rightarrow rr_1 = rr_2$$
$$\forall\ rr_1,\ rr_2: routingRecords$$
$$\quad \bullet\ rr_1.destinationEncryptorAddress = rr_2.destinationEncryptorAddress$$
$$\quad \Rightarrow rr_1 = rr_2$$

Fig. 8. Z Specification Invariant Sample

This specification indicates that no duplicate destination addresses or duplicate encryptor addresses may appear in the Routing Table. These invariants are checked by the "no_dups" model (shown in Fig. 9 below). Given a routing table input ("Rt_Tbl"), the model checks that no duplicate destination encryptor addresses exist in the data structure and sets the output booleans accordingly. Note that the number of boolean outputs in the model is determined by the internal representation of the routing table data structure, and that the condition in which all four boolean outputs are "false" indicates that both invariants hold.

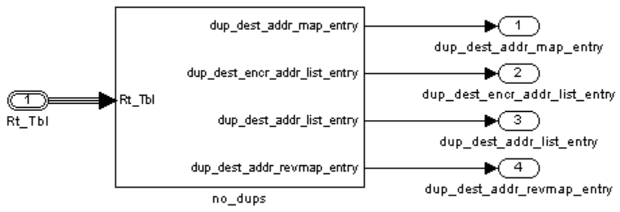


Fig. 9. Sample Invariant Model

3.4 Formal Verification

This section illustrates the approach to performing formal analysis on a database transaction. The necessary models include both the transaction model and any invariant models associated with the relevant database(s). In the formal analysis we are establishing two kinds of properties: 1.) data invariants over the databases (as defined by the Z schemas defining each database) and 2.) transaction requirements that ensure that the operation performed by a model matches the Z schema for that transaction.

3.4.1 Proof Strategy

The proof strategy employed for the data invariants is induction over the sequence of transactions that are performed. We first verify that the Simulink models responsible for initializing each database establish the data invariant for that database. This step provides the basis for our induction. We then prove every transaction that modifies a database maintains the invariant for that database. More concretely, on the "init" models, we use the model checker to determine whether or not the data invariants hold on the model outputs. For the other transactions, the proof strategy is to assume the invariants in the input "pre" database (prior to performing the transaction), and then use the model checker to determine whether the invariants hold in the output "post" database (resulting from performing the transaction).

We prove all the invariants required by the Z specification and also additional invariants involving implementation details related to realizing the Z databases in Simulink/Stateflow. For example, a linked-list representation is used for many of the finite sets described in the Z document. In this case, additional invariants establish that the linked list is a faithful representation of the finite set.

The transaction requirements for each operation are specified as additional properties that must hold on the "post" database. For example, when deleting an element, these properties ensure that the element in question has been removed from the database.

3.4.2 Stateless Transactions

Transactions on HSCC databases (and transactional database systems in general) are designed to perform a complete unit of work, or fail without making any modifications to the database. This indicates that each transaction can be developed without the use of internal state. It is this property of transactional databases that greatly simplifies the model-checking effort, because proofs can be obtained using just bounded model-checking to depth 1, instead of (for example) automated k-inductive reasoning. Taking advantage of this property made the most difficult proof obligations tractable, and reduced the time of simpler proofs from hours to minutes.

3.4.3 Routing Table Example

As an example, we present the proof system for the DeleteRoutingEntry transaction, which deletes an entry from a routing record in the routing table. The Z schema which specifies the RoutingTable is shown below in Fig. 10. The specification describes the contents of the database, the maximum size of the database, and further constraints on the data (no duplicate addresses, as discussed previously).

RoutingTable

$routingRecords: \mathbb{F}\ RoutingRecord$
$knownDestAddresses: \mathbb{F}\ NETWORK\_ADDRESS$
$maxNumRoutingRecords: \mathbb{N}$

$\#\, routingRecords \leqslant maxNumRoutingRecords$
$knownDestAddresses = \bigcup \{\ rt: routingRecords \bullet rt.destinationAddresses\ \}$
$\forall\ kda: knownDestAddresses;\ rr_1, rr_2: routingRecords$
$\quad \bullet\ kda \in rr_1.destinationAddresses \wedge kda \in rr_2.destinationAddresses$
$\quad\quad \Rightarrow rr_1 = rr_2$
$\forall\ rr_1, rr_2: routingRecords$
$\quad \bullet\ rr_1.destinationEncryptorAddress = rr_2.destinationEncryptorAddress$
$\quad\quad \Rightarrow rr_1 = rr_2$

Fig. 10. Z specification of the Routing Table database

For a complete understanding of the proof strategy, it is necessary to present the underlying representation of the Routing Table, shown in Fig. 11.

The number of routing records currently in the database is represented as "num_routing_records". The "addr_count_list" array maintains the number of valid destination addresses, on a "per routing record" basis. The "dest_addr_map" 2-D array holds pointers to the destination addresses. Each row corresponds to a routing record. The "dest_encr_addr_list" contains the destination encryptor addresses (one per routing record). Since an "address" may be a IPv4 or IPv6, this array is sized to contain eight 16-bit values. Taken collectively, a slice across the three arrays (addr_count_list, dest_addr_map, and dest_encr_addr_list), as shown by the dashed arrows, represents a single routing record, which comprises one or more destination addresses mapped to a single destination address. Since 3-D arrays are not supported by the toolchain (specifically, not by Reactis), a mapping array is used. The dest_addr_map array contains

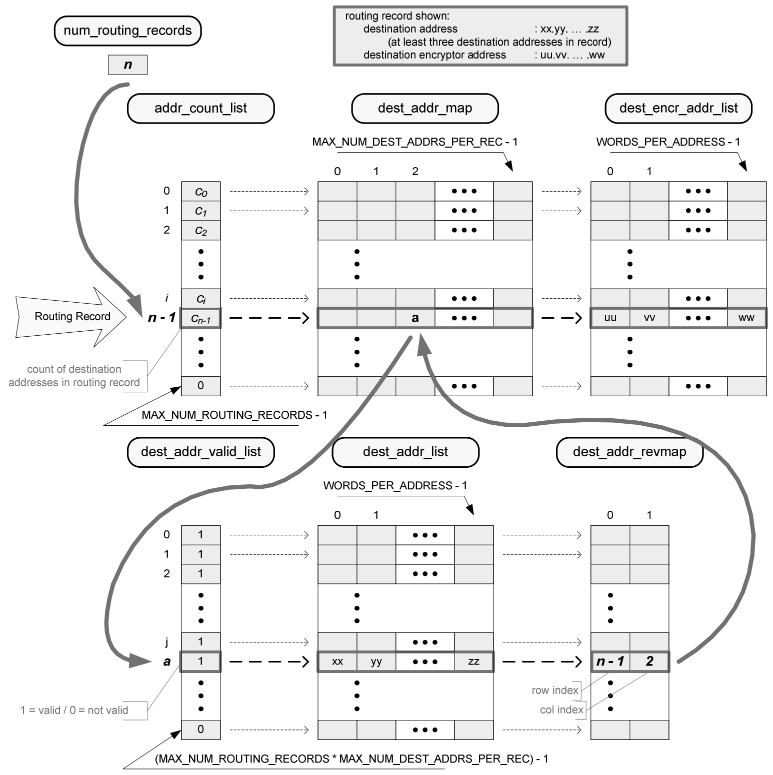

routing record shown:
destination address : xx.yy. … .zz
(at least three destination addresses in record)
destination encryptor address : uu.vv. … .ww
num_routing_records
n
addr_count_list
dest_addr_map
dest_encr_addr_list
MAX_NUM_DEST_ADDRS_PER_REC - 1
WORDS_PER_ADDRESS - 1
Routing Record
n - 1
a
uu
vv
ww
count of destination addresses in routing record
MAX_NUM_ROUTING_RECORDS - 1
dest_addr_valid_list
dest_addr_list
dest_addr_revmap
WORDS_PER_ADDRESS - 1
xx
yy
zz
n - 1
2
row index
col index
1 = valid / 0 = not valid
(MAX_NUM_ROUTING_RECORDS * MAX_NUM_DEST_ADDRS_PER_REC) - 1

Fig. 11. Routing Table Data Structure Implementation

not the actual destination address, but rather an index into another array which holds all of the known destination addresses. Together, the "dest_addr_valid_list" array and the "dest_addr_list" array form the list of known destination addresses. The dest_addr_valid_list is an array of booleans, indicating whether the corresponding row of the dest_addr_list array contains a known address. (Note that all-zeros is accommodated as a valid address, so a valid flag must accompany the list of addresses.) As with the dest_encr_addr_list array, the dest_addr_list is sized to hold either an IPv4 or IPv6 address (eight 16-bit values). In order to support reverse lookups (destination encryptor address → destination address), we employ a mapping array which contains the row and column indices (into the dest_addr_map array) for a given destination address. Constant array bounds are indicated in the figure. The maximum number of routing records in the table is defined as MAX_NUM_ROUTING_RECORDS. The maximum number of destination addresses mapped to a destination encryptor address is MAX_NUM_DEST_ADDRS_PER_REC. The number of values required to represent the largest address (IPv6, in the current implementation) is WORDS_PER_ ADDRESS.

The transaction to be model-checked in this example will be the DeleteRoutingEntry transaction. Its Z specification is shown in Fig. 12.

DeleteRoutingEntry

$\Delta RoutingTable$
$ControlDB$
$destinationAddress?: NETWORK\_ADDRESS$
$modRoutingRecord, delRoutingRecord: RoutingRecord$
$response!: Response$

$destinationAddress? \in knownDestAddresses$
$delRoutingRecord$
$\quad = (\mu\ rr: routingRecords \mid destinationAddress? \in rr.destinationAddresses)$
$\# delRoutingRecord.destinationAddresses \geqslant 2$
$\Rightarrow modRoutingRecord.destinationAddresses$
$\quad = delRoutingRecord.destinationAddresses \setminus \{destinationAddress?\}$
$\quad \wedge modRoutingRecord.destinationEncryptorAddress$
$\quad\quad = delRoutingRecord.destinationEncryptorAddress$
$\quad \wedge routingRecords' = routingRecords \setminus \{delRoutingRecord\} \cup \{modRoutingRecord\}$
$\# delRoutingRecord.destinationAddresses = 1$
$\wedge delRoutingRecord.destinationEncryptorAddress$
$\quad \notin \{\ cr: controlRecords \bullet cr.destinationEncryptorAddress\ \}$
$\Rightarrow routingRecords' = routingRecords \setminus \{delRoutingRecord\}$
$response! = success$

Fig. 12. Z Specification of DeleteRoutingEntry

From the specification of the DeleteRoutingEntry function, the inputs are the routing table object, the destination address (the address to be deleted), and a boolean ("Dest_Encryptor_Addr_Found") which indicates whether the destination encryptor address (mapped to the destination address) exists in the Control database. (This lookup is performed by the caller of the DeleteRoutingEntry function.) Although the Z specification is for only the successful case (error cases are handled in separate schemas), the model must properly trap and handle all error cases. The RoutingTable Delete model is shown in Fig. 13 below. Outputs are the updated routing table object ("Routing_Table_out") and the response code ("Response"). Note that this is the same Simulink model (Delete.mdl) used for code generation. The model for the invariant discussed previously is also shown in Fig. 13. This model checks for any duplicate entries in the routing table.

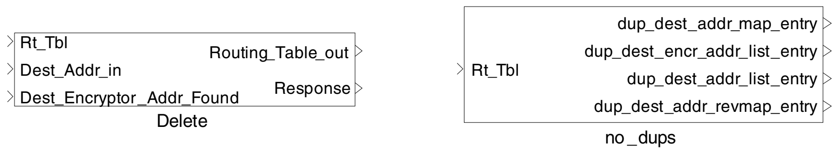


Fig. 13. DeleteRoutingEntry Model and "no_dups" Invariant Model

Given the underlying representation of the routing table, additional invariants must be constructed to complete the infrastructure for model verification. One such model, “is_consistent” determines whether the database is in a valid, consistent state. For example, checks are performed to ensure that each “count” value (e.g. num_routing_records, addr_count_list array elements) is within valid bounds. Mapping array elements are checked to ensure they point to valid entries in their target arrays. Finally, all unused array elements are checked to ensure they have been cleared (set to zero) by any previous Delete operations. (The data structure is initialized to all zeros upon reset by calling an “init” function.) The “is_consistent” model is shown in Fig. 14 below.

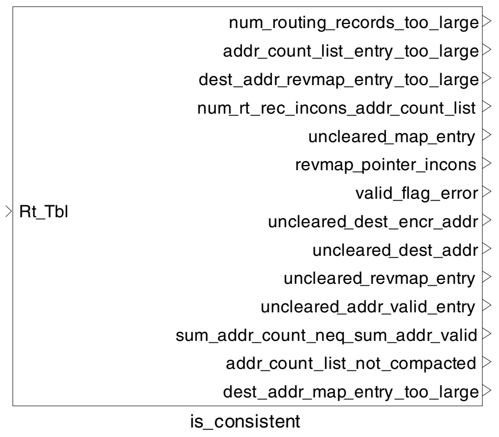

Rt_Tbl
num_routing_records_too_large
addr_count_list_entry_too_large
dest_addr_revmap_entry_too_large
num_rt_rec_incons_addr_count_list
uncleared_map_entry
revmap_pointer_incons
valid_flag_error
uncleared_dest_encr_addr
uncleared_dest_addr
uncleared_revmap_entry
uncleared_addr_valid_entry
sum_addr_count_neq_sum_addr_valid
addr_count_list_not_compacted
dest_addr_map_entry_too_large
is_consistent

Fig. 14. Routing Table "is_consistent" Invariant Model

Since the Delete operation removes at most one destination address, the resulting routing table must be a subset of the original table. A transaction requirement model has been constructed to check that the “post” routing table object is a subset of the “pre” object. The t2_contains_tl model shown in Fig. 15 checks if Rt_Tbl_2 “contains” Rt_Tbl_1, and sets the output boolean (“t2_contains_t1”) accordingly. Additionally, the model returns the difference in counts (of destination addresses) between the two input routing table objects.

Finally, the resulting routing table must not contain the destination address which was to be deleted. The “address_deleted” requirement model in Fig. 15 checks for the existence of the specified address and returns a boolean result.

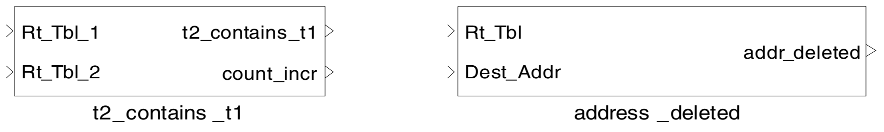


Fig. 15. Routing Table "t2_contains_t1" and "address_deleted" Invariant models

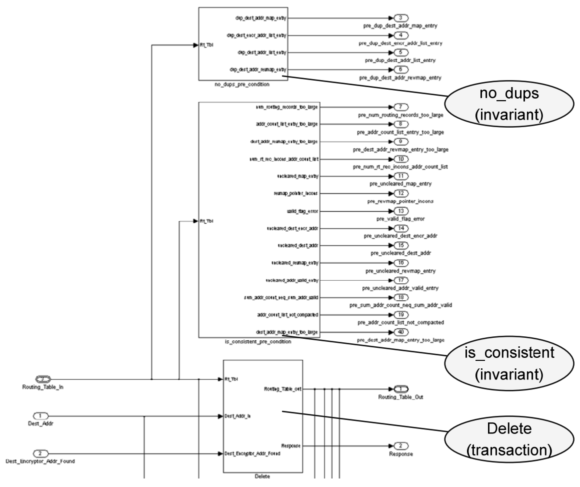


Fig. 16. Sample Formal Analysis Model (part 1)

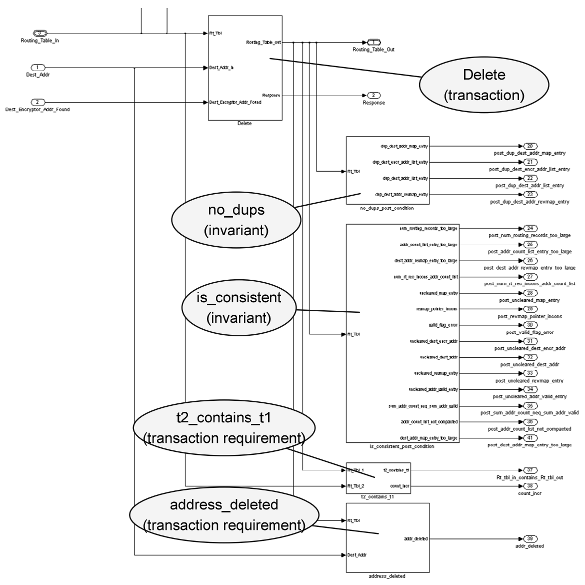


Fig. 17. Sample Formal Analysis Model (part 2)

Given the transaction model, the collection of invariant models, and the transaction requirement models, the approach is to assume the data invariants hold prior to performing the Delete operation, and then verify that the invariants and transaction requirement models hold following the Delete operation. The proof model is shown in Fig. 16 and Fig. 17. Due to the size of the model, the screenshot has been split into two parts for better readability.

Fig. 16 shows the preconditions portion of the proof model. This contains the delete transaction model along with the invariants which are assumed to be true for the "pre" table, namely the "no_dups" (check for duplicated addresses) and "is_consistent" (check for a valid, consistent state).

Fig. 17 shows the postconditions portion of the proof model. This includes the delete model and the invariants which are to be verified against the resulting routing table. The invariants are "no_dups", "is_consistent", "t2_contains_t1", and "address_deleted". If all these invariants are true (i.e. the post-conditions are true), then the correctness of the delete operation is verified.

The model-hierarchy used to perform the proof analysis is translated to the Prover imperative language using the Gryphon translation framework and discharged by the Prover model checker.

3.4.4 Formal Verification Results Summary

The formal verification effort for the project as a whole resulted in the proof of some 840 properties for the HSCC databases, of which 140 were written by the verification team, and the remainder (mainly well-formedness checks) automatically generated by the Gryphon framework. Verification required less than 5% of total project effort over the course of seven calendar months.

3.4.5 Caveat

Due to the demands of a fully-functioning cryptographic system, it will be necessary to populate databases with hundreds, if not thousands of elements. The extreme size of these databases makes model-checking the fully-sized system impossible, due to state-space explosion. Instead, in this effort, a small representative sized database (n=3) was verified in formal analysis. The design of each transaction model allows the user to configure the size of a database to be configured at compile time with a global variable. The claim can be made this is sufficient, due to the design of the database transaction models, however no attempt to prove this claim have been made. Future work at Rockwell Collins will be focused on proving this claim, and it will be discussed further in the Future Work section of this document.

3.5 Code Generation

Code generation is performed after a transaction is proven to satisfy all of its invariant properties. This is accomplished through the use of a proprietary translation tool that leverages the existing Gryphon framework to generate SPARK-compliant Ada95 source code for use on the AAMP7G.

All of the transactions are compiled into single Ada95 package for use by the system programmer. The procedures in the package declaration are shown in Fig. 7.

4 Future Work

Whenever all the subsystems are available, NSA researchers will integrate them into a demonstration system. We also plan to organize the assurance evidence produced in this phase and ask our evaluation organization for their assessment. We won't get a full evaluation of our prototype system, but expect to get expert advice on possible areas for improvement to our assurance case. Depending on available resources, we plan to add functionality missing from the current system. For example, this version can use connections to peer system established and entered by hand. A full system must be capable of automatically establishing and tearing down connections as needed. For another example, this version manages cryptographic key material internally, including limited key update capability. It does not have the capability of requesting and receiving new key material from an external source. This capability would be essential in a real system.

Rockwell Collins will continue this research initiative by investigating two key issues.

The selection of Simulink/Reactis tool-chain caused considerable difficulty due to the lack of an orthogonal type system: Simulink currently does not support arrays of record-types (called *buses* in Simulink). Reactis further restricts the type system by only supporting arrays of one or two dimensions. Consequently, databases could not be designed in an intuitive way, simply to satisfy the Simulink type system. This factor drove up transaction development time, increased the number and complexity of proof obligations, and hampered the development of the proprietary code generator. SCADE, by Esterel Technologies, is a model-development environment similar to Simulink, but features an orthogonal type system, and fits directly into the Gryphon toolchain off the shelf. A comparison of the transaction development process with both Simulink/Reactis and SCADE tool-chains would reveal whether or not the use of SCADE results in significant reductions in development time and cost.

Each transaction was checked to satisfy invariants over a finite-sized database. This partial proof provides a high degree of assurance, however it would be ideal to obtain a proof that each transaction satisfies invariants over databases of arbitrary length. One major obstacle to obtaining this general proof is the use of various FOR loops within each transaction to traverse the elements of a database. Because these loops are typically limited by constants defined at compile time, it would be necessary to prove these loops preserve the transaction invariant over an arbitrary number of iterations. Software model-checkers, such as SLAM by Microsoft, handle this problem with some degree of success. Research would focus on techniques employed by software model-checkers to overcome this problem.

5 Conclusion

Our experiences developing the HSCC system have shown that the methodology described in this paper is a viable process for the development of high-assurance software for use in cryptographic systems.

NSA-provided specifications written in the Z formal notation proved to be superior to those written in English-language in producing a complete and unambiguous set of software requirements. Using these specifications as the main development artifact, Rockwell Collins was able to quickly and accurately determine the necessary pre and post conditions for each database transaction.

The use of a model-based approach to transaction development provides early simulation capabilities, leading to earlier discovery of errors in both the specification and in the implementation. The use of automated code generation removes the possibility of human coding errors. The application of automated model checkers provides a proof of correctness at a level unattainable through traditional software testing methods. With all of these components in our software development approach, we have exercised a viable methodology to deliver high-assurance software with a much greater level of confidence than software developed through traditional approaches.

The use of SPARK information flow annotations for Ada95 code at the system level provides assurance the system code is properly routing information to each of the devices in the HSCC architecture. Hardware enforced (AAMP7G partitioning) red/black separation serves as the final sentinel in preventing unintended red/black communication. In our judgment, the methodology described in this paper is sturdy enough to support full EAL-7 certification of a production encryptor based on this research prototype.

Despite the apparent advantages of the proposed methodology, some aspects of the approach need further development before the methodology can readily be employed.

The most serious limitation is the relatively poor support in portions of the tool-chain for composite data structures. For many of the databases in the HSCC system, multi-dimensional arrays would have been the natural representation chosen, had Reactis allowed their use. Instead, multiple two-dimensional arrays were used, leading to a more complex implementation and an increased burden in development of the proof infrastructure models. Arrays of structures would have been a natural choice to represent some of the databases, but Simulink does not support such data structures. The use of the SCADE tool, or improvements to the Mathworks tool-suite may address some or all of these issues.

Another limitation is the relatively small size of databases for which transaction proofs are able to be obtained. Further research will look into identifying methods to leverage existing transaction proofs to obtain general proofs of arbitrarily sized databases.

References

1. AdaCore Inc, Tokeneer product description at AdaCore, `http://www.adacore.com/home/products/sparkpro/tokeneer/`
2. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, W.: Engineering the Tokeneer Enclave Protection Software. In: Proceedings of IEEE International Symposium on Secure Software Engineering (ISSSE) (2006)
3. Esterel Technologies, Inc., SCADE Suite product description, `http://www.esterel-technologies.com/products/scade-suite`
4. Greve, D., Richards, R., Wilding, M.: A Summary of Intrinsic Partitioning Verification. In: Proceedings of ACL 2004, Austin, TX (November 2004)
5. Hardin, D.: Invited Tutorial: Considerations in the Design and Verification of Microprocessors for Safety-Critical and Security-Critical Applications. In: Cimatti, A., Jones, R. (eds.) Proceedings of the Eighth International Conference on Formal Methods in Computer-Aided Design (FMCAD 2008), pp. 1–8 (November 2008)
6. McComb, T., Wildman, L.: Verifying Abstract Information Flow Properties in Fault Tolerant Security Devices. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 621–638. Springer, Heidelberg (2006)
7. Prover Technologies, Inc. Prover SL/DE plug-in product description, `http://www.prover.com/products/prover_plugin`

8. Reactive Systems, Inc. Reactis product description, `http://www.reactive-systems.com`
9. Saaltink, M.: The Z/EVES System. In: Bowen, J.P., Hinchey, M.G., Till, D. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 72–86. Springer, Heidelberg (1997)
10. Spivey, J.M.: The Z Notation: A Reference Manual, 2nd edn. International Series in Computer Science. Prentice Hall, Englewood Cliffs (1992)
11. The Mathworks, Inc., Simulink product description, `http://www.mathworks.com/Simulink`
12. Whalen, M., Cofer, D., Miller, S., Krogh, B.H., Storm, W.: Integration of formal analysis into a model-based software development process. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 68–84. Springer, Heidelberg (2008)