AN ATTEMPT TO EXAMINE TOKENEER USING BAKAR KIASAN.

by

HARI HARA KUMAR EARLAPATI.

B.TECH, Kakatiya University, INDIA, 2009

A REPORT

Submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences.
College of Engineering.

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2011

Approved by:

Major Professor
Dr. John Hatcliff

Abstract

In order to demonstrate that developing highly secure systems to the level of rigor required by the higher assurance levels of the common criteria is possible, the NSA asked Praxis High Integrity Systems to undertake a research project to develop a high integrity variant of part of an existing secure system (the Tokeneer System) in accordance with Praxis' own high-integrity development process. Their objective is to show the security community that it is possible to develop secure systems rigorously in a cost-effective manner. Hence part of the Tokeneer (ID Station) is redeveloped in Spark programming language and is verified using the Spark proof tools. Bakar Kiasan is a symbolic execution tool for the Spark programming language built in Kansas State University, it can be used for bug finding, test case generation and contract checking. This tool's proof process does not include the conventional Spark tools like the Examiner, Simplifier and Proof Checker. It mainly allows the programmer to focus entirely on the source code level. The goal of this MS report is to assess the extent to which symbolic execution techniques in Bakar Kiasan can be applied to the Tokeneer example implemented in Spark.

Table of Contents

List of figures iv
Acknowledgements v
1. Introduction 1
1.1 Goal: 1
1.2 Motivation: 1
1.3 Challenges: 2
1.4 Software requirements: 2
2. Tokeneer: 3
2.1 Overall Tokeneer System: 3
2.2 System boundaries: 5
2.3 Scenarios: 6
3. Formal Specification in Z: 8
4. Introduction to Spark. 9
4.1 Steps (Life Cycle) in the process of proving the correctness of a Spark program: 13
4.2 External variables: 17
5. Changes made in existing project to examine using Kiasan. 18
5.1. Flattening the nested Methods: 18
5.2. Eliminating Abstraction: 22
5.3 Replacing the Proof functions with normal Spark Executable functions: 31
5.4. Removing the private child packages: 34
6. Summary of changes in the project for examining using Kiasan. 37
7. Future Work: 40
8. References 41

List of figures

Figure 1: Tokeneer System 3
Figure 2: System Boundaries 5
Figure 3: Diagram showing the Lifecycle: 13
Figure 4 - Specification Before eliminating Abstraction 26
Figure 5 - Body before eliminating abstraction 27
Figure 6 - Specification after removing abstraction 28
Figure 7 - Body after eliminating abstraction 29

Acknowledgements

I would like to thank my major professor Dr. John Hatcliff for his constant guidance and help throughout the project. I would also like to thank Dr. Robby and Dr. Mitch Neilsen for graciously accepting to be on my committee. Finally, I wish to thank my family and friends for all their support and encouragement.

1. Introduction

In order to demonstrate that developing highly secure systems to the level of rigor required by the higher assurance levels of the Common Criteria is possible, the NSA asked Praxis High Integrity Systems to undertake a research project to develop a high integrity variant of part of an existing secure system (the Tokeneer System) in accordance with Praxis' own high-integrity development process. Their objective is to show the security community that it is possible to develop secure systems rigorously in a cost-effective manner.

The development project for this high integrity variant has an objective:
To redevelop part of the software for the Identification Station (ID Station — part of the Tokeneer system) according to Praxis High Integrity System's formal, high-integrity system development process.

1.1 Goal:

The principle goal of the project is to test to what extent we can examine the IDStation implemented in Spark using the symbolic execution tool Bakar Kiasan.

1.2 Motivation:

The prime motivation for using a tool like Bakar Kiasan to examine the Tokeneer is:
Existing Spark tools like the Examiner, Simplifier and Proof checker uses Verification Conditions (VC) approach for proving the correctness of the Spark programs. Examiner takes the Spark program as input and generates verification conditions; it is the duty of the automated tools like simplifier or proof checker to prove the verification conditions to true or false for checking the Spark programs correctness. The above proof process involves few issues:

1. We need to write the rule files (with extension .rlu) manually in order to prove certain verification conditions (VC). Rule files contain the 'replacement rules' that are used by the Spark tools in the process of proving the program correctness.

2. Rule files are written in FDL (Functional Description Language) so we need to learn the FDL syntax and semantics.

3. To understand the proof process thoroughly, we need to study other files that are generated by the examiner (for example: .rls, .vcg, .fdl) and how the simplifier and proof checker are dealing with them. We will discuss about all the intermediate files in detail in next sections.

Considering the above issues we want to explore the Tokeneer using Bakar Kiasan in order to see if it offers any advantages over the VC approach.

1.3 Challenges:

The main challenge was to understand the life cycle of the proof process for a Spark program completely. This involved studying all the intermediate files that are generated by the Examiner, learning the FDL syntax and semantics, understanding the rule files that hold the replacement rules for the proof functions. During the initial days, to understand how the tokeneer is modelled, I had to study basics of the 'Z' specifications language. 'Z' is used to model the tokeneer at the specifications level.

1.4 Software requirements:

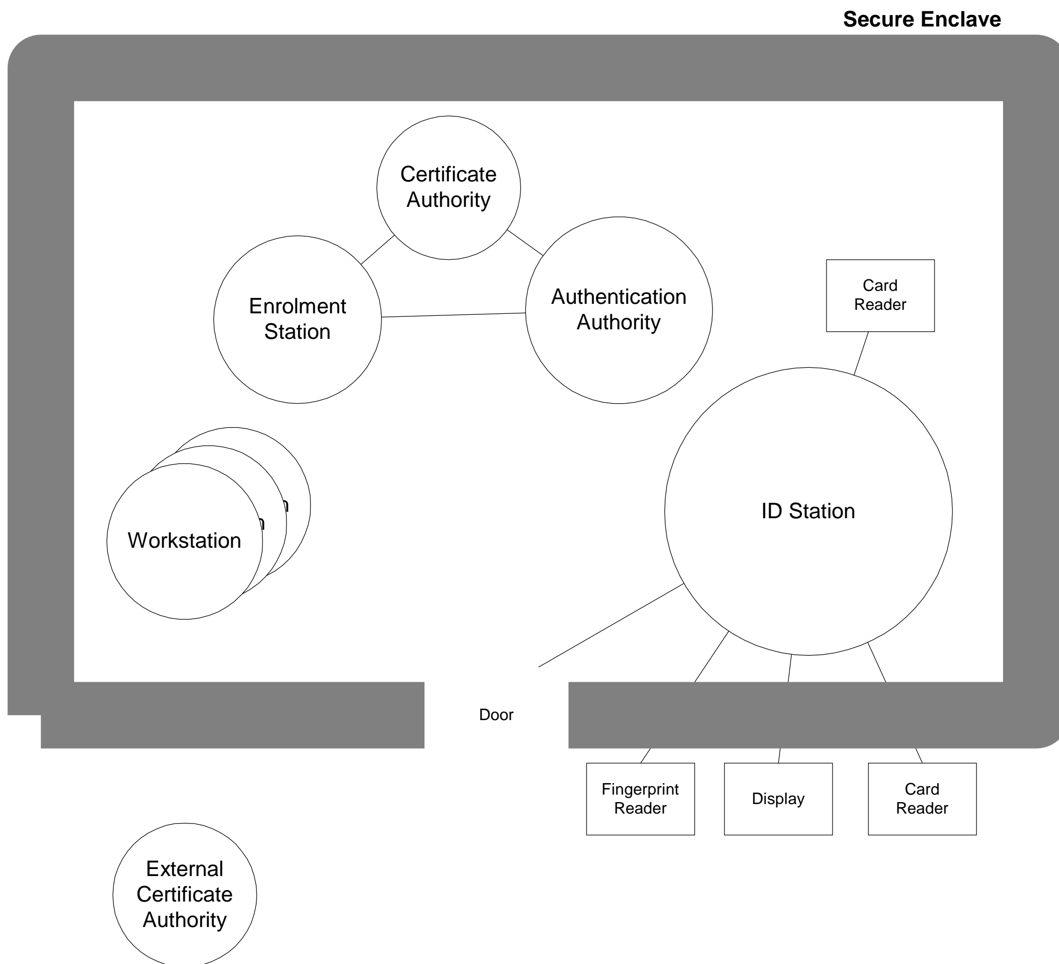
1. Eclipse platform (version 3.6.2 used).
2. Spark.
3. Operating system: Linux/Mac/Windows.

2. Tokeneer:

Tokeneer is a security system that demonstrates the use of smart cards and biometrics for access control. It provides protection to secure information held on a network of workstations situated in a physically secured enclave.

2.1 Overall Tokeneer System:

Figure 1: Tokeneer System



The complete tokeneer system consists of a secure enclave and set of system components, some present inside the enclave and some outside. As it can be seen, ID Station is component of the Tokeneer system and its functionality is to provide user authentication.

There are two system boundaries: one boundary between ID Station machine and its environment, the other boundary between the IDStation core functions and its support functions. The existing ID Station has four connected peripherals and a number of internal drivers and libraries.

The physical devices that are interfaced to ID Station are:

1. Fingerprint reader.
2. Smartcard reader.
3. Door
4. Visual display.

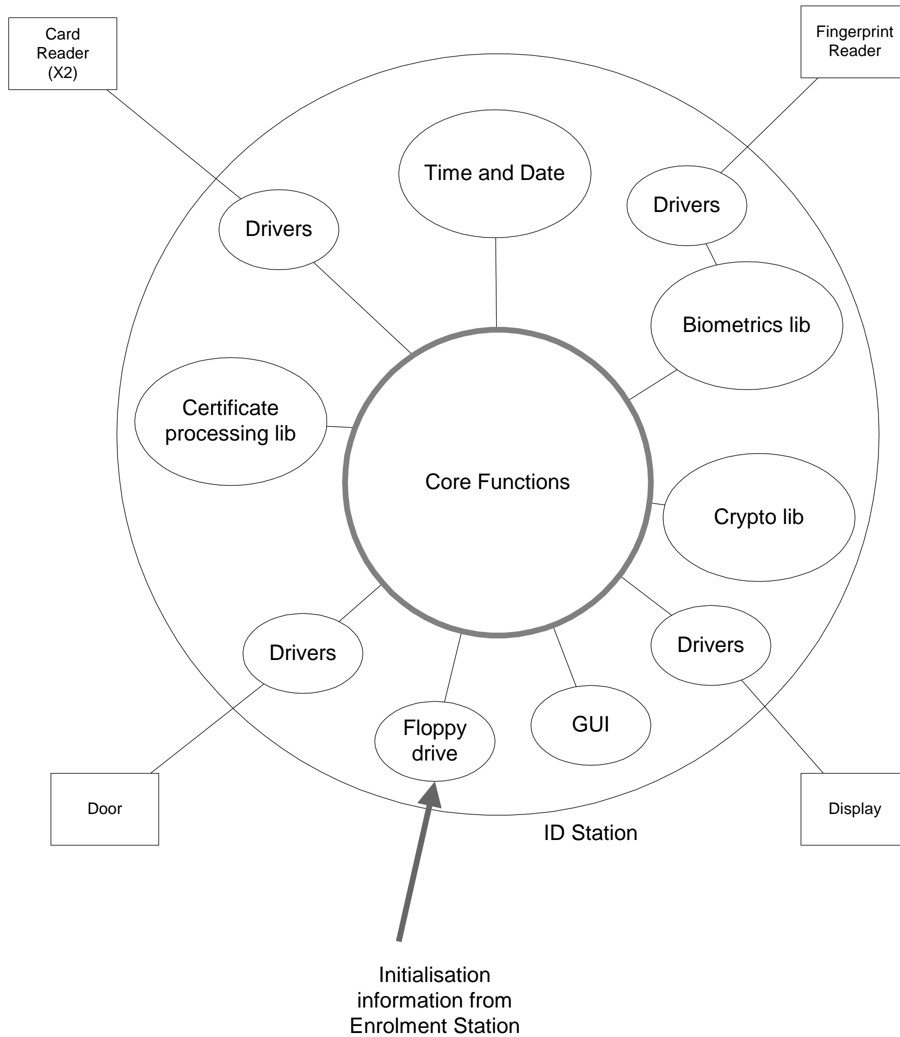
Individuals enter the secured enclave via ‘door’ by providing the credentials either to ‘fingerprint reader’ or the ‘card reader’. ‘Visual display’ displays the messages that help to track the progress of user entry process into secured enclave.

We will now look at how the ID Station that controls the user access to the enclave is considered for development by the programmers.

Diagram showing System boundaries:

2.2 System boundaries:

Figure 2: System Boundaries



Praxis redeveloped the core functions of the ID Station according to their high-integrity development process. The required behavior of the ID Station is specified using scenarios, which run through typical uses of the ID Station and define the interaction between the ID Station and its connected systems. In each case the scenario focuses on a successful outcome, but it also covers various conditions that may arise that do not allow the successful outcome to be achieved. The full behavior of the system, including both successful and failed outcomes, constitutes the system requirements.

2.3 Scenarios:

1. User gains allowed initial access to Enclave.
2. User is denied prohibited initial access to Enclave.
3. User gains allowed repeat access to Enclave.
4. ID Station is started and enrolled with input from the Enrolment Station.
5. ID Station is started already enrolled.
6. ID Station is shut down.
7. Security Officer updates the configuration of the ID Station.
8. Audit log is archived.
9. Guard manually unlocks the door.
10. Administrator logs on.
11. Administrator logs off.

We now consider a sample scenario, give a high level description of it and then give small code snippets of how it is modeled in Z and then programmed in Spark.

This is done only to show how the actual scenarios are modeled in different levels (The reader is not expected to understand modeling in Z and programming in Spark at present). The below specifications in 'Z' and program snippet in Spark can be understood completely by looking into the documentation.

Consider the scenario: Administrator logs on (Scenario 10)

1. Administrator logs on by inserting card into the card reader.
2. We assume that the ID Station is quiescent. (No others are attempting to operate it)
3. We assume that door is closed and the card inserted by the admin is valid.
4. We also consider the success end conditions:
    a. Administrator is able to successfully log in to the enclave.
    b. Once admin comes in, door is closed and locked.
    c. All the events are recorded in the audit log (It logs all the activities).
5. We now consider the failure conditions:
    a. Admin is unable to log in as all the data on the card is not read successfully.
    b. Unable to write the audit files to the audit log.

c. Audit log has no place.

6. The constraint that should hold is: user use or ID Station shutdown is not allowed during the entire scenario.

Above scenario modeled in Z as: (FD.Admin.AdminLogon: page 76 of [1])

AdminLogonC

ΔAdminC

requiredRole? : ADMINPRIVILEGE

rolePresentC = nil

the rolePresentC′ = requiredRole?

currentAdminOpC′ = nil.

It is modeled in Spark as:

(Admin package in ‘core’ directory; Refer to code download for Tokeneer)

```
procedure Logon (TheAdmin :    out T; Role     : in     PrivTypes.AdminPrivilegeT)

--# derives TheAdmin from Role;
--# post ( Role = PrivTypes.Guard <-> Prf_rolePresent(TheAdmin) = PrivTypes.Guard ) and
--#      not IsDoingOp(TheAdmin) and IsPresent(TheAdmin);
is begin
    TheAdmin.RolePresent := Role;
    TheAdmin.CurrentOp   := NullOp;
end Logon;
```

3. Formal Specification in Z:

The behavior of the core of the ID Station is first specified using Z formal notations. The specifications model the ID Station as a number of state components and a number of operations that change the state.

These formal specifications are then programmed in Spark programming language.

Considering a simple example:

(token is the card that is inserted by the user to enter the enclave.)

'user token presence' (it defines whether user token is present or not) is modeled in Z as:

PRESENCE ::= present | absent

userTokenPresenceC : PRESENCE

Different privileges of the administrator are modeled as:

(Admin can be either a guard or auditManager or SecurityOfficer)

ADMINPRIVILEGE == {guard, auditManager, securityOfficer}

It is programmed in Spark as: (Shift it later to Spark section.)

```
TokenPresence : BasicTypes.PresenceT;
type PresenceT is (Present, Absent);
```

Few commonly used Z notations: (Z notation: What it denotes)

1. P: Set of.

2. Theta: State of.

3. Z: Set Of Integers

4. Delta: A change in the variable.

5. A |--> B: (A, B) is a set of elements.

6. a? : Input to a.

7. a! : Output from a.

4. Introduction to Spark.

Spark is a high level programming language designed for writing software for high integrity applications where safety and security are important.

Spark language comprises a Kernel, which is a subset of Ada plus additional features inserted as annotations in the form of Ada comments. These annotations are ignored by Ada compiler and can be compiled separately by a standard compiler. Spark annotations are written prefixed with ‘ --# ‘. Packages are general means of encapsulation in Spark. Packages provide key facilities of Object oriented programming by controlling access to hidden entities through Subprogram methods. Spark subprograms can be procedures or functions. The purpose of a procedure is to perform an updating action of some kind. A function in general doesn’t have any side effects.

Spark annotations are in two categories:

1. Core Annotations:

A. Global definitions: using keyword ' global '. They are used to access the ' own ' variables inside a subprogram. ‘own’ variables are the package variables.
B. Annotations that concern coupling between variables: using ' derives ' clause.

If a variable is prefixed with ‘ in ’, it means the value it holds at the beginning of the method must be used inside the method’s body.

If a variable is prefixed with ‘ out ’, it means there is a change in the value it holds at the beginning and in the end of the method.

Sample Spark program (1) using ‘global’ and ‘derives’ clauses:

```
procedure Add (X: in Integer)
--# global in out Total;
--# derives Total from Total, X;
```

Considering the sample program 'Add' we can say that:

- procedure Add uses the global variable ‘Total’ in its method body.
- Initial values of variable 'Total' and 'X' must be used in the procedure Add (initial values are the values upon entering the procedure Add).
- A new value of 'Total' will be produced at the end of procedure Add.
- 'derives' statement says that, initial values of 'X' and 'Total' must be used in deriving the final value of 'Total'.

There are also other core annotations that concern access to variables in packages.

- ' inherit ' clauses: they control the visibility of packages.
- ' own ' variable clauses: they control access to package variables (global variables).
- ' initialization ' annotations: indicate the initialization of own variables.

'own' variable is the one declared inside a package and it contains state preserved between calls of subprograms in the package. ' global ' is used to access the ' own ' variables inside a subprogram.

2. Proof Annotations:

A. Pre and Post conditions of subprograms.
B. assertions such as loop invariants and type assertions.
C. Declarations of proof functions and proof types.

Sample Spark program (2) using ‘own’, ‘global’, ‘derives’, 'post' annotations:

The example shown has two procedures and two functions. It can be seen that function does not have any side effects on the package; they just have the return statements. But procedures perform a number of operations.

~ Operator indicates 'prestate' value (initial value) for an identifier. An identifier that is both an import and export of a procedure may be decorated with a tilde and the identifier with tilde indicates the initial imported value of the identifier.

```
package Odometer
--# own Trip, Total: Integer;           --package variable declarations.
is
  procedure Zero_Total
  --# global out Trip;                  -- package variable 'Trip' will be given a new value.
  --# derives Trip from ;               -- 'Trip' is not derived from any variable.
  --# post Trip=0;                      -- At the end of the procedure, Trip value is 0.

  function Read_Trip return Integer;
  --# global in Trip;                   --uses the initial value of package variable 'Trip'.

  function Read_Total return Integer;
  --# global in Total;

  procedure Inc;
  --# global in out Trip, Total;
  --# derives Trip from Trip & Total from Total;
  --# post Trip = Trip~ + 1 and Total = Total~ + 1;

end Odometer;

package body Odometer is

   Trip, Total: Integer;                       --declarations of package variables in ada code.

   procedure Zero_Trip is
   begin
     Trip:=0;
   end Zero_Trip;

   function Read_Trip return Integer is
   begin
       return Trip;
   end Read_Trip;

  function Read_Total return Integer is
  begin
      return Total;
  end Read_Total;

  procedure Inc is
  begin
    Trip := Trip + 1; Total := Total + 1;              --Initializing the 'initialize' variable.
  end Inc;

end Odometer;
```

‘Inherit’ and ‘Initialize’ annotations:

- ' inherit ' clauses are used to control access to global entities outside packages.
- 'initialization' annotations ensure that 'own' variables are properly initialized.

Sample program (3) using ‘inherit’ and 'initialize' annotations:

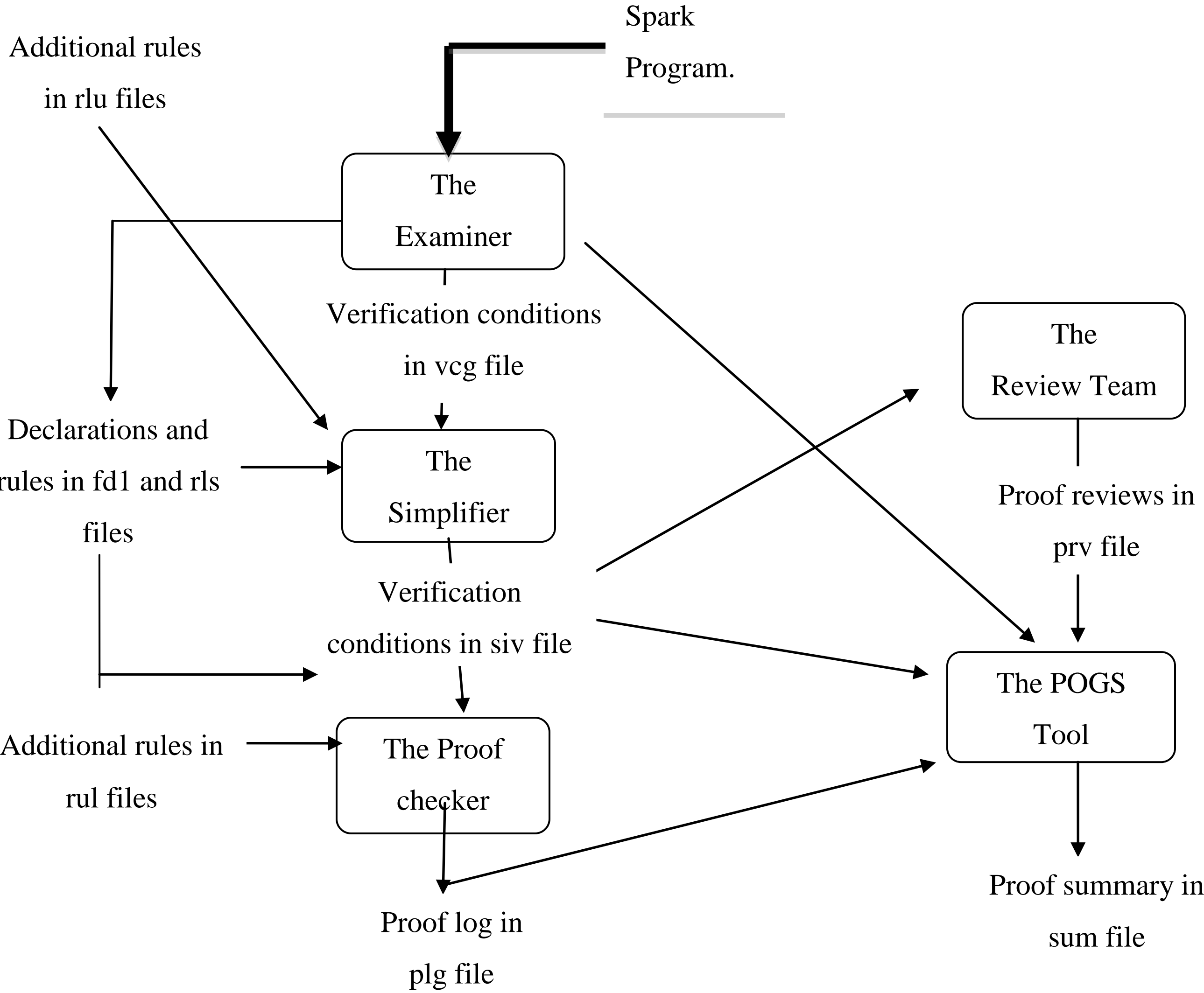
```
--# inherit Odometer;
package Capture_Odometer_Total
--# own Temp;
--# initializes Temp;
is
  procedure capture
  --#global out Temp;
  --#          in   Odometer.Total;
 end Odometer;

with Odometer;
package body Capture_Odometer_Total is

  Temp:Integer;

   procedure capture
   is
      begin
      Temp=Odometer.Total;
   end capture;
begin
  Temp:=0;                                        -- Initializing the variable.
  end Capture_Odometer_Total;
```

In the above example the package ‘Capture_Odometer_Total’ wants to capture the value of the variable ‘Total’ of package Odometer. Hence we include ‘inherit’ clause for that package on the top of package ‘Capture_Odometer_Total’. We declared the variable ‘Temp’ prefixing with ‘initializes’ annotations, so we initialized it in the initialization part(between ‘begin’ and ‘end’) of the package body ‘with’ clause is used on the package body. It allows accessing the outside package variables via dotted notation in the ada code.

4.1 Steps (Life Cycle) in the process of proving the correctness of a Spark program:

I studied the proof process in detail because, in the process of working on the project, I had to make a number of replacements in the original code. This required a deeper understanding of the entire proof process.

Figure 3: Diagram showing the Lifecycle:



Consider the life cycle: At first the Spark program is passed to the Examiner. It then generates a number of ‘verification conditions’. Proving the programs correctness amounts to proving the verification conditions. Automated tools like ‘The Simplifier’ and ‘The Proof checker’ are used for proving these verification conditions. POGS stands for Proof Obligation Summarizer. This tool summarizes what are the verification conditions that are discharged (proved) and that are not discharged. We now consider Spark Examiner, Simplifier and Proof checker in detail:

Spark Examiner: It is a key Spark tool that has been written in Spark. Its main functionality is:

1. It checks whether the code is syntactically correct.
2. It also checks whether Spark annotations conform to the Ada code in the program by performing three different levels of analysis:

A. **Data flow analysis:** It checks that the usage of parameters and global variables corresponds to their modes ('in' and 'out') and also checks that variables are not read before given a value and values are not overwritten without being used and also that all imported variables are used.

B. **Information flow analysis:** This needs 'derives' annotation. As well as carrying out the Data flow analysis, it checks that the modes of parameters and global variables and their usage in the code of the body matches the interdependencies given in the ‘derives’ annotation.

C. **Generation** of Verification conditions (VC's). These VC's need to be proved either manually or by using an automated tool like Simplifier or proof checker to prove that the program is correct with respect to the proof annotations.

Verification conditions and how the Examiner generates them: The Examiner generates Verification conditions (VC) by considering both the code and proof annotations. Verification conditions are written in FDL (Functional Description Language). Examiner performs the mapping from Spark expressions into FDL statements.

VC's take the form:

H1: ....

H2: ....

H3: ....

....

->

C1:.....

C2:.....

....

Here H1, H2, H3 are called hypotheses (Assumptions) and they represent the precondition written for each subprogram.

C1, C2. C3 are called conclusions and these are to be proved.

VC's are generated for each procedure in the package. Each procedure can have more than one VC depending on the number of paths from procedure begin to end.

Example: A procedure that swaps two variables has only one path usually, but a procedure that has iterative loops will have more than one path from begin to end.

Considering a sample example (4) and then showing how VC's are generated:

```
procedure Exchange (X, Y: in out Float)
--# derives X from Y &
--#         Y from X;
--# post X=Y~ and Y=X~;
is
    T : Float;
begin
    T:=X;
    X:=Y;
    Y:= T;
end Exchange;
```

Procedure ‘Exchange’ exchanges the variables ‘X’ and ‘Y’ that are passed as parameters to it. Post condition states that final value of X holds the initial value of Y and vice versa.

We now consider the post condition and then make all the replacements that are in the code from ‘bottom to top ‘. Remember that VC's are generated by considering both the code and the annotations.

Post condition: X=Y~ and Y=X~.

Replacement 1: Y:=T, then the post condition becomes

X=Y~ and T=X~.

Replacement 2: X:=Y, post condition becomes

Y=Y~ and T=X~.

Replacement 3: T:=X, post condition becomes

Y=Y~ and X=X~.

While replacing, Y~ means the initial value so we did not replace it with T in the first replacement. The process of moving the post condition backward through some operation (replacement) is known as “hoisting“.

The condition that is obtained after making all the replacements on post condition is called the "weakest precondition". So the conclusion of a VC for a path is the weakest precondition obtained by hoisting its postcondition to the beginning of the path. Proving the VC amounts to showing that the "weakest precondition" is implied by the given precondition.

Now considering our example, the weakest precondition is: Y=Y~ and X=X~. It says that, at the beginning the current value of X must be initial value of X and the same is with Y. It is clear that, at the beginning the current values of X and Y are equal to their initial values. So tildes can be dropped. There is no precondition for the given procedure, so it can be taken as 'true'.

Verification Condition turns to be:

H1: true.

->

C1: Y=Y.

C2: X=X.

Both the conclusions are correct, and hence the procedure is proved to be correct. If there are

multiple VC's for a procedure then each of these need to be discharged to prove the procedure correctness.

Simplifier and Proof checker:

Simplifier and Proof checker are the automated tools that are used for proving the verification conditions. Role of proof checker comes after the simplifier. If the simplifier is unable to reduce a verification condition to true then proof checker does that job. There may be number of reasons why simplifier is unable to reduce a verification condition to true:

1. The verification condition may not be true. This could be the case that program is wrong or some proof statement is wrong.
2. Simplifier is unable to process the additional rules that are supplied. (This will be discussed in further sections).
3. Simplifier may not be clever enough.

4.2 External variables:

Moded 'own' variables are referred to as external variables. They are to be initialized by the environment. Mode can be in (for input) or out (for output). External variables cannot be 'in out'. Sometimes external variables do not actually correspond to explicit variables in the program at all but only exists in the spark annotations for the purpose of reasoning. They generally represent some external state that changes independently of the program.

Example:

```
Package P
--# own in State;                    -- Moded own variable.
is
….
end P;

Package body P is
…
end P;
```

5. Changes made in existing project to examine using Kiasan.

Tokeneer ID Station is implemented in Spark making use of most of the programming language features. Bakar Kiasan currently cannot deal with certain Spark features. For it to be able to examine the ID Station, we need to find replacements for these features. List of features that I considered for replacement, as Kiasan currently cannot deal with them are:

1. Nested methods.
2. Abstraction and Refinement.
3. Proof functions.
4. Child Packages.

5.1. Flattening the nested Methods:

Kiasan currently cannot deal with the nested methods in the packages. So the nested methods need to be converted to arbitrary normal methods that are not nested any more. Converting the nested methods to normal methods is called flattening the nested methods.

Nested Methods:

```
Method A
--#...
is
    Method B
    --# …..
    is
    begin
    ......
    end B;
begin
......
end A;
```

After flattening:

```
Method B
--# …..
is
begin
   ......
end B;
--#.....
Method A
is
begin
  ......
end A;
```

In the above example it can be seen that method B is nested in method A. After flattening B is converted to a normal method.

Sample project code that is flattened in the above format:

Consider the ‘enclave’ package in the Tokeneer code dump.

Nested function in enclave.adb:

```
function CurrentAdminActivityPossible return Boolean
 --# global AdminToken.State,
 --#        Status;
 --# return R => (R -> Status in NonQuiescentStates);
 is

    function AdminActivityInProgress return Boolean
      --# global Status;
    is
    begin
       return Status in ActiveEnclaveStates ;
    end AdminActivityInProgress;

 begin
    return AdminHasDeparted or AdminActivityInProgress;

 end CurrentAdminActivityPossible;
```

After flattening the nested methods:

```
  function AdminActivityInProgress return Boolean
     --# global Status;
    is
    begin
       return Status in ActiveEnclaveStates ;
  end AdminActivityInProgress;

  function CurrentAdminActivityPossible return Boolean
  --# global AdminToken.State,
  --#        Status;
  --# return R => (R -> Status in NonQuiescentStates);
  is

  begin
     return AdminHasDeparted or AdminActivityInProgress;

  end CurrentAdminActivityPossible;
```

In the above example before flattening, the function ‘AdminActivityInProgress’ is nested inside the function ‘CurrentAdminActivityPossible’. Flattening the methods involves lifting the nested function ‘AdminActivityInProgress’ and placing it with the other methods in the package. Nested methods generally appear between "is" and "begin" of the method in which they are

nested. This is the region where the variables used in the method's body are declared. Following the above process to flatten methods in most of the packages in the project did not involve excessive refactoring as the nested methods are called only from the method in which they are nested. Considering the above example, the function ‘AdminActivityInProgress’ is called only from the function ‘CurrentAdminActivityPossible’ in the entire enclave package.

In few packages, nested method uses variables that are passed as parameters to the method in which they are nested; in such cases those variables are again passed as parameters to the flattened methods. The other case may be that, nested methods use variables that belong to method in which they are nested; in that case those variables are also passed as parameters to the flattened methods.

Example describing the above scenarios:

procedure UpdateEndTimeFromFile of AuditLog package.

It has one nested method **before flattening**: function OverwriteTimeInText.

```
procedure UpdateEndTimeFromFile (TheFile     : in out File.T;
                                 Description : in out AuditTypes.DescriptionT)
--# global in out AuditSystemFault;
--# derives AuditSystemFault,
--#         TheFile,
--#          Description     from *,
--#          TheFile;
is
   OK : Boolean;
   LastTime  : Clock.TimeTextT;
   TimeCount : Natural;
   TimeOK    : Boolean := True;

   function OverwriteTimeInText(Description : AuditTypes.DescriptionT )
                                return AuditTypes.DescriptionT
   --# global LastTime,
   --#       TimeOK;
   is begin
      ………..
   end OverwriteTimeInText;

begin
      ………………..
      Description := OverwriteTimeInText(Description);

end UpdateEndTimeFromFile;
```

After flattening the nested method:

```
function OverwriteTimeInText(Description : AuditTypes.DescriptionT;
                              LastTime  : Clock.TimeTextT;
                               TimeOK    : Boolean)
                            return AuditTypes.DescriptionT
--# global LastTime,
--#        TimeOK;
is
begin
   ………..
   end OverwriteTimeInText;

procedure UpdateEndTimeFromFile (TheFile     : in out File.T;
                                 Description : in out AuditTypes.DescriptionT)
is
     OK : Boolean;
     LastTime  : Clock.TimeTextT;
     TimeCount : Natural;
    TimeOK    : Boolean := True;

begin
        ………………..
        Description := OverwriteTimeInText(Description, LastTime, TimeOK);

end UpdateEndTimeFromFile;
```

In the above example, variable 'TimeOK' belongs to procedure UpdateEndTimeFromFile. After flattening the methods, variable 'TimeOk' will not be present in the scope of the function OverwriteTimeInText but it uses that variable in the function body. Hence this variable is passed as parameter to OverwriteTimeInText.

In the project sometimes we find nested methods that are nested inside other methods. We need to follow the same procedure for flattening them.

Spark examiner and simplifier is run and the POGS (Proof Obligation Summarizer) report has been considered to check the number of undischarged VC's .The report states that all the VC's are discharged. Hence the changes that are made to the code in the process of lifting back the nested methods did not have any implications. Hence flattening the methods this way is both semantically and syntactically correct.

5.2. Eliminating Abstraction:

Abstraction and Refinement:

Spark packages typically come in two parts:

1. Specifications describing the external interface. They appear in .ads files (ads stands for ada specification).
2. Body providing the implementation details. They are written in .adb files (adb stands for ada body).

Specification itself may have two parts, a visible part that declares the various entities (types, constants and subprograms) that may be used outside the package and a private part that appears at the end of the package specification. Private part begins with the reserved word 'private'. The entities that are in the private part cannot be referenced outside the package. Body contains the implementation details and these details are generally hidden from the users.

We now define 'Abstract State machine' and 'Abstract own variable' that helps us understand the concept of abstraction.

Abstract State machine is an entity that has well defined states plus a set of operations that cause state transitions. It can be represented by a package with variables, which record its state declared in its body. Procedures that act on the machine and functions that observe its state are specified in the visible part of the package specification. All other details are hidden in the package body.

Abstract own variable is not an ordinary variable, it represents a set of variables that are used in the implementation, and through this they provide the 'refinement' mechanism. Abstract own variable occurs in two annotations, the 'own' variable clause in the package specification and also in the refinement definition in the body that gives the set of variables onto which it is mapped (refined).

From now on, .ads files are referred as “specifications section” and .adb files are referred as “body section”. Annotations (Spark contracts) in the specifications section are called Abstract annotations and annotations present in the body are called refined annotations. Preconditions and Postconditions referring to abstract variables are called ‘Abstract preconditions’ and ‘Abstract Post conditions respectively’. The pre and post conditions referring to the refined variables are called ‘Refined preconditions’ and ‘Refined Post conditions’.

We already discussed that the Spark Examiner can generate more than one Verification condition for a procedure depending on the number of ‘paths’ present in that procedure. In addition, if there is refinement in the annotations of that procedure we can have more verification conditions one for each of the following:

1. Abstract Preconditions → Refined Preconditions.
2. Refined Preconditions → Refined Postconditions.
3. Refined Postconditions → abstract Postconditions.

Sample Example program showing Abstraction and refinement:

The_Stack.ads:

```
package The_Stack
--# own State;                          -- This is an abstract variable.
Is
    procedure Push(X: in Integer);
    --# global in out State;            -- The annotations use the abstract variable
    --# derives State from State, X;

    procedure Pop(X: out Integer)
    --# global in out State;
    --# derives State, X from State;
end The_Stack;
```

The_Stack.adb:

```
 package body The_Stack
 --# own State is S, Pointer;              -- The refinement definition for abstract variable.
is
   -- Ada code declaring variables S and Pointer...
   procedure Push(X: in Integer);
   --# global in out S, Pointer;           -- The annotations use the refined variables
   --# derives S from S, Pointer X &
   --#          Pointer from Pointer;
   is begin
        …………..
   end Push;
   procedure Pop(X: out Integer)
    --# global in S; in out Pointer;
   --# derives Pointer from Pointer &
   --#          X from S, Pointer;
   is begin
        …………..
   end Pop;
is
begin
      ……………..
end The_Stack;
```

In the above example, we can see that State is an abstract variable present in the specifications section of the package, it is refined to S & Pointer in the body. Refined state is not visible as the implementation details in the body are hidden from the external users. Hence all the external users who access the package 'Stack' do not have any idea bout the refinement of the variable 'State'. Observe the difference in the annotations for the same procedure present in the specifications section and body section of the package: annotations for the procedure 'Push' in stack.ads use the abstract variable 'State' where as annotations in stack.adb for the same procedure uses ' S & pointer '.

Kiasan currently cannot deal with the concept of abstraction and refinement, so we decided to eliminate it completely by lifting the refined state variables from body section (adb files) into specification section (ads file). Hence the state variables are declared only once and this is in the specifications section of the package.

The strategy followed for eliminating the abstraction is: First we transfer all the refined variables from body section into the specifications section, this is done by placing all the refined variables as fields in the record in .ads file. We leave the 'own' variable in the specifications section with the same name but it is declared to be of the type 'record' (Previously it was of the type 'abstract'). The proof annotations for methods present in the specifications section of package are replaced by the proof annotations for the same methods present in the body section of the package. After this is done, all the annotations in the body section for those methods are eliminated, as we don't want to repeat the same annotations twice. The reason for not replacing the core annotations is explained with an example. We finally remove all the 'own' annotations in the body section. This is the strategy that is followed for eliminating the abstraction.

We can even place the refined variables as normal variables in the specification section but the reason for transferring the refined variables into a record is to reduce the amount of refactoring that has to be done in the rest of the project once the abstraction is removed.

Eliminating abstraction from the sample project code: (Example: 5)
'door' package: consider door.ads and door.adb files.

(The example package that is described here will be used to explain all other transformations that are made in the rest of the report)

Before eliminating the Abstraction: (door.ads and door.adb)

Figure 4 - Specification Before eliminating Abstraction

door.ads: (Considering the length of the files, only the necessary parts are written here)

```
package Door
--# own State : StateType;
--#       in Input;
is

   --# type StateType is Abstract;

   --# function prf_alarmTimeout(DoorState : StateType)
   --#    return Clock.TimeT;

   function TheDoorAlarm return AlarmTypes.StatusT;
   --# global State;

   procedure Poll(SystemFault :    out Boolean);
   --# global in     Input;
   --#          …….
   --#          …….
   --#        in out State;
   --#         ………
   --#         ………
   --# derives AuditLog.State,
   --#         AuditLog.FileState from State,
   --#                            Input,
   --#             ……………… &
   --#         State              from *,
   --#                            Input,
   --#               …………….  &
   --#            …………..  &
   --#         SystemFault        from Input;

   --#         post
   --#            …………..      and
   --#          TheCurrentDoor(State) = Open and
   --#          Clock.GreaterThanOrEqual(Clock.TheCurrentTime(Clock.CurrentTime),
   --#                               prf_alarmTimeout(State)) ) <->
   --#        TheDoorAlarm(State) = AlarmTypes.Alarming ) and
   --#         ………………….

end Door;
```

Figure 5 - Body before eliminating abstraction

door.adb:

```
with Door.Interface;
package body Door
--# own State is CurrentDoor,
--#            AlarmTimeout,
--#            DoorAlarm &
--#     Input is in Door.Interface.Input;
is
        CurrentDoor  : DoorType.T;
        DoorAlarm    : AlarmTypes.StatusT;
        AlarmTimeout : Clock.TimeT;

       function TheDoorAlarm return AlarmTypes.StatusT
       --# global DoorAlarm;
       is begin
       return DoorAlarm;
       end TheDoorAlarm;

       procedure Poll(SystemFault :    out Boolean)
       --# global in     Clock.Now;
       --#          …………..
       --#        in     AlarmTimeout;
       --#        in     Interface.Input;
       --#        in out DoorAlarm;
       --#        in out CurrentDoor;
       --# derives AuditLog.State,
       --#         AuditLog.FileState from AuditLog.State,
       --#          …………….
       --#                             DoorAlarm,
       --#                             AlarmTimeout,
       --#                             CurrentDoor,
       --#                             Interface.Input &
       --#         SystemFault        from Interface.Input &
       --#         DoorAlarm          from Clock.CurrentTime,
       --#                ……………
       --#                             AlarmTimeout,
       --#                             CurrentDoor,
       --#                             Interface.Input &
       --#         CurrentDoor        from *,
       --#                             Interface.Input;
       --# post
       --#      ( ( CurrentDoor = Open and
       --#         ……………………. and
       --#         Clock.GreaterThanOrEqual(Clock.TheCurrentTime(Clock.CurrentTime),
       --#                              AlarmTimeout) ) <->
       --#        DoorAlarm = AlarmTypes.Alarming ) and
       --#         ……………………..;
       is
                ………………
       begin
            ………………….
       end Poll;
end Door;
```

After eliminating the Abstraction from the package door:
Figure 6 - Specification after removing abstraction

door.ads:

```
--# inherit DoorInterface;
          …………..;
package Door
--# own State;
is

      type StateType is
      record
               CurrentDoor  : DoorType.T;
               DoorAlarm    : AlarmTypes.StatusT;
               AlarmTimeout : Clock.TimeT;
      end record;

      State: StateType;

      --This is the replacement for the proof function…
      function SparkExecutable_alarmTimeout
               return Clock.TimeT;
      --# global State;

      function TheDoorAlarm return AlarmTypes.StatusT;
      --# global State;

     procedure Poll(SystemFault :    out Boolean);
      --# global in     Clock.Now;
      --#         …………..
      --#       in     DoorInterface.Input;
      --#       in out State;
      --# derives AuditLog.State,
      --#        AuditLog.FileState from AuditLog.State,
      --#          …………….
      --#                          State,
      --#                          DoorInterface.Input &
      --#        SystemFault       from DoorInterface.Input &
      --#        State         from *, Clock.CurrentTime,
      --#               ……………
      --#                          AlarmTimeout,
      --#                          CurrentDoor,
      --#                          DoorInterface.Input &
      --# post
      --#     ( ( State.CurrentDoor = DoorType.Open and
      --#         …………………….  and
      --#         Clock.GreaterThanOrEqual(Clock.TheCurrentTime(Clock.CurrentTime),
      --#                            State.AlarmTimeout) ) <->
      --#       State.DoorAlarm = AlarmTypes.Alarming ) and
      --#         ……………………..
      --#         ……………………..;
      end Poll;
end Door;
```

Figure 7 - Body after eliminating abstraction

door.adb:

```
with DoorInterface;
        ………….;
package body Door
 is
          -- This is the replacement for the proof function:
        function SparkExecutable_alarmTimeout
        return Clock.TimeT
          is
          begin
                 return State.AlarmTimeout;
          end SparkExecutable_alarmTimeout;

          function TheDoorAlarm return AlarmTypes.StatusT
          is
          begin
                 return State.DoorAlarm;
          end TheDoorAlarm;
          procedure Poll(SystemFault :    out Boolean)
        is
        ……………
        …………...
          begin
          ………………
        end Poll;

end Door;
```

Package Door has many procedures and functions, but we considered only one procedure, one function and one proof function for describing the refinement mechanism. For now one can ignore the replacement of the proof function with spark executable function (Its purpose is discussed later).

We have the abstract own variable 'State' in door.ads. This is refined to 'CurrentDoor', 'DoorAlarm', and 'AlarmTimeOut' in door.adb. Following the refinement mechanism: 'State' is changed to be just 'own' variable in door.ads. StateType is now not of the type Abstract, but we make it a record. We then transfer the refined variables from door.adb to door.ads, we declare the variables CurrentDoor, DoorAlarm, and AlarmTimeOut to be fields in the record (StateType) in door.ads. 'State' is then declared to be of type 'StateType'. Hence 'State' is a record with three fields. We then remove all the 'own' variable annotations in the body section (door.adb). Then, for function TheDoorAlarm, procedure Poll, we copy the proof annotations from door.adb into

door.ads. We then remove all the annotations for those procedures from the door.adb file. As we made the three variables part of record, we cannot refer to them directly, but by 'recordname.fieldname' notation.

The reason for not replacing the core annotations in the specifications section is: The global and derives annotations cannot refer to the fields in the records. If we still want them to refer to fields in the record, we just mention the record name in those annotations. In the above example, own variable 'State' remains the same (we just changed its type to be record). Hence we need not change the core annotations.

We can say that semantics are preserved, as we did not remove any state variables but just shifted the refined variables from body to the specifications and made necessary refactoring in rest of the project. I considered only the packages that had proof functions in them for removing the abstraction; there may be few packages that have abstraction but no proof functions in them. Spark examiner is not run on the transformed project because the transformations had certain implications. These implications have to be resolved and then spark tools must be used.

Issues:

There might be some situations where the above mechanism for removing abstraction does not work. Consider a situation where the refined state has an abstract variable. Considering the above example, the abstract own variable 'input' is refined to Door.Interface.Input. Input is imported from the package Door.interface (this is child of door) and is an external variable. We already know 'external' variables get its input from external world and so it is not refined. As the input in Door.Interface is not refined, the variable 'input' in Door.ads also cannot be refined. Hence 'input' in Door.ads is removed. All the references to 'input' in Door and other packages are replaced with DoorInterface.input, this is explained in "Removing the child packages" section.

Implications:

By eliminating the abstraction we are actually exposing the implementation details. This is fine; because the motivation is not delivering the transformed project to end user but only use it for tool analysis.

5.3 Replacing the Proof functions with normal Spark Executable functions:

Proof function:

There may be some situations where spark tools will not be able to prove the correctness of a procedure because we are unable to express the pre or postcondition in a suitable form. This issue can be resolved by introducing a new type of function called proof function. Its syntax is just the same as for a normal ada function declaration. A proof function is only used in the spark annotations and is not defined in the ada program.

Spark Examiner uses it in the process of producing the verification conditions but it does not need to know what the proof function actually means, because the process of producing verification conditions simply involve formal substitution through the Hoisting process. Hoisting process is already discussed earlier. It is better to prefix the proof functions with 'prf', as it can avoid the confusion with other normal function declarations in specification section.

Example: --# function prf_statusIsGotAdminToken (State: StateType) return Boolean;

Files with .rlu and .rls extension are called rule files. They contain rules that are used by the examiner or proof checker in the process of proving the verification conditions. All the rule files are written in FDL (functional description language). There are many predefined rules available to the tools covering the usual theorems of mathematics and logic. .rls (**Rul**es**S**park) are automatically generated by the examiner and they contain the replacement rules for each subprogram processed.

rlu (**Rul**es**U**ser), as the name suggests these rules are manually written by the user. All the additional rules that are needed for the proof process and which are not generated by the examiner in .rls files are written in .rlu files. For example: The replacement rules for the proof functions are written in .rlu files, as spark examiner cannot generate them. .rlu files can be present either as < procedurename >.rlu or < packagename >.rlu.

Kiasan cannot deal with Proof functions because it doesn't employ the VC approach (used by simplifier and proof checker) for proving the correctness of the program. So all the proof

functions in the Tokeneer project have to be replaced with normal Spark executable functions.
I started looking for proof functions from the main program (tismain) and at first listed all the proof functions that are directly and indirectly referenced through the main program.

When the spark examiner is run on the door package, we have following files generated:

poll.vcg: This has all the Verification conditions corresponding to procedure Poll.

poll.fdl: This has FDL declarations of all the variables found in the verification conditions.

poll.rls: This has the some of the rules that are needed during the proof process.

poll.prv: This is a proof review file, which will be discussed further.

Along with these, poll.rlu which is written by the user and that has the additional rules is supplied to the simplifier. This is done because the proof function prf_alarmTimeout is referenced in the annotations of the procedure poll. Verification conditions that are generated for procedure poll has FDL statements related to prf_alarmTimeout. Considering the Simplifier uses rules in poll.rlu for proving this verification conditions. Along with these additional files are generated for the function TheDoorAlarm like TheDoorAlarm.vcg, TheDoorAlarm.rls, TheDoorAlarm.fdl.

The strategy I employed for replacing the proof functions can be better explained by considering an example package from the project. Considering the same package Door that we already coded with a single function and procedure. The proof function in the package Door:

```
--# function prf_alarmTimeout(DoorState : StateType)
--#           return Clock.TimeT;
```

First we consider the poll.vcg, which has a number of VC's, among them we consider the VC that talks about the "refinement integrity". This is the VC that gives us an idea how the state is refined (Recall we will have a VC that represent the Refined Postcondition as hypothesis and has Abstract Post condition as conclusion). We then consider poll.slg. .slg files are "SparkLog files" that logs how each VC is discharged. By following the log files, VC's and using the rules in .rlu files, we can conclude what the proof function actually models (semantic meaning).

Once we understand the semantics of the proof function, we can replace the proof function with Spark executable function by following those semantics. Hence this replacement is semantically correct.

Following the above process, we give the replacement for prf_alarmTimeout() as:

```
function SparkExecutable_alarmTimeout  return Clock.TimeT
is
begin
        return State.AlarmTimeout;
end SparkExecutable_alarmTimeout;
```

This is the normal spark executable function that has a declaration in door.ads and body in the door.adb.This replacement is done in all the other packages that make a reference to the proof function.

Issues:

Some of the proof functions do not have replacement rules in the .rlu files. In those situations a type of files called Proof review files are used for proving the VC's. These files hold identifiers that indicate specific Verification conditions in the .vcg files and the review team proves them. These identifiers may change once we make any changes in the original code. Hence prv files are to be removed, as we are making many transformations in the code. Proof review files have the extension .prv.

Implications:

The .rlu files are no longer needed as the proof functions are removed.

5.4. Removing the private child packages:

Child Packages:

Top-level packages and subprograms that are not nested in any other packages or subprograms are called library units. Library package can have child packages and these are themselves library packages. The name of the child uses the dotted notation, in which prefix is the name of its parent.

Consider an example:

```
package P is
  ……..          -- Visible part.
private
  …….           -- private part.
end P;

package body P is
…….
begin
……..
end P;
```

Now its child package looks like:

```
package P.child is
  …….
end P.child;
```

We can call such child package as a ‘public child’. There is a difference between a child package and an arbitrary package: the private part and body of a child package can access the private part of its parent. The other difference is, a child package does not need a ‘with’ clause on its body section for inheriting its parent. A child package can have with clauses for its siblings. Body of the parent cannot have with clauses for its public children and cannot access them.

Now considering the private child package:

```
private package P.child is
    ……..
end P.Child;
```

The key thing about a private child is it is not visible to external users. It can only be visible to body of its parent and both the body and specifications of its private siblings. Another key

difference between private and public children is that whole of a private child can see the private part of its parent. Hence public child can seen as adding to the specification of the package where as private child is like part of its body.

Kiasan currently cannot deal with the child packages. We need to convert them to the normal arbitrary packages. This includes refactoring the parent package and the child package as well. Now considering the example package Door. This package has a private child Door.Interface that looks like:

```
--# inherit Door;
private package Door.Interface
--# own in     Input;
is
procedure GetDoorState (DoorState  :    out Door.T; Fault      :    out Boolean);
--# global in     Input;
--# derives DoorState,
--#         Fault      from Input;
end Door.Interface;
```

This package inherits Door, because Door.T is used in the contracts in GetDoorState procedure. Now converting it into arbitrary package and making it public.

```
--# inherit Door;
package DoorInterface
--# own in     Input;
is
procedure GetDoorState (DoorState  :    out Door.T; Fault      :    out Boolean);
--# global in     Input;
--# derives DoorState,
--#         Fault      from Input;
end DoorInterface;
```

This package is made public because, earlier in the process of removing the abstraction we removed a field called ‘input’ from the Door.ads package (Remember it is refined to Door.Interface.input in Door.adb). There are few other packages that referred to ‘input’ in Door.ads earlier, as we removed this field completely we need to consider the ‘input’ in DoorInterface. So all the other packages that referred to ‘input’ in ‘Door’ earlier now refers to

‘input’ in DoorInterface. As the other packages are referring to fields (‘input’) in DoorInterface package, this cannot be a private package. Hence it is made public.

Issues in dealing with private child packages:

Some of the child packages inherit their parent in order to access the fields in parent. If we follow the above strategy of converting the private child packages into normal arbitrary package it leads to circular inheritance (DoorInterface is inherited by Door, because it uses ‘input’ in DoorInterface and Door is in turn inherited by DoorInterface, because it uses ‘T’ in Door). This issue can be solved by considering the fields that cause the circular inheritance and lifting them to a separate package. In the above packages, ‘input’ in DoorInterface and type ‘T’ in Door are causing circular inheritance. So we lift one of them, type ‘T’ into a separate package ‘DoorType’. This package has just one field, type ‘T’. Both Door and DoorInterface will inherit this package.

Some of the private child packages have proof functions. I did not find a systematic way to convert them to the Sparkexecutables, as the rule files did not have any useful semantics for replacement.

Implications:

The private child packages are converted to public arbitrary packages. This implies that any other arbitrary packages can inherit the contents of these packages. Earlier they were private child packages and no other packages were able to inherit the contents.

6. Summary of changes in the project for examining using Kiasan.

I listed few of the packages in the project that I worked on. For each of the package, features that are to be replaced are noted.

1. At first **Enclave** package is considered. It has nested methods, proof functions and abstraction. It is inheriting 21 other packages.

   Status: Applying the techniques mentioned in the report, I was able to flatten all the nested methods, replace the proof functions with Spark executable functions and remove the abstraction in the enclave package. But Kiasan cannot examine it because it inherits other packages that have either abstraction or proof functions or child packages in them.

2. Package **Admin**: It has proof functions.

   Status: Applying the above techniques, I was able to replace the proof functions with the Spark executable methods. But Kiasan cannot examine it as it inherits other packages that either has abstraction or proof functions or child packages in them.

3. Package **AdminToken**: It has nested methods, proof functions, child package and abstraction.

   Status: Applying the above techniques, I was able to replace the nested methods. Kiasan cannot examine it because abstraction and proof functions are not replaced.

   A. The child package **AdminToken.Interface** has abstract variables that are not refined.

      Status: Unable to examine using Kiasan as it has external variables.

4. Package **Alarm**: It has proof functions, abstraction and child package.

   Status: Applying the above techniques, child package can be removed. But abstraction is not removed. Proof functions are not yet replaced.

   A. The child package **Alarm.Interface** has proof function, abstract variables that are not refined.

      Status: Kiasan cannot examine it as it has external variables.

5. Package **AlarmTypes**: Kiasan can examine it.

6. Package **AuditLog**: It has abstraction, nested methods.

   Status: Nested methods are rolled back. Abstraction can be removed.

7. Package AuditTypes: Kiasan can examine it.

8. Package BasicTypes: Kiasan can examine it.
9. Package **Bio**: It has child package, abstraction, and nested methods.

   Status: Applying the above techniques, Nested methods can be removed and abstraction also can be removed. But Kiasan cannot examine it as it has child packages.

   A. The child package **Bio.Interface** has external variable, abstract variables that are not refined.

      Kiasan cannot examine it as it has external variables.
10. Package **Cert**: It has proof function, child packages.

    Status: Proof function can be removed. Unable to examine it using Kiasan as it has child packages.

    A. The child package **Cert.attr**: It has child packages.

       Status: Kiasan cannot examine it as it inherits other packages that have either proof functions or child packages or abstraction.

       a. **Cert.attr.auth**: Nested method. They can be flattened. After flattening Kiasan can examine it.

       b. **Cert.attr.ianda**: Kiasan can examine it.

       c. **Cert.attr.priv**: Kiasan can examine it.

    B. The child package **Cert.id**: Kiasan can examine it.
11. Package **Configuration**: It has nested methods.

    Status: They can be flattened, but Kiasan cannot examine it, as it inherits other packages that either have child packages or abstraction or proof functions.
12. Package **Door**: It has child packages, abstraction and proof function.

    Status: Unable to remove the abstraction completely. Proof function is replaced.

    A. **Door.Interface**: It has external variables. Abstract variables are not refined.

       Status: Kiasan cannot examine it as it has external variables.
13. Package **Display**: It has abstraction, child packages.

    Status: Abstraction cannot be completely removed.

    A. **Display.Interface**: It has external variables. Abstract variables are not refined.

       Status: Kiasan cannot deal with it as it has external variables.

14. Package **Enrolment**: It has nested methods. They can be flattened.

    Status: Kiasan cannot examine it as it inherits other packages that either have abstraction or child packages or proof functions.

15. Package **File**: Abstract variables are not refined.

    Status: Kiasan cannot examine it.

16. Package **Floppy**: Abstract variables are not refined.

    Status: Kiasan cannot examine it.

17. Package **KeyStore**: It has proof functions, nested methods, abstraction and child package.

    Status: Nested methods are flattened but abstraction is not removed completely.

    A. **KeyStore.Interface**: It has External variables. Abstract variables are not refined.

       Status: Kiasan cannot examine it as it has external variables.

18. Package **KeyBoard**: It has child packages, abstraction.

    Status: Abstraction cannot be removed.

    A. **KeyBoard.Interface**: It has external variables.

       Status: Kiasan cannot examine it as it has external variables.

19. Package **Latch**: Proof functions, abstraction and child package.

    Status: Abstraction cannot be removed completely.

    A. **Latch.Interface**: It has external variables, proof functions. Abstract variables are not refined.

       Status: Kiasan cannot examine it as it has external variables.

20. Package **Screen**: Child package, abstraction and nested methods.

    Status: Abstraction cannot be completely removed. Nested methods can be flattened.

    A. **Screen.Interface**: It has external variables. Abstract variables are not refined.

       Status: Kiasan cannot examine it as it has external variables.

21. Package **Usertoken**: Child package, abstraction and nested methods.

    Status: Abstraction cannot be completely removed. Nested methods are rolled back.

    A. **UserToken.Interface**: It has external variables. Abstract variables are not refined.

       Status: Kiasan cannot examine it as it has external variables.

7. Future Work:

Strings and operations on strings such as concatenations are used in the tokeneer project. Kiasan currently cannot deal with these; we need to find an alternative for handling them effectively. One way to deal them is to use ‘hash code’ replacements for the strings. Another way is to replace the strings with Integers (Kiasan can deal with integers). An alternative for proof functions that do not have the replacement rules in the .rlu files needs to be found. We also need to find a way to deal with external variables. One way would be to simulate the external devices with files that can supply the values for the external variables or take values from them.

8. References

[1] 50_1_Formal_Design (docs directory in the tokeneer download).

[2] 41_1_System_Requirements_Specification (docs directory in the tokeneer download)

[3] John Barnes, High Integrity Software. The Spark Approach to Safety and Security,