CIS 890: Safety Critical Systems

Lecture:
SPARK – Contract Checking with Bakar Kiasan

Interface Contracts

Implementation level specification & checking plays an important role in developing high-assurance systems.

```plaintext
function FindSought
   (A: Table; Sought: Integer) return Index;
   --# pre for some M in Index => ( A(M) = Sought );
   --# return ............( A(Z) = Sought) and
   --# [for all K in Index range 1 .. (Z - 1) =>
   --# (A(M) /= Sought));
```

“SPARK Examiner with Run-time Checker…”; p. 22
SPARK Language & Tools?

Language and verification framework designed for critical systems

**Interface Specification Language**
- Annotations for pre/post-conditions, assertions, loop invariants, information flow specifications

**Programming Language**
- Subset of Ada appropriate for critical systems — no heap data, pointers, exceptions, recursion, gotos, aliasing

+ 

**Automated Verification Tools**

- **Examiner** simple static analysis and verification condition generator
- **Simplifier** decision procedure package that simplifies and tries to automatically prove verification conditions
- **Proof Checker** semi-automated framework for manually carrying out proof steps to discharge remaining verification conditions

Uses of SPARK

SPARK has been (is being) used in a number of safety and security critical applications

- **Tokeneer** -- biometrics and smart authentication in card technology. Demonstration project sponsored by US National Security Agency.
- Several large scale security critical projects at Rockwell Collins such as the Janus crypto-graphic engine.
- Avionics systems in the Lockheed C130J and EuroFighter Typhoon projects
- iFACTS - United Kingdom next generation air-traffic control system (team of 100+ developers at Praxis).
- [Rockwell Collins / KSU] development of certified embedded security devices
What are the obstacles?

Unfortunately, none of projects makes extensive use of SPARK’s contract language (most don’t use it at all)

Let’s review what developers must do to verify SPARK contracts

Run the Examiner

```haskell
function Value_Present (A: AType; X : Integer) return Boolean
--# return for some M in Index => (A(M) = X);
if

Result : Boolean;
begin
Result := False;
for I in Index loop
  if A(I) = X then
    Result := True;
    exit;
  end if;
--# assert I in Index and
--# not Result and
--# (for all M in Index range Index'First .. I => (A(M) /= X));
end loop;
return Result;
end Value_Present;
```

Simple post-condition

Developer must insert loop invariants

Examiner simple static analysis and verification condition generator

Verification conditions written in a separate “proof language” called FDL
Run the Simplifier

1 of 7 Verification Conditions written in FDL proof language...

```markdown
function_value_present_3.
H1: true.
H2: for_all(i___1: integer, (i___1 >= index__first and i___1 <= index__last) -> (element(a, [i___1]) >= integer__first and element(a, [i___1]) <= integer__last)).
H3: x >= integer__first.
H4: x <= integer__last.
H5: index__first >= index__first.
H6: index__first <= index__last.
H7: not (element(a, [index__first]) = x).
H8: index__first >= index__first.
C1: index__first <= index__last.
C2: not false.
C3: for_all(m_ : integer, 1 <= m_ and m_ <= 10 -> element(a, [m_]) <> x).
C4: for_all(i___1: integer, 1 <= i___1 and i___1 <= 10 -> integer__first <= element(a, [i___1]) and element(a, [i___1]) <= integer__last).
C5: x >= integer__first.
C6: x <= integer__last.
```

Feed Into the Proof Checker

1 of 3 Remaining Verification Conditions...

```markdown
function_value_present_6.
H1: for_all(m_ : integer, 1 <= m_ and m_ <= 10 -> element(a, [m_]) <> x).
H2: for_all(i___1: integer, 1 <= i___1 and i___1 <= 10 -> integer__first <= element(a, [i___1]) and element(a, [i___1]) <= integer__last).
H3: x >= integer__first.
H4: x <= integer__last.
H5: integer_size >= 0.
H6: integer__first <= integer__last.
H7: integer_base__first <= integer_base__last.
H8: integer_base__first <= integer__first.
H9: integer_base__last >= integer__last.
H10: integer_size >= 0.
H11: integer_base__first <= integer_base__last.
H12: integer_base__first = 1.
H13: integer_base__last >= 10.
C1: not for_some(m_: integer, m_ >= 1 and m_ <= 10 and element(a, [m_]) = x).
```
Feed Into the Proof Checker

Proof steps that must be manually entered to prove 1 of the 3 remaining VCs...

6. replace c # 1 : not for_some(_1, _2) by for_all(_1, not _2) using quant.
   y
   replace h # 11 : not (_1 and _2) by not _1 or not _2 using logical.
   y
   replace c # 1 : not (_1 and _2) by not _1 or not _2 using logical.
   y
   replace c # 1 : not _1 or _2 by _1 -> _2 using logical.
   y
   replace c # 1 : not _1 or _2 by _1 -> _2 using logical.
   y
   unwrap h # 1.
   unwrap c # 1.
   inst int_M__1 with int_m__1.
   replace c # 1 : int_m__1 => 1 by not 1 > int_m__1 using negation.
   y
   replace h # 1 : not _1 > _2 by _1 <= _2 using negation.
   y
   done

Commands / rules to remember when operating proof checker

...no lemmas, no tactics, etc.
About 15 mins for an expert to prove this very simple method/contract

All or Nothing Useful

function Value_Present (A: AType; X : Integer) return Boolean
   --# return for some M in Index => (A(M) = X);
   is
      Result : Boolean;
      begin
      Result := False;
      for I in Index loop
         if A(I) = X then
            Result := True;
            exit;
         end if;
         --# assert I in Index and
         --# not Result and
         --# (for all M in Index range Index'First .. I => (A(M) /= X));
      end loop;
      return Result;
   end Value_Present;

Some paths are verified; some paths are not verified – not very useful
Obstacles

- Loop invariants required
- In many cases, developers get only segmented evidence of a contract’s correctness (all or nothing)
- Basic behavioral properties have to be specified in a separate “proof language” (FDL)
- Technique is not connected with other quality assurance techniques (e.g., testing)

In reality, the burden of use is so high that it is preventing the SPARK contract language from being used to do anything useful.

Our Work

Better integration of contract checking into workflows
- Functional Contracts (pre/post, assertions)
- Automatic Checking
- Kiasan
  - Symbolic Execution Engine

Themes

- Payback is on par with investment; control costs/benefits of the analysis
- When verification engine processes code, communicate “knowledge” gained to developer
- Keep focus on the source code level, instead of using a separate formalism
- Support both declarative and executable specs
- Allow progress without loop invariants
- Connect to other quality assessment techniques; phrase actions in terms of what developers already understand

Use symbolic execution, not just for bug-finding or test-case generation, but for contract checking
Symbolic Execution [King:ACM76]

```c
void foo(int x, int y, int z) {
    z = x + y;
    if (z > 0) {
        z++;
    }
}
```

Symbolic values:
- $x \leftarrow \alpha$
- $y \leftarrow \beta$
- $z \leftarrow \delta$

Constraints:
- $\Phi = \{\}$

New symbolic value:
- $x \leftarrow \alpha$
- $y \leftarrow \beta$
- $z \leftarrow \pi$
- $\Phi = \{x = \alpha + \beta\}$
Symbolic Execution [King:ACM76]

```c
void foo(int x, int y, int z) {
    z = x + y;
    if (z > 0) {
        z++;
    }
}
```

1. $x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \delta, \Phi = \{\}$
2. $z = x + y$
3. $x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \pi, \Phi = \{\pi = \alpha + \beta\}$

**new constraint for conditional**

1. $x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \pi, \Phi = \{\pi = \alpha + \beta, \pi > 0\}$

**new symbolic value**

1. $x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \pi', \Phi = \{\pi = \alpha + \beta, \pi > 0, \pi' = \pi + 1\}$

**new constraint**
Symbolic Execution [King:ACM76]

void foo(int x, int y, int z) {
    z = x + y;
    if (z > 0) {
        z++;
    }
}

...symbolic execution characterizes (theoretically) infinite number of real executions!

Kiasan -- Solving Constraints

The path condition characterizes the set of program states that flow to this point in the path.

Solving constraints on input variables yields input values (a test case) that drives execution down the current path.
Bakar Kiasan Architecture

In Indonesian...
Bakar = Spark/Fire
Kiasan = Symbolic

Visualizing Properties of Paths

Kiasan exhaustively explores all paths through a procedure.

Kiasan builds constraints that characterize each state along each path. These constraints are solved to provide an example ("flow scenarios") of input & output along each path.

Each instance can be turned into a concrete test case.
Kiasan Output

- Tests provide “evidence” to people not familiar with formal methods that something “interesting” is happening in the tool.
- When a bug is found along one path, the test provides a counter-example illustrating the bug.
- Kiasan’s exhaustive exploration automatically yields test suites with very high levels of MCDC coverage.

Each instance can be turned into a concrete test case.

---

Kiasan Output

Sample path input/output for a more complicated example with nested arrays/records.
Early Payback

Kiasan does not need contracts to provide useful semantic information.

This procedure has four paths, so Kiasan provides four “examples”.

Controlling Cost/Coverage

To ensure the path exploration always terminates, Kiasan employs several bounding techniques which are configurable by the user.

- Start with small bounds
- Coverage information provided by the tool indicates if your missing any statements / branches
- Increase bounds to increase coverage
  - increasing bounds increases time required for analysis
  - run analysis with high bounds for high-confidence as part of over-night regression testing
- Most bugs are found with relatively low bounds
Coverage Information

- Summary of coverage information
- Source code. Green code indicates executable code that is covered by analysis. Yellow code indicates that code is partially covered (e.g., only one branch of a conditional).

Working at Source Code Level

- Code + coverage information
- Selectable paths w/ pre/post-state diagrams
- Variable constraints at each step of the computation.
Benefits of Bakar Kiasan

Bakar Kiasan provides a number of capabilities that help integrate SPARK contract checking directly into developer workflows

- Helpful visualization of paths explored
- Connection to existing quality assurance techniques that developers are familiar with (i.e., testing)
- Capabilities are brought together in the Eclipse IDE
- Technique can be applied very early in the development (even before contracts are written)

Example -- Swap

The Swap example below illustrates a very simple contract...

```plaintext
procedure Swap(X, Y : in out Integer)
--# derives X from Y & Y from X;
--# post X = Y~ and Y = X~;
is
T : Integer;
begin
  T := X; X := Y; Y := T;
end Swap;
```
Example - Swap with Error

Kiasan can detect the erroneous code in the version of Swap below...

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

Path for the error case

Red indicates the variables whose values have changed

Y has not changed from the pre-state; it did not receive X's original value.
Example - Swap with Pre-Condition

Adding a (nonsensical) pre-condition allows us to see how the generated pre/post-state examples are affected...

```haskell
procedure SwapPre(X, Y : in out Integer)
--# derives X from Y & Y from X;
--# pre X >= 0 and Y < 0;
--# post X = Y~ and Y = X~;
is
T : Integer;
begin
T := X; X := Y; Y := T;
end SwapPre;
```

For You To Do

Pause the lecture and complete the following exercise (FYTD #1)...

- Modify the contract for the procedure SwapPre on the previous slide so that the post-condition requires that \( X \geq 0 \). Check the procedure using Bakar Kiasan and verify that an error results. This example illustrates that there can be situations where nothing is wrong with the code per se, but that the assumptions made on the input values are inappropriate for establishing the post-condition.
Example - Swap Elements in Array
Let us now consider the more complicated case of swapping elements at certain positions of an array...

procedure Swap_Elements_weak(T : in out Array_Type; I, J : in Index_Type)
--# derives T from T,I,J;
--# post T(I) = T~(J) and T(J) = T~(I);
is
Temp : Contents_Type;
beg
Temp := T(I);
T(I) := T(J);
T(J) := Temp;
end Swap_Elements_weak;

This contract sort of looks like our non-array example. The code conforms to the contract, but the contract should be “stronger”: it needs to specify that the values at the other positions of the array do not change.

Example - Swap Elements in Array
Let us now consider the more complicated case of swapping elements at certain positions of an array...

procedure Swap_Elements_strong_quant(T : in out Array_Type; I, J : in Index_Type)
--# derives T from T,I,J;
--# post T(I) = T~(J) and T(J) = T~(I);
--# (for all K in Index_Type =>
--# ((K /= I) and (K /= J)) -> T(K) = T~(K));
is
Temp : Contents_Type;
beg
Temp := T(I);
T(I) := T(J);
T(J) := Temp;
end Swap_Elements_strong_quant;

We can use quantification to state the “frame condition” that the other parts of the array don’t change.
Example - Swap Elements in Array

Let us now consider the more complicated case of swapping elements at certain positions of an array...

```haskell
procedure Swap_Elements_strong(T : in out Array_Type; I, J : in Index_Type)
--# derives T from T,I,J;
--# post T = T~[I => T~(J); J => T~(I)];
is
Temp : Contents_Type;
begin
  Temp := T(I);
  T(I) := T(J);
  T(J) := Temp;
end Swap_Elements_strong;
```

Fortunately, SPARK provides "array update notation" that allows us to capture the desired condition in a much more compact manner.

For You To Do

Pause the lecture and complete the following exercise (FYTD #2)...

- We claimed that the Swap_Elements_weak procedure did not have a contract strong enough to specify our intentions for the behavior of the procedure. Introduce an error into the code that will violate our intention for the procedure (i.e., we only want to swap the indicated elements and leave everything else untouched) but will actually yield a procedure body that satisfies the weak contract. Use Bakar Kiasan to confirm that the contract is verified.
Kiasan Explores all Paths

This example illustrates how Kiasan systematically explores all paths through a method...

```
procedure Fault_Integrator(Fault_Found : in Boolean;
                          Trip     : in out Boolean;
                          Counter  : in out Integer);
  --# derives Trip from Fault_Found, Trip, Counter &
  --#         Counter from Fault_Found, Counter;
is begin
  if Fault_Found then
    Counter := Counter + Up_Rate;
    if Counter >= Upper_Limit then
      Trip := True; Counter := Upper_Limit;
      end if;
  else
    Counter := Counter - Down_Rate;
    if Counter <= Lower_Limit then
      Trip := False; Counter := Lower_Limit;
      end if;
  end if;
end Fault_Integrator;
```

Kiasan Explores all Paths

This example illustrates how Kiasan systematically explores all paths through a method...

```
procedure Fault_Integrator(Fault_Found : in Boolean;
                          Trip     : in out Boolean;
                          Counter  : in out Integer);
  --# derives Trip from Fault_Found, Trip, Counter &
  --#         Counter from Fault_Found, Counter;
is begin
  if Fault_Found then
    Counter := Counter + Up_Rate;
    if Counter >= Upper_Limit then
      Trip := True; Counter := Upper_Limit;
      end if;
  else
    Counter := Counter - Down_Rate;
    if Counter <= Lower_Limit then
      Trip := False; Counter := Lower_Limit;
      end if;
  end if;
end Fault_Integrator;
```

Kiasan discovers the four paths in this procedure
This example illustrates how Kiasan systematically explores all paths through a method...

```plaintext
if Fault_Found then
    Counter := Counter + Up.Rate;
    if Counter >= Upper.Limit then
        Trip := True; Counter := Upper.Limit;
    end if;
else
    Counter := Counter - Down.Rate;
    if Counter <= Lower.Limit then
        Trip := False; Counter := Lower.Limit;
    end if;
end if;
end Fault_Integrator;
```

Path 0

Path 1

Test of outermost conditional is true.
Kiasan Explores all Paths

This example illustrates how Kiasan systematically explores all paths through a method...

```plaintext
if Fault_Found then
  Counter := Counter + Up_Rate;
  if Counter >= Upper_Limit then
    Trip := True; Counter := Upper Limit;
    end if;
  else
    Counter := Counter - Down_Rate;
    if Counter <= Lower_Limit then
      Trip := False; Counter := Lower Limit;
      end if;
    end if;
```

Path 2: Test of outermost conditional is false.

Path 3: Test of outermost conditional is false.
Enhancing Contract Functionality

Existing SPARK is limited to first-order logic. Any "helper functions" are treated as uninterrupted functions.

```plaintext
--# (Response = ... -> (    
--#   (for all I in Item_List_Index_Type ->    
--#     (ID /= Item_List~(I).ID)) and then    
--#   (for some I in Item_List_Index_Type ->    
--#     (ID = Item_List(I).ID)))
```

Bakar Kiasan adds support for functions in contracts whose semantics is specified directly using SPARK functions (guaranteed "pure")

```plaintext
--# post Invariant(...)    
--# and then    
--# (Response = ... <->    
--#   contains(ID, ...) = False)    
--# and then
```

Some invariants are extremely cumbersome to specify under these limitations.

Executable Contracts

Kiasan supports contracts with executable helper functions...

```plaintext
procedure BubbleSort(A : in out Vector);    
--# derives A from A;    
--# post isSorted(A) and then isPerm(A, A~);
```

```plaintext
function isSorted(Z : Vector)    
return Boolean is    
  B : Boolean;    
begin    
  for I in Index_Type range    
  Index_Type'First ..    
  Index_Type'Last - 1 loop    
  B := Z(I) <= Z(I + 1);    
  exit when not B;    
  end loop;    
  return B;    
end isSorted;
```

```plaintext
function isPerm(A, B : Vector)    
return Boolean is    
  Result : Boolean := True;    
begin    
  for I in Index_Type loop    
  if I = Index_Type'First or else    
  A(I - 1) /= A(I) then    
  Result := Num_Repetitions(A, A(I))    
  := Num_Repetitions(B, A(I));    
  end if;    
  exit when not Result;    
  end loop;    
  return Result;    
end isPerm;
```
For You To Do

Pause the lecture and complete the following exercises (FYTD #3)...

- (a) Starting with the code fytd-03-code.ada, add a helper function
  
  ```ada
  function isSet(Z : Vector) return Boolean
  end function
  ``

  that returns true when the input array Z has no duplicate elements (represents a set). Add a pre-condition to the `BubbleSort` procedure that calls the helper function to ensure that its input has no duplicate elements.

- (b) Continuing with the code above, write a procedure `InsertionSort` sort that implements the standard insertion sort algorithm. Use contracts similar to those used in `BubbleSort` to verify the correctness of your implementation.

Kiasan Methodology

- Checking in IDE
  - start with small bounds
  - incrementally check
  - scenario and test case generation for violations

- More exhaustive checking
  - higher bounds with overnight/parallel checking
  - Kiasan tells you if coverage criteria has been met

- Code understanding
  - select any block of code, Kiasan generates flow scenarios giving path coverage

- Test case generation for regression testing
  - automatically generate tests (full MCDC coverage) from code

- Add loop invariants for complete verification
Summary

- Introduced the concept of symbolic execution – executing a program with symbolic values instead of concrete values and propagating constraints.
- Illustrated how symbolic execution can be used to check SPARK contracts and to generate examples of input/output along procedure paths.
- Illustrated multiple ways of leveraging the information produced by symbolic execution
  - Pre/post-state visualization
  - Test case generation

Acknowledgements

- Some of the material in this lecture is based on the following paper...