Bakar Kiasan: Flexible Contract Checking for Critical Systems using Symbolic Execution

SAnToS Laboratory, Kansas State University, USA

santoslab.org

Funding

NSF AFOSR AIR FORCE OFFICE OF SCIENTIFIC RESEARCH Rockwell Collins

John Hatcliff
Jason Belt
Patrice Chalin
William Deng (Google Inc.)
David Hardin (Rockwell Collins Inc.)
Robby

What is SPARK?

One of the best available commercially supported frameworks for code-level development of safety critical systems

- Developed by Praxis High Integrity Systems
  - http://www.praxis-his.com/sparkada/
- Marketed in a partnership with AdaCore
  - http://www.adacore.com/
  - integrated with AdaCore GnatPro compiler and integrated development environment
- SPARK tools are GPL open source
  - Examiner is implemented in SPARK

What is SPARK?

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

SPARK Contracts

SPARK includes annotations for assertions, pre/post-conditions that can be used to express *software contracts*

Simple pre-condition with existential quantification…

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

Post-condition constraining return value to inputs using universal quantification…

"SPARK Examiner with Run-time Checker…", p. 22

What is SPARK?

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 caring 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 Praxis and NSA
  - http://www.adacore.com/home/products/sparkpro/tokeneer/
- 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] development of certified embedded security devices

…this talk will emphasize experiences with Rockwell Collins on a DoD-funded research project that involved developing a prototype of high-speed crypto-controller

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

```
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;
```

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…

```
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) .
        ->
C1:    index__first >= index__first .
C2:    index__first <= index__last .
C3:    not false .
C4:    for_all(m_: integer, ((m_ >= index__first) and (m_ <=
           index__first)) -> (element(a, [m_]) <> x)) .
C5:    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))) .
C6:    x >= integer__first .
C7:    x <= integer__last .
C8:    index__first >= index__first .
C9:    index__first <= index__last .
C10:   index__first >= index__first .
C11:   index__first <= index__last .
```

Simplifier
decision procedure package that simplifies and tries to automatically prove verification conditions

4 of 7 VCs proved, the rest are simplified

Feed Into the Proof Checker

1 of 3 Remaining Verification Conditions…

```
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:   index__size >= 0 .
H11:   index__base__first <= index__base__last .
H12:   index__base__first <= 1 .
H13:   index__base__last >= 10 .
       ->
C1:    not for_some(m_ : integer, m_ >= 1 and m_ <= 10 and element(a, [m_]) = x)
           .
```

Proof Checker

semi-automated framework for manually caring out proof steps to discharge remaining verification conditions

Manually carry out proofs

Feed Into the Proof Checker

Proofs 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.
 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 = _2 by _1 <> _2 using negation(1).
 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 neg
 y
 replace c # 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;
```

Places where VCs need to be discharged in proof checker

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

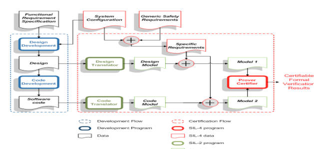
Rockwell Collins Workaround

Customer

Prover Model Checker

PROVER®

engineering a safer world™



SPARK

Detailed contract specs from customer in Z

Expressed design in Prover modeling language and checked for array size = 3

Translated to SPARK code (no contracts)

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 almost everyone from using it – We want to change that!

Our Work

Better integration of contract checking into SPARK developer workflows

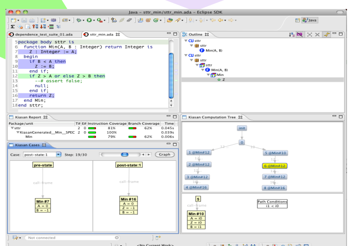
Functional Contracts (pre/post, assertions)

Developers

```
procedure Add(E: Element_Type)
 --# pre Member_Count < Size_Range'Last;
 --# post (isMember(E, Elem_Array, Member_Count) -> (
 --#       Elem_Array = Elem_Array~ and Member_Count = Member_Count~))
 --#   and (not isMember(E, Elem_Array, Member_Count) -> (
 --#       Elem_Array = Elem_Array~[Member_Count => E]
 --#       and Member_Count = Member_Count~ + 1));
```

Automatic Checking

Kiasan
Symbolic Execution Engine



SPARK Eclipse IDE

Use symbolic execution, not just for bug-finding or test-case generation, but for contract checking that complements the existing facilities of SPARK

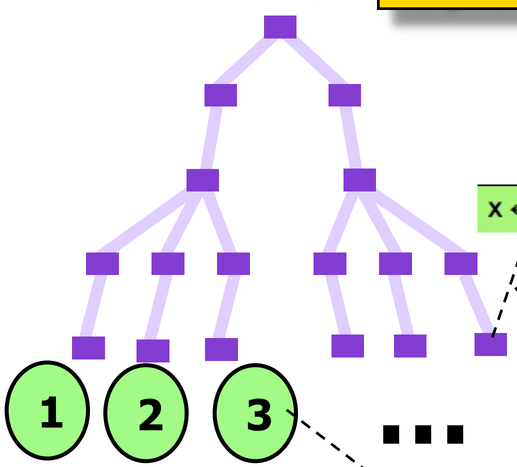
Themes

- Highly automated; payback is on par with investment;
- Meaningful checking without loop invariants (bounded loop unfolding)
- When verification engine processes code, communicate "knowledge" gained to developer
- Keep focus on the source code level, instead of using a separate formalism like FDL
- SPARK supports only declarative contracts; we want to support both declarative and executable specs
- Connect to other quality assurance techniques; phrase results in terms of what developers already understand

Visualizing Properties of Paths

Verification *(bounded, in some cases)*

Kiasan exhaustively explores all paths through a procedure



paths

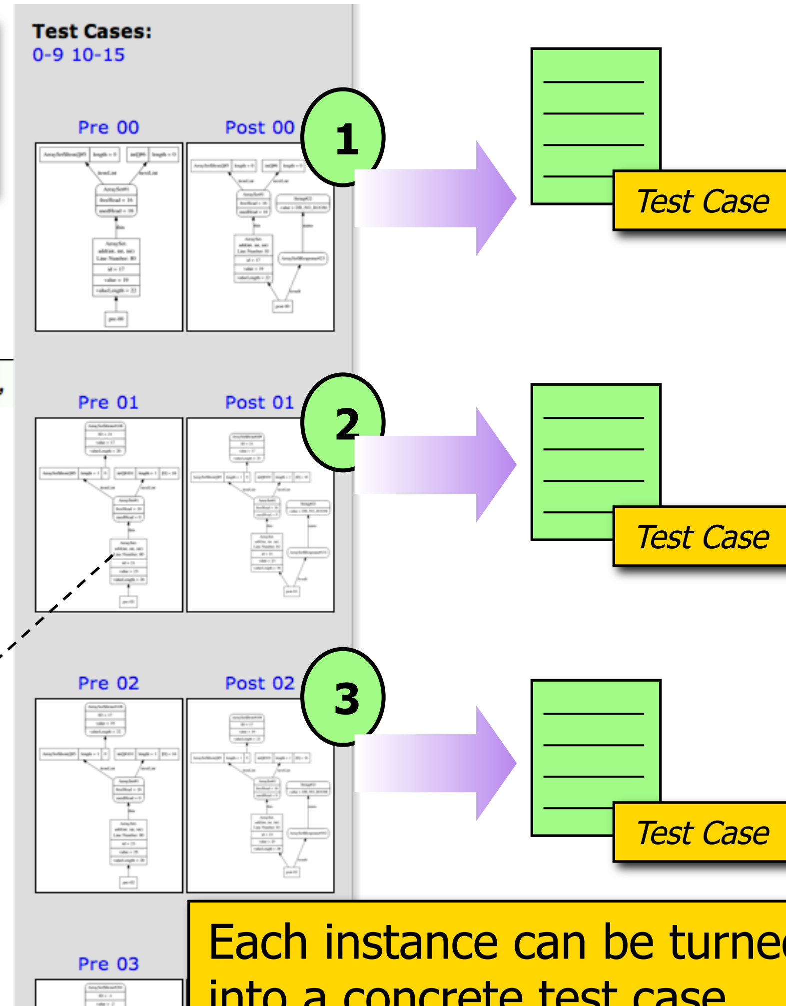
constraints

$x \leftarrow \alpha$, $y \leftarrow \beta$, $z \leftarrow \pi$, $\Phi = \{\pi = \alpha + \beta,$

check if post-condition is implied

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.

Examples / Tests

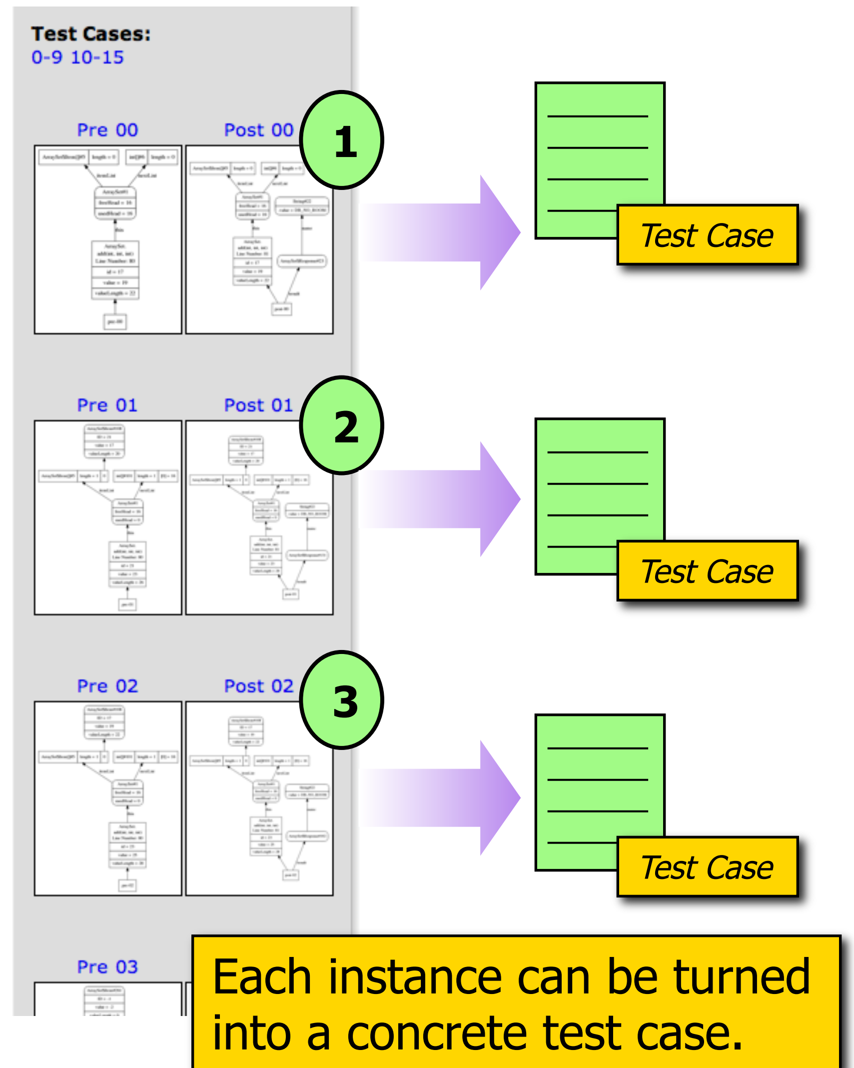


Each instance can be turned into a concrete test case.

Kiasan Output

Why provide examples/tests if we are verifying?

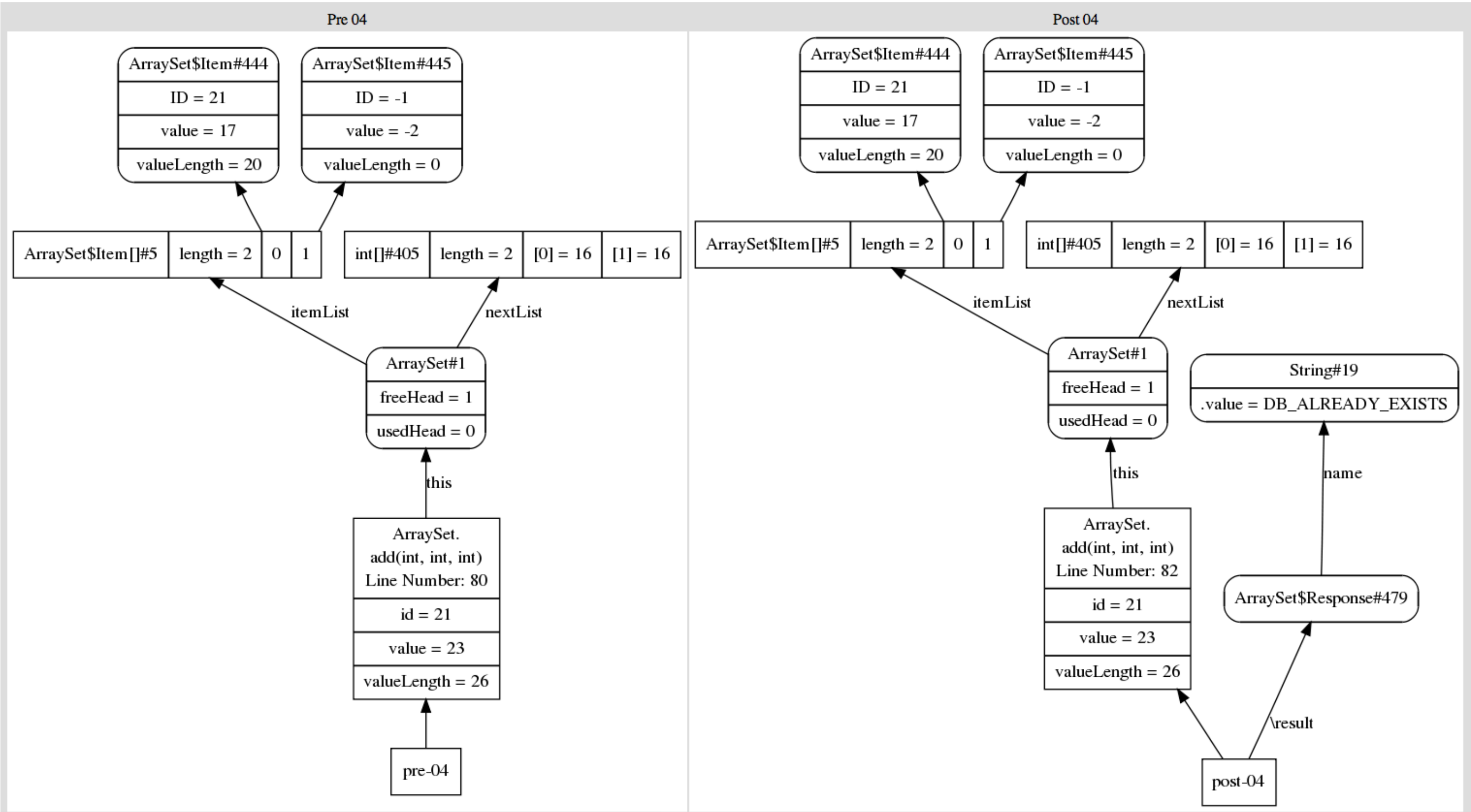
- Tests provide “evidence” to people not familiar with formal methods that something “interesting” is happening in the tool
- When a bug is found along one path, the test provides a counter-example illustrating the bug
- Kiasan’s exhaustive exploration automatically yields test suites with very high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



NB: Jeff Joyce (DO-178C FM) – explain formal method contribution in terms of coverage / tests

Kiasan Output

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



Input -- program state at start of path

Output -- program state at end of path

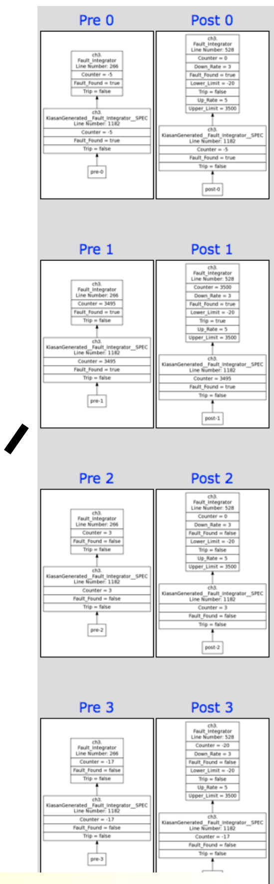
Early Payback

Kiasan does not need contracts to provide useful semantic information – immediately can explore to look for possible run-time exceptions

```
procedure Fault_Integrator(Fault_Found : in Boolean;
                           Trip : in out Boolean;
                           Counter : in out Integer)
is
```

```
  if Fault_Found then
    Counter := 
    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;
```

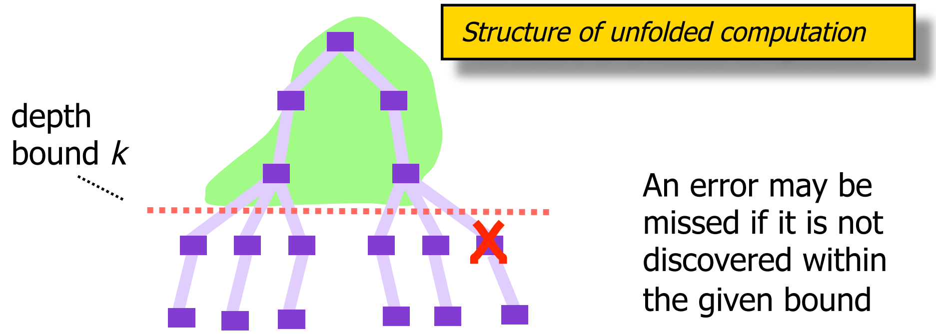
Fully Covered Line



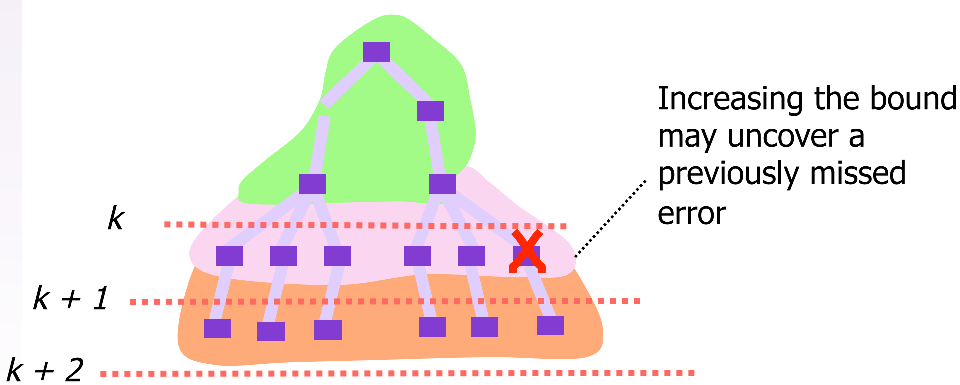
This procedure has four paths, so Kiasan provides four “examples”.

Controlling Cost/Coverage

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



Increasing *k* increases coverage & cost



- 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

Package Name: *ArraySet*

Report Rendered: Mon Apr 18 12:36:56 PDT 2011, by Sireum/Kiasan for SPARK v0.1.20100729

Branches Covered For Tests: 23/24 (95.83%)

Branches Covered For Package: 23/69 (33.33%)

Summary of coverage information

Method Covered:

Percent Ratio

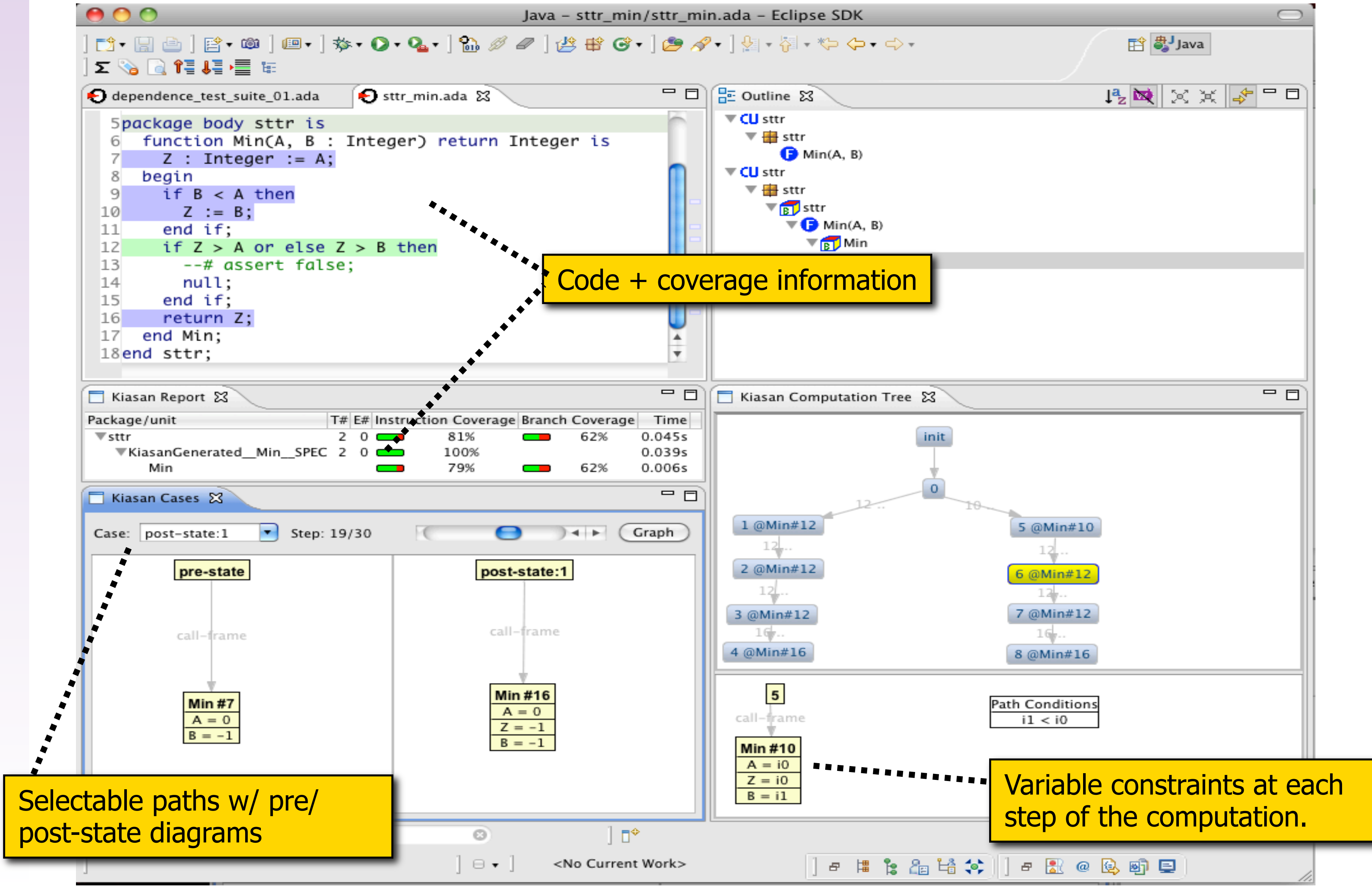
Method	T	E	Instruction Coverage	Branch Coverage	Time
Add	34	0	94.12%	83.33%	0.016s
Delete	10	0	100%	100%	0.050s
Get_Value	10	0	100%	100%	0.020s

Source Code:

```
with ArraySetDefs;
with ArraySetUnsigned;
use type ArraySetDefs.ID_Type;

package body ArraySet
--# own State is Item_List, Next_List, Free_Head, Used_Head;
is
  Max_Items : constant := 3; -- belt: originally 16

  type Item_Type is record
    ID          : ArraySetDefs.ID_Type;
    Value        : ArraySetDefs.Value_Type;
  end record;

  subtype Item_List_Index_Type is ArraySetUnsigned.Word range 0 .. Max_Items - 1;
  subtype Link_Type is ArraySetUnsigned.Word range 0 .. Max_Items;

  type Item_List_Type is array (Item_List_Index_Type) of Item_Type;
  type Next_List_Type is array (Item_List_Index_Type) of Link_Type;

  Item_List : Item_List_Type;
  Next_List : Next_List_Type;

  Terminator : constant := Link_Type'Last;
  Inf_Length : constant := Max_Items + 1;
  Free_Head : Link_Type;
  Used_Head : Link_Type;

  ------------------Invariant

  ---
  --- @param head the index of the start of a list (expected to be either
  ---           Free_Head or Used_Head)
  --- @return the number of elements reachable from head, or Inf_Length
  ---        if the list is cyclic.
  ---
  function Size_Of_List(head : Link_Type) return ArraySetUnsigned.Word
    --# global in Next_List;
  is
    Cursor : Link_Type;
    Result : ArraySetUnsigned.Word := 0;
  begin
    Cursor := head;
    while Cursor /= Terminator and Result < Inf_Length loop
      Result := Result + 1;
```

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



Kiasan Methodology

- Checking in IDE
  - start with small bounds
  - incrementally check / verify
  - 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