KSU / AdaCore / Altran Praxis Discussions on Enhancing Information Flow Framework of SPARK

http://SantosLab.org

Kansas State University

John Hatcliff
Torben Amtoft
Zhi Zhang
Simon Ou
Andrew Cousino

Princeton University

Josiah Dodds
Andrew Appel
Lennart Beringer

Rockwell Collins

David Hardin

Funding: *this work supported by Air Force Office of Scientific Research (Dr. Robert Herklotz) and Rockwell Collins*

Goals

Make SPARK the language of choice for certified embedded information assurance systems by adding "developer support" and enhanced information flow specification verification

- Phase I:
  - IDE support for information flow browsing & querying based on existing SPARK information flow framework
  - Require no changes to underlying semantics and annotations
- Phase II:
  - High-level information flow policy language for specifying domains, allowable flows between domains
  - Requires minimal changes to underlying semantics and annotations
- Phase III:
  - Conditional information flow / conditional declassification
  - Requires moderate changes to underlying semantics and annotations
- Phase IV
  - Precise array frame conditions and information flow
  - Requires substantial changes to underlying semantics and annotations

Phase I – Technical Background

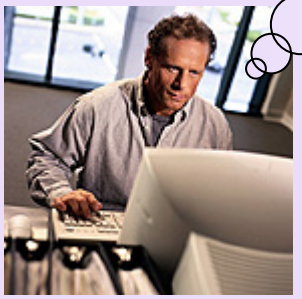
- Two algorithms for reasoning about information flow within the existing derives framework
  - Bergeretti and Carre "matrix" approach used in Examiner
  - Conventional "program dependence graph" data-dependence / control-dependence used in KSU info flow browsing components
- Both of these compute same results and have the same correctness properties
- Neither of these directly support more precise reasoning needed for conditional flow or precise array flows

Phase I – Flow Browsing

- Intra-procedural forwards/backwards slice
- Inter-procedural forwards/backwards slice
- Developer selects
  - Program statement
  - Procedure parameter / global
  - Variable in a derives annotation
- Tool calculates
  - Incoming flows or outgoing flows (based on choice of forward/backward)

Basic Info Flow Browsing

Where does information flowing into RESPONSE come from?



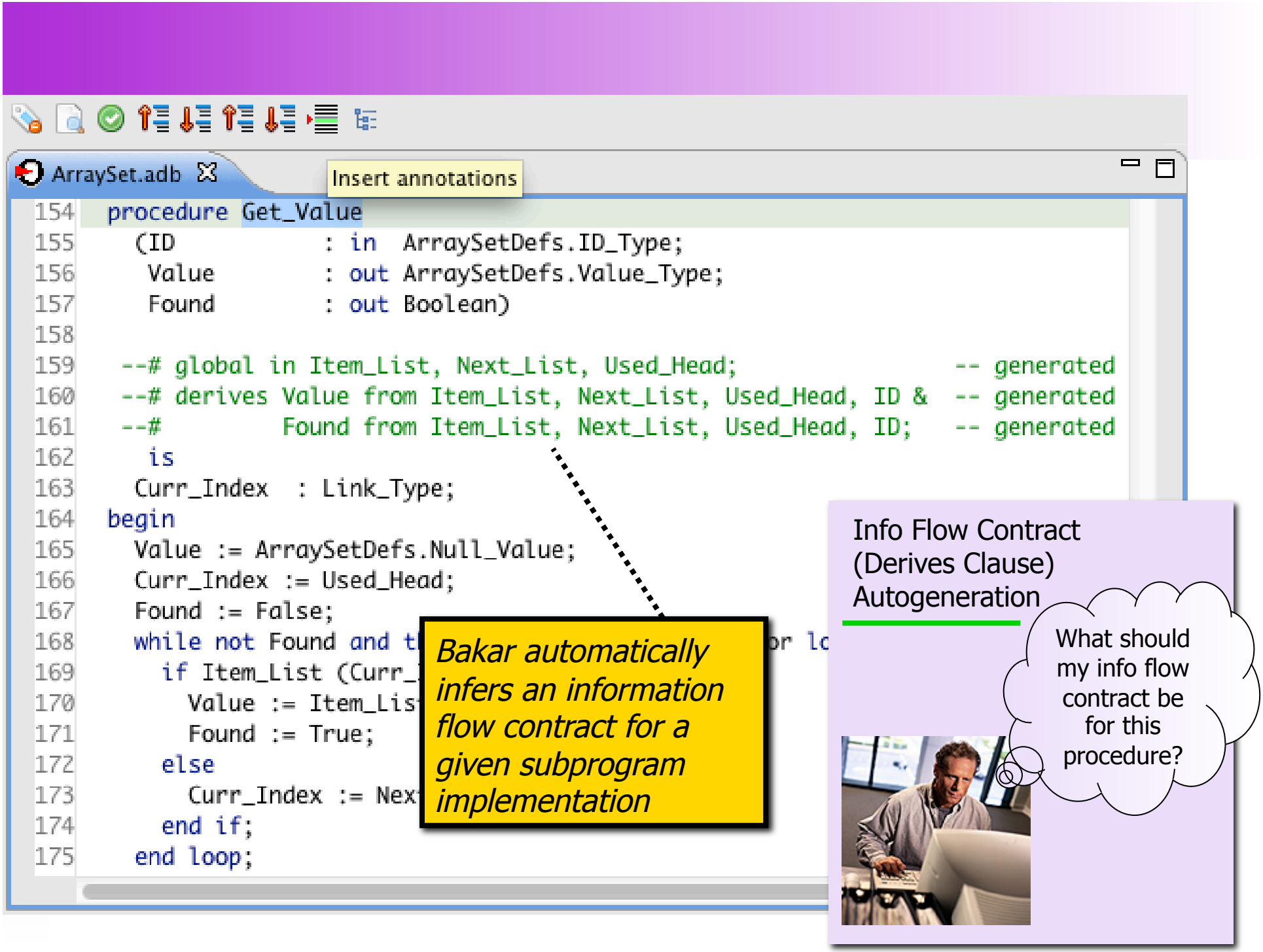
Developer highlights RESPONSE and selects "Backward Flow" visualization.

Bakar automatically marks the lines and input vars from which information flows into Response.

```
180  procedure Add
181    (ID            :
182     Value         :
183     Response      :
184    --# global in out Item_List, Next_List, Free_Head, Used_Head;
185    --# derives Item_List from ID, Value, Item_List, Next_List,
186    --#             Free_Head, Used_Head &
187    --#           Response, Next_List, Free_Head, Used_Head from
188    --#             ID, Item_List, Next_List, Free_Head, Used_Head;
189  is
190    Curr_Index : Link_Type;
191    Temp_Value : ArraySetDefs.Value_Type;
192    Found : Boolean;
193  begin
194    if ID /= ArraySetDefs.Null_ID then
195      if Free_Head /= Terminator then
196        Get_Value(ID, Temp_Value, Found);
197        if not Found then
198          Curr_Index := Free_Head;
199          Free_Head := Next_List (Free_Head);
200          Item_List (Curr_Index).ID := ID;
             Item_List (Curr_Index).Value :=  Value;
             Next_List (Curr_Index) := Used_Head;
             Used_Head              := Curr_Index;
             Response := ArraySetDefs.DB_Success;
        else
          Response := ArraySetDefs.DB_Already_Exists;
        end if;
      else
        Response := ArraySetDefs.DB_No_Room;
      end if;
    else
      Response := ArraySetDefs.DB_Input_Check_Fail;
    end if;
  end Add;
```

Phase I – Derives Support

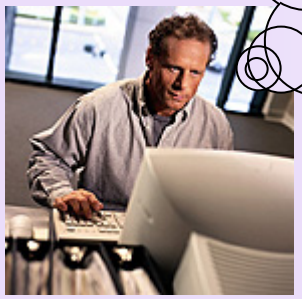
- Automatically calculate and insert derives clauses into code
  - Per procedure
  - Per package
  - Entire program
- Derives clause refactoring
  - Allow user to select if changes made in derives lower in the call tree propagate to derives contracts for procedures higher in call tree


ArraySet.adb
Insert annotations
154 procedure Get_Value
155 (ID : in ArraySetDefs.ID_Type;
156 Value : out ArraySetDefs.Value_Type;
157 Found : out Boolean)
158
159 --# global in Item_List, Next_List, Used_Head; -- generated
160 --# derives Value from Item_List, Next_List, Used_Head, ID & -- generated
161 --# Found from Item_List, Next_List, Used_Head, ID; -- generated
162 is
163 Curr_Index : Link_Type;
164 begin
165 Value := ArraySetDefs.Null_Value;
166 Curr_Index := Used_Head;
167 Found := False;
168 while not Found and t
169 if Item_List (Curr_
170 Value := Item_Lis
171 Found := True;
172 else
173 Curr_Index := Nex
174 end if;
175 end loop;
Bakar automatically infers an information flow contract for a given subprogram implementation
Info Flow Contract (Derives Clause) Autogeneration
What should my info flow contract be for this procedure?

Phase I – Security Domain Highlighting

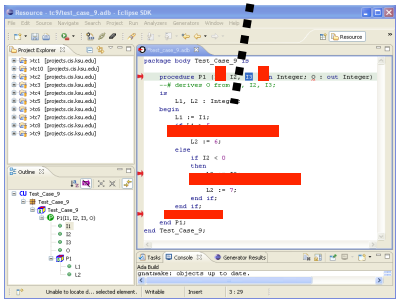
- Allow colors to be associated with declared security domains
- Developer can request visualization of program statements/variables that manipulate data from selected domains

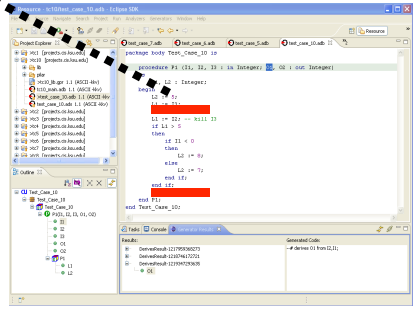
MLS Flow Markup

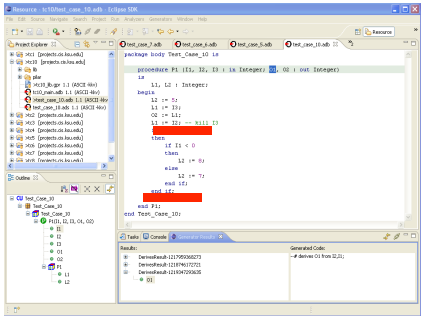


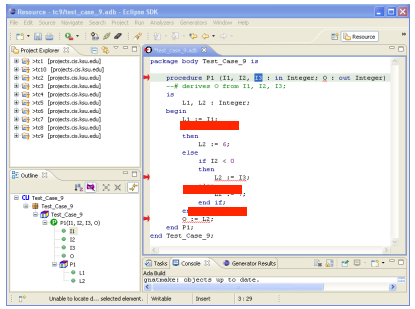
Show me all variables and statements in my program that manipulate top secret data

Given an initial tagging of input variables according to security level, SpAda propagates and marks up source code to indicate MLS level of statement/variable









Principals	***Security Lattice***	
A1	**Top Secret**	
A2	**High**	
A3	**Low**	
…	**Public**	
An		

MLS security lattice with color mark-up tags

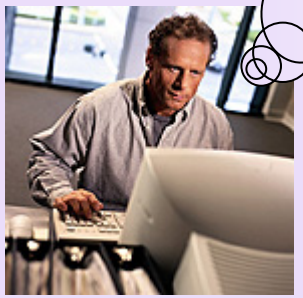
(not yet implemented -- target May 2009)

Phase I - Named Queries

Provide a simple information flow "scripting language" (inspired by CodeSurfer / Indus)

- Allow developers/auditors to name/save common information flow queries
- Can be used to support declaration of high-level information flow policies
  - Certain flow never overlap / intersect
  - Certain program features and contained within specified flows
- Can also be used to support partitioning of code by criticality levels (e.g., code that provides safety-critical calculations vs code that performs logging)

Info Flow Contract (Derives Clause) Autogeneration



I'd like to name and save my common queries…

Bakar provides a simple query language that be used to define common queries/policies

Can reuse the results of one query in other

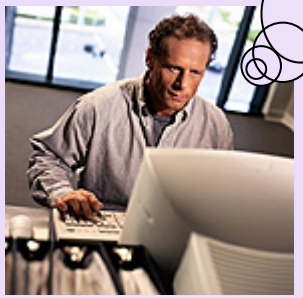
mailbox-rv3.ada | project.qry

```
295  procedure WRITE_OUTPUT_1(Data : in INTEGER) is
296  begin
297    INTEGER_OUTPUT_1_DATA := Data;
298  end WRITE_OUTPUT_1;
299
300  procedure MACHINE_STEP
301  is
302      DATA_0 : INTEGER;
303      DATA_1 : INTEGER;
304    begin
305      if INPUT_0_READY and OUTPUT_1_CONSUMED then
306        DATA_0 := READ_INPUT_0;
307        NOTIFY_INPUT_0_CONSUMED;
308        WRITE_OUTPUT_1(DATA_0);
309        NOTIFY_OUTPUT_1_READY;
310      end if;
311      if INPUT_1_READY and OUTPUT_0_CONSUMED then
312        DATA_1 := READ_INPUT_1;
313        NOTIFY_INPUT_1_CONSUMED;
314        WRITE_OUTPUT_0(DATA_1);
315        NOTIFY_OUTPUT_0_READY;
316      end if;
317  end MACHINE_STEP;
```

QueryView

Query Name	Value	Expression
pathA_1	38	slice FORWARD with {body Mailbox.MACHINE_STEP(Mailbox.INTEGER_INPUT_1_DATA)}
pathB_0	38	slice FORWARD with {body Mailbox.MACHINE_STEP(Mailbox.INTEGER_INPUT_0_DATA)}
pathA_0	68	slice BACKWARD with {body Mailbox.MACHINE_STEP(Mailbox.INTEGER_OUTPUT_0_DATA)}
pathB_1	75	slice BACKWARD with {body Mailbox.MACHINE_STEP(Mailbox.INTEGER_OUTPUT_1_DATA)}
pathA1_0	17	chop FULL from {body Mailbox.MACHINE_STEP(Mailbox.INTEGER_INPUT_1_DATA)} to {body
pathB0_1	17	chop FULL from {body Mailbox.MACHINE_STEP(Mailbox.INTEGER_INPUT_0_DATA)} to {body
A_B	true	assert (pathA_1 DISJOINT pathB_0)
AintersectA	18	pathA_1 INTERSECTION pathA_0
AintersectB		pathA_1 INTERSECTION pathB_0
A1_0_B0_1	true	assert (pathA1_0 DISJOINT pathB0_1)

Info Flow Contract (Derives Clause) Autogeneration



I'd like to name and save my common queries…

Bakar provides a simple query language that be used to define common queries/policies

mailbox-rv3.ada | project.qry

```
295  procedure WRITE_OUTPUT_1(Data : in INTEGER) is
296  begin
297    INTEGER_OUTPUT_1_DATA := Data;
298  end WRITE_OUTPUT_1;
299
300  procedure MACHINE_STEP
301  is
302      DATA_0 : INTEGER;
303      DATA_1 : INTEGER;
304    begin
305      if INPUT_0_READY and OUTPUT_1_CONSUMED then
306        DATA_0 := READ_INPUT_0;
307        NOTIFY_INPUT_0_CONSUMED;
308        WRITE_OUTPUT_1(DATA_0);
309        NOTIFY_OUTPUT_1_READY;
310      end if;
311      if INPUT_1_READY and OUTPUT_0_CONSUMED then
312        DATA_1 := READ_INPUT_1;
313        NOTIFY_INPUT_1_CONSUMED;
314        WRITE_OUTPUT_0(DATA_1);
315        NOTIFY_OUTPUT_0_READY;
316      end if;
317  end MACHINE_STEP;
```

QueryView

Query Name	Value	Expression
pathA_1	38	slice FORWARD with {body Mailbox.MACHINE_STEP(Mailbox.INTEGER_INPUT_1_DATA)}
pathB_0	38	slice FORWARD with {body Mailbox.MACHINE_STEP(Mailbox.INTEGER_INPUT_0_DATA)}
pathA_0	68	slice BACKWARD with {body Mailbox.MACHINE_STEP(Mailbox.INTEGER_OUTPUT_0_DATA)}
pathB_1	75	slice BACKWARD with {body Mailbox.MACHINE_STEP(Mailbox.INTEGER_OUTPUT_1_DATA)}
pathA1_0	17	chop FULL from {body Mailbox.MACHINE_STEP(Mailbox.INTEGER_INPUT_1_DATA)} to {body
pathB0_1	17	chop FULL from {body Mailbox.MACHINE_STEP(Mailbox.INTEGER_INPUT_0_DATA)} to {body
A_B	true	assert (pathA_1 DISJOINT pathB_0)
AintersectA	18	pathA_1 INTERSECTION pathA_0
AintersectB		pathA_1 INTERSECTION pathB_0
A1_0_B0_1	true	assert (pathA1_0 DISJOINT pathB0_1)

Goals

Make SPARK the language of choice for certified embedded information assurance systems by adding "developer support" and enhanced information flow specification verification

- Phase I:
  - IDE support for information flow browsing & querying based on existing SPARK information flow framework
  - Require no changes to underlying semantics and annotations
- Phase II:
  - High-level information flow policy language for specifying domains, allowable flows between domains
  - Requires minimal changes to underlying semantics and annotations
- Phase III:
  - Conditional information flow / conditional declassification
  - Requires moderate changes to underlying semantics and annotations
- Phase IV
  - Precise array frame conditions and information flow
  - Requires substantial changes to underlying semantics and annotations

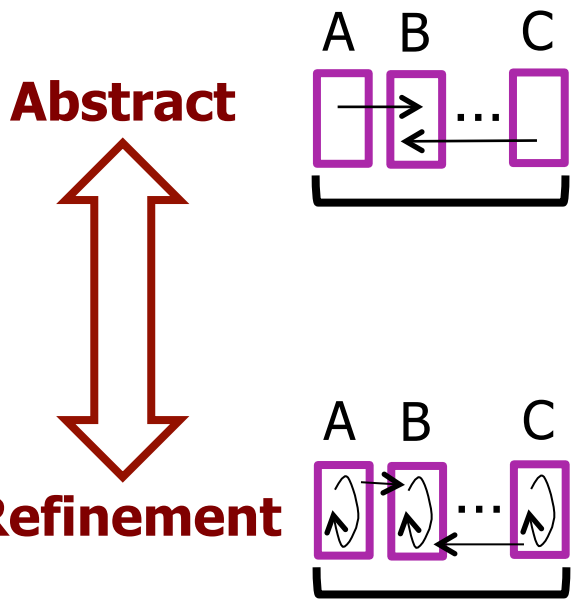
Contract Language (1)

- Domain represent either
  - an integrity level, which constructs a lattice, or
  - a system's partition to show noninterference between them

Contract Language (1)

```
p ::=
      | Domain A, B, C, …
      | Ordering A > B, B > C, …        (* can be null {} *)
      | x … in A, …
      | Declassify (exp, A → C), …     (* can be null {} *)
```

```
p ::=
      | Derives o From i On guard
      | Derives Forall j With p(j) h[exp (j)] From i On guard
```



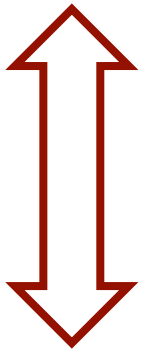
It expresses information contract both within one domain and across-domain

In abstract policy, **Declassify()** constraints the mount of information that can be released through both the data flow (exp) and control flow (guard) mentioned in refinement policy.

Contract Language (1) - case 1

```
p ::=
      | Domain A, B, C, …
      | Ordering A > B, B > C, …
      | x … in A, …
      | Declassify { }
```

Abstract



Refinement

```
p ::=
      | Derives o From i On guard
      | Derives Forall j With p(j) h[exp (j)] From i On guard
```

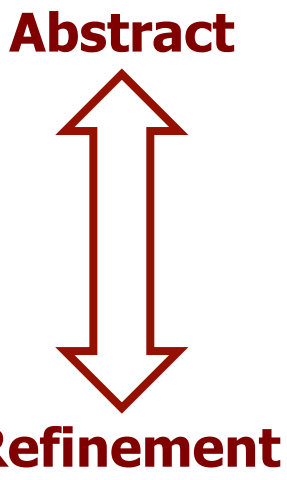
It expresses information contract both within one domain and across-domain

Isolation: tag data with different integrity levels (for example MILS), and enforce that data with higher integrity level can never be released into lower data. However, it allows data with lower integrity level flowing into higher data, for example: h := h + l.

Contract Language (1) - case 2

```
p ::=
      | Domain A, B, C, …
      | Ordering   { }
      | x … in A, …
      | Declassify { }
```

```
p ::=
      | Derives o From i On guard
      | Derives Forall j With p(j) h[exp (j)] From i On guard
```



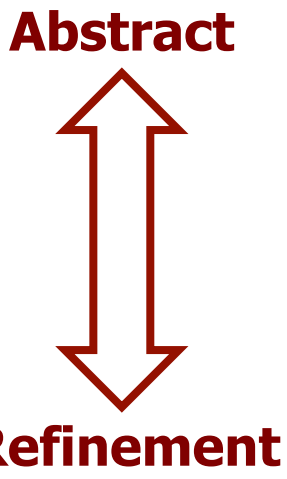
It expresses information contract both within one domain and across-domain

Isolation: divide the system into several partitions, and enforce the noninterference between these partitions

Contract Language (1) - case 3

```
p ::=
      | Domain A, B, C, …
      | Ordering A > B, B > C, …
      | x … in A, …
      | Declassify { exp: A → C }
```

```
p ::=
      | Derives o From i On guard
      | Derives Forall j With p(j) h[exp (j)] From i On guard
```



It expresses information contract both within one domain and across-domain

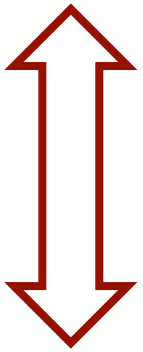
Declassify can express both (1) the information release channel from high to low and (2) explicitly specified information channel between two non-interference partitions. No restriction on domains appearing in Declassify, they can be element of a lattice or just partitions.

Contract Language (1) - case 4

```
p ::=
      | Domain A, B, C, …
      | Ordering   { }
      | x … in A, …
      | Declassify { exp: A → C }
```

```
p ::=
      | Derives o From i On guard
      | Derives Forall j With p(j) h[exp (j)] From i On guard
```

Abstract



Refinement

It expresses information contract both within one domain and across-domain

Declassify can express both (1) the information release channel from high to low and (2) explicitly specified information channel between two non-interference partitions. No restriction on domains appearing in Declassify, they can be element of a lattice or just partitions.

Security (1)

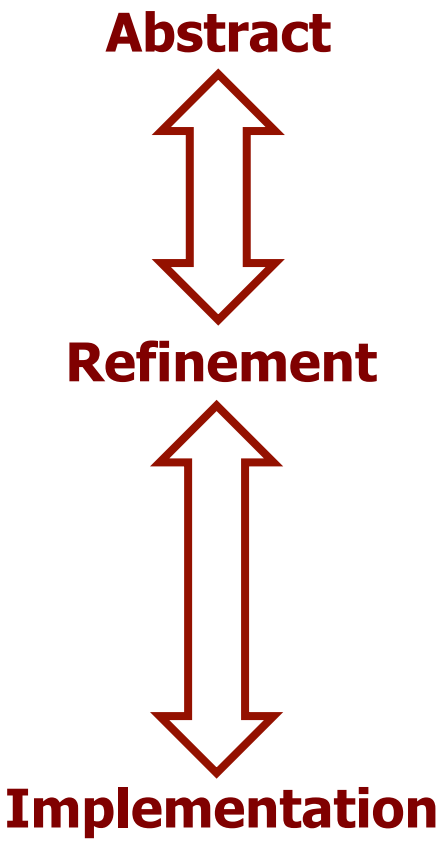
Two-level required contract

```
p ::=
      | Domain A, B, C, ...
      | Ordering A > B, B > C, ...      (* can be null {} *)
      | x ... in A, ...
      | Declassify (exp, A → C), ...    (* can be null {} *)
```

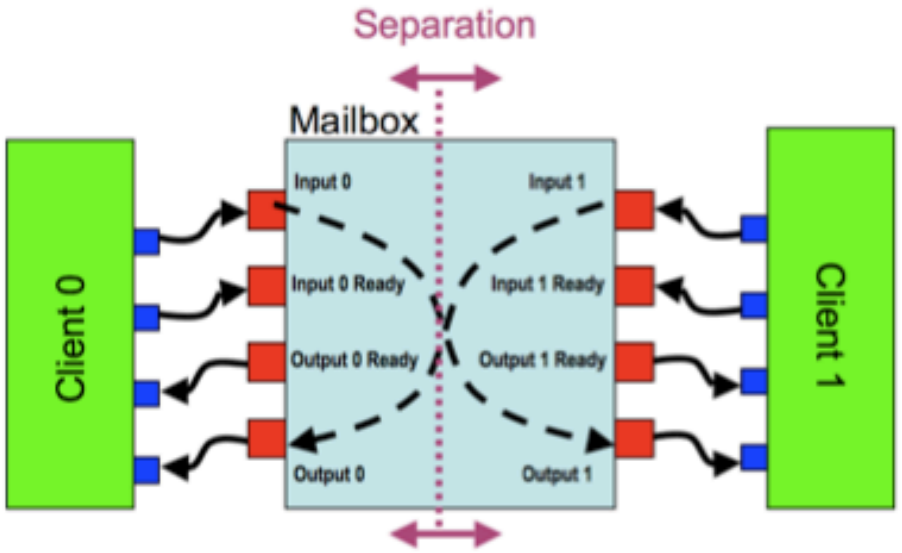
```
p ::=
      | Derives o From i On guard
      | Derives Forall j With p(j) h[exp (j)] From i On guard
```

Actual contract by SIFL algorithm

```
Procedure foo( ... )
  is
  begin
      ...
  end foo;
```



Contract Language (1) - Example 1




```
procedure MACHINE_STEP
-- INFORMATION FLOW CONTRACT (Figure 3)
is D_0, D_1 : CHARACTER;
begin
  if IN_0_RDY and not OUT_1_RDY then
    D_0 := IN_0_DAT;  IN_0_RDY := FALSE;
    OUT_1_DAT := D_0; OUT_1_RDY := TRUE;
  end if;
  if IN_1_RDY and not OUT_0_RDY then
    D_1 := IN_1_DAT;  IN_1_RDY := FALSE;
    OUT_0_DAT := D_1; OUT_0_RDY := TRUE;
  end if;
end MACHINE_STEP;
```

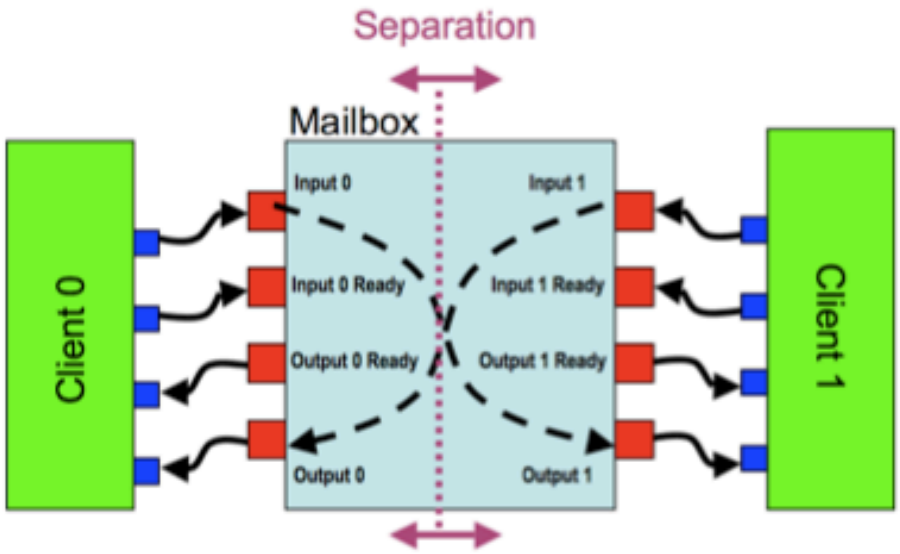
```
Procedure MACHINE_STEP
--# Domain P0In, P0Out, P1In, P1Out;
--# Ordering {} ;
--# IN_0_DAT, IN_0_RDY in P0In ; OUT_0_DAT, OUT_0_RDY in P0Out ;
--# IN_1_DAT, IN_1_RDY in P1In ; OUT_1_DAT, OUT_1_RDY in P1Out ;
--# Declassify (IN_0_DAT, IN_0_RDY: P0In → P1Out), (IN_1_DAT, IN_1_RDY: P1In → P0Out) ;
--# Derives OUT_1_DAT from IN_0_DAT  On (IN_0_RDY && ¬ OUT_1_RDY)
--#                       *          On ¬(IN_0_RDY && ¬ OUT_1_RDY)
--#                       (IN_0_RDY) & (OUT_1_RDY)
--# Derives OUT_0_DAT from IN_1_DAT  On (IN_1_RDY && ¬ OUT_0_RDY)
--#                       *          On ¬(IN_1_RDY && ¬ OUT_0_RDY)
--#                       (IN_1_RDY) & (OUT_0_RDY)
```

Abstract

Refinement

Both declassification policy and Conventional information contract

Contract Language (1) - Example 1

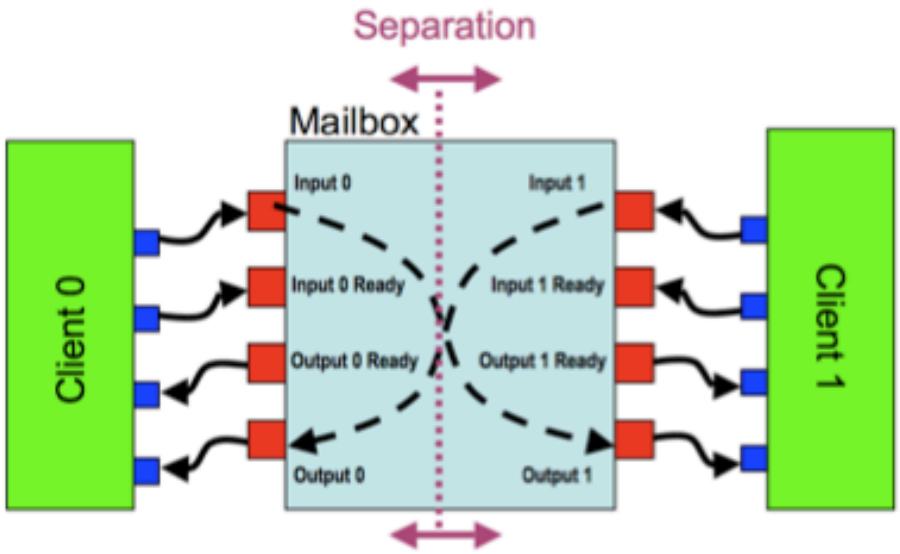



```
procedure MACHINE_STEP
--- INFORMATION FLOW CONTRACT (Figure 3)
is D_0, D_1 : CHARACTER;
begin
  if IN_0_RDY and not OUT_1_RDY then
    D_0 := IN_0_DAT;  IN_0_RDY := FALSE;
    OUT_1_DAT := D_0; OUT_1_RDY := TRUE;
  end if;
  if IN_1_RDY and not OUT_0_RDY then
    D_1 := IN_1_DAT;  IN_1_RDY := FALSE;
    OUT_0_DAT := D_1; OUT_0_RDY := TRUE;
  end if;
end MACHINE_STEP;
```

```
Procedure MACHINE_STEP
--# Domain Channel1, Channel2;
--# Ordering {};
--# IN_0_DAT, IN_0_RDY, OUT_1_DAT, OUT_1_RDY in Channel1;
--# IN_1_DAT, IN_1_RDY, OUT_0_DAT, OUT_0_RDY in Channel2;
--# Declassify {};
--# Derives OUT_1_DAT from IN_0_DAT  On (IN_0_RDY && ¬ OUT_1_RDY)
--#                          *       On ¬(IN_0_RDY && ¬ OUT_1_RDY)
--#                          (IN_0_RDY && ¬ OUT_1_RDY)
--# Derives OUT_0_DAT from IN_1_DAT  On (IN_1_RDY && ¬ OUT_0_RDY)
--#                          *       On ¬(IN_1_RDY && ¬ OUT_0_RDY)
--#                          (IN_1_RDY && ¬ OUT_0_RDY)
```

Partitions' Isolation

Contract Language (1) - Example 1




```
procedure MACHINE_STEP
-- INFORMATION FLOW CONTRACT (Figure 3)
is D_0, D_1 : CHARACTER;
begin
  if IN_0_RDY and not OUT_1_RDY then
    D_0 := IN_0_DAT;  IN_0_RDY := FALSE;
    OUT_1_DAT := D_0; OUT_1_RDY := TRUE;
  end if;
  if IN_1_RDY and not OUT_0_RDY then
    D_1 := IN_1_DAT;  IN_1_RDY := FALSE;
    OUT_0_DAT := D_1; OUT_0_RDY := TRUE;
  end if;
end MACHINE_STEP;
```

{ (IN_0_RDY /\ ¬OUT_1_RDY) ⇒ IN_0_DAT#,
¬(IN_0_RDY /\ ¬OUT_1_RDY) ⇒ OUT_1_DAT#,
(IN_0_RDY /\ ¬OUT_1_RDY)#}

Mailbox

{ OUT_1_DAT # }

{ (IN_1_RDY /\ ¬OUT_0_RDY) ⇒ IN_1_DAT#,
¬(IN_1_RDY /\ ¬OUT_0_RDY) ⇒ OUT_0_DAT#,
(IN_1_RDY /\ ¬OUT_0_RDY)#}

Mailbox

{ OUT_0_DAT # }

Contract Language (1) - Example 1

Actual information flow

{ (IN_0_RDY /\ ¬OUT_1_RDY) ⇒ IN_0_DAT#,
¬(IN_0_RDY /\ ¬OUT_1_RDY) ⇒ OUT_1_DAT#,
(IN_0_RDY /\ ¬OUT_1_RDY)#}

{ OUT_1_DAT# }

Step 1*: Generate data dependence between variables in input and output channels*

Contract Language (1) - Example 1

```
--# Derives OUT_1_DAT from IN_0_DAT    On (IN_0_RDY && ¬ OUT_1_RDY)   &
--#                      OUT_1_DAT  On ¬(IN_0_RDY && ¬ OUT_1_RDY)  &
--#                      (IN_0_RDY)                                 &
--#                      (OUT_1_RDY) ;

--# Derives OUT_0_DAT from IN_1_DAT    On (IN_1_RDY && ¬ OUT_0_RDY)   &
--#                      OUT_0_DAT  On ¬(IN_1_RDY && ¬ OUT_0_RDY)  &
--#                      (IN_1_RDY)                                 &
--#                      (OUT_0_RDY) ;
```

Step 2: *Transfer from user-supplied contract to 2-assertions*

Required information flow

{ (IN_0_RDY /\ ¬OUT_1_RDY) ⇒ IN_0_DAT#,
¬(IN_0_RDY /\ ¬OUT_1_RDY) ⇒ OUT_1_DAT#,
IN_0_RDY#, OUT_1_RDY#} ⇝ ***{ OUT_1_DAT# }***

Actual information flow

{ (IN_0_RDY /\ ¬OUT_1_RDY) ⇒ IN_0_DAT#,
¬(IN_0_RDY /\ ¬OUT_1_RDY) ⇒ OUT_1_DAT#,
(IN_0_RDY /\ ¬OUT_1_RDY)#} ⇝ ***{ OUT_1_DAT# }***

Step 1: *Generate data dependence between variables in input and output channels*

Contract Language (1) – Example 1

```
--# Derives OUT_1_DAT from IN_0_DAT    On (IN_0_RDY && ¬ OUT_1_RDY)   &
--#                      OUT_1_DAT  On ¬(IN_0_RDY && ¬ OUT_1_RDY)  &
--#                      (IN_0_RDY)                                  &
--#                      (OUT_1_RDY) ;

--# Derives OUT_0_DAT from IN_1_DAT    On (IN_1_RDY && ¬ OUT_0_RDY)   &
--#                      OUT_0_DAT  On ¬(IN_1_RDY && ¬ OUT_0_RDY)  &
--#                      (IN_1_RDY)                                  &
--#                      (OUT_0_RDY) ;
```

Step 2: *Transfer from user-supplied contract to 2-assertions*

Required information flow

{ (IN_0_RDY /\ ¬OUT_1_RDY) ⇒ IN_0_DAT#,
¬(IN_0_RDY /\ ¬OUT_1_RDY) ⇒ OUT_1_DAT#,
IN_0_RDY#, OUT_1_RDY#} ⇝ ***{ OUT_1_DAT# }***

Step 3: *2-assertions inference*

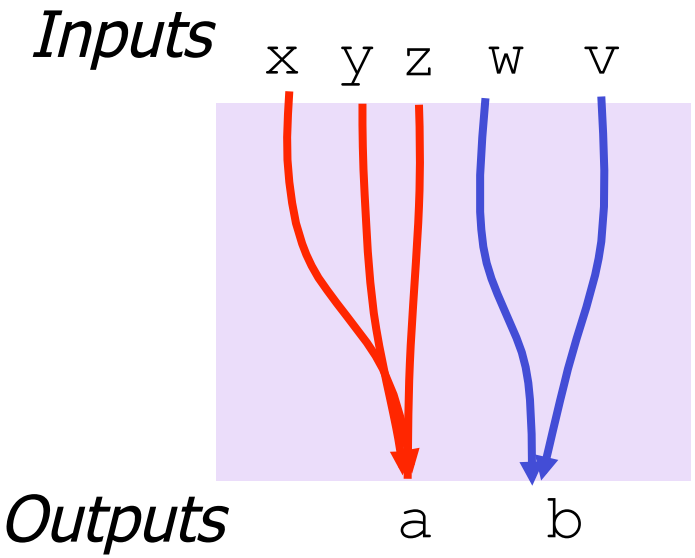
Actual information flow

{ (IN_0_RDY /\ ¬OUT_1_RDY) ⇒ IN_0_DAT#,
¬(IN_0_RDY /\ ¬OUT_1_RDY) ⇒ OUT_1_DAT#,
(IN_0_RDY /\ ¬OUT_1_RDY)#} ⇝ ***{ OUT_1_DAT# }***

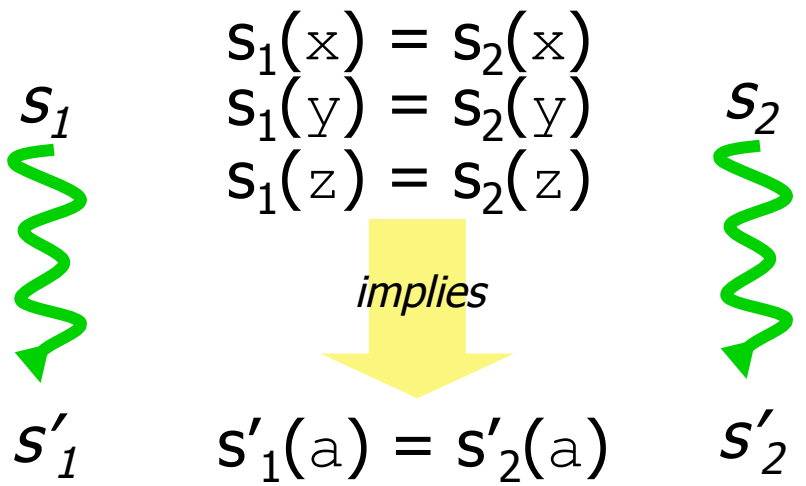
Step 1: *Generate data dependence between variables in input and output channels*

Non-interference

The classical notion of *non-interference* (Goguen & Meseguer) provides semantic foundation for describing the required separation…



Proving "nothing interferes with red channel" (i.e., `a` only depends on `x`, `y` and `z`)…

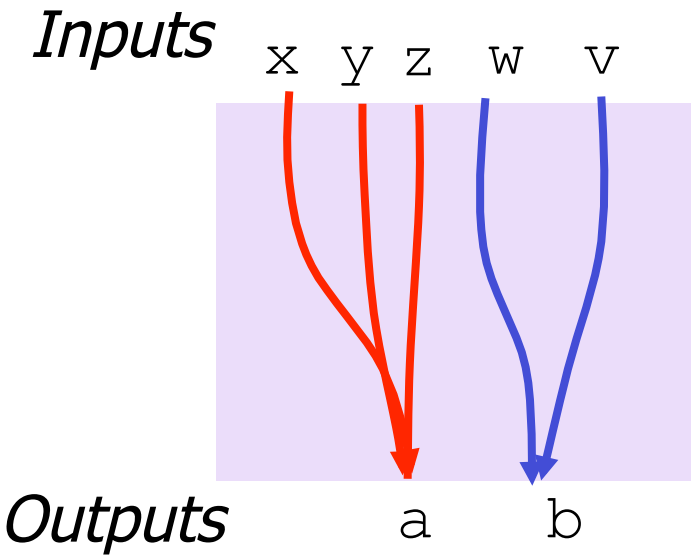


Non-interference theorem:

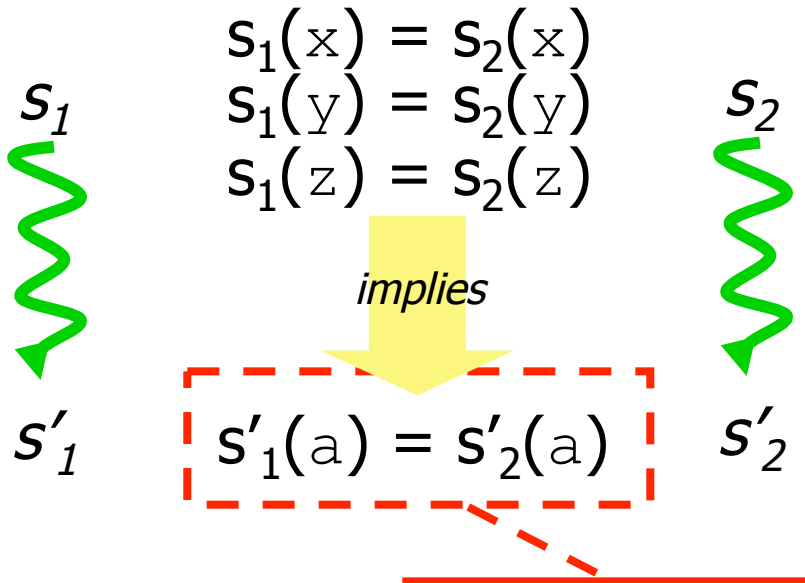
For any two executions with initial states s_1, s_2,
if s_1 and s_2 both **agree** on the values of `x`, `y` and `z`,
then final states s'_1 and s'_2 **agree** on the value of `a`

Non-interference

How can we lift reasoning about non-interference to code-level annotations?



Proving "nothing interferes with red channel" (i.e., `a` only depends on `x`, `y` and `z`)...



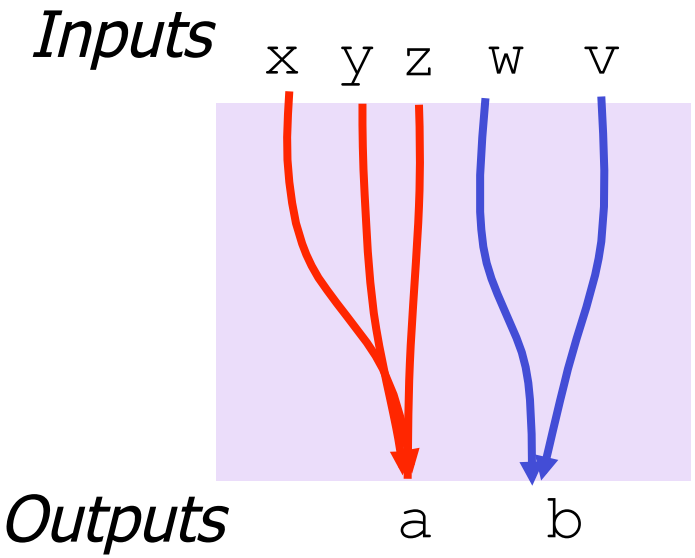
Let's have an abbreviation for the concept of "agreement"

Non-interference theorem:

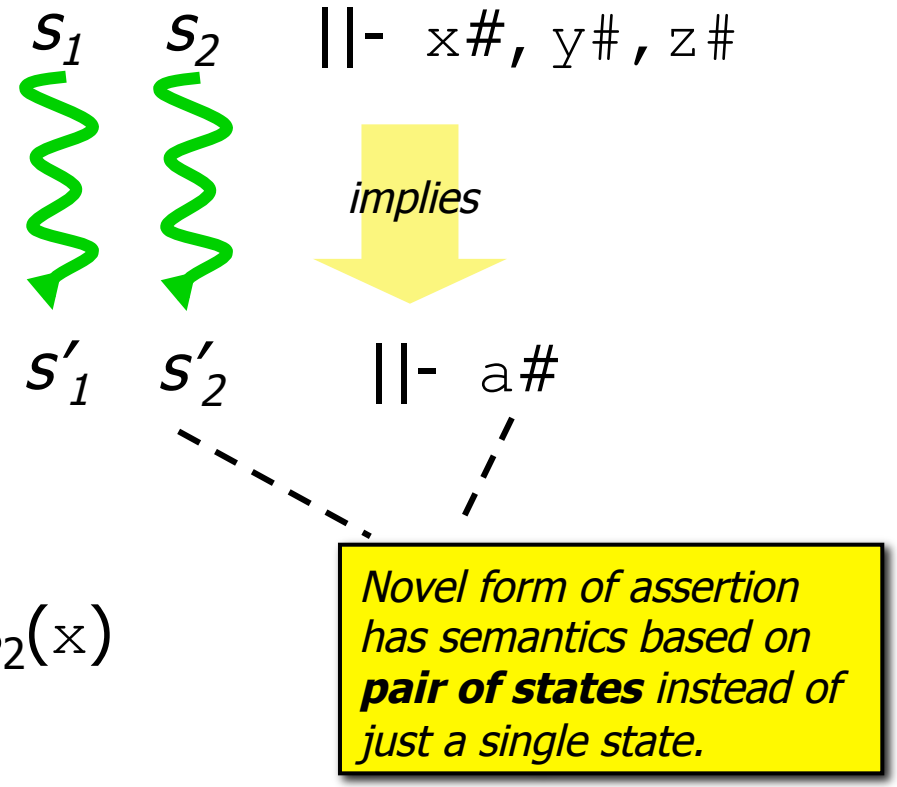
For any two executions with initial states s_1, s_2, if s_1 and s_2 both **agree** on the values of `x`, `y` and `z`, then final states s'_1 and s'_2 **agree** on the value of `a`

Non-interference

The classical notion of *non-interference* provides semantic foundation for describing the required separation…



Proving "nothing interferes with red channel" (i.e., `a` only depends on `x`, `y` and `z`)…



Agreement Assertion:

s_1, s_2 ||- x# iff $s_1(x) = s_2(x)$

Conditional Agreement Assertions

We'll use a more general form of the logic described by Amtoft and Banerjee in [FMSE 07]…

$s_1, s_2 \,||\!-\, x\#$ *iff* $[[x]]s_1 = [[x]]s_2$

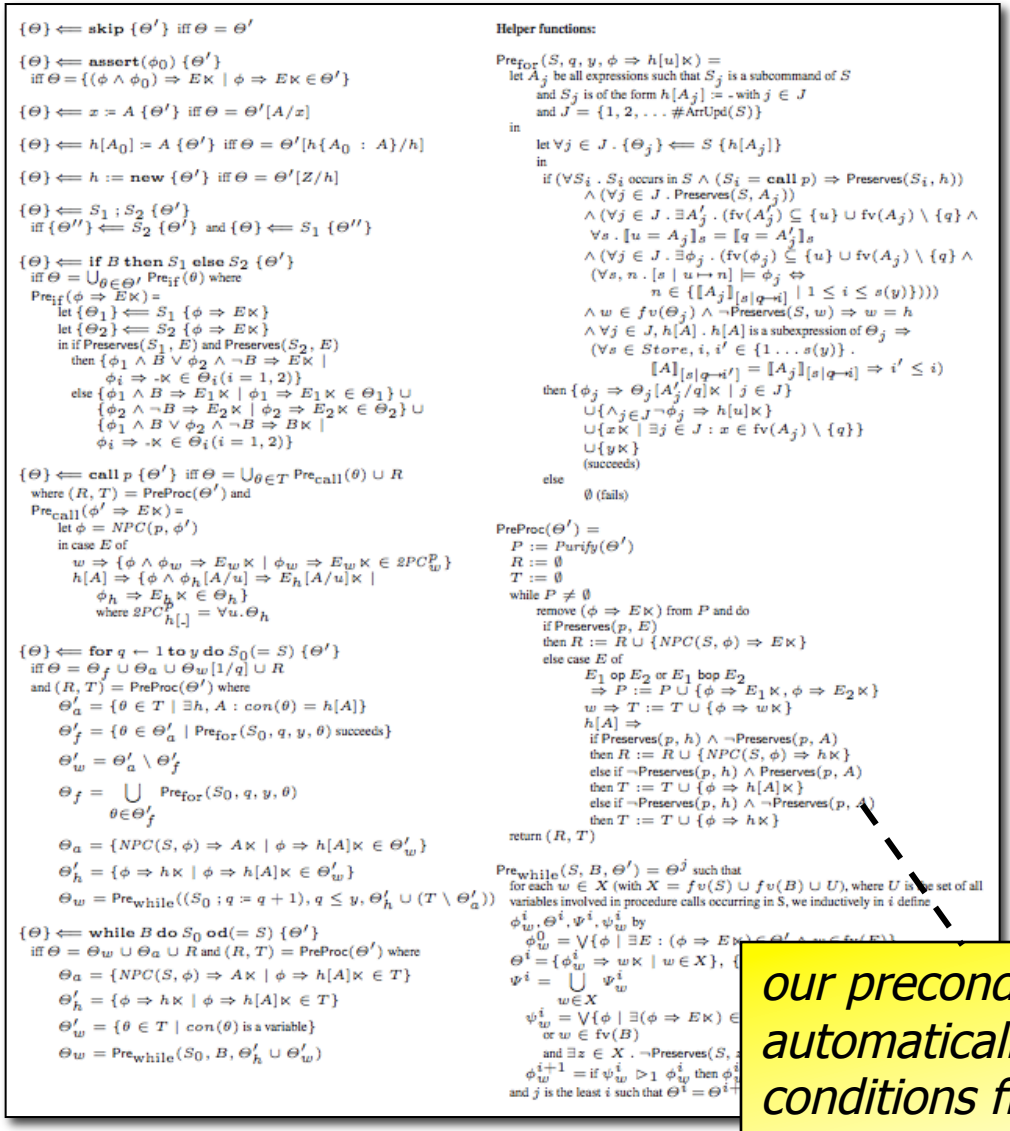
generalize

$s_1, s_2 \,||\!-\, P \Rightarrow E\#$ *iff* whenever $[[P]]s_1$ and $[[P]]s_2$ hold then $[[E]]s_1 = [[E]]s_2$

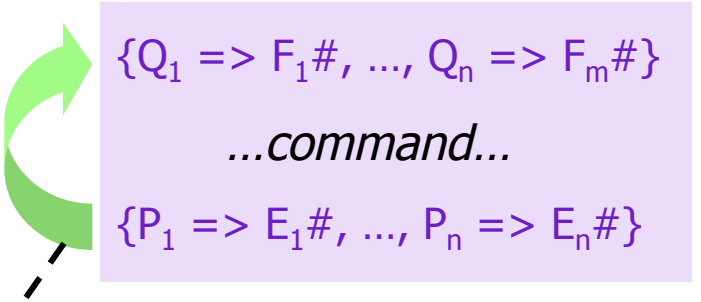
Agreement on expression E under condition P

Note: x# *abbreviates* true => x#

Automating the Analysis



- Can check contracts
- Can (in many cases) automatically infer contracts



our precondition algorithm automatically generates pre-conditions from post-conditions

Example Derivation

```
if INP_0_RDY and
   not OUT_1_RDY then

  DATA_0 := INP_0_DAT;

  INP_0_RDY := false;

  OUT_1_DAT := DATA_0;

  OUT_1_RDY := true;
fi
```

OUT_1_DAT# - - - - - - - - - - -

What do we need agreement on in the pre-state to obtain agreement on OUT_1_DAT *in the post-state?*

…on what does OUT_1_DAT *depend?*

Example Derivation

```
if INP_0_RDY and
   not OUT_1_RDY then

  DATA_0 := INP_0_DAT;

  INP_0_RDY := false;

  OUT_1_DAT := DATA_0;
   OUT_1_DAT# ----------
  OUT_1_RDY := true;
fi
```

Irrelevant to OUT_1_DAT*'s value, so* OUT_1_DAT# *still required*

OUT_1_DAT#

Rule: NotModE

Example Derivation

```
if INP_0_RDY and
   not OUT_1_RDY then

  DATA_0 := INP_0_DAT;

  INP_0_RDY := false;
   DATA_0#
  OUT_1_DAT := DATA_0;
   OUT_1_DAT#
  OUT_1_RDY := true;
fi
```

OUT_1_DAT#

OUT_1_DAT *gets its value from* DATA_0 *and so* DATA_0# *is required*

Rule: AssignE

Example Derivation

```
if INP_0_RDY and
   not OUT_1_RDY then

  DATA_0 := INP_0_DAT;
   DATA_0#
  INP_0_RDY := false;
   DATA_0#
  OUT_1_DAT := DATA_0;
   OUT_1_DAT#
  OUT_1_RDY := true;
fi
```

OUT_1_DAT#

DATA_0 *not modified…*

Rule: NotModE

Example Derivation

```
if INP_0_RDY and
   not OUT_1_RDY then
    INP_0_DAT#
  DATA_0 := INP_0_DAT;
   DATA_0#
  INP_0_RDY := false;
   DATA_0#
  OUT_1_DAT := DATA_0;
   OUT_1_DAT#
  OUT_1_RDY := true;
fi
```

OUT_1_DAT#

DATA_0 *gets its value from* INP_0_DAT *and so* INP_0_DAT# *is required*

Rule: NotModE

Example Derivation

(true && (INP_0_RDY && !OUT_1_RDY)) => INP_0_DAT#

We need agreement on INP_0_DAT# if we enter the true branch

...and we also satisfy any conditions already associated with INP_0_DAT# (none in this case)

```
if INP_0_RDY and
   not OUT_1_RDY then
true => INP_0_DAT#
   DATA_0 := INP_0_DAT;
    DATA_0#
   INP_0_RDY := false;
    DATA_0#
   OUT_1_DAT := DATA_0;
    OUT_1_DAT#
   OUT_1_RDY := true;
fi
```

OUT_1_DAT#

Rule: CondE

Example Derivation

(true && (INP_0_RDY && !OUT_1_RDY)) => INP_0_DAT#

(true && !(INP_0_RDY && !OUT_1_RDY)) => OUT_1_DAT#

If we don't enter the true branch, then OUT_1_DAT *is not modified, and so we need agreement on it in the pre-state*

```
if INP_0_RDY and
   not OUT_1_RDY then
    INP_0_DAT#
  DATA_0 := INP_0_DAT;
   DATA_0#
  INP_0_RDY := false;
   DATA_0#
  OUT_1_DAT := DATA_0;
   OUT_1_DAT#
  OUT_1_RDY := true;
fi
```

OUT_1_DAT#

Rule: CondE

Example Derivation

(true && (INP_0_RDY && !OUT_1_RDY)) => INP_0_DAT#

(true && !(INP_0_RDY && !OUT_1_RDY)) => OUT_1_DAT#

(true && (INP_0_RDY && !OUT_1_RDY) ||
true && !(INP_0_RDY && !OUT_1_RDY)) => (INP_0_RDY && !OUT_1_RDY)#

```
if INP_0_RDY and
   not OUT_1_RDY then
    INP_0_DAT#
  DATA_0 := INP_0_DAT;
    DATA_0#
  INP_0_RDY := false;
    DATA_0#
  OUT_1_DAT := DATA_0;
    OUT_1_DAT#
  OUT_1_RDY := true;
fi
```

OUT_1_DAT#

If either of the guarding expressions above can be satisfied, we need agreement on the test expression to ensure agreement on which branch is taken (control dependence)

Rule: CondE

Example Derivation

(true && (INP_0_RDY && !OUT_1_RDY)) => INP_0_DAT#

(true && !(INP_0_RDY && !OUT_1_RDY)) => OUT_1_DAT#

(true && (INP_0_RDY && !OUT_1_RDY) ||
 true && !(INP_0_RDY && !OUT_1_RDY)) => (INP_0_RDY && !OUT_1_RDY)#

```
if INP_0_RDY and
   not OUT_1_RDY then
    INP_0_DAT#
  DATA_0 := INP_0_DAT;
   DATA_0#
  INP_0_RDY := false;
   DATA_0#
  OUT_1_DAT := DATA_0;
   OUT_1_DAT#
  OUT_1_RDY := true;
fi
```

OUT_1_DAT#

Optimizations are detected, and a decision procedure is called to check if the optimizations are sound.

Performing simplification steps to optimize the size of expressions is a common action in the algorithm

Rules: DecisionImpl, ContraVar

Example Derivation

INP_0_RDY && !OUT_1_RDY => INP_0_DAT#

INP_0_RDY && !OUT_1_RDY => OUT_1_DAT#

INP_0_RDY#
!OUT_1_RDY#

```
if INP_0_RDY and
   not OUT_1_RDY then
    INP_0_DAT#
  DATA_0 := INP_0_DAT;
    DATA_0#
  INP_0_RDY := false;
    DATA_0#
  OUT_1_DAT := DATA_0;
    OUT_1_DAT#
  OUT_1_RDY := true;
fi
```

OUT_1_DAT#

Enhanced SPARK Contracts

INP_0_RDY && !OUT_1_RDY => INP_0_DAT#

INP_0_RDY && !OUT_1_RDY => OUT_1_DAT#

INP_0_RDY#
!OUT_1_RDY#

```
if INP_0_RDY and
   not OUT_1_RDY then
    INP_0_DAT#
  DATA_0 := INP_0_DAT;
   DATA_0#
  INP_0_RDY := false;
   DATA_0#
  OUT_1_DAT := DATA_0;
   OUT_1_DAT#
  OUT_1_RDY := true;
fi
```

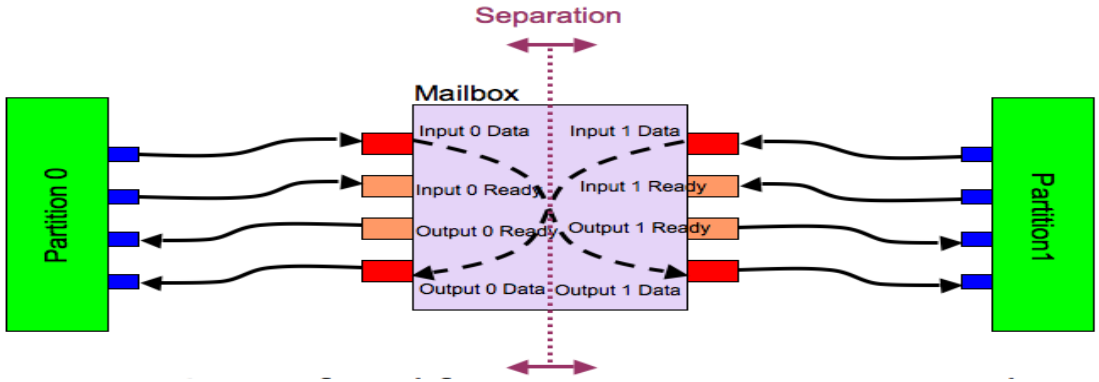
OUT_1_DAT#

New enhanced SPARK contract language hides details of underlying logic

```
--# derives
--#    OUT_1_DAT from
--#      INP_0_DAT
--#        when (INP_0_RDY and
--#              not OUT_1_RDY),
--#      OUT_1_DAT,
--#        when (not INP_0_RDY or
--#              OUT_1_RDY),
--#      INP_0_RDY,
--#      OUT_1_RDY
```

Enhanced SPARK Contracts

Enhanced SPARK contract language can now capture the appropriate policy for the mailbox example...



flows exist only under certain conditions

Original SPARK

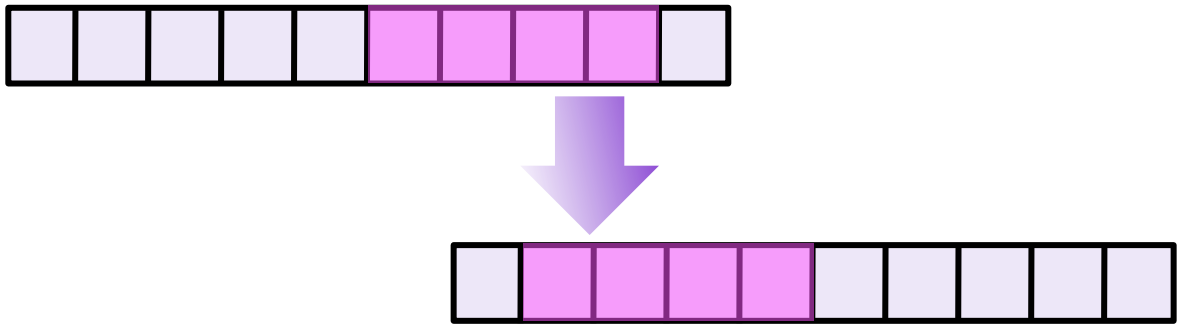
```
--# derives
--#    Output_1_Data from
--#       Input_0_Data,
--#       Output_1_Data,
--#       Input_0_Ready,
--#       Output_1_Ready
```

New conditional SPARK

```
--# derives
--#    Output_1_Data from
--#       Input_0_Data
--#         when (Input_0_Ready and
--#                 not Output_1_Ready),
--#       Output_1_Data,
--#         when (not Input_0_Ready or
--#                 Output_1_Ready),
--#       Input_0_Ready,
--#       Output_1_Ready
```

SPARK Contract Limitations

Many applications focus on processing of array-based message buffers that involves copying portions of buffers, moving message chunks, etc.



- We now show a collection of micro-examples that capture common idioms…
  - illustrate the short-comings of existing SPARK annotations
  - show more precise contracts that are enabled by our *secure information flow logic.*
- See previous papers for illustration of larger examples – including a MILS message router

Precise Array Information Flow

Conditional agreement assertions can be used to state precise conditions on indexing expressions…

Precondition generation rule for array assignment…

Pre: $x = y \Rightarrow w\ltimes,\ x \neq y \Rightarrow A[y]\ltimes,\ (x = y)\ltimes$

$A[x] := w$

Post: $A[y]\ltimes$

Precise Array Information Flow

Original SPARK

```
procedure SinglePositionAssign
(Flag  : in Int;
 Value : in Types.Flagvalue)
--# global in out Flags;
--# derives Flags
--#     from Flags,
--#          Flag, Value;
is
begin
   Flags(Flag) := Value;
end SinglePositionAssign;
```

*Contract describes a flow to **all** index positions*

Enhanced SPARK

Precise frame condition -- only a specific position is affected

```
--# global out Flags(Flag);
--# derives Flags(Flag)
--#     from Value;
```

Contract describes a flow to a specific index position

***Idiom**: updating a single position within an array*

Precise Array Information Flow

Original SPARK

```
procedure Scrub
(B : in out Buffer)
--# derives B
--#      from B;
is
begin
   for I in Buffer_Range loop
      B(I) := 0;
   end loop;
end Scrub;
```

Contract states that final value of B depends on its initial value; cannot capture the fact that all buffer contents are overwritten with 0;

Enhanced SPARK

Enhanced contracts allow universal quantification over array ranges…

```
--# derives forall I in
--#                Buffer_Range
--#    B(I)from {};
```

Contract states that each element in the final value of the buffer gets its value from some unspecified constant (all old information is scrubbed)

***Idiom**: scrubbing a buffer*

Precise Array Information Flow

Original SPARK

```
procedure CopySegment
 (B1 : in Buffer;
  B2 : in out Buffer)
--# derives B2
--#      from B1, B2;
is
begin
   for I in RangeToCopy loop
     B2(I) := B1(I+2);
  end loop;
end CopySegment;
```

Contract imprecision means that flow description is mangled; cannot capture the fact a specific segment of B2 comes from a specific section of B1, and the rest depends its initial value

Enhanced SPARK

Contract is able to specify a fine-grained flow policy…

```
--# derives forall I in
--#             RangeToCopy
--#    B2(I) from B1(I+2)
--# and forall I notin
--#             RangeToCopy
--#    B2(I)  from B2(I)
```

Contract states that each B2 element in the RangeToCopy gets its value from a specific destination of B1, and otherwise depends on its initial value.

***Idiom**: copy one part of a buffer to another*

Precise Array Information Flow

Flipping the halves of an array…

```
procedure flipHalves
--# global in out h;
--# derives for all I in h.Range =>
--#     (h(I) from h(I+h.Range'Last/2)
--#         when I >= 1 && I <= h.Range'Last/2,
--#       h(I-h.Range'Last/2)
--#         when I > h.Range'Last/2 && I <= h.Range,
--#       * when I < 1 || I > h.Range);
is
 t : h.content;
 m : h.Range;
 begin
    m := h.Range'Last/2;
    for q in range h.Range'First .. m loop
       t := h(q);
       h(q) := h(q+m);
       h(q+m) := t;
    end loop;
end flipHalves;
```

Assessment

When dealing with loops…

- **while** loops
  - In many cases, can infer an appropriate loop invariant
  - A loop invariant can always be inferred by dropping all conditions on assertions, but that can lead to imprecision
- **for** loops
  - Treated in clever way
  - Detects loops that are parallelizable, introduces a "schematic" variable for loop index, need only make one pass over the loop, and then generalize with universal quantification
  - Many **for** loops in information assurance applications are parallelizable and thus can be treated precisely
  - Those that aren't are treated as **while** loops