The SPARK2014 verifying compiler

A wide and shallow introduction

October 27, 2015

Deductive verification

A short overview

Florian Schanda

So who is Altran...

- Altran has around 25000 consultants
- In the UK we focus on the development of high-integrity software:
  - iFACTS (part of the UK air traffic control)
  - an engine monitoring unit for a family of commercial aircraft engines
  - etc.
- ... and we also develop SPARK!

Content

Motivation

No bugs, please!

Motivation

Otherwise...

Static analysis

Reason about source code without executing it:

```c
int divide_and_fail(int a, int b) {
  return a / b;
}
```

How many bugs can you find?

Static analysis

Reason about source code without executing it:

```
int divide_and_fail(int a, int b) {
  return a / b;
}
```

How many bugs can you find?

Test 1

```
result = divide_and_fail(500, 0);
```

Static analysis

Reason about source code without executing it:

```
int divide_and_fail(int a, int b) {
  return a / b;
}
```

How many bugs can you find?

Test 1

```
result = divide_and_fail(500, 0);
```

Test 2

```
result = divide_and_fail(-2147483648, -1);
```

Static analysis

- Testing can only find bugs, not prove their absence
- Static analysis can be sound (no missed bugs)
- Static analysis can be complete (all alarms are bugs)
- Static analysis can be automatic (no human intervention)

Static analysis

- Testing can only find bugs, not prove their absence
- Static analysis can be sound (no missed bugs)

```bash
#!/bin/bash
echo "your program is wrong"
```

- Static analysis can be complete (all alarms are bugs)
- Static analysis can be automatic (no human intervention)

Static analysis

- Testing can only find bugs, not prove their absence
- Static analysis can be sound (no missed bugs)

```bash
#!/bin/bash
echo "your program is wrong"
```

- Static analysis can be complete (all alarms are bugs)

```bash
#!/bin/bash
echo "your program is perfect"
```

- Static analysis can be automatic (no human intervention)

Static analysis

- Testing can only find bugs, not prove their absence
- Static analysis can be sound (no missed bugs)

```
#!/bin/bash
echo "your program is wrong"
```

- Static analysis can be complete (all alarms are bugs)

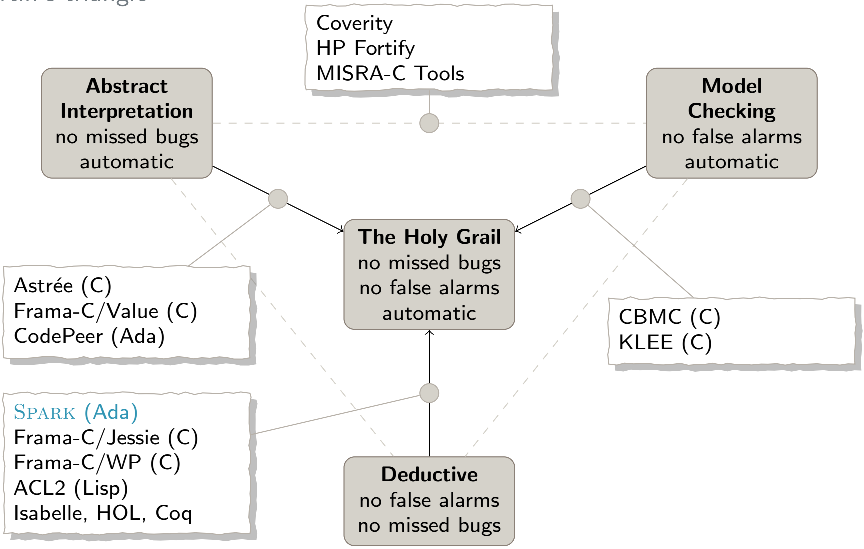
```
#!/bin/bash
echo "your program is perfect"
```

- Static analysis can be automatic (no human intervention)

```
#!/bin/bash
echo "potato"
```

Approaches to static analysis

Martin's triangle



What is Ada?

- Around since 1980
- Influenced by ALGOL, Pascal
- Emphasis readable code
- General purpose, but main users is aerospace and defence industry

Ada
Example

```ada
type U32 is mod 2 ** 32;

procedure Swap (A, B : in out U32)
is
   Tmp : U32;
begin
   Tmp := A;
   A   := B;
   B   := Tmp;
end Swap;
```

What is SPARK?

It is a language and a toolset.

- The language is a variant of Ada:
  - unsafe and difficult constructs excluded
  - contracts
  - reference manual publicly available (GFDL)
- The toolset is a static analysis tool
  - sound, but with a low false alarm rate
  - deductive
  - publicly available (GPL3)

What is SPARK?

It is a language and a toolset.

- The language is a variant of Ada:
  - unsafe and difficult constructs excluded
  - contracts
  - reference manual publicly available (GFDL)
- The toolset is a static analysis tool
  - sound, but with a low false alarm rate
  - deductive
  - publicly available (GPL3)
- ...and it's actually used in real life

What is SPARK?

A brief history

1985 University of Southampton - SPADE

1987 PVL, Praxis, Altran - SPARK 83, 95, 2005

2011 Altran + AdaCore - GPLv3 release of SPARK

2013 Altran + AdaCore - SPARK2014

What is SPARK?

An example

```
type U32 is mod 2 ** 32;

procedure Swap (A, B : in out U32)
with
   Global => null,
   Post   => A = B'Old and
             B = A'Old
is
begin
   A := A xor B;
   B := A xor B;
   A := A xor B;
end Swap;
```

What is SPARK?

Another example

```
type Int_Array is array (1 .. 10) of Integer;

procedure Find_Element (A     : in     Int_Array;
                        Elem  : in     Integer;
                        Idx   :    out Natural;
                        Found :    out Boolean)
with
   Post => (if Found then A (Idx) = Elem)
is
begin
   for I in A'Range loop
      Idx   := I;
      Found := A (I) = Elem;
      exit when Found;
   end loop;
end Find_Element;
```

What is SPARK?

Headline features

- Strong typing

```
type Grams  is new Float range 0.0 .. 100.0;
type Pounds is new Float range 0.0 .. 100.0;
```

- Many contracts...
  - Parameter modes (in, in out, out)
  - Globals, Depends, Abstract_State
  - Pre, Post, Contract_Cases
  - No_Return, Volatile, Atomic
  - Asserts, Invariants, Variants

  ...most of which are executable
- Ghost code
- No pointers (or aliasing), instead:
  - Pass by reference
  - Container library
- No side-effects

What is SPARK?

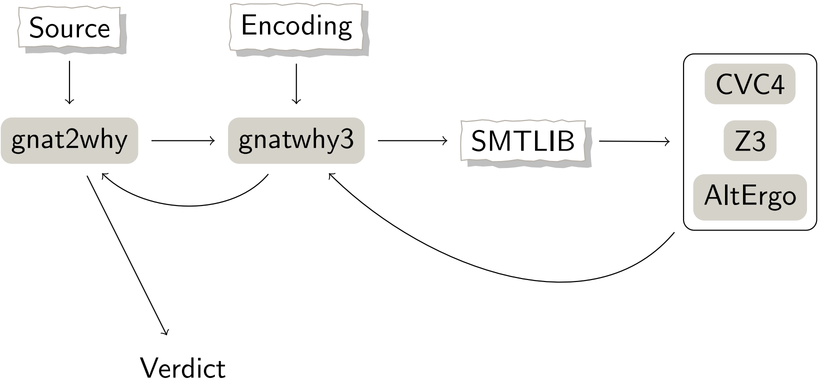
The tools

- Developed jointly at Altran (Bath) and AdaCore (Paris, and more)
  - 4 in Bath
  - 5 in Paris
  - plus more (language design, gcc, etc.)
- Based on free software (gcc, Why3, CVC4, etc.)
- Based on published results, and ongoing collaborations with universities
- Very fun!

Tool architecture
User view

Tool architecture

More detailed view...



Tool architecture
gnat2why

This is what we develop ourselves.

- Contains SPARK/Ada parser
- Does some preliminary analysis
- Compilation to intermediate language

Other tools are also free software that are used by us and others.

GNAT Frontend
Overview

- Ada 2012 and SPARK2014 lexer,
- parser,
- semantic analyser,
- expander,
- code generator (with gcc via intermediate language)

GNAT Frontend

Lexer and Parser

- Hand-written lexer
- Hand-written parser, recursive descent, with arbitrary look-ahead
  - better error messages, especially for serious structural errors
  - closely aligned with published Ada RM grammar
- Produces AST
  - Nodes (structure of program)
  - Entities (identifiers, operator symbols, etc.)
- There is no classical symbol table
- Entities are the symbol table

GNAT Frontend
The AST

- Table indexed by integers (so no pointers as such)
- No OO, instead records with generic fields (field1, field2, field3, etc.) and access subprograms that give them meaning based on node type:

```
function Etype               (N : Node_Id) return Node_Id;
function Exception_Choices   (N : Node_Id) return List_Id;
function Exception_Handlers (N : Node_Id) return List_Id;
...
```

- 12,000+ line comment that is tool-enforced for documentation

GNAT Frontend

AST example

Partial tree for `return a / b;`

```
Node #1 N_Simple_Return_Statement (Node_Id=2346) (source,analyzed)
 Sloc = 8249  foo.adb:4:4
 Return_Statement_Entity = Node #5 N_Defining_Identifier "R1b" (Entity_Id=2350)
 Expression = Node #2 N_Op_Divide "Odivide" (Node_Id=2347)

  Node #2 N_Op_Divide "Odivide" (Node_Id=2347) (source,analyzed)
   Parent = Node #1 N_Simple_Return_Statement (Node_Id=2346)
   Sloc = 8258  foo.adb:4:13
   Chars = "Odivide" (Name_Id=300000413)
   Left_Opnd = Node #3 N_Identifier "a" (Node_Id=2345)
   Right_Opnd = Node #4 N_Identifier "b" (Node_Id=2348)
   Entity = N_Defining_Identifier "Odivide" (Entity_Id=1919s)
   Do_Overflow_Check = True
   Etype = N_Defining_Identifier "integer" (Entity_Id=1035s)
   Do_Division_Check = True

    Node #3 N_Identifier "a" (Node_Id=2345) (source,analyzed)
     Parent = Node #2 N_Op_Divide "Odivide" (Node_Id=2347)
     Sloc = 8256  foo.adb:4:11
     Chars = "a" (Name_Id=300000099)
     Etype = N_Defining_Identifier "integer" (Entity_Id=37s)
     Entity = N_Defining_Identifier "a" (Entity_Id=2324)
     Associated_Node = N_Defining_Identifier "a" (Entity_Id=2324)

    ...
```

GNAT Frontend

The expander and code generation

- Generics are expanded here
- Also deals with tasks, protected objects and object orientation
- AST is transformed to something the GCC code generator expects

gnat2why

Overview

- Just another GNAT back-end
- An elaborate semantic analysis pass over the AST:

  filter: Note which areas of the program are "in SPARK"

  globals: Generate frame conditions (global contracts if they have not been specified) at varying levels of details

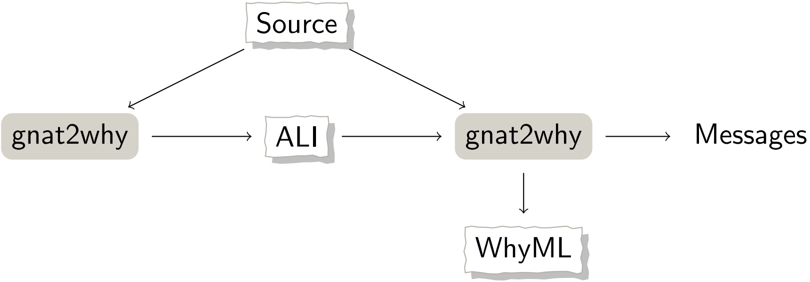
  flow: Check initialization, non-aliasing, global contracts, and information flow contracts

  translation: Transform SPARK subprograms into WhyML subprograms

altran

gnat2why

Overview



gnat2why

Filter

- Useful for analysing programs partly in SPARK, partly in Ada
- Essentially assigns Boolean flag to all entities
- Produces error messages if things have to be in SPARK
- Fairly obvious top-down tree-walk

Example

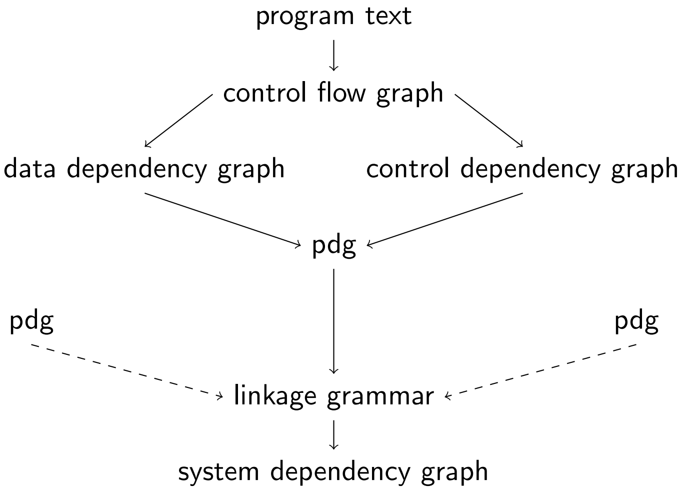
```
package P1 is
   type T1 is range 1 .. 10;  -- T in SPARK
   type T2 is access Float;   -- T2 not in SPARK
end P1;

package P2 with SPARK_Mode is
   type T3 is access Integer; -- ERROR
end P2;
```

gnat2why

Flow

- Over-approximation, automatic
- More or less implements “system dependence graphs”
- Checks initialization of all variables
  - an important assumption for proof
  - an important check for information leaks
- Checks non-aliasing, an important assumption for proof
- Determines or checks frame conditions
- Checks flow contracts
- Checks for data races (for tasking)
- Warns of some suspicious constructs (ineffective code, etc.)

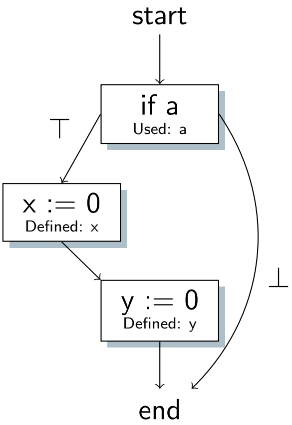
gnat2why
Flow



gnat2why

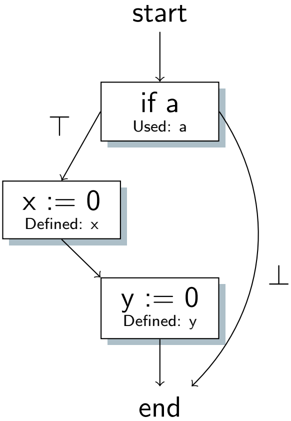
Flow - Control flow graph

```
if a then
   x := 0;
else
   return;
end if;
y := 0;
```



gnat2why

Flow - Data dependence graph



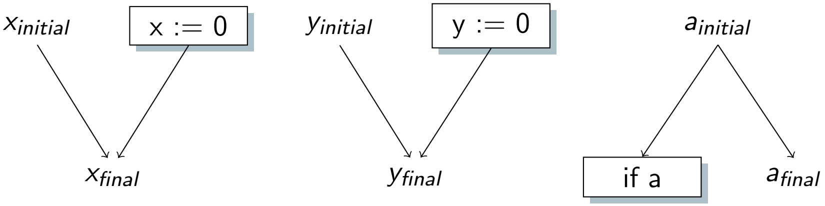
Def-use chains:

- *a*
  - start → if a
  - start → end
- *x*
  - start → end
  - x := 0 → end
- *y*
  - start → end
  - y := 0 → end

gnat2why

Flow - Data dependence graph

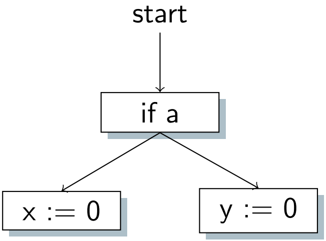
We can now draw the DDG:



gnat2why

Flow - Control dependence graph

```
if a then
   x := 0;
else
   return;
end if;
y := 0;
```

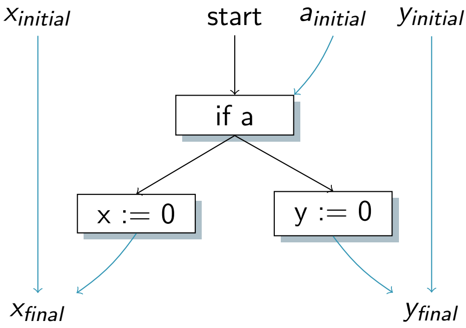


Algorithm: produce dominance frontier of the reversed control flow graph ("post dominance frontier").

gnat2why

Flow - Program dependence graph

To construct the program dependence graph (PDG) we simply overlay the CDG and DDG.



gnat2why

Flow

- There is much more detail in flow:
  - Inter-procedural analysis and recursion
  - Non-terminating subprograms
  - Tasking
  - Component-level analysis of records
  - Volatile variables
  - etc.
- Debug output is done through graphviz and has been a primary design concern

gnat2why

Translation to WhyML

- SPARK is still an extremely complicated language
- Key properties need to be proven for a program to be correct ("verification conditions", or "VCs")
- Translation to a smaller, intermediate language WhyML
  - Simpler control flow
  - Simpler types
- Verification condition generation based on this IL

gnat2why

Translation to WhyML

```
function Example
   (A, B : Natural)
    return Natural
is
   R : Natural;
begin
   if A < B then
      R := A + 1;
   else
      R := B - 1;
   end if;
   return R;
end Example;
```

gnat2why

Translation to WhyML

```
function Example
   (A, B : Natural)
    return Natural
is
   R : Natural;
begin
   if A < B then
      R := A + 1;
   else
      R := B - 1;
   end if;
   return R;
end Example;
```

→

```
let example (a: int) (b: int)
    requires { a >= 0 /\ a <= 2147483647 }
    requires { b >= 0 /\ b <= 2147483647 }
    returns { r -> r >= 0 /\
                   r <= 2147483647 }
  = let r = ref 0 in
    if a < b then
       r := a + 1
    else
       r := b - 1;
    (!r)
```


altran

gnat2why

Translation to WhyML

- Another traversal over AST (for SPARK), building another AST (for Why3)
- Tree is "pretty" printed, but not meant to be human readable
- One or more Why3 modules per SPARK entity
  - Types
  - Entity definitions, axioms
  - Subprogram definitions, axioms, bodies

  All of which are dumped into a single file for `gnatwhy3`.
- Not as nice as the previous example, a lot of extra information embedded:
  - Original source locations of all VCs
  - Checks ($x \neq 0$, or $x < 2^{32}$, etc.)

gnat2why

Translation to WhyML

Yep, not very readable... VC fragment for $r = a/b$:

```
( ( "GP_Sloc:overflow.adb:7:7" ( #"overflow.adb" 7 0 0#
overflow__example__result.int__content <- ( (
#"overflow.adb" 7 0 0# "GP_Sloc:overflow.adb:7:16"
"GP_Shape:return__div" "keep_on_simp" "model_vc"
"GP_Reason:VC_OVERFLOW_CHECK" "GP_Id:1"
(Standard__integer.range_check_(( #"overflow.adb" 7 0 0#
"GP_Reason:VC_DIVISION_CHECK" "GP_Id:0"
"GP_Sloc:overflow.adb:7:16" "GP_Shape:return__div"
"keep_on_simp" "model_vc" (Int_Division.div_
(Overflow__example__a.a) (Overflow__example__b.b))
))) ) ); #"overflow.adb" 7 0 0# raise Return__exc ) );
#"overflow.adb" 3 0 0# raise Return__exc )
```

gnat2why

Translation to WhyML

Yep, not very readable... VC fragment for $r = a/b$:

```
( ( "GP_Sloc:overflow.adb:7:7" ( #"overflow.adb" 7 0 0#
overflow__example__result.int__content <- ( (
#"overflow.adb" 7 0 0# "GP_Sloc:overflow.adb:7:16"
"GP_Shape:return__div" "keep_on_simp" "model_vc"
"GP_Reason:VC_OVERFLOW_CHECK" "GP_Id:1"
(Standard__integer.range_check_(( #"overflow.adb" 7 0 0#
"GP_Reason:VC_DIVISION_CHECK" "GP_Id:0"
"GP_Sloc:overflow.adb:7:16" "GP_Shape:return__div"
"keep_on_simp" "model_vc" (Int_Division.div_
(Overflow__example__a.a) (Overflow__example__b.b))
))) ) ); #"overflow.adb" 7 0 0# raise Return__exc ) );
#"overflow.adb" 3 0 0# raise Return__exc )
```

But we eventually get nice output...

overflow.adb:7:16: medium: divide by zero might fail (e.g. when B = 0)
overflow.adb:7:16: medium: overflow check might fail

gnat2why

Translation to WhyML

Features of the IL:

- Based on first order logic + theories
- In vague ML syntax with programming constructs:
  - (mutable) variables
  - sequences
  - loops, if, etc.
  - assertions
  - exceptions
- Built-in types are Boolean, Int, Real, Arrays, Records, Lists, Sets, etc. but more can be defined

gnat2why

Translation to WhyML

All checks come from a specification:

- Some checks are user defined (user asserts, postconditions)
- Ada RM defines basic checks (overflow, range, index, division by zero, discriminants, etc.)
- SPARK RM defines more (LSP checks, loop variants and invariants, etc.)

... we just follow that spec, and err on side of redundant checks.

SAT, SMT and SMTLIB

Recap: we now have the SPARK program in a different language (WhyML), but have not verified much...

- It's still difficult to prove anything, so we need to start talking to (automatic) theorem provers
- Language of choice is SMTLIB, but others exist
- So, next step is another language transformation
- But, to appreciate this step, let's first talk about SMT...

SAT, SMT and SMTLIB

Recap of SAT

Let's start with SAT:

- Is there an assignment for $(a \vee \neg c) \wedge (\neg b \vee c) \wedge (d)$ that makes everything true?

[1] conflict-driven clause learning

SAT, SMT and SMTLIB
Recap of SAT

Let's start with SAT:

- Is there an assignment for $(a \vee \neg c) \wedge (\neg b \vee c) \wedge (d)$ that makes everything true?
- Yes, at least one: $\neg a, \neg b, \neg c, d$
- Significant advances in the last 15 years
- Modern CDCL[1] solvers can solve huge problems with millions of variables

[1] conflict-driven clause learning

SAT modulo theories: SMT

- Is some first-order logic formula SAT (given some background theory)?

SAT modulo theories: SMT

- Is some first-order logic formula SAT (given some background theory)?
- $(0 \heartsuit x) \wedge (0 \heartsuit y) \wedge (x \spadesuit y \heartsuit x)$

SAT, SMT and SMTLIB

Overview of SMT

SAT modulo theories: SMT

- Is some first-order logic formula SAT (given some background theory)?
- $(0 \heartsuit x) \wedge (0 \heartsuit y) \wedge (x \spadesuit y \heartsuit x)$
- We need an interpretation for $\heartsuit$ and $\spadesuit$ to decide!

SAT, SMT and SMTLIB

Overview of SMT

SAT modulo theories: SMT

- Is some first-order logic formula SAT (given some background theory)?
- $(0 \heartsuit x) \wedge (0 \heartsuit y) \wedge (x \spadesuit y \heartsuit x)$
- We need an interpretation for $\heartsuit$ and $\spadesuit$ to decide!
- $\heartsuit =<$ and $\spadesuit = +$ and $D = \mathbb{R}$:
  $(0 < x) \wedge (0 < y) \wedge (x + y < x)$

SAT modulo theories: SMT

- Is some first-order logic formula SAT (given some background theory)?
- $(0 \heartsuit x) \wedge (0 \heartsuit y) \wedge (x \spadesuit y \heartsuit x)$
- We need an interpretation for $\heartsuit$ and $\spadesuit$ to decide!
- $\heartsuit =<$ and $\spadesuit = +$ and $D = \mathbb{R}$:
  $(0 < x) \wedge (0 < y) \wedge (x + y < x)$ - UNSAT

SAT, SMT and SMTLIB

Overview of SMT

SAT modulo theories: SMT

- Is some first-order logic formula SAT (given some background theory)?
- $(0 \heartsuit x) \wedge (0 \heartsuit y) \wedge (x \spadesuit y \heartsuit x)$
- We need an interpretation for $\heartsuit$ and $\spadesuit$ to decide!
- $\heartsuit =<$ and $\spadesuit = +$ and $D = \mathbb{R}$:
  $(0 < x) \wedge (0 < y) \wedge (x + y < x)$ - UNSAT
- $\heartsuit =<$ and $\spadesuit = +_{mod256}$ and $D = uint8\_t$:
  $(0 < x) \wedge (0 < y) \wedge (x +_{mod256} y < x)$

SAT, SMT and SMTLIB
Overview of SMT

SAT modulo theories: SMT

- Is some first-order logic formula SAT (given some background theory)?
- $(0 \heartsuit x) \wedge (0 \heartsuit y) \wedge (x \spadesuit y \heartsuit x)$
- We need an interpretation for $\heartsuit$ and $\spadesuit$ to decide!
- $\heartsuit =<$ and $\spadesuit = +$ and $D = \mathbb{R}$:
  $(0 < x) \wedge (0 < y) \wedge (x + y < x)$ - UNSAT
- $\heartsuit =<$ and $\spadesuit = +_{mod256}$ and $D = uint8\_t$:
  $(0 < x) \wedge (0 < y) \wedge (x +_{mod256} y < x)$ - SAT $(x = 255,\ y = 1)$

altran

SAT, SMT and SMTLIB

SMT Example

- $(0 < x) \wedge (0 < y) \wedge (x + y < x)$

SAT, SMT and SMTLIB
SMT Example

- $(0 < x) \wedge (0 < y) \wedge (x + y < x)$
- Is this SAT? $A_1 \wedge A_2 \wedge A_3$

SAT, SMT and SMTLIB

SMT Example

- $(0 < x) \wedge (0 < y) \wedge (x + y < x)$
- Is this SAT? $A_1 \wedge A_2 \wedge A_3$
- YES - (A_1, A_2, A_3)

SAT, SMT and SMTLIB
SMT Example

- $(0 < x) \wedge (0 < y) \wedge (x + y < x)$
- Is this SAT? $A_1 \wedge A_2 \wedge A_3$
- YES - (A_1, A_2, A_3)
- Hand over to theory solver:
  $0 < x$
  $0 < y$
  $x + y < x$

altran

SAT, SMT and SMTLIB
SMT Example

- $(0 < x) \wedge (0 < y) \wedge (x + y < x)$
- Is this SAT? $A_1 \wedge A_2 \wedge A_3$
- YES - (A_1, A_2, A_3)
- Hand over to theory solver:
  $0 < x$
  $0 < y$
  $x + y < x$ (substract by x)

altran

SAT, SMT and SMTLIB
SMT Example

- $(0 < x) \wedge (0 < y) \wedge (x + y < x)$
- Is this SAT? $A_1 \wedge A_2 \wedge A_3$
- YES - (A_1, A_2, A_3)
- Hand over to theory solver:
  $0 < x$
  $0 < y$
  $y < 0$

SAT, SMT and SMTLIB

SMT Example

- $(0 < x) \wedge (0 < y) \wedge (x + y < x)$
- Is this SAT? $A_1 \wedge A_2 \wedge A_3$
- YES - (A_1, A_2, A_3)
- Hand over to theory solver:
  $0 < x$
  $0 < y$
  $y < 0$ (contradiction between A_2 and A_3!)

SAT, SMT and SMTLIB

SMT Example

- $(0 < x) \wedge (0 < y) \wedge (x + y < x)$
- Is this SAT? $A_1 \wedge A_2 \wedge A_3$
- YES - (A_1, A_2, A_3)
- Hand over to theory solver:
  $0 < x$
  $0 < y$
  $y < 0$ (contradiction between A_2 and A_3!)
- Add $\neg(A_2 \wedge A_3)$ to the SAT problem

SAT, SMT and SMTLIB

SMT Example

- $(0 < x) \wedge (0 < y) \wedge (x + y < x)$
- Is this SAT? $A_1 \wedge A_2 \wedge A_3$
- YES - (A_1, A_2, A_3)
- Hand over to theory solver:
  $0 < x$
  $0 < y$
  $y < 0$ (contradiction between A_2 and A_3!)
- Add $\neg(A_2 \wedge A_3)$ to the SAT problem
- Is this SAT? $A_1 \wedge A_2 \wedge A_3 \wedge (\neg A_2 \vee \neg A_3)$

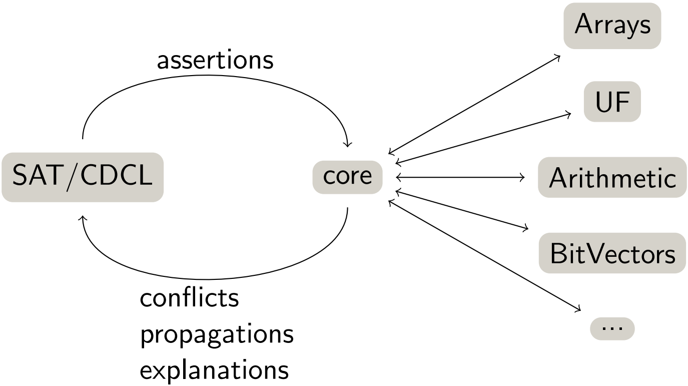
SAT, SMT and SMTLIB

SMT Example

- $(0 < x) \wedge (0 < y) \wedge (x + y < x)$
- Is this SAT? $A_1 \wedge A_2 \wedge A_3$
- YES - (A_1, A_2, A_3)
- Hand over to theory solver:
  $0 < x$
  $0 < y$
  $y < 0$ (contradiction between A_2 and A_3!)
- Add $\neg(A_2 \wedge A_3)$ to the SAT problem
- Is this SAT? $A_1 \wedge A_2 \wedge A_3 \wedge (\neg A_2 \vee \neg A_3)$
- UNSAT

SAT, SMT and SMTLIB

Architecture of modern SMT solver (thanks to Liana for the picture)



SAT, SMT and SMTLIB

Theories

Many theories have been implemented:

- Boolean
- Integer
- Reals
- Quantifiers
- Arrays
- Uninterpreted functions
- Bitvectors
- IEEE-754 Floating Point
- Strings
- Sets
- Algebraic Datatypes

SAT, SMT and SMTLIB

Overview of SMTLIB

- In the beginning all SMT solvers used their own input language
- This made it hard to compare solvers
- SMTLIB is both a standard language and a huge library of benchmarks
- SMTLIB only describes a search problem
- No control flow (if statements, loops, etc.) - so very far away from “programming language”

SAT, SMT and SMTLIB

SMTLIB is just s-expressions – I hope you remember your LISP?

```
; quantifier-free linear integer arithmetic
(set-logic QF_LIA)
; declarations
(declare-const x Int)
(declare-const y Int)
; hypothesis - things we know are true
(assert (<= 1 x 10)) ; 1 ≤ x ≤ 10
(assert (<= 1 y 10)) ; 1 ≤ y ≤ 10
; goal - what we want to prove
(define-const goal Bool (< (+ x y) 15)) ; x + y < 15
; search for a model where the goal is not true
(assert (not goal))
(check-sat)
```

SAT, SMT and SMTLIB

SMTLIB is just s-expressions – I hope you remember your LISP?

```
; quantifier-free linear integer arithmetic
(set-logic QF_LIA)
; declarations
(declare-const x Int)
(declare-const y Int)
; hypothesis - things we know are true
(assert (<= 1 x 10))  ; 1 ≤ x ≤ 10
(assert (<= 1 y 10))  ; 1 ≤ y ≤ 10
; goal - what we want to prove
(define-const goal Bool (< (+ x y) 15))  ; x + y < 15
; search for a model where the goal is not true
(assert (not goal))
(check-sat)
```

CVC4 output

```
sat
((x 10) (y 5))
```

SAT, SMT and SMTLIB

SMTLIB language overview

- Functions

```
(define-fun double (Int) Int)
(declare-fun triple ((x Int)) Int (+ x x x))
```

- Assertions and function calls

```
(assert (forall ((x Int)) (= (double x) (+ x x))))
```

- Predefined functions for theories

Core	=, =>, and, or, xor, not, ite, ...
Ints	+, -, *, /, >, >=, ...
Arrays	select, store
BV	bvadd, bvudiv, bvsdiv, bvlte, ...
FP	fp.add, fp.mul, fp.eq, fp.isInfinite, ...

SAT, SMT and SMTLIB

You can encode difficult problems with this...

```
(declare-fun fib (Int) Int)

(assert (= (fib 0) 0))
(assert (= (fib 1) 1))

; read this as: ∀x ∈ Int • x ≥ 2 ⟹ fib(x) = fib(x − 2) + fib(x − 1)
(assert (forall ((x Int))
          (=> (>= x 2)
              (= (fib x) (+ (fib (- x 2))
                            (fib (- x 1)))))))

; let's try to prove fib(10) < 10
(assert (not (< (fib 10) 10)))
(check-sat)
```

SAT, SMT and SMTLIB

You can encode difficult problems with this...

```
(declare-fun fib (Int) Int)

(assert (= (fib 0) 0))
(assert (= (fib 1) 1))

; read this as: ∀x ∈ Int • x ≥ 2 ⟹ fib(x) = fib(x − 2) + fib(x − 1)
(assert (forall ((x Int))
          (=> (>= x 2)
              (= (fib x) (+ (fib (- x 2))
                            (fib (- x 1))))))))

; let's try to prove fib(10) < 10
(assert (not (< (fib 10) 10)))
(check-sat)
```

CVC4 output

```
unknown
(((fib 10) 55))
```

SAT, SMT and SMTLIB

Solvers

Many solvers exist - (partial) table from Wikipedia:

Platform			Features						Notes
Name	OS	License	SMT-LIB	CVC	DIMACS	Built-in theories	API	SMT-COMP [2]	
ABsolver	Linux	CPL	v1.2	No	Yes	linear arithmetic, non-linear arithmetic	C++	no	DPLL-based
Alt-Ergo	Linux, Mac OS, Windows	CeCILL-C (roughly equivalent to LGPL)	partial v1.2 and v2.0	No	No	empty theory, linear integer and rational arithmetic, non-linear arithmetic, polymorphic arrays, enumerated datatypes, AC symbols, bitvectors, record datatypes, quantifiers	OCaml	2008	Polymorphic first-order input language à la ML, SAT-solver based, combines Shostak-like and Nelson-Oppen like approaches for reasoning modulo theories
Barcelogic	Linux	Proprietary	v1.2			empty theory, difference logic	C++	2009	DPLL-based, congruence closure
Beaver	Linux, Windows	BSD	v1.2	No	No	bitvectors	OCaml	2009	SAT-solver based
Boolector	Linux	GPLv3	v1.2	No	No	bitvectors, arrays	C	2009	SAT-solver based
CVC3	Linux	BSD	v1.2	Yes		empty theory, linear arithmetic, arrays, tuples, types, records, bitvectors, quantifiers	C/C++	2010	proof output to HOL
CVC4	Linux, Mac OS, Windows	BSD	Yes	Yes		rational and integer linear arithmetic, arrays, tuples, records, inductive data types, bit-vectors, strings, and equality over uninterpreted function symbols	C++	2010	version 1.4 released July 2014
Decision Procedure Toolkit (DPT)	Linux	Apache	No				OCaml	no	DPLL-based
iSAT	Linux	Proprietary	No			non-linear arithmetic		no	DPLL-based
MathSAT	Linux	Proprietary	Yes		Yes	empty theory, linear arithmetic, bitvectors, arrays	C/C++, Python, Java	2010	DPLL-based
MiniSmt	Linux	LGPL	partial v2.0			non-linear arithmetic		2010	SAT-solver based, Yices-based
OpenCog	Linux	AGPL	No	No	No	probabilistic logic, arithmetic. relational models	C++, Scheme, Python	no	subgraph isomorphism
OpenSMT	Linux, Mac OS, Windows	GPLv3	partial v2.0		Yes	empty theory, differences, linear arithmetic, bitvectors	C++	2011	lazy SMT Solver
SatEEn	?	Proprietary	v1.2			linear arithmetic, difference logic	none	2009	
SMTInterpol	Linux, Mac OS, Windows	LGPLv3	v2.0			uninterpreted functions, linear real arithmetic, and linear integer arithmetic	Java	2012	Focuses on generating high quality, compact interpolants.
SMCHR	Linux, Mac OS, Windows	GPLv3	No	No	No	linear arithmetic, nonlinear arithmetic, heaps	C	no	Can implement new theories using Constraint Handling Rules.
SMT-RAT	Linux, Mac OS	MIT	v2.0	No	No	linear arithmetic, nonlinear arithmetic	C++	2015	Toolbox for strategic and parallel SMT solving consisting of a collection of SMT compliant implementations.
SONOLAR	Linux, Windows	Proprietary	partial v2.0			bitvectors	C	2010	SAT-solver based
Spear	Linux, Mac OS, Windows	Proprietary	v1.2			bitvectors		2008	
STP	Linux, OpenBSD, Windows, Mac OS	MIT	partial v2.0	Yes	No	bitvectors, arrays	C, C++, Python, OCaml, Java	2011	SAT-solver based
SWORD	Linux	Proprietary	v1.2			bitvectors		2009	
UCLID	Linux	BSD	No	No	No	empty theory, linear arithmetic, bitvectors, and constrained lambda (arrays, memories, cache, etc.)		no	SAT-solver based, written in Moscow ML. Input language is SMV model checker. Well-documented!

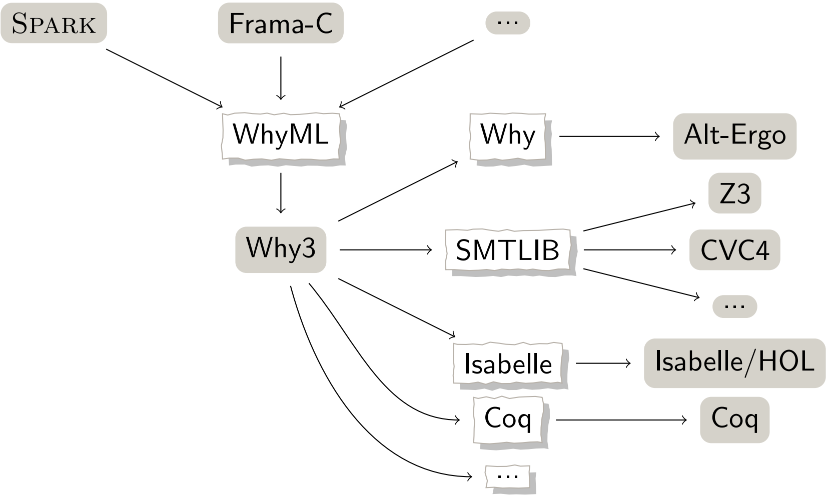
... different strengths and logic support.

Why3 and WP

- So - SPARK/WhyML and SMTLIB are quite different
- Last step is to go from the intermediate language to verification conditions expressed in SMTLIB

Why3 and WP

Why3 is a general purpose intermediate language:



Why3 and WP

Consider Hoare triplets:

- $\{precondition\}$ statement $\{postcondition\}$

Why3 and WP

Consider Hoare triplets:

- $\{$*precondition*$\}$ statement $\{$*postcondition*$\}$
- $\{\top\}$ `x := 42` $\{\top\}$

Why3 and WP

Consider Hoare triplets:

- $\{$*precondition*$\}$ statement $\{$*postcondition*$\}$
- $\{\top\}$ `x := 42` $\{\top\}$
- $\{\top\}$ `x := 42` $\{x > 0\}$

altran

Why3 and WP

Consider Hoare triplets:

- $\{$*precondition*$\}$ statement $\{$*postcondition*$\}$
- $\{\top\}$ `x := 42` $\{\top\}$
- $\{\top\}$ `x := 42` $\{x > 0\}$
- $\{\top\}$ `x := 42` $\{x = 42\}$

Why3 and WP

Consider Hoare triplets:

- $\{$*precondition*$\}$ statement $\{$*postcondition*$\}$
- $\{\top\}$ `x := 42` $\{\top\}$
- $\{\top\}$ `x := 42` $\{x > 0\}$
- $\{\top\}$ `x := 42` $\{x = 42\}$
- $\{\bot\}$ `x := 42` $\{x = 5\}$

Why3 and WP

Consider Hoare triplets:

- $\{precondition\}$ statement $\{postcondition\}$
- $\{\top\}$ `x := 42` $\{\top\}$
- $\{\top\}$ `x := 42` $\{x > 0\}$
- $\{\top\}$ `x := 42` $\{x = 42\}$
- $\{\bot\}$ `x := 42` $\{x = 5\}$
- $\{x > 0\}$ `z := x + y` $\{z > y\}$

Why3 and WP

Consider this example:

```
-- precondition: 1 ≤ a ≤ 10
v1 := a + 5;
v2 := v1 / 2;
-- postcondition 1 ≤ v2 ≤ 10
```

Weakest Precondition

$\{\}$

v1 := a + 5;

$\{\}$

v2 := v1 / 2;

$\{1 \leq a \leq 10 \implies 1 \leq v_2 \leq 10\}$

Why3 and WP

Consider this example:

```
-- precondition: 1 ≤ a ≤ 10
v1 := a + 5;
v2 := v1 / 2;
-- postcondition 1 ≤ v2 ≤ 10
```

Weakest Precondition

$\{\}$

v1 := a + 5;

$\{1 \leq a \leq 10 \implies 2 \leq v_1 \leq 20\}$

v2 := v1 / 2;

$\{1 \leq a \leq 10 \implies 1 \leq v_2 \leq 10\}$

Why3 and WP

Consider this example:

```
-- precondition: 1 ≤ a ≤ 10
v1 := a + 5;
v2 := v1 / 2;
-- postcondition 1 ≤ v2 ≤ 10
```

Weakest Precondition

$\{1 \leq a \leq 10 \implies -3 \leq a \leq 15\}$
v1 := a + 5;
$\{1 \leq a \leq 10 \implies 2 \leq v_1 \leq 20\}$
v2 := v1 / 2;
$\{1 \leq a \leq 10 \implies 1 \leq v_2 \leq 10\}$

Why3 and WP

Consider this example:

```
-- precondition: 1 ≤ a ≤ 10
v1 := a + 5;
v2 := v1 / 2;
-- postcondition 1 ≤ v2 ≤ 10
```

Weakest Precondition

$\{\}$

v1 := a + 5;

$\{1 \leq a \leq 10 \implies 1 \leq v_1/2 \leq 10\}$

v2 := v1 / 2;

$\{1 \leq a \leq 10 \implies 1 \leq v_2 \leq 10\}$

Why3 and WP

Consider this example:

```
-- precondition: 1 ≤ a ≤ 10
v1 := a + 5;
v2 := v1 / 2;
-- postcondition 1 ≤ v2 ≤ 10
```

Weakest Precondition

$\{1 \leq a \leq 10 \implies 1 \leq (a+5)/2 \leq 10\}$

v1 := a + 5;

$\{1 \leq a \leq 10 \implies 1 \leq v_1/2 \leq 10\}$

v2 := v1 / 2;

$\{1 \leq a \leq 10 \implies 1 \leq v_2 \leq 10\}$

Why3 and WP

We've had this: $1 \leq a \leq 10 \implies 1 \leq (a + 5)/2 \leq 10$

Why3 and WP

We've had this: $1 \leq a \leq 10 \implies 1 \leq (a+5)/2 \leq 10$

example.smt2

```
(set-logic QF_LIA)
(declare-const a Int)
(assert (<= 1 a 10))
(assert (not (<= 1 (/ (+ a 5) 2) 10)))
(check-sat)
```

Why3 and WP

We've had this: $1 \leq a \leq 10 \implies 1 \leq (a+5)/2 \leq 10$

example.smt2

```
(set-logic QF_LIA)
(declare-const a Int)
(assert (<= 1 a 10))
(assert (not (<= 1 (/ (+ a 5) 2) 10)))
(check-sat)
```

Running CVC4...

```
$ cvc4 example.smt2
unsat
```

... postcondition proven!

Why3 and WP

But programs are a bit more complicated...

- $\{?\}$ if a then x := y; $\{x > 0\}$

Why3 and WP

But programs are a bit more complicated...

- $\{?\}$ if a then x := y; $\{x > 0\}$
- $\{(\neg a \implies x > 0) \wedge (a \implies y > 0)\}$
  if a then x := y;
  $\{x > 0\}$

Why3 and WP

But programs are a bit more complicated...

- $\{?\}$ if a then x := y; $\{x > 0\}$
- $\{(\neg a \implies x > 0) \wedge (a \implies y > 0)\}$
  if a then x := y;
  $\{x > 0\}$
- ...or split in graph, producing 2 VCs
- (but this means exponential explosion)

Why3 and WP

Loops are the main issue:

- $\{P\}$ while c do s $\{Q\}$

Why3 and WP

Loops are the main issue:

- $\{P\}$ while c do s $\{Q\}$
- Requires a loop invariant I, which is difficult to find
- Split into three VCs:

$$\begin{tikzcd} P \implies I \arrow[d] \\ \text{while c do s} \arrow[loop right, "\{I \wedge c\}\ \text{s}\ \{I\}"] \arrow[d] \\ I \wedge \neg c \implies Q \end{tikzcd}$$

Why3 and WP

Loop invariant example

```
procedure Find_Element (A     : in      Int_Array;
                        Elem  : in      Integer;
                        Idx   :     out Natural;
                        Found :     out Boolean)
with
   Global => null,
   Post   => (if Found
              then A (Idx) = Elem
              else (for all J in A'Range => A (J) /= Elem))
is
begin
   for I in A'Range loop
      Idx   := I;
      Found := A (I) = Elem;
      exit when Found;
      pragma Loop_Invariant
        ((for all J in A'First .. I => A (J) /= Elem)
         and not Found);
   end loop;
end Find_Element;
```

Conclusion

Today we've seen:

- Architecture of modern static analysis systems:
  Source $\rightarrow$ Intermediate Language $\rightarrow$ SMTLIB
- Overview of SPARK tool-set architecture
- Brief introduction to SMT
- Brief introduction to WP
- How this all comes together in SPARK2014

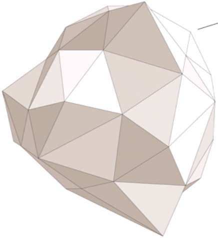
Conclusion

Today we've seen:

- Architecture of modern static analysis systems:
  Source $\rightarrow$ Intermediate Language $\rightarrow$ SMTLIB
- Overview of SPARK tool-set architecture
- Brief introduction to SMT
- Brief introduction to WP
- How this all comes together in SPARK2014

Thank you for your time and attention.
Also, we're hiring!
Any questions?

Made using only Free Software.



INNOVATION MAKERS

altran