THALES

Applications of Contracts for Security Certifications

Nikolai KOSMATOV

Joint work with Loïc CORRENSON, Lionel BLATTER, Adel DJOUDI, Martin HANA, Pascale LE GALL, Virgile PREVOSTO, Louis RILLING, Virgile ROBLES

Lorenz workshop “Contract Languages”
March 4-8, 2024

Tool context: ACSL, Frama-C and its deductive verification plugin WP

Frama-C is a platform for analysis and verification of C programs

- **ACSL (ANSI C Specification Language)** supported by Frama-C

WP plugin: Weakest Precondition based tool for deductive verification

- **Proof** of **semantic properties** of the program
- **Modular** verification (function by function)
- **Input**: a program and its specification in ACSL
- WP generates verification conditions (VCs)
- Relies on **Why3** and **Automatic Theorem Provers** to discharge VCs
  - **Alt-Ergo**, Z3, CVC4, CVC5, …

THALES

Example of a C program annotated in ACSL

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result != 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Can be proven with Frama-C/WP

THALES

Outline

- Motivation: Specification and verification of global (security) properties
- High-Level ACSL Requirements (HILARE), or Metaproperties, and MetAcsl tool
- Examples of Proof with MetAcsl and WP
- Application to certification of JavaCard Virtual Machine
- Relational properties
  - Specification and verification using self-composition
  - Verification using a VCGen: formalization and proof in Coq
- Conclusion and perspectives

Motivation: Why security properties are hard to specify and verify in Frama-C?

Integrity for some data area `data`: no satisfactory solutions:

`assigns \nothing;` or `assigns <what can be modified>;`

- Specifies that data area `data` ***cannot be modified by the function***
- **Does not work** if `data` can be modified only **under some condition** `cond` (access rights,…)

```
assigns data;
ensures !\old(cond)==> \at(data,Pre)== \at(data,Post);
```

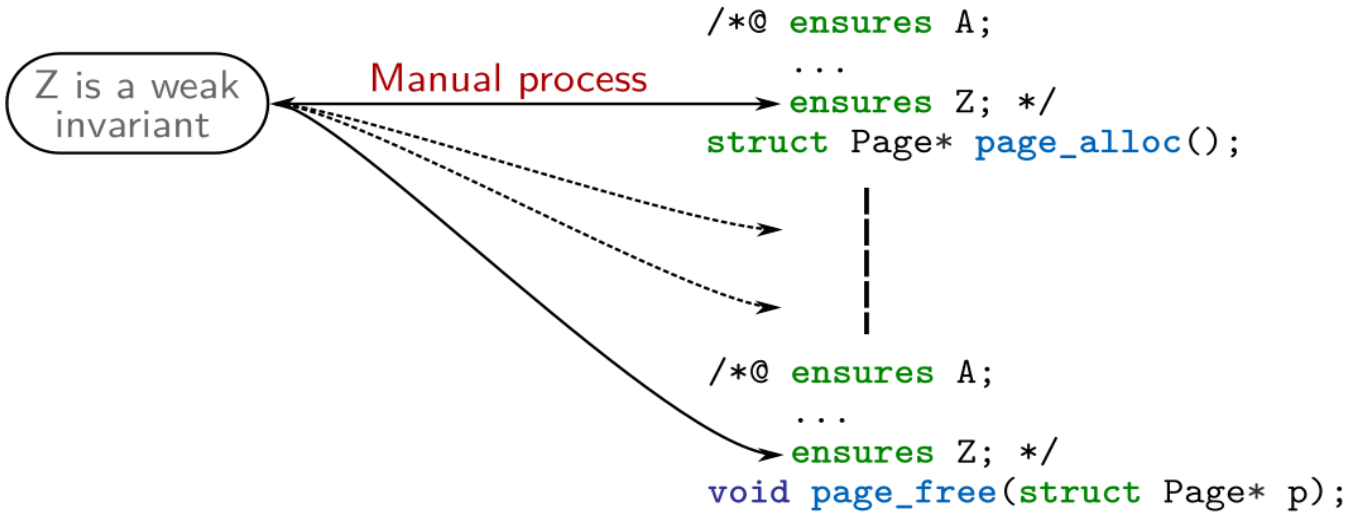
- Specifies that data area `data` is unchanged ***after the function*** if `cond` was false
- It does not forbid **internal modifications inside the function** (risk of attack)
- What if `cond` **can be modified?**
- How to ensure that `data` **cannot be modified even temporarily inside the function**?

Confidentiality: no direct solution at all in Frama-C

THALES

Motivation: Global (High-Level) properties are hard to specify and to maintain

Specifying global properties with contracts: manual and tedious. No explicit link between clauses.



Assessing if contracts form a global property is difficult, especially after an update.

THALES

Examples of High-Level Properties

- A non-privileged user never reads a privileged (private) data page
- A privileged user never writes to a non-privileged (public) page
- The privilege level of a page cannot be changed unless...
- The privilege level of a user cannot be changed unless…
- A free page cannot be read or written, and must contain zeros
- Object data can be written only by the object owner
- Object data can be read only by the object owner

Such properties can be expressed as

- Constraints on reading / writing operations, calls to some functions,
- Strong or weak invariants

Solution: Metaproperties, or HILARE (High-Level ACSL Requirements)

We introduce meta-properties, which are a combination of:

- **A set of targets functions**, on which the property must hold.
  `foo` `{foo, bar}` `\ALL` `\diff(\ALL,{foo, bar})`
- **A context**, which characterizes the situation in which the property must hold.
  `\strong_invariant` `\writing` `\reading`
- **An ACSL predicate**, expressed over the set of global variables.
  `A < B` `*p == 0` `\separated(\written, p)`

```
meta \prop,
    \name(A < B everywhere in foo and bar),
    \targets({foo, bar}),
    \context(\strong_invariant),
    A < B;
```


THALES

Available Contexts

- **Strong invariant:** Everywhere in the function
- **Weak invariant:** Before and after the function
- **Upon writing:** Whenever the memory is modified. The predicate can use a special meta-variable `\written`, referencing the address(es) being written to at a particular point.

```
meta \prop, \name(X is only modified if null),
       \targets(\ALL), \context(\writing),
       !\separated(\written, &X) ⇒ X == 0;
```

- **Upon reading:** Similarly, when memory is read
- **Upon calling:** Similarly, when a function is called

```
meta \prop, \name(foo can only be called from bar),
       \targets(\diff(\ALL, bar)),
       \context(\calling), \called ≠ &foo;
```

Example: Integrity Metaproperty Verified with MetAcsl – Writing context

Resulting code after generating assertions with MetAcsl and proof with Frama-C/WP:

```
/*@ meta "A_unchanged_unless";
  */
/*@ requires
    ensures
      (C ≥ 0
      (C < 0
    assigns A,
  */
void foo(voi
{
  if (C >= 0) {
    /*@ check A_unchanged_unless: _1: meta: C < 0 → \separated(&A, &A);
    A = C;
    /*@ check A_unchanged_unless: _2: meta: C < 0 → \separated(&B, &A);
    B = C;
  }
  return;
}
```

If all instances are proved, the metaproperty is true

MetAcsl

Contrary to an assert, a check is not kept in the proof context and does not overload the proof

MetAcsl instantiates a metaproperty in all relevant locations

Initial C code:

test5.c

```
1 int A, B, C;
2 /*@
3   meta \prop, \name(A_unchanged_unless),
4        \targets(\ALL), \context(\writing),
5   C < 0 ==> \separated(\written, &A);
6 */
7 /*@
8   requires A==B;
9   assigns A,B;
10  ensures C>=0 && A==C && B==C ||
11    C<0 && A==\old(A) && B==\old(B); */
12 void foo(){
13   if ( C >= 0 ){
14     A = C;
15     B = C;
16   }
```


THALES

Example: Confidentiality Metaproperty Verified with MetAcsl – Reading context

Resulting code after generating assertions with MetAcsl and proof with Frama-C/WP:

```
/*@ meta "A_not_read";
*/
/*@ requires A ≡ B;
    ensures
      (C ≥ 0 ∧ A ≡ C ∧ B ≡ C) ∨
      (C < 0 ∧ A ≡ \old(A) ∧ B ≡ \old(B));
    assigns A, B;
 */
void foo(void)
{
  /*@ check A_not_read: _1: meta: \separated(&C, &A); */
  if (C >= 0) {
    /*@ check A_not_read: _2: meta: \separated(&C, &A); */
    A = C;
    /*@ check A_not_read: _3: meta: \separated(&C, &A); */
    B = C;
  }
  return;
}
```

MetAcsl

Initial C code:

test4.c

```
1 int A, B, C;
2 /*@
3   meta \prop, \name(A_not_read),
4     \targets(\ALL), \context(\reading),
5     \separated(\read, &A);
6 */
7 /*@
8   requires A==B;
9   assigns A,B;
10  ensures C>=0 && A==C && B==C ||
11    C<0 && A==\old(A) && B==\old(B); */
12 void foo(){
13   if ( C >= 0 ){
14     A = C;
15     B = C;
16   }
17 }
18
```


THALES

Example: Strong Invariant

Resulting code after generating assertions with MetAcsl and proof with Frama-C/WP:

```
int A;
int B;
int C;
/*@ meta "A_B_eq_strong";
*/
/*@ check requires A_B_eq_strong: _1: meta: A ≡ B;
    requires A ≡ B;
    check ensures A_B_eq_strong: _1: meta: A ≡ B;
    ensures
      (C ≥ 0 ∧ A ≡ C ∧ B ≡ C) ∨
      (C < 0 ∧ A ≡ \old(A) ∧ B ≡ \old(B));
    assigns A, B;
 */
void foo(void)
{
  if (C >= 0) {
    A = C;
    /*@ check A_B_eq_strong: _3: meta: A ≡ B; */ ;
    B = C;
    /*@ check A_B_eq_strong: _4: meta: A ≡ B; */ ;
  }
  /*@ check A_B_eq_strong: _2: meta: A ≡ B; */ ;
  return;
  /*@ check A_B_eq_strong: _5: meta: A ≡ B; */ ;
}
```

Initial C code:

test2.c

```
1 int A, B, C;
2 /*@
3   meta \prop, \name(A_B_eq_strong),
4     \targets(\ALL), \context(\strong_invariant),
5     A == B;  // FAILS
6 */
7 /*@
8   requires A==B;
9   assigns A,B;
10  ensures C>=0 && A==C && B==C ||
11    C<0 && A==\old(A) && B==\old(B); */
12 void foo(){
13   if ( C >= 0 ){
14     A = C;
15     B = C;
16   }
17 }
18
```

Does not hold as a strong invariant betw. lines 14,15

THALES

Example: Weak Invariant

Resulting code after generating assertions with MetAcsl and proof with Frama-C/WP:

```
int A;
int B;
int C;
/*@ meta "A_B_eq_weak";
*/
/*@ check requires A_B_eq_weak: _1: meta: A ≡ B;
    requires A ≡ B;
    check ensures A_B_eq_weak: _1: meta: A ≡ B;
    ensures
      (C ≥ 0 ∧ A ≡ C ∧ B ≡ C) ∨
      (C < 0 ∧ A ≡ \old(A) ∧ B ≡ \old(B));
    assigns A, B;
 */
void foo(void)
{
  if (C >= 0) {
    A = C;
    B = C;
  }
  return;
}
```

Initial C code:

test3.c

```
1 int A, B, C;
2 /*@
3   meta \prop, \name(A_B_eq_weak),
4     \targets(\ALL), \context(\weak_invariant),
5     A == B;
6 */
7 /*@
8   requires A==B;
9   assigns A,B;
10  ensures C>=0 && A==C && B==C ||
11    C<0 && A==\old(A) && B==\old(B); */
12 void foo(){
13   if ( C >= 0 ){
14     A = C;
15     B = C;
16   }
17 }
18
```

Holds as a weak invariant

THALES

Examples of HILAREs

```
meta \prop, \name(Do not write to lower pages outside free),
  \targets(\diff(\ALL , {page_free})),
  \context( \writing ),

  \forall integer i; 0 <= i < MAX_PAGE_NB ==>
  \let p = pages + i;
  p->status == PAGE_ALLOCATED &&
  user_level > p->confidentiality_level ==>
  \separated(\written, p->data + (0.. PAGE_SIZE - 1));

meta \prop, \name(Free pages are never read),
  \targets(\ALL),
  \context( \reading ),

  \forall integer i; 0 <= i < MAX_PAGE_NB &&
  pages[i].status == PAGE_FREE ==>
  \separated(\read, pages[i].data + (0 .. PAGE_SIZE - 1));
```

Application to certification of JavaCard Virtual Machine

- Common Criteria Certification
- JavaCard Virtual Machine
- General approach
- Proof issues and statistics

THALES

Common Criteria: Evaluation assurance levels (EAL)

Assurance class	Assurance Family	Assurance Components by Evaluation Assurance Level						
		EAL1	EAL2	EAL3	EAL4	EAL5	EAL6	EAL7
Development	ADV_ARC		1	1	1	1	1	1
	ADV_FSP	1	2	3	4	5	5	6
	ADV_IMP				1	1	2	2
	ADV_INT					2	3	3
	ADV_SPM						1	1
	ADV_TDS		1	2	3	4	5	6
Guidance documents	AGD_OPE	1	1	1	1	1	1	1
	AGD_PRE	1	1	1	1	1	1	1
Life-cycle support	ALC_CMC	1	2	3	4	4	5	5
	ALC_CMS	1	2	3	4	5	5	5
	ALC_DEL		1	1	1	1	1	1
	ALC_DVS			1	1	1	2	2
	ALC_FLR							
	ALC_LCD			1	1	1	1	2
	ALC_TAT				1	2	3	3
Security Target evaluation	ASE_CCL	1	1	1	1	1	1	1
	ASE_ECD	1	1	1	1	1	1	1
	ASE_INT	1	1	1	1	1	1	1
	ASE_OBJ	1	2	2	2	2	2	2
	ASE_REQ	1	2	2	2	2	2	2
	ASE_SPD		1	1	1	1	1	1
	ASE_TSS	1	1	1	1	1	1	1
Tests	ATE_COV		1	2	2	2	3	3
	ATE_DPT			1	1	3	3	4
	ATE_FUN		1	1	1	1	2	2
	ATE_IND	1	2	2	2	2	2	3
Vulnerability assessment	AVA_VAN	1	2	2	3	4	5	5

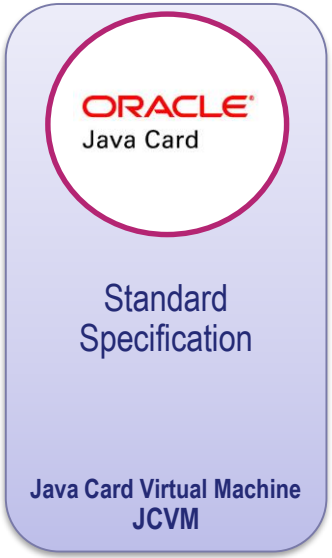
EAL1	Functionally tested
EAL2	Structurally tested
EAL3	Methodically tested and checked
EAL4	Methodically designed, tested and reviewed
EAL5	Semiformally designed and tested
EAL6	**Semiformally verified design and tested**
EAL7	**Formally verified design and tested**

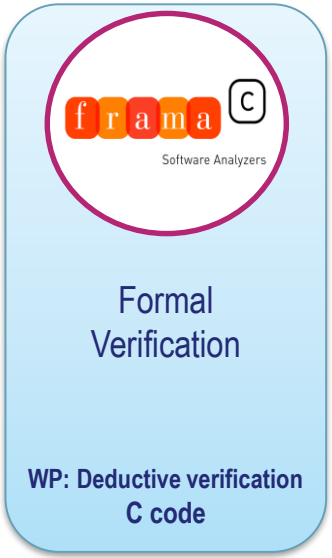
Source:
CCpart3v3.1 - Table 1
(https://www.commoncriteriaportal.org/cc/)

Common Criteria: Certified products (consulted on March 7, 2024)

Certified Products by Assurance Level and Certification Date																
EAL	**2010**	**2011**	**2012**	**2013**	**2014**	**2015**	**2016**	**2017**	**2018**	**2019**	**2020**	**2021**	**2022**	**2023**	**2024**	**Total**
Basic	0	0	0	0	0	0	0	0	0	0	1	4	38	44	0	87
EAL1	0	0	0	0	0	0	0	0	0	3	4	3	2	0	1	13
EAL1+	0	0	0	0	0	0	0	0	0	0	1	0	2	2	0	5
EAL2	0	0	0	0	0	0	0	1	0	17	15	39	12	12	1	97
EAL2+	0	0	0	0	0	2	1	5	2	28	43	35	30	33	1	180
EAL3	0	0	0	0	0	2	0	0	0	9	9	4	0	2	2	28
EAL3+	0	0	0	0	0	3	1	0	1	4	12	18	29	13	1	82
EAL4	0	0	0	0	0	0	3	0	5	6	5	3	2	3	0	27
EAL4+	1	0	0	0	1	3	7	5	6	44	60	66	73	90	8	364
EAL5	0	0	0	0	0	0	1	0	0	0	2	0	4	2	0	9
EAL5+	0	0	0	0	2	2	4	17	12	41	69	44	39	77	14	321
EAL6	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	2
EAL6+	0	0	0	0	0	1	0	0	0	20	20	30	33	37	2	143
EAL7	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	3
EAL7+	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	2
Medium	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
None	0	0	0	0	0	0	0	0	0	38	44	77	75	113	13	360
US Standard	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Totals:	**1**	**0**	**0**	**0**	**3**	**13**	**17**	**28**	**26**	**210**	**286**	**324**	**341**	**431**	**43**	**1723**

Context: three fields of expertise





THALES
Building a future we can all trust

- **C** implementation of the **Standard Specification** of the **JCVM**
- **Formal Security Properties** meet **Security Assurance Requirements**
- **Formal verification** of global formal security properties using **Frama-C/WP**

THALES

JCVM: Standard Specification (1/2)

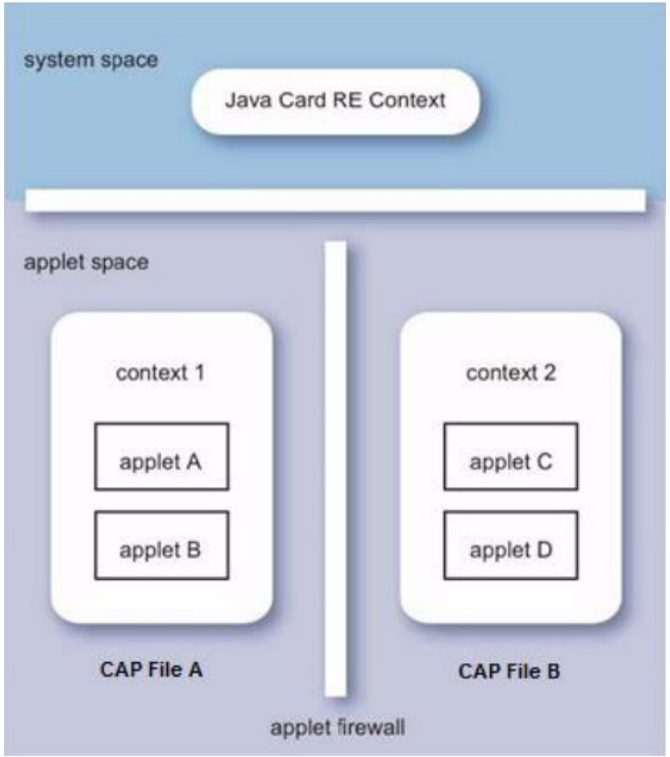
- Execute Java Card applications' bytecode with **basic operations**
- Bytecodes are read iteratively inside the main **dispatch loop**

- 3 main memory areas: Java **stack**, data **heap** and **code** area
- 3 types of **heap** memory: **persistent**, **transient reset/deselect**

- A **unique context** assigned to each Java Card binary (CAP file)
- **Object owner context** is stored inside the object header

JCVM: Standard Specification (2/2)

- The **Firewall guarantees isolation** of heap data between different contexts
- Java Card Runtime Environment (**JCRE**) context is a **privileged context** devoted to system operations
- **Well-defined exceptions:** global arrays, shareable interfaces,…



EAL6-EAL7: Formal verification of Security Properties

Security Aspect

#.Firewall: "*The Firewall shall ensure controlled sharing of class instances, and **isolation of their data and code between packages** (that is, controlled execution contexts) as well as between packages and the JCRE context...*"

[Java Card System – Open Configuration Protection Profile – V3.1]

Security properties (simplified examples)

- **integrity_header**: allocated objects' headers cannot be **modified** during a VM run.
- **integrity_data**: allocated objects' data can be **modified** only by the owner.
- **confidentiality_data**: allocated objects' data can be **read** only by the owner.

Evaluation Assurance Levels

Formal verification of security properties

Frama-C/WP: Formal Deductive Verification

```
/*@
requires P;
assigns  L;
ensures  E;
*/
<type> function(<type> arg1,<type> arg2, …){

    …

    /*@
    loop invariant I;
    loop assigns   L;
    loop variant   m;
    */
    while (c){
     …
    }

     …
}
```

ACSL function contract

ACSL loop contract

Formal Specification Structure

Basic level

STEP1: Write ACSL annotations
(Formal Specification)

STEP2: Frama-C/WP computes proof goals
(Based on Hoare logic)

STEP3: Discharge proof goals with
(QED, Alt-Ergo via Why3, …)

Advanced level features

Ghost code

Predicates, Lemmas

Proof scripts

THALES

Verification of security properties with MetAcsl

THALES
Building a future we can all trust

- **Integrity_data and Confidentiality_data cannot be verified with WP** as global invariants
- We use metaproperties:

name

targets all function(s)

application context:
whenever a location is read

```
meta \prop,\name(meta_persi_objects_confident),\targets(\ALL),\context(\reading),

( \forall integer i; 0 <= i < gNumObjs && !gIsTrans[i] &&
                     ObjHeader[gHeadStart[i] + 0] != JCC ==>
  \separated(\read,PersiData+(gDataStart[i]..gDataEnd[i])) ); */
```

The read location must be separated from the data of any persistent object if the current context is not its owner.

- **MetAcsl** translates metaproperties into **assertions/checks** at each relevant program point.
- If all **assertions/checks** are proved, the metaproperty is proved.
- Thanks to the translation of metaproperties into **checks** that do not overload proof contexts, the metaproperty-based approach scales very well, despite a great number of generated annotations.

THALES

Specification effort

JCVM C code		ACSL Annotations			
		User provided annotations		MetAcsl	RTE
# Functions	# Loc C	# Loc Ghost	# Loc ACSL	# Loc ACSL	# Loc ACSL
381	7,014	162	35,480	396,603	2,290

Large code

A few yet necessary

12,432 before preprocessing macros that gather redundant annotations

Still a considerable effort

Automatically generated from **36** metaproperties only

- **User-provided annotations**: predicates, lemmas, function contracts, loop contracts and other assertions
- **MetAcsl**: automatically generated annotations according to user-defined metaproperties
- **RTE:** automatically generated annotations in order to prevent undefined behaviors

THALES

Some Issues (I), Solutions (S) and Perspectives (P)

Companion ghost model

- **I**: **Automatic proof fails** on low-level code (bit-fields)
- **S:** Linking bits to ghost integer variables brings the **prover back into its comfort zone**
- **P:** **Proof at the abstract level** for some properties can help [as discussed at Dagstuhl]

Proof scripts for complex predicates

- **I:** **Automatic proof fails** to use the right predicates
- **S:** **Guide the first proof steps** by unfolding relevant predicates or instantiating values
- **S&P:** **New proof strategy mechanism** to generate scripts automatically [TACAS'24]

Carefully chosen lemmas

- **I:** **Automatic proof fails** repeatedly in similar cases
- **S:** Lemmas help to **re-use the same reasoning**

THALES

New proof strategy mechanism to generate scripts automatically [TACAS'24]

Instead of creating the proof script interactively in Frama-C/WP

- **With much time** spent for try-and-wait-and-debug attempts

The verification engineer creates a proof strategy

- Written directly **in the source code** as a special annotation
- Including one or **several alternatives** (proof tactics) to try
  - unfolding, rewriting, enumerating, calling a solver,…
- Indicating possible strategies to apply on the **resulting proof goals** (children)
- Possibly attached to **specific proof goals**
- Typically, applied to **help automatic SMT solvers** to prove the goal

The tool automatically tries to apply provided strategies and records a proof script when the proof succeeds

New proof strategy mechanism : examples

Example 1: a lemma unproven in Frama-C/WP with Alt-Ergo

```
lemma vhm_preserved{L1,L2}:
  valid_heap_model{L1} ∧
  mem_model_footprint_intact{L1,L2} ∧
  \at(gNumObjs,L1) == \at(gNumObjs,L2) ∧
  object_headers_intact{L1,L2}
 ⇒ valid_heap_model{L2};
```

A proof strategy that generates a script proving the lemma

```
1   strategy FastAltErgo:  \prover("alt-ergo", 1); // run Alt-Ergo for 1s
2   strategy EagerAltErgo: \prover("alt-ergo",10); // run Alt-Ergo for 10s
3   strategy UnfoldVhmThenProver:                  // Strategy with three steps:
4     FastAltErgo,                                 // 1) fast prover attempt
5     \tactic("Wp.unfold",                         // 2) if unproved, unfold
6       \pattern(P_valid_heap_model((..))),        // predicate valid_heap_model
7       \children(UnfoldVhmThenProver) ),          // and apply itself recursively
8     EagerAltErgo;                                // 3) longer prover attempt
9   proof UnfoldVhmThenProver: vhm_preserved;      // Associate strategy to goal
```


THALES

New proof strategy mechanism : examples

Example 1: a lemma unproven in Frama-C/WP with Alt-Ergo

```
1 lemma dn3:
2   ∀ unsigned char c d;
3  (c & 0x8E) == 2 ∧
4  (c & 0x01) == 1 ∧
5  (d & 0x8F) == 0
6   ⇒ ((c+d) & 0x03) == 0x03;
```

A proof strategy that generates a script proving the lemma

```
1 strategy RangeThenProver:
2   \tactic ("Wp.range",
3     \pattern(is_uint8(e)),
4     \select(e),
5     \param("inf",0),\param("sup",255),
6     \children(RangeThenProver) ),
7   \prover("alt-ergo",2);
8 proof RangeThenProver: dn3;
```


THALES

New proof strategy mechanism : initial experiments

Applied to the proof of the real-life JCVM code at Thales

- 8,000+ lines of C and 30,000+ lines of ACSL
- Complete proof for 85,000 goals using Alt-Ergo with a 250s timeout requires 800+ proof scripts.

With the new extension: significant time savings

- after a manual creation of strategies (~2 days),
- WP automatically produces more than 50% of the required scripts, whose
- Their manual creation would take ~1 person-month.

An even greater number of proof scripts is expected to be generated from strategies

- This will strongly facilitate industrial verification

THALES

Relational properties

- Specification and verification using self-composition
- Verification using a VCGen: formalization and proof in Coq

Relational properties (RPs): how to relate program calls

Monotonicity ?

$$\forall x_1, x_2,\ x_1 < x_2 \Rightarrow f(x_1) < f(x_2)$$

```
/*@ requires 1000 > x;
  @ ensures ?;
  @ assigns \nothing;*/
int f (int x){
   return x + 1;
}
```

How to specify this property ?

How to prove this property ?

How to use this property ?

Specification of relational properties (RPs) in Frama-C

Extension of ACSL:

- New clause relational
- New built-in \callpure

Example with pure function:

```
/*@ requires 1000 > x;
  @ relational \forall int x1,x2; x1 < x2  ==>
  @              \callpure(f,x1) < \callpure(f,x2);
  @ assigns \nothing;*/
int f (int x){
    return x + 1;
}
```

Proposal 1: Proof of relations properties by self-composition [TACAS'17]

- Inspired by Self-composition [Barthe et al (2011)]
- Inline involved function calls
- Express the RP as a standard ACSL assertion

```
void relational_wrapper_1(int x1, int x2){
    int return_1 = x1 + 1;
    int return_2 = x2 + 1;
    /*@ assert x1 < x2 ==> return_1 < return_2; */
    return;
}
```

Express the RP in ACSL

THALES

Proposal 1: Use of relations properties as hypotheses

```
/*@ axiomatic RP_axiom {
  @  logic int f_acsl(int x);
  @
  @ lemma RP_lemma:
  @   \forall int x1, int x2; x1 < x2 ==>
  @         f_acsl(x1) < f_acsl(x2);
}*/

/*@ requires 1000 > x;
  @ assigns \nothing;
  @ ensures \result == f_acsl(x);*/
int f(int x) {
   return x + 1;
}

/*@ relational \forall int x1,x2; x1 < x2  ==>
  @            \callpure(g,x1) < \callpure(g,x2);*/
int g(int x){
    return f(x) + 1;
}
```

Valid if the inserted assertion holds

Assumed: bridge betw. `f` and `f_acsl`

Proved by using the RP for `f`

THALES

A large range of relational properties addressed

Relational Properties (RPs) :

- Invoking at least two function calls
- Invoking possibly dissimilar functions
- Invoking possible nested calls

$$\forall \sigma_1, \ldots, \sigma_n$$

$$P(\sigma_1, \ldots, \sigma_n, [\![f_1]\!]\sigma_1, \ldots, [\![f_n]\!]\sigma_n)$$

A large range of relational properties addressed: some examples

$$\forall x1, x2 \in \mathbb{Z} : \\ x1 < x2 \Rightarrow f(x1) < f(x2)$$

$$\forall\ x; \\ f(x+1) = f(x) * (x+1)$$

$$\forall x,\ f_1(x) \leq f_2(x) \leq f_3(x)$$

$$\forall x, f(f(x)) = f(x)$$

$$\forall\ Msg, Key; \\ Decrypt(Encrypt(Msg, Key), Key) = Msg$$

$$\forall\ t, sub_{t1}, \ldots, sub_{tn}; \\ t = sub_{t1} \cup \ldots \cup sub_{tn} \Rightarrow \\ max(t) = max(max(sub_{t1}), \ldots, max(sub_{tn}))$$

$$\forall x1, x2, y, f(x1 + x2, y) = f(x1, y) + f(x2, y)$$

$$\forall a, b, c,\ \mathrm{Med}(a, b, c) = \mathrm{Med}(a, c, b)$$

THALES

Relational Property Verification: Related Work

- Proof systems (Relational Hoare Logic [Benton, POPL 2004], Relational Separation Logic [Yang. TCS 2007] Cartesian Hoare Logic [Sousa and Dillig. PLDI 2016], Equivalence proofs [Beckert, Ulbrich]...):
  - Separated memory state for each program.
  - Require a dedicated decision system.
  - No modular proofs or relational contracts.
- Code transformations (Self-Composition [Barthe et al. MSCS 2011], Program Products [Barthe, LAMP. 2016], ...):
  - Allow use of verification methods for Hoare Triples.
  - Require renaming and joining the memory state of each tag.
  - No modular proofs or relational contracts.

Proposal 2: Proof of relations properties via a VCGen [iFM'22, ISOLA'22]

- Relational property verification based on a VCGen (verification condition generator)
- Enabling modular verification of relational properties
- Fully formalized and proved sound in the Coq proof assistant for a while language with procedures and aliasing.

THALES

Relational property, formally

A Relational Property is a property about n programs $c_1, ..., c_n$: if each program c_i starts in σ_i and ends in $\sigma'_i = [\![c_i]\!]_\psi \sigma_i$ such that $(\sigma_1, ..., \sigma_n)$ satisfies $\widehat{P}$, then $(\sigma'_1, ..., \sigma'_n)$ satisfies $\widehat{Q}$.

$$\forall \sigma_1, \ldots, \sigma_n, \widehat{P}(\sigma_1, \ldots, \sigma_n) \Rightarrow \widehat{Q}(\sigma_1, \ldots, \sigma_n, [\![c_1]\!]_\psi \sigma_1, \ldots, [\![c_n]\!]_\psi \sigma_n)$$

We use following notation for Relational Properties:

$$\psi : \{\widehat{P}\}(c_k)^n\{\widehat{Q}\}.$$

THALES

Example of relational property

We want to prove that both programs below are equivalent:

$$\{\ x_2\langle 1\rangle = x_2\langle 2\rangle\ \}\quad \begin{array}{l} x_1 := 1; \\ x_3 := 0; \\ \textbf{call}(y_{\mathrm{sum}}) \end{array} \quad \langle 1\rangle \sim \quad \begin{array}{l} x_1 := 0; \\ x_3 := 0; \\ \textbf{call}(y_{\mathrm{sum}}) \end{array} \quad \langle 2\rangle\ \{\ x_3\langle 1\rangle = x_3\langle 2\rangle\ \}$$

With procedure environment $\{y_{\mathrm{sum}} \to \mathrm{body}(y_{\mathrm{sum}})\}$ where:

$$\mathrm{body}(y_{\mathrm{sum}}) = \begin{array}{l} \textbf{if } x_1 < x_2 \textbf{ then } \{ \\ \quad x_3 := x_3 + x_1; \\ \quad x_1 := x_1 + 1; \\ \quad \textbf{call}(y_{\mathrm{sum}}) \\ \} \textbf{ else } \{ \textbf{ skip } \} \end{array}$$

Verification Condition Generator (VCGen): main principle

Construction of first-order formulas whose validity implies the validity of the Hoare Triple.

- Naive generation: *if* statements make the size of the formulas grow exponantially.
- Optimized generation: the formulas size is lineare in the size of the initial program.

Verification Condition Generator (VCGen): main principle

- $\mathcal{T}_c$ generates the main verification condition: the postcondition holds in the final state, assuming auxiliary annotations hold;
- $\mathcal{T}_a$ generates auxiliary verification conditions stemming from assertions, loop invariants, and preconditions of called procedures;
- $\mathcal{T}_f$ generates verification conditions for the auxiliary procedures.

Theorem: VCGen is sound

If

$$\forall \sigma \in \Sigma, P(\sigma) \Rightarrow \mathcal{T}_c[\![c]\!](\sigma, \phi, Q),$$

$$\forall \sigma \in \Sigma, P(\sigma) \Rightarrow \mathcal{T}_a[\![c]\!](\sigma, \phi),$$

$$\mathcal{T}_f(\phi, \psi),$$

then we have $\psi : \{P\}c\{Q\}$.

Verification of relational properties with VCGen

What do we want

For each command in a relational property, we separately generate the associated logical formulas.

Proposal

Using extensions of $\mathcal{T}_c$, $\mathcal{T}_a$ and $\mathcal{T}_f$, we translate the relational verification problem directly into first-order formulas.

Relational Verification Condition Generator (VCGen)

Construction of first-order formulas whose validity implies the validity of the Relational Property.

- $\mathcal{T}_{cr}$ generates the main verification condition (using $\mathcal{T}_c$): the relational postcondition holds in the final state, assuming auxiliary annotations hold;
- $\mathcal{T}_{ar}$ generates auxiliary verification conditions (using $\mathcal{T}_a$) stemming from assertions, loop invariants, and preconditions of called procedures;
- $\mathcal{T}_{pr}$ translates relational contracts into first order formulas;
- $\mathcal{L}$ lifts relational contract environment into a standard contract environment;
- $\mathcal{T}_{fr}$ generates verification conditions for the procedures: procedure bodies respect their relationals contracts.

Relational Verification Condition Generator (VCGen) is sound

Theorem: Relational VCGen is sound

If

$$\forall (\sigma_k)^n, (\sigma'_k)^n, \psi', \quad \widehat{P}((\sigma_k)^n) \wedge \mathcal{T}_{pr}(\widehat{\phi}, \psi') \Rightarrow$$
$$\mathcal{T}_{cr}((c_k)^n, (\sigma_k)^n, (\sigma'_k)^n, \mathcal{L}(\widehat{\phi}, \psi'))\ \lambda p.\, p \Rightarrow \widehat{Q}((\sigma_k)^n, (\sigma'_k)^n),$$

and

$$\forall (\sigma_k)^n, \psi', \quad \widehat{P}((\sigma_k)^n) \wedge \mathcal{T}_{pr}(\widehat{\phi}, \psi') \Rightarrow$$
$$\mathcal{T}_{ar}((c_k)^n, (\sigma_k)^n, \mathcal{L}(\widehat{\phi}, \psi')),$$

and

$$\mathcal{T}_{fr}(\widehat{\phi}, \psi),$$

then we have $\psi : \{\widehat{P}\}(c_k)^n\{\widehat{Q}\}$.

Conclusion

Successful industrial application of deductive verification

- World-first proof of real-life JavaCard VM code
- EAL7 certificate issued by ANSSI, the French certification body
- Careful combination of: ghost code, lemmas, proof scripts, …
- High level of automation (99% of goals proved automatically)
- MetAcsl is crucial for specification of security properties
- Proof integrated into the Continuous Integration process
- Efficient tool support from Frama-C developers was essential

Promising approach for relational property verification based on a VCGen

- Modular verification of Relational Properties
- Separated memory state for each program
- Proven sound in the Coq proof assistant
- Relies on optimized verification condition generation

THALES

Ongoing and future work directions

Custom and more flexible proof strategies to save manual script creation effort

- New extension of Frama-C/WP for proof strategies to be presented at TACAS 2024
- About 50% of necessary proof scripts are generated automatically!

Reasoning about metaproperties and other annotations can be helpful

- Preliminary ideas proposed in Virgile Robles' PhD thesis
- Externalizing verification of metaproperties at the callsite for two functions reduced proof time by 1 hour!!

Scaling to large programs having parts with and without low-level operations, or where some of the maintained properties are irrelevant

- Collaborative memory models
- More abstract levels of reasoning

Developing industry-ready sound tools for verification of relational properties

Participating in collaborative projects to apply innovative verification techniques to Thales products

THALES

References

- Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall and Virgile Prevosto.
  “RPP: Automatic Proof of Relational Properties by Self-Composition.” **TACAS 2017**. Springer.
- Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall.
  “MetAcsl: Specification and Verification of High-Level Properties.” **TACAS 2019.** Springer.
- Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall.
  “Tame your annotations with MetAcsl: Specifying, Testing and Proving High-Level Properties”. **TAP 2019**. Springer.
- Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall.
  “Methodology for Specification and Verification of High-Level Properties with MetAcsl”. **FormaliSE 2021.** IEEE.
- Adel Djoudi, Martin Hana and Nikolai Kosmatov.
  “Formal verification of a JavaCard virtual machine with Frama-C”. **FM 2021.** Springer.
- Adel Djoudi, Martin Hána, Nikolai Kosmatov, Milan Kříženecký, Franck Ohayon, Patricia Mouy, Arnaud Fontaine and David Féliot.
  “A Bottom-Up Formal Verification Approach for Common Criteria Certification:
  Application to JavaCard Virtual Machine”. **ERTS 2022, Best paper award**.
- Lionel Blatter, Nikolai Kosmatov, Virgile Prevosto and Pascale Le Gall.
  “An Efficient VCGen-based Modular Verification of Relational Properties.” **ISOLA 2022.** Springer.
- Lionel Blatter, Nikolai Kosmatov, Virgile Prevosto and Pascale Le Gall.
  “Certified Verification of Relational Properties.” **iFM 2022**. Springer.
- Loïc Correnson, Allan Blanchard, Adel Djoudi and Nikolai Kosmatov.
  “Automate where Automation Fails: Proof Strategies for Frama-C/WP.” **TACAS 2024.** Springer. To appear.

THALES