

Formal verification of a JavaCard Virtual Machine with Frama-C

Initially presented at FM 2021

Nikolai KOSMATOV

Joint work with Adel DJOUDI, Martin HANA

DISC, Besançon, March 1, 2022



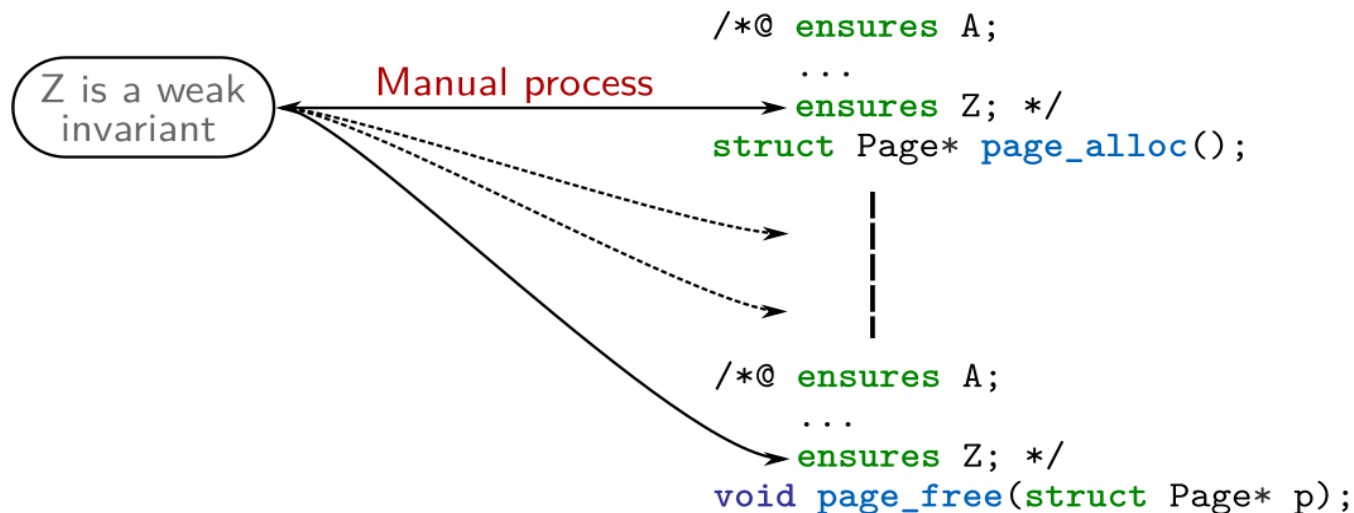
Part I*. Global Property Verification with MetAcsI

- **Motivation: Specification and verification of global properties**
- **Solution: High-Level ACSL Requirements (HILARE)**
- **Examples of Proof with MetAcsI and WP**

*Some slides prepared by Virgile Robles

Motivation: Global (High-Level) Properties

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.

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: Meta-properties, 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;
```

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)  $\Rightarrow$  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  $\neq$  &foo;
```

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));
```

Example: Strong Invariant

```
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; */ ;
}
```

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
```


Example: Weak Invariant

```
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;
}
```

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
```

Example: Reading Context

```
/*@ 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;
}
```

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
```

Example: Writing Context

```
/*@ meta "A_unchanged_unless";
*/
/*@ requires 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) {
        /*@ 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;
}
```

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     }
17 }
18
```

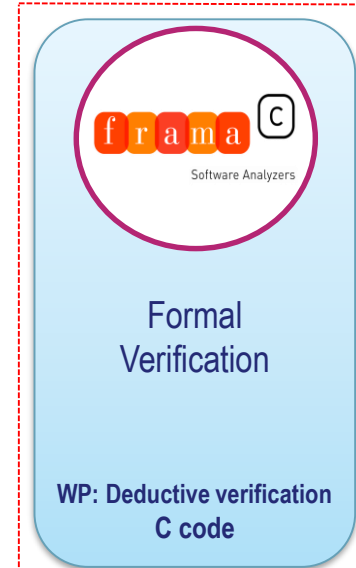
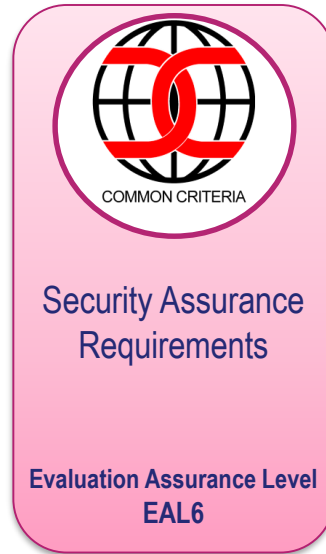
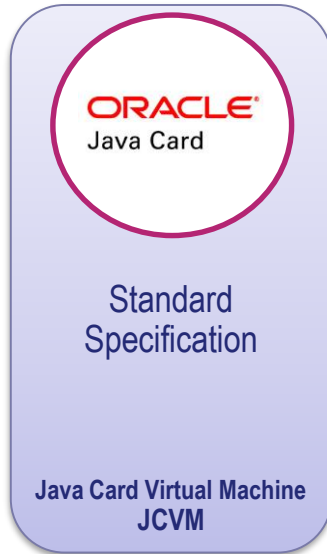
Introduction

General approach

Proof issues and solutions

Results and conclusion

Context: three fields of expertise





THALES
Building a future we can all trust

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

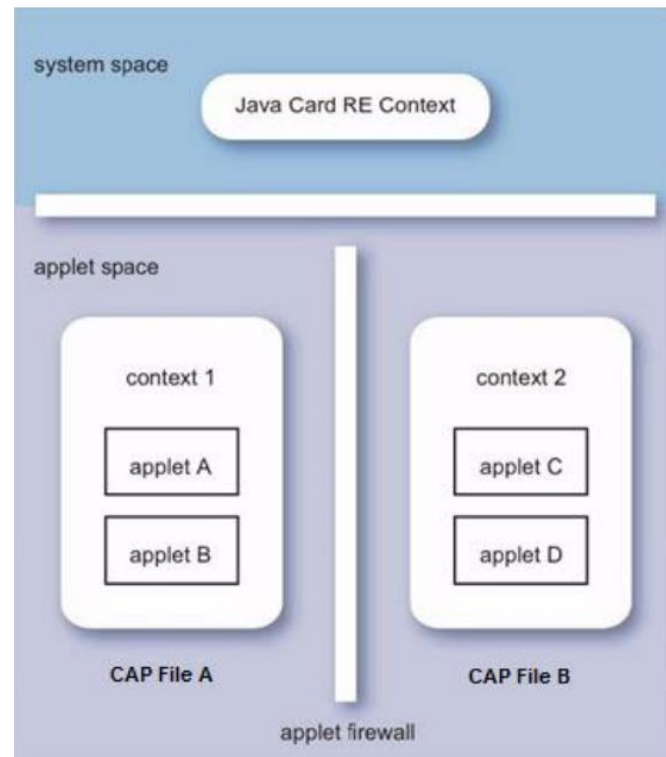


- Execute Java Card applications' bytecode with **basic operations**
- Bytecodes are read iteratively inside the main **dispatch loop**

- 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

- 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,...



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
Guidance documents	ADV_TDS		1	2	3	4	5	6
	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
Tests	ASE_TSS	1	1	1	1	1	1	1
	ATE_COV		1	2	2	2	3	3
	ATE_DPT			1	1	3	3	4
	ATE_FUN		1	1	1	1	2	2
Vulnerability assessment	ATE_IND	1	2	2	2	2	2	3
	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/>)

OPEN

THALES

EAL6: 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

EAL1	EAL2	EAL3	EAL4	EAL5	EAL6	EAL7
------	------	------	------	------	------	------

Formal verification

Formal verification of security properties

■ Introduction

■ General approach

■ Proof issues and solutions

■ Results and conclusion

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

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

    ...
}
```

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

This document may not be reproduced, modified, adapted, published, translated, in any way, in whole or in part or disclosed to a third party without the prior written consent of Thales - © Thales 2018 All rights reserved.

- **MetAcsI**: proves global properties (user needs only to specify metaproperties)

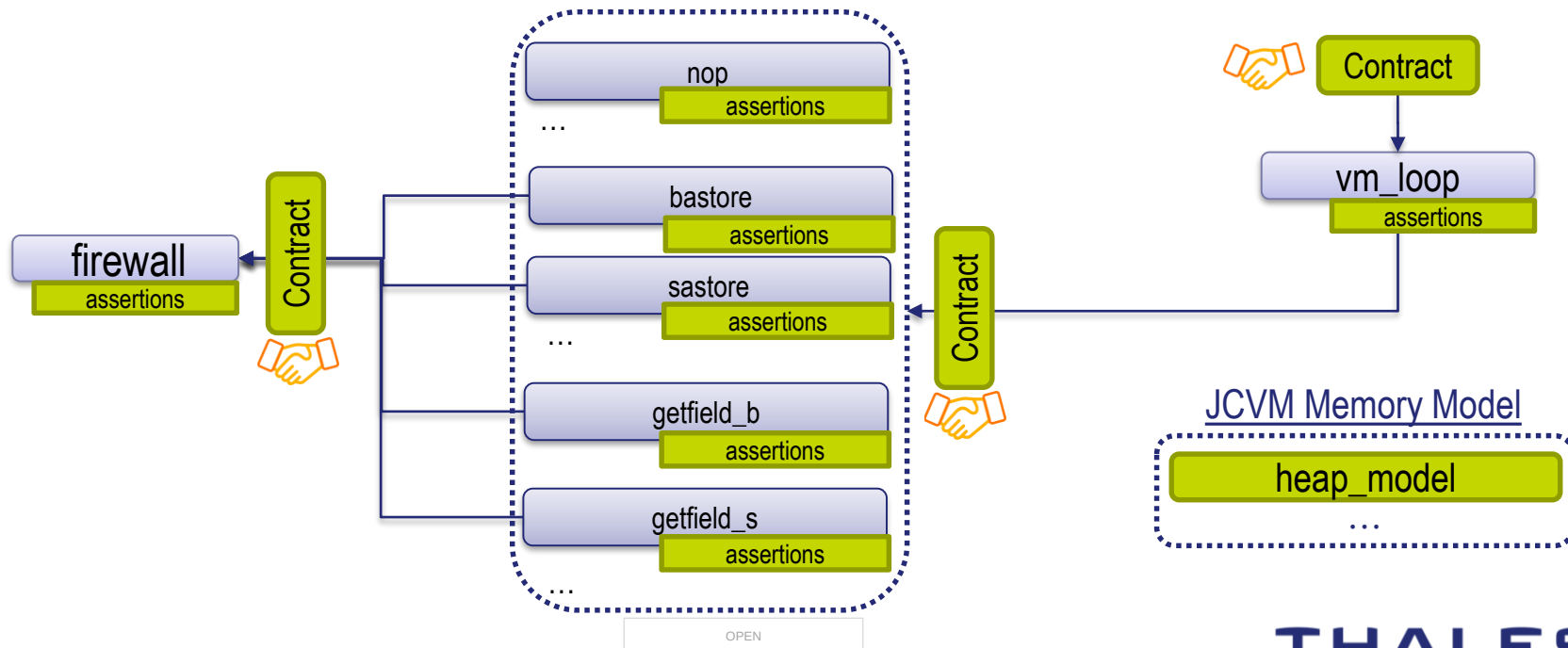
property

JCVM Call Graph (toy example)

Formal verification of security properties with Frama-C/WP

JCVM C code

ACSL annotations



Memory segments

- Object headers: `unsigned char ObjHeader[SEGM_SIZE];`
- Persistent/Transient object data:
`unsigned char PersiData[SEGM_SIZE], TransData[SEGM_SIZE];`

ACSL predicates for memory model constraints

- Ex. **predicate** `valid_heap_model`
 - Number of allocated objects is within allowed bounds
 - Headers are in corresponding segment bounds and do not overlap
 - Data are in corresponding segment bounds and do not overlap

ACSL predicates for security properties

- Ex. **predicate** `object_headers_intact{L1, L2}`
 - Object headers of allocated objects do not change between labels L1 and L2 (**integrity_header**)

Bastore : function contract



```
99  /*@
100 requires vhm: valid_heap_model;
101 requires ...;
102 assigns  ...;
103 ensures  ...;
104 ensures  vhm: valid_heap_model;
105 ensures  oh: object_headers_intact{Pre, Post};
106 */
107 void bastore(u4 ObjRef, u4 DestOff, u1 Val){           // u1/u4: unsigned char/int
108     if( ! firewall(ObjRef, DestOff) )            // Check access and
109     return;                                           // exit if forbidden
110     if( GET_FLAG(ObjHeader+ObjRef) & 0x08 )           // If transient bit set,
111         TransData[GET_OFF(ObjHeader+ObjRef) + DestOff] = Val; // write to transient body
112     else                                           // Otherwise
113         PersiData[GET_OFF(ObjHeader+ObjRef) + DestOff] = Val; // write to persistent body
114     updateJPC();
115 }
```

toy example

- **Bastore**: write value **Val** into a given array at a given offset
- **valid_heap_model** is maintained both as **pre-condition** and **post-condition**
- Line 105 ensures security property **integrity_header**
- **Firewall** is called to check the access

Main dispatch loop



```
171 void vm_loop() {
172     /*@
173     loop invariant vhm: valid_heap_model;
174     loop invariant ...
175     loop invariant oh: object_headers_intact{LoopEntry, Here};
176     loop assigns ...
177     ...
178     ...
179     ...
180     */
181     while(1) {
...
        // calls opcode functions (bastore, ...)
...
194     }
195 }
```

valid_heap_model is maintained as a loop invariant

Security Property is maintained as a loop invariant

toy example

Integrity_data and Confidentiality_data cannot be specified (easily) with WP as global invariants

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.

- **MetAcsI** 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.

■ Introduction

■ General approach

■ Proof issues and solutions

■ Results and conclusion

Some Issues (I) and Solutions (S)

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**



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



Carefully chosen lemmas

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



Low-level operations (examples)

Low-level bit-field operations

- Transient bit obtained from flag byte with mask 0x08
- Companion ghost encoding to help the prover: `unsigned char gIsTrans [MAX_OBJS];`

```
predicate valid_heap_model =  
...  
  (\forall integer i; 0 <= i < gNumObjs ==>  
    ( gIsTrans[i] <==> (GET_FLAG(ObjHeader+gHeadStart[i]) & 0x08) ) ) &&  
...
```

Frama-C/WP “Typed” memory model does not allow pointer casts

- We rewrite some pointer casts

```
typedef unsigned char u1; typedef unsigned short u2;  
...  
#define GET_OFF(addr) ( (u2)((*(u1*)addr + 4))*256 + (*(u1*)addr + 5) )  
// instead of  
// #define GET_OFF(addr) ((u2)(*(u2*)(addr + 4)))
```

Lemmas to deduce some complex predicates (1/2)

Complex preservation properties

- Lemmas help automatic provers to prove complex preservation properties

```
/*@  
lemma vhm_preserved{L1,L2}:  
  mem_model_footprint_intact{L1,L2} &&  
  object_headers_intact{L1,L2} &&  
  valid_heap_model{L1} &&  
  \at(gNumObjs,L1) == \at(gNumObjs,L2) ==>  
    valid_heap_model{L2};  
*/
```

Lemmas to deduce some complex predicates (2/2)

```
67 /*@ // === A security property: object headers remain intact ===
68 predicate object_headers_intact{L1, L2} =
69   \forall integer i, off; 0 <= i < \at(gNumObjs,L1) &&
70   \at(gHeadStart[i],L1) <= off < \at(gHeadStart[i],L1) + 8 ==>
71   \at(ObjHeader[off],L1) == \at(ObjHeader[off],L2);
72
73 // === Memory footprint predicate and lemma example ===
74 predicate mem_model_footprint_intact{L1,L2} =
75   \at(gNumObjs,L1) <= \at(gNumObjs,L2) &&
76   ( \forall integer i; 0 <= i < \at(gNumObjs,L1) ==>
77     \at(gIsTrans[i],L1) == \at(gIsTrans[i],L2) &&
78     \at(gHeadStart[i],L1) == \at(gHeadStart[i],L2) &&
79     \at(gDataStart[i],L1) == \at(gDataStart[i],L2) &&
80     \at(gDataEnd[i],L1) == \at(gDataEnd[i],L2) );
81
82 lemma vhm_preserved{L1,L2}: mem_model_footprint_intact{L1,L2} &&
83   object_headers_intact{L1,L2} && valid_heap_model{L1} &&
84   \at(gNumObjs,L1) == \at(gNumObjs,L2) ==> valid_heap_model{L2}; */
```

toy example

- Object headers do not change between labels L1 and L2

- Relevant memory footprint does not change between labels L1 and L2 for objects that existed at label L1.

- Helps automatic provers to prove complex preservation properties.

Overview

Introduction

General approach

Proof issues and solutions

Results and conclusion

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

Proof results for 4 increasing code subsets

The proof scales well with an increasing number of goals

Code subset	Prover	User-provided ACSL	MetAcsl	RTE	Total	
		#Goals	#Goals	#Goals	#Goals	Time
Bastore	Qed	1,019	3,304	106	4,429 (77.92%)	0h47m45s
	Script	78	131	1	210 (3.69%)	0h11m12s
	SMT	305	590	148	1,043 (18.35%)	0h17m23s
	All	1,402 (24.67%)	4,025 (70.81%)	255 (4.48%)	5,684	0h49m37s
Sample 1	Qed	1,491	5,037	120	6,648 (79.76%)	1h00m49s
	Script	111	149	7	267 (3.20%)	0h13m41s
	SMT	437	784	199	1,420 (17.03%)	0h28m24s
	All	2,039 (24.46%)	5,970 (71.63%)	326 (3.91%)	8,335	0h59m59s
Sample 2	Qed	2,413	6,884	126	9,423 (79.43%)	1h04m33s
	Script	144	257	20	421 (3.55%)	0h18m15s
	SMT	682	1,088	249	2,019 (17.01%)	0h37m01s
	All	3,239 (27.30%)	8,229 (69.36%)	395 (3.33%)	11,863	1h09m47s
All	Qed	18,925	22,361	168	41,454 (79.42%)	2h58m15s
	Script	330	212	30	572 (1.1%)	0h44m48s
	SMT	4,683	4,588	902	10,173 (19.49%)	2h36m18s
	All	23,938 (45.85%)	27,435 (52.55%)	1,117 (2.13%)	52,198	3h28m07s

99% of proof goals discharged automatically (QED + Alt-Ergo)
Manual effort is still important (a few days to update scripts !)

Successful industrial application of deductive verification

- EAL 6 certificate issued by ANSSI after evaluation by CEA Leti
- Careful combination of: ghost code, lemmas, proof scripts, ...
- High level of automation (99% of goals proved automatically)
- MetAcsI is crucial for specification of security properties
- Efficient tool support from Frama-C developers was essential

Future work directions

- Introduce proof into a continuous integration process
- Better support for custom proof strategies to save manual script effort
- Accelerate QED simplification for complex functions with lots of branches
- 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
- Better parallelization of the proof with WP (in particular, QED)

References

- Adel Djoudi, Martin Hana and Nikolai Kosmatov.
“Formal verification of a JavaCard virtual machine with Frama-C”.
In *Proc. of the 24rd International Symposium on Formal Methods (FM 2021)*,
Online, November 2021, pages 427-444. LNCS, vol. 13047. Springer.
- Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall.
“MetAcsl: Specification and Verification of High-Level Properties.”
In *Proc. of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019)*, pages 358-364, LNCS, vol. 11427. 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”.
In *Proc. of the 13th International Conference on Tests & Proofs (TAP 2019)*, pages 167-185. LNCS, vol. 11823. Springer.
- Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall.
“Methodology for Specification and Verification of High-Level Properties with MetAcsl”.
In *Proc. of the 9th IEEE/ACM International Conference on Formal Methods in Software Engineering (FormalISE 2021)*, IEEE.