Toward a Verified Software Toolchain for Java

David Pichardie - INRIA Rennes

How do you trust your software ?

How do you trust your software ?

The increasing complexity of safety critical systems requires efficient validation techniques

How do you trust your software ?

The increasing complexity of safety critical systems requires efficient validation techniques

- Manual verifications
  - do not scale

manual verification

yesterday

How do you trust your software ?

The increasing complexity of safety critical systems requires efficient validation techniques

- Manual verifications
  - do not scale
- Automatic bug finders
  - may miss some bugs

manual verification	bug finders
yesterday	today

How do you trust your software ?

The increasing complexity of safety critical systems requires efficient validation techniques

- Manual verifications
  - do not scale
- Automatic bug finders
  - may miss some bugs
- Automatic, sound verifiers
  - find all bugs, may raise false alarms

  ex: the Astrée static analyzer

http://www.astree.ens.fr/

~1M loc of a critical control-command software analyzed

0 false alarms

manual verification	bug finders	sound verifiers
yesterday	today	tomorrow

How do you trust the tool that verifies your software ?

The increasing complexity of safety critical systems requires efficient validation techniques

- Manual verifications
  - do not scale
- Automatic bug finders
  - may miss some bugs
- Automatic, sound verifiers
  - find all bugs, may raise false alarms

  ex: the Astrée static analyzer

http://www.astree.ens.fr/

~1M loc of a critical control-command software analyzed

0 false alarms

manual verification	bug finders	sound verifiers
yesterday	today	tomorrow

How do you trust the tool that verifies your software ?

The increasing complexity of safety critical systems requires efficient validation techniques

- Manual verifications
  - do not scale
- Automatic bug finders
  - may miss some bugs
- Automatic, sound verifiers
  - find all bugs, may raise false alarms

  ex: the Astrée static analyzer
- Formally-verified verifiers
  - the verifier comes with a soundness proof
  - that is machine checked

http://www.astree.ens.fr/

~1M loc of a critical control-command software analyzed

0 false alarms

manual verification	bug finders	sound verifiers	verified verifiers
yesterday	today	tomorrow	after tomorrow

How do we verify the verifier?

How do we verify the verifier?

A simple idea:

How do we verify the verifier?

A simple idea:

Program and prove your verifier in the same language!

How do we verify the verifier?

A simple idea:

Program and prove your verifier in the same language!

Which language ?

How do we verify the verifier?

A simple idea:

Program and prove your verifier in the same language!

Which language ?

Coq

Coq: an animal with two faces

Coq: an animal with two faces

First face:

Coq: an animal with two faces

First face:

- a proof assistant that allows to interactively build proof in constructive logic

Coq: an animal with two faces

First face:

- a proof assistant that allows to interactively build proof in constructive logic

Second face:

Coq: an animal with two faces

First face:

- a proof assistant that allows to interactively build proof in constructive logic

Second face:

- a functional programming language with a very rich type system

Coq: an animal with two faces

First face:

- a proof assistant that allows to interactively build proof in constructive logic

Second face:

- a functional programming language with a very rich type system

  example:

```
sort: ∀ l: list int, { l': list int |
                              Sorted l' ∧ PermutationOf l l' }
```

Coq: an animal with two faces

First face:

- a proof assistant that allows to interactively build proof in constructive logic

Second face:

- a functional programming language with a very rich type system
  example:

```
sort: ∀ l: list int, { l': list int |
                                   Sorted l' ∧ PermutationOf l l' }
```

- with an extraction mechanism to Ocaml

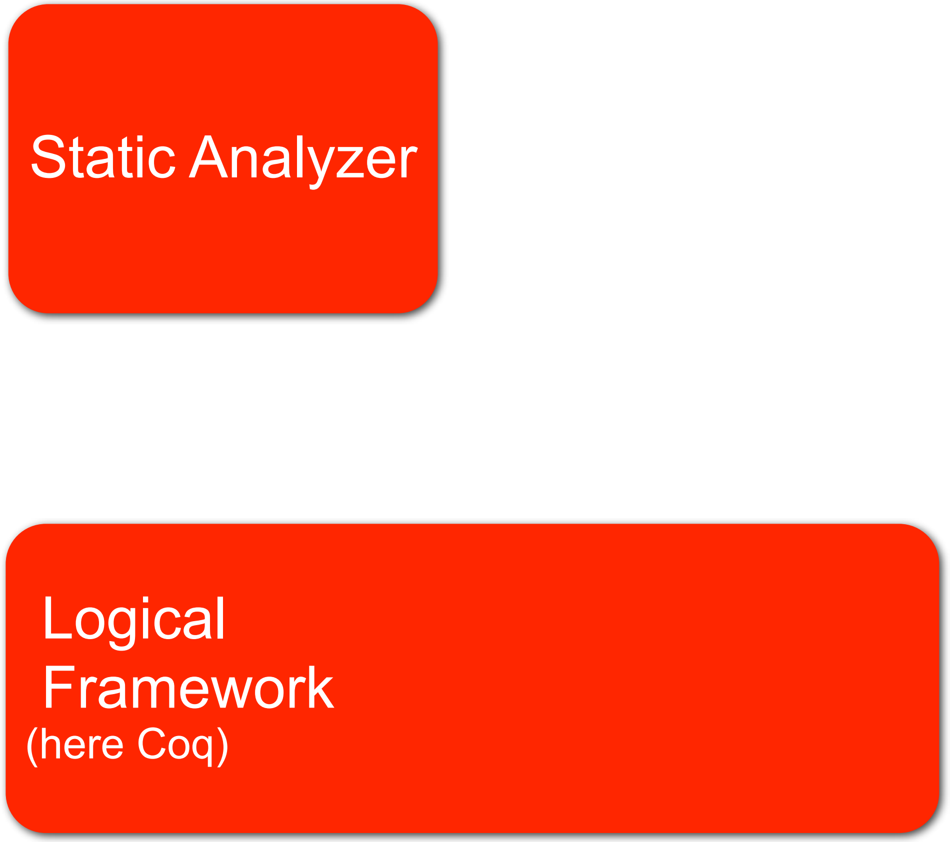
```
sort: int list → int list
```

Our Methodology

Our Methodology

We program the static analyzer inside Coq

```
Definition analyzer (p:program) := ...
```



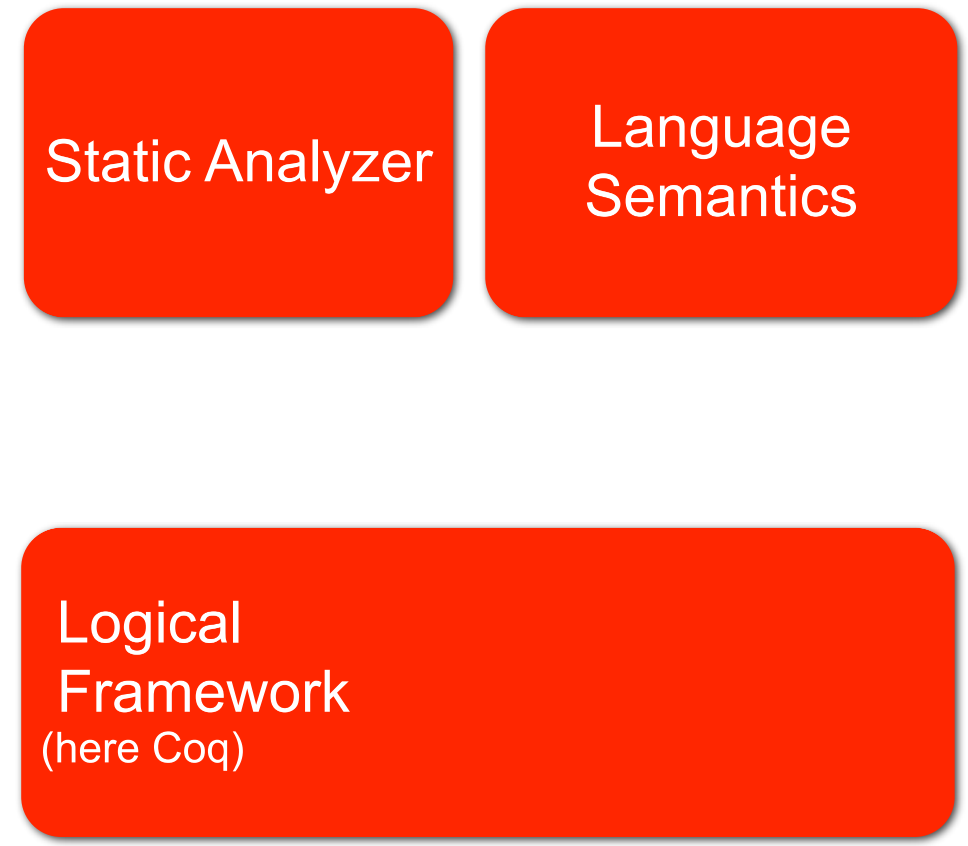
Our Methodology

We program the static analyzer inside Coq

```
Definition analyzer (p:program) := ...
```

We state its correctness wrt. a formal specification of the language semantics

```
Theorem analyser_is_sound :
  ∀ p, analyser p = Yes → Sound(p)
```



Our Methodology

We program the static analyzer inside Coq

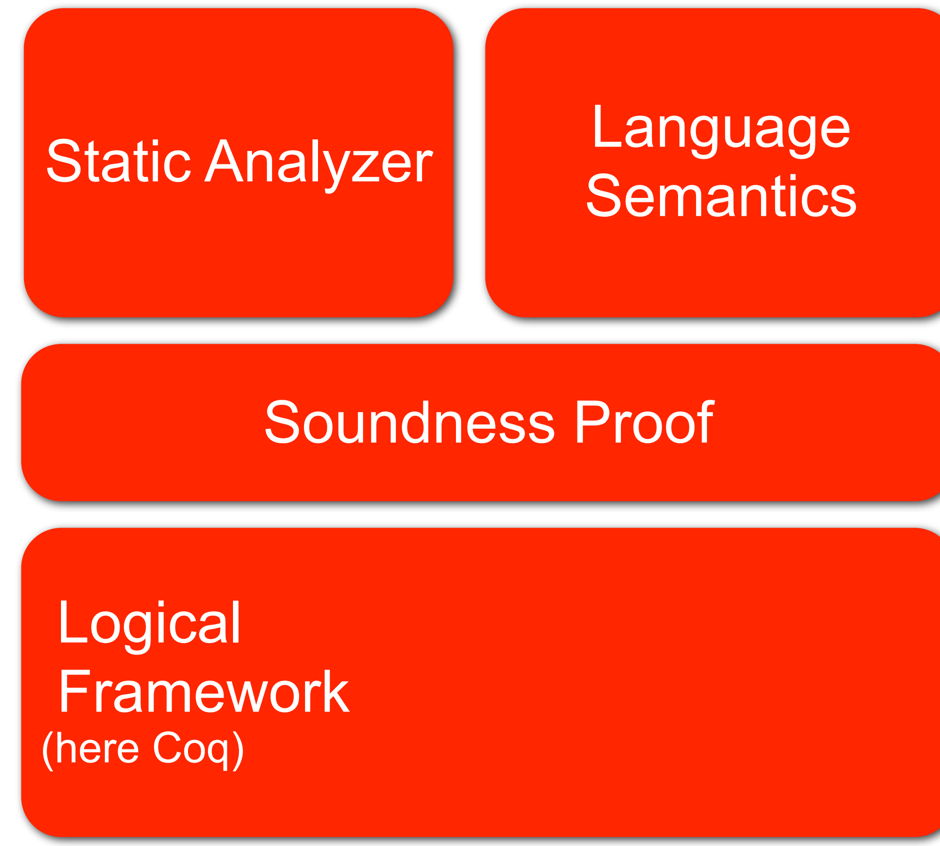
```
Definition analyzer (p:program) := ...
```

We state its correctness wrt. a formal specification of the language semantics

```
Theorem analyser_is_sound :
  ∀ p, analyser p = Yes → Sound(p)
```

We interactively and mechanically prove this theorem

```
Proof. ... (* few days later *) ... Qed.
```



Our Methodology

We program the static analyzer inside Coq

```
Definition analyzer (p:program) := ...
```

We state its correctness wrt. a formal specification of the language semantics

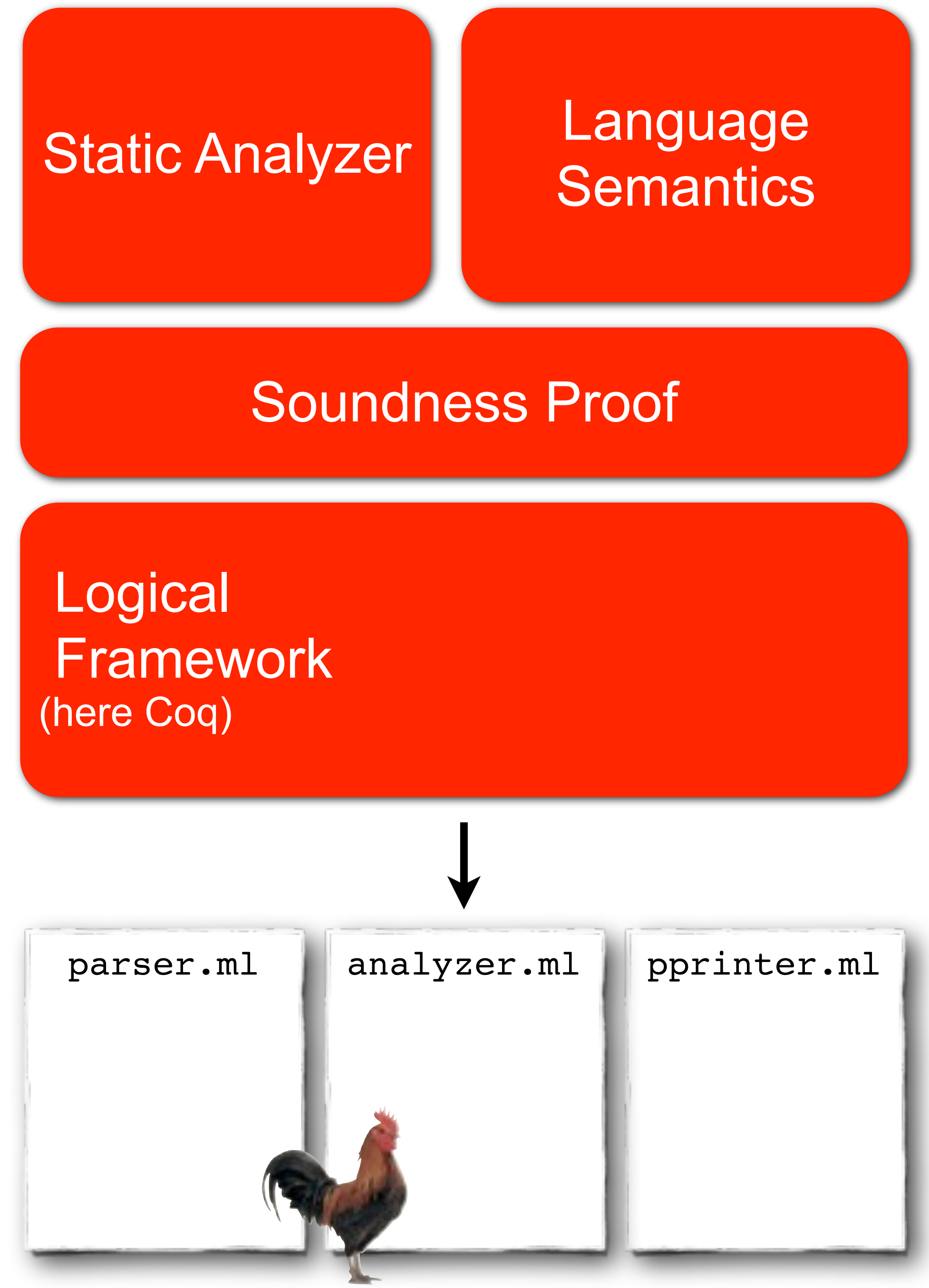
```
Theorem analyser_is_sound :
  ∀ p, analyser p = Yes → Sound(p)
```

We interactively and mechanically prove this theorem

```
Proof. ... (* few days later *) ... Qed.
```

We extract an OCaml implementation of the analyzer

```
Extraction analyzer.
```



A Posteriori Validation

An important tool in our toolbox

We program the *full* static analyzer inside Coq

```
Definition analyzer (p:program) :=
  ...
  let x := complex_computation p in
  ...
```

... or ...

A Posteriori Validation

An important tool in our toolbox

We program the *full* static analyzer inside Coq

```
Definition analyzer (p:program) :=
  ...
  let x := complex_computation p in
  ...
```

... or we program some parts in Coq, other parts in OCaml and use a verified validator

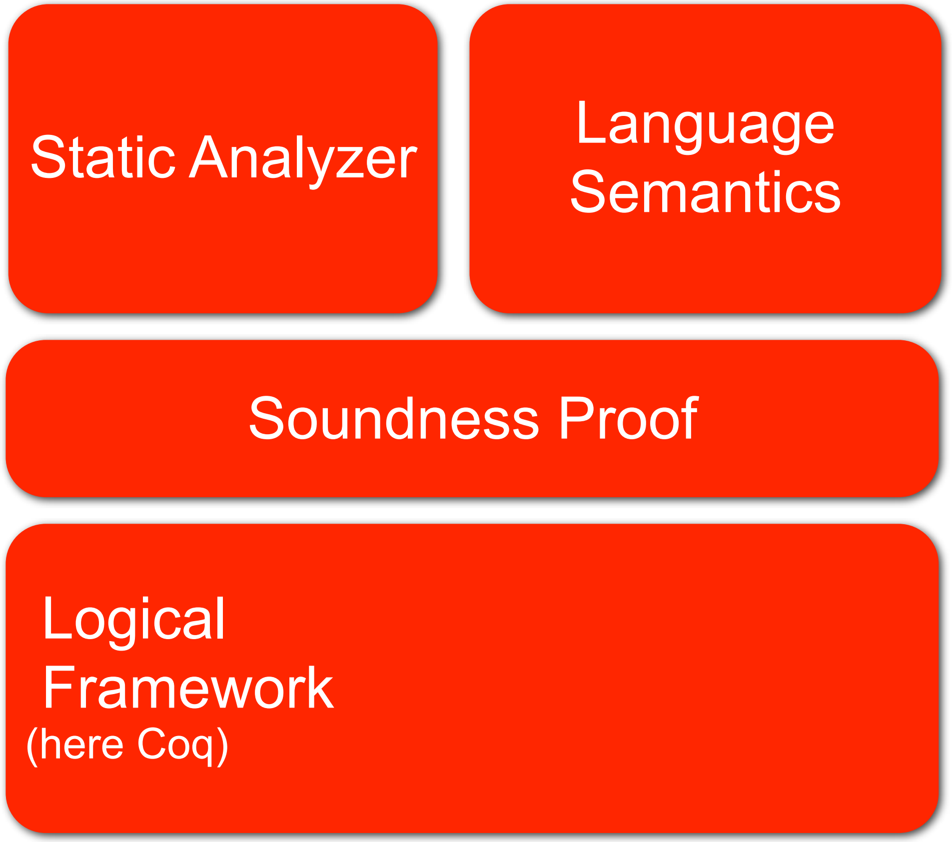
```
Definition analyzer (p:program) :=
  ...
  let x := ocaml_external_complex_computation p in
  match validator x p with
    | OK ⇒ ...
    | Error ⇒ abort
  end.
```

Ideally we also prove (on paper) that if the external implementation implements correctly a well-known algorithm then the validator will always succeed (completeness)

Trusted Computing Base (TCB)

1. Formal specification of the programming language semantics
   - (informally) shared by any end-user programmer, compiler, static analyzer
   - less specialized than static analyzer's abstract semantics

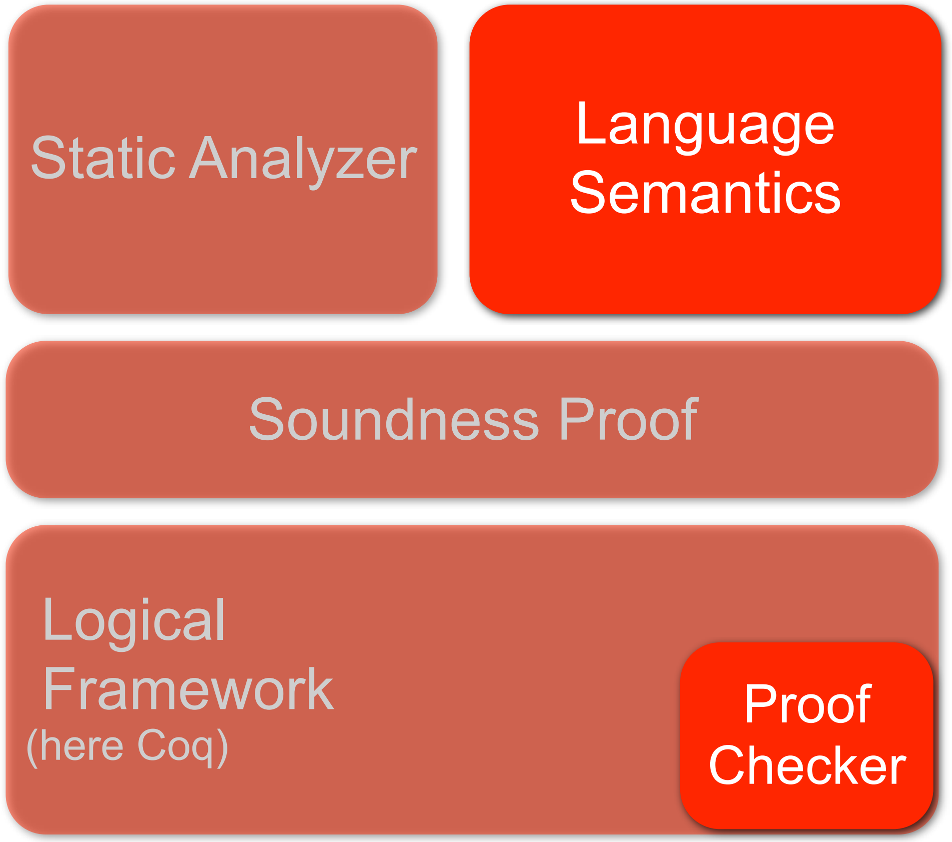
2. Logical Framework
   - only the proof checker needs to be trusted
   - we don't trust sophisticated decision procedures



Trusted Computing Base (TCB)

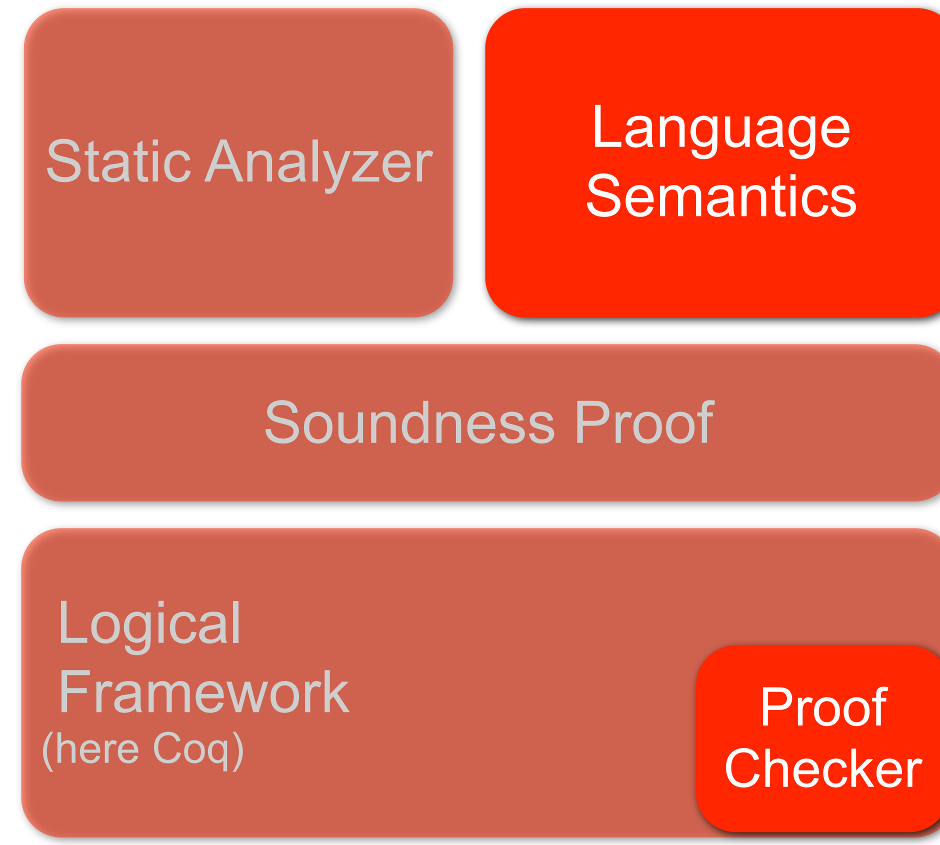
1. Formal specification of the programming language semantics
   - (informally) shared by any end-user programmer, compiler, static analyzer
   - less specialized than static analyzer's abstract semantics

2. Logical Framework
   - only the proof checker needs to be trusted
   - we don't trust sophisticated decision procedures



Trusted Computing Base (TCB)

1. Formal specification of the programming language semantics
   - (informally) shared by any end-user programmer, compiler, static analyzer
   - less specialized than static analyzer's abstract semantics



2. Logical Framework
   - only the proof checker needs to be trusted
   - we don't trust sophisticated decision procedures

Still a large code base but at least a *foundational* code base: logic & semantics

Verified Static Analysis: Challenges

«Once you have a good proof of your tool on paper, mechanizing it is just a matter of time!»

Verified Static Analysis: Challenges

Common belief

«Once you have a good proof of your tool on paper,
mechanizing it is just a matter of time!»

Verified Static Analysis: Challenges

Common belief

«Once you have a good proof of your tool on paper, mechanizing it is just a matter of time!»

I think such a reasoning is severely flawed

Verified Static Analysis: Challenges

Common belief

«Once you have a good proof of your tool on paper, mechanizing it is just a matter of time!»

I think such a reasoning is severely flawed

- You never have a proof of a tool. You have a proof of an algorithm and implementation details sometimes matter.

Verified Static Analysis: Challenges

Common belief

«Once you have a good proof of your tool on paper, mechanizing it is just a matter of time!»

I think such a reasoning is severely flawed

- You never have a proof of a tool. You have a proof of an algorithm and implementation details sometimes matter.
- Not all proofs share the same vocabulary. We seek for a proof at the level of the language semantics.

Verified Static Analysis: Challenges

Common belief

«Once you have a good proof of your tool on paper, mechanizing it is just a matter of time!»

I think such a reasoning is severely flawed

- You never have a proof of a tool. You have a proof of an algorithm and implementation details sometimes matter.
- Not all proofs share the same vocabulary. We seek for a proof at the level of the language semantics.
- You rarely have a full proof. You reason on a core subset of a language but interactions between all features may invalidate this proof.

Verified Static Analysis: Challenges

Common belief

«Once you have a good proof of your tool on paper, mechanizing it is just a matter of time!»

I think such a reasoning is severely flawed

- You never have a proof of a tool. You have a proof of an algorithm and implementation details sometimes matter.
- Not all proofs share the same vocabulary. We seek for a proof at the level of the language semantics.
- You rarely have a full proof. You reason on a core subset of a language but interactions between all features may invalidate this proof.
- «matter of time»: do not confound decidability with tractability. Without a good methodology a mechanized proof can overwhelm human capacities.

Verified PL Stacks: Achievements

Some major achievements have changed our expectations about programming language mechanized proofs

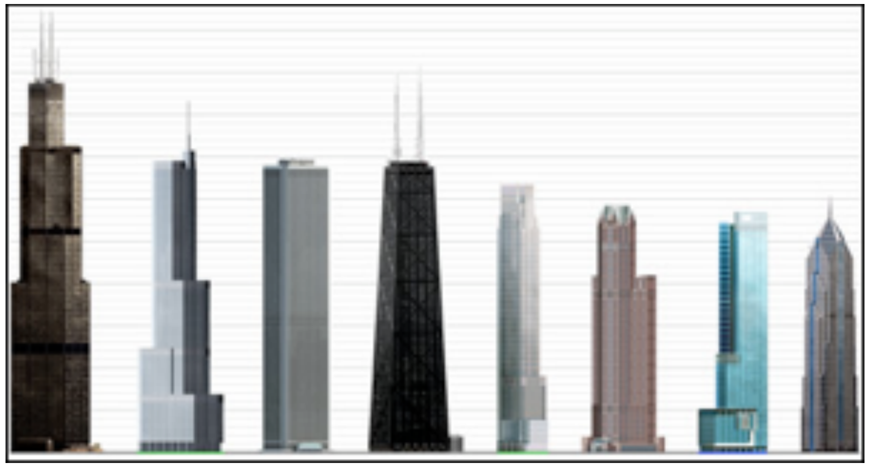
- M6: JVM bytecode interpreter in ACL2 (Liu)
- Jinja: source & bytecode Java, compiler, BCV in Isabelle/HOL (Klein & Nipkow, and extensions by Lochbihler)
- CompCert: realistic C compiler in Coq (Leroy, Blazy et al.)
- Verified Software ToolChain: extension of CompCert for concurrent C programs (Appel et al.)
- seL4: verified OS kernel in Isabelle/HOL (Klein, Norrish et al.)

Verified Static Analysis: Objectives

We identify two major objectives

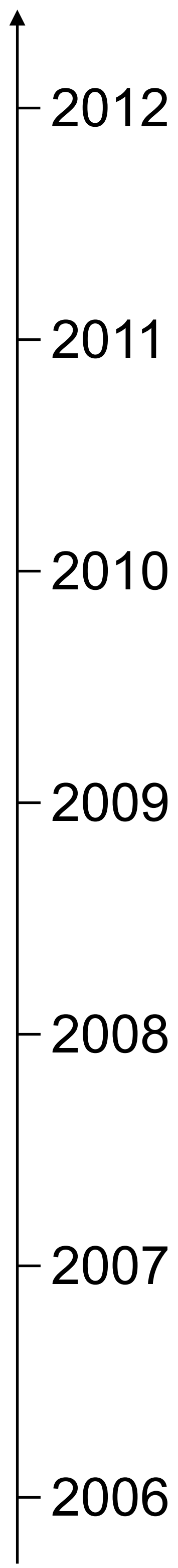
1. building new proof methods for the working (mechanized) semanticist
    - using Abstract Interpretation theory, we can provide generic interfaces between analyses
    - we have to discover new *a posteriori* validation algorithms (sound and efficient)

2. building big proofs
    - proving in the small will not give us all the lessons we want to learn
    - large case studies are important to build a new *proof engineering* knowledge



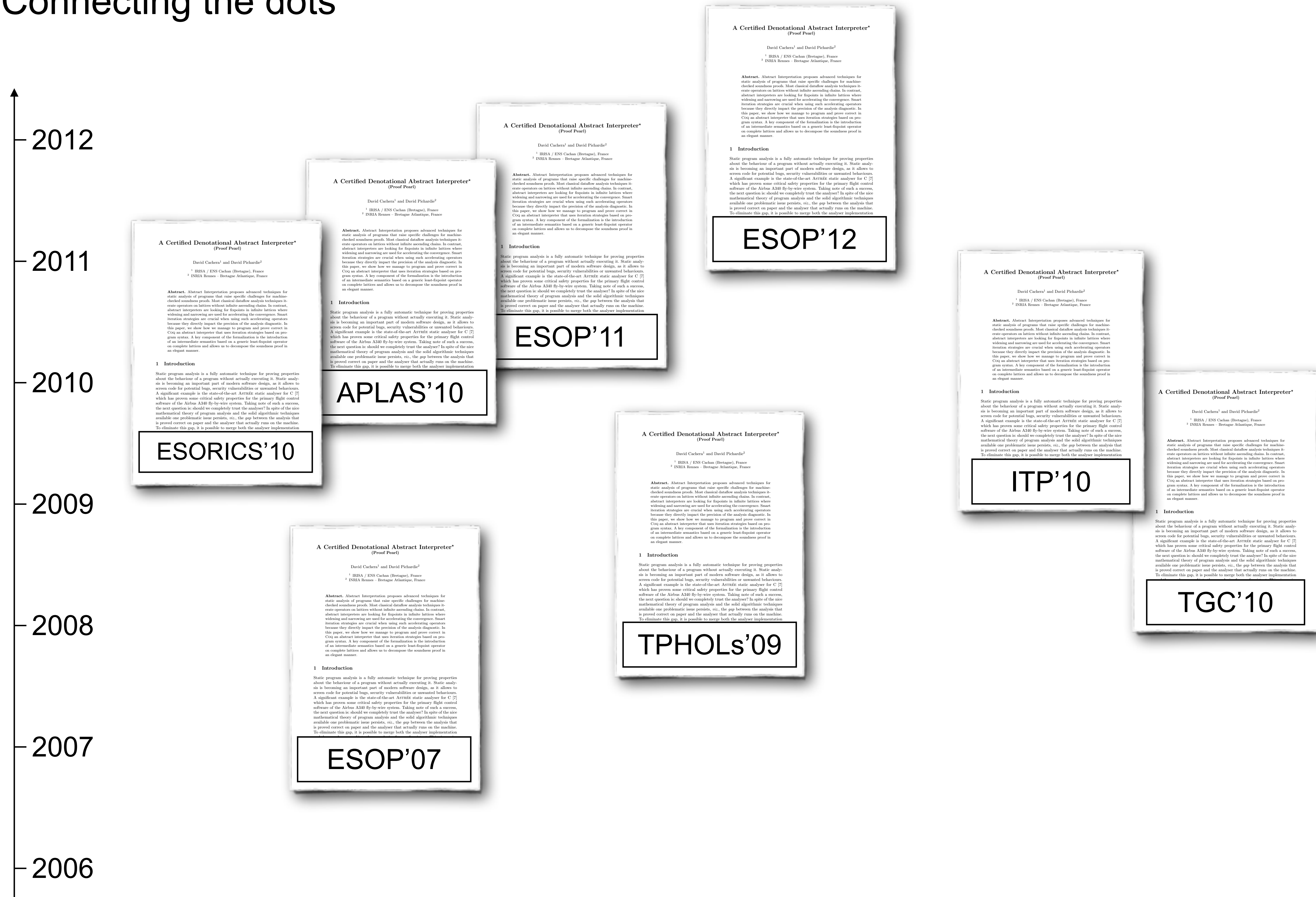
This *HDR*

Connecting the dots



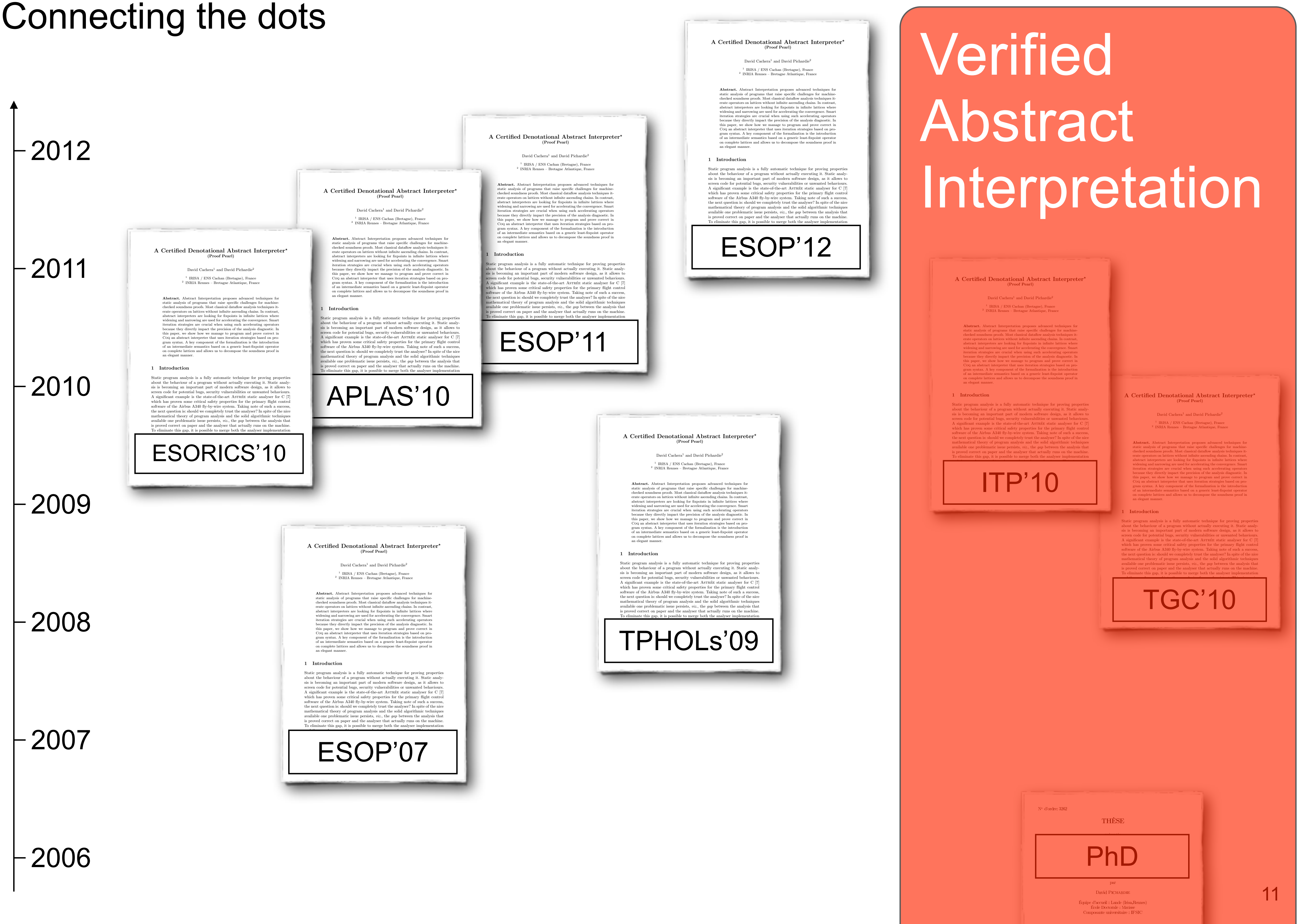
This *HDR*

Connecting the dots



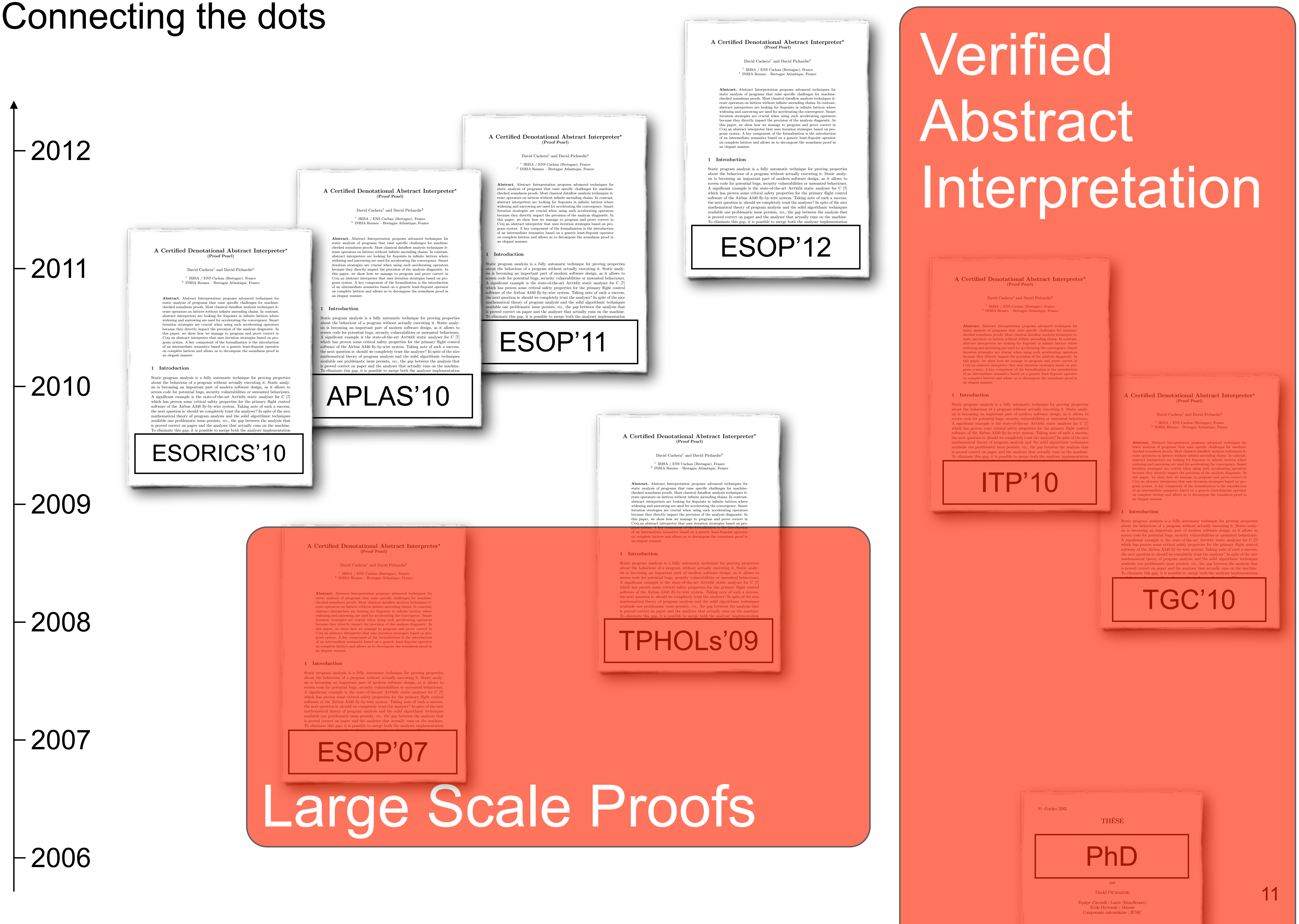
This *HDR*

Connecting the dots



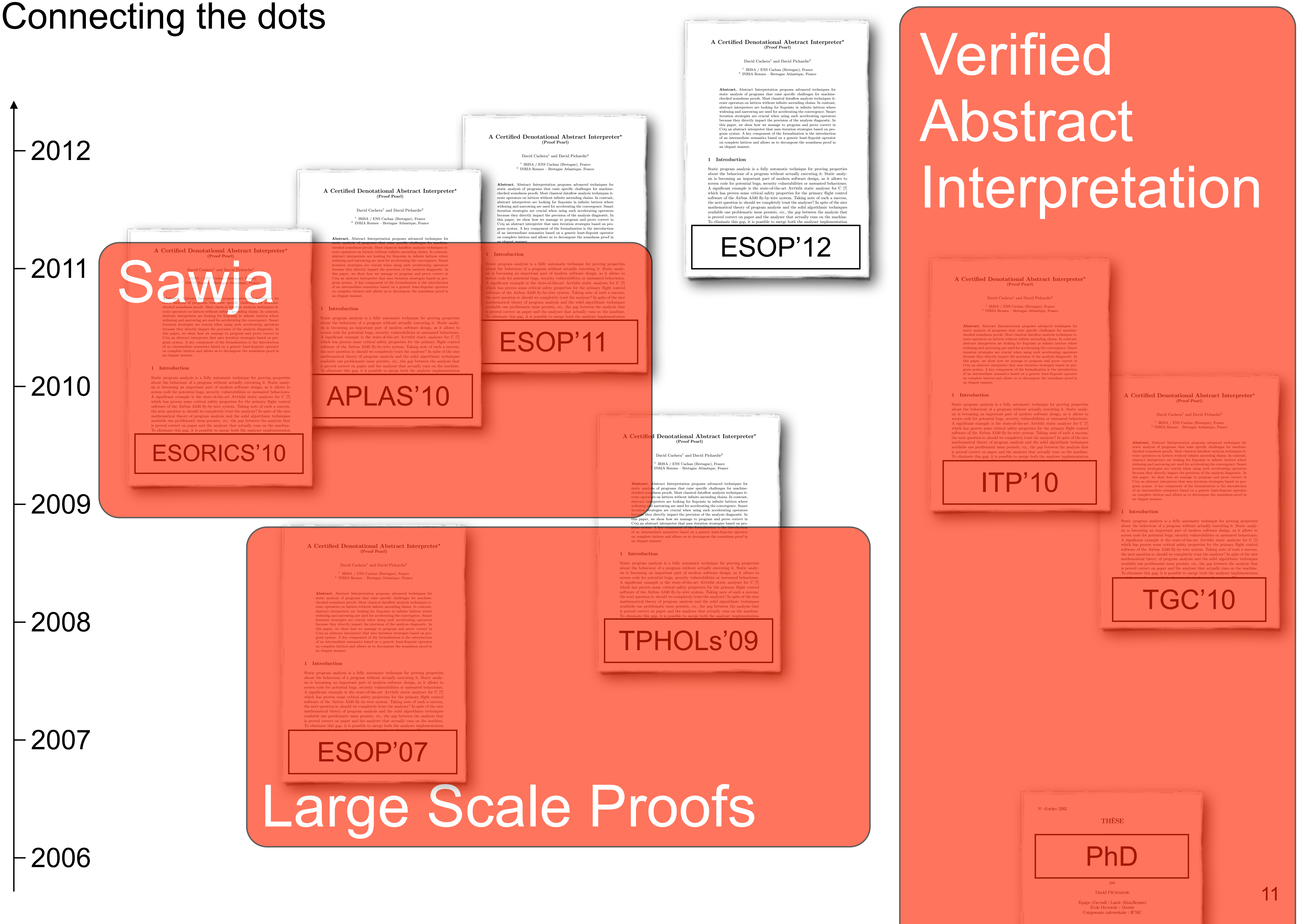
This *HDR*

Connecting the dots



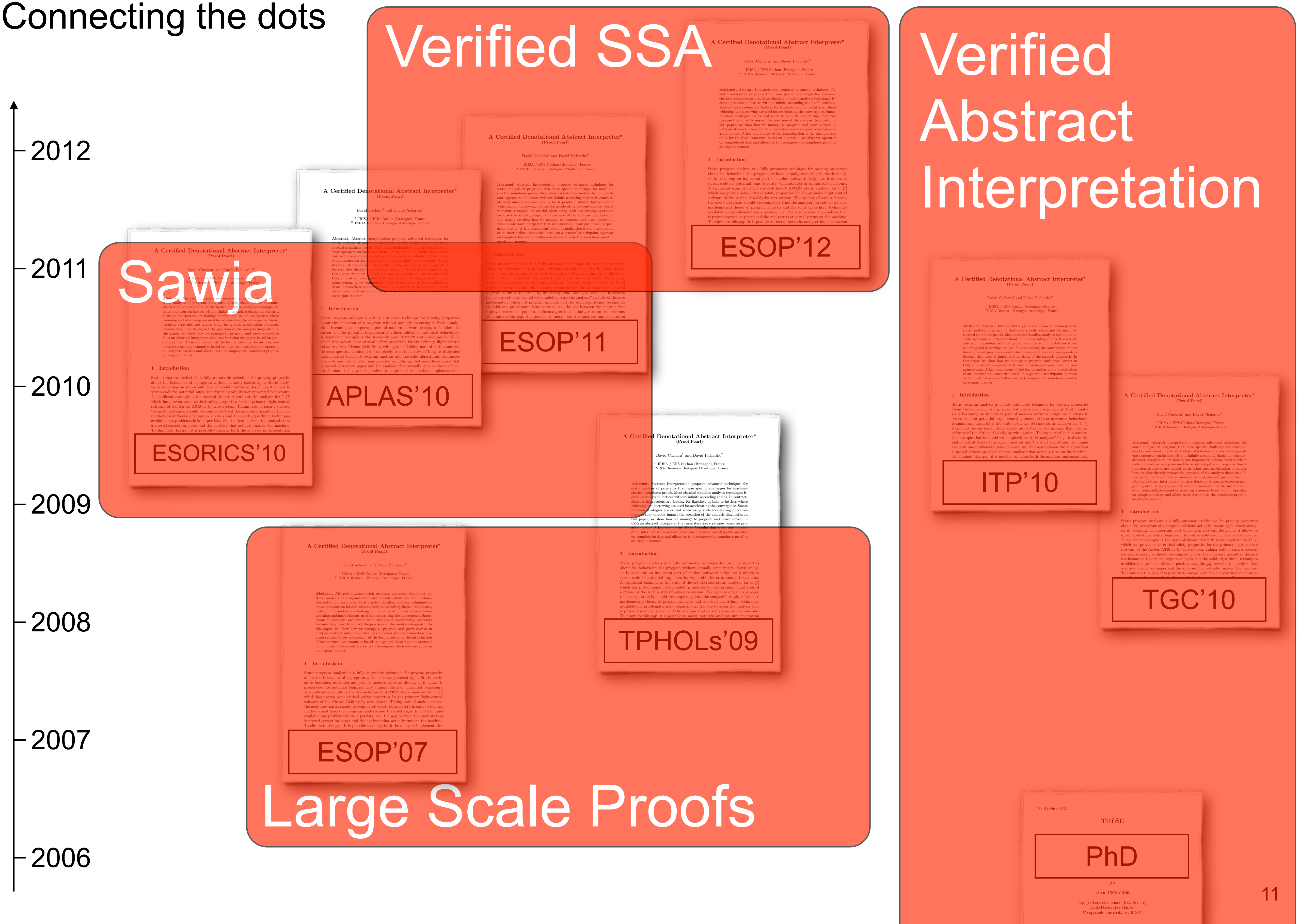
This *HDR*

Connecting the dots



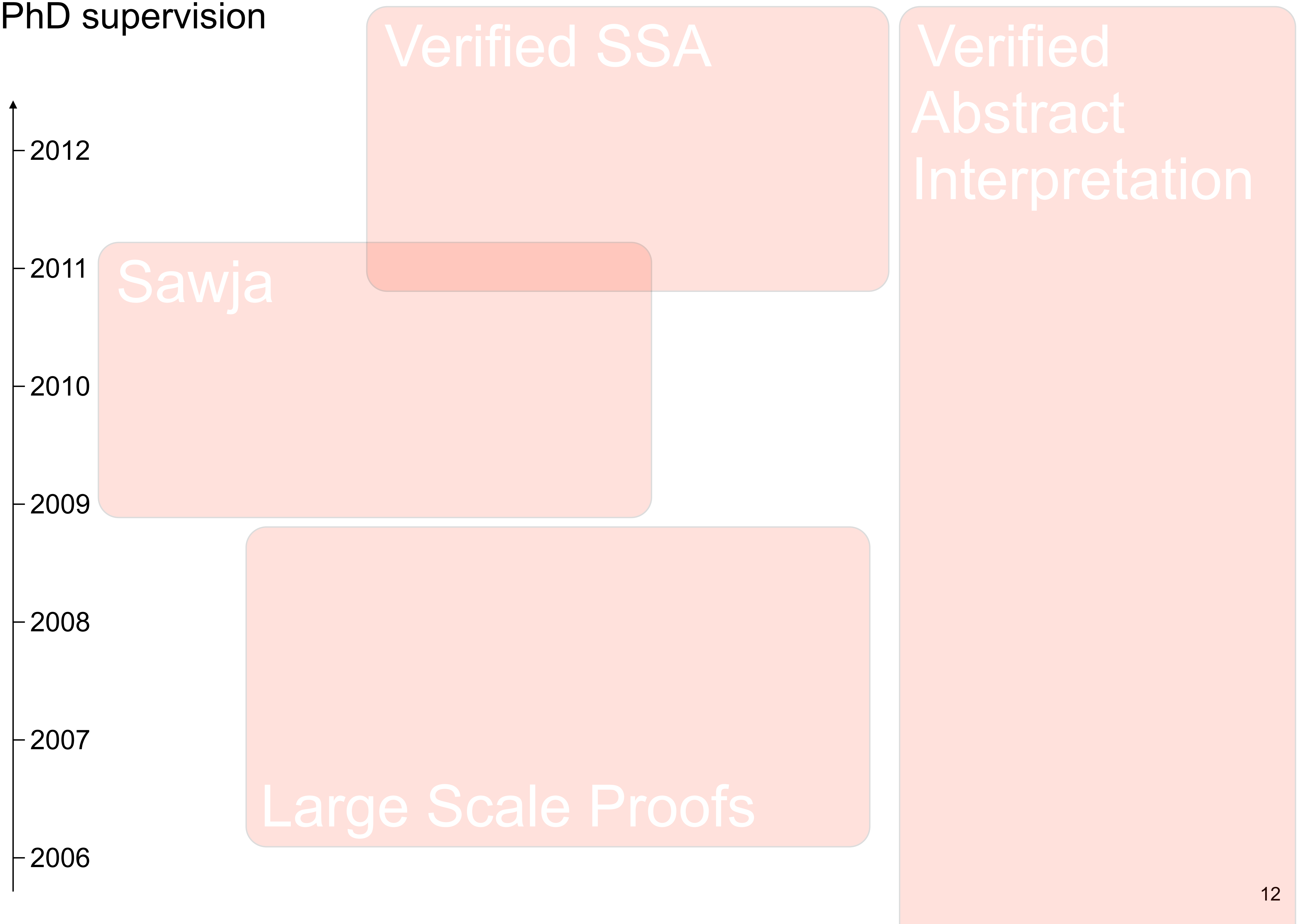
This *HDR*

Connecting the dots



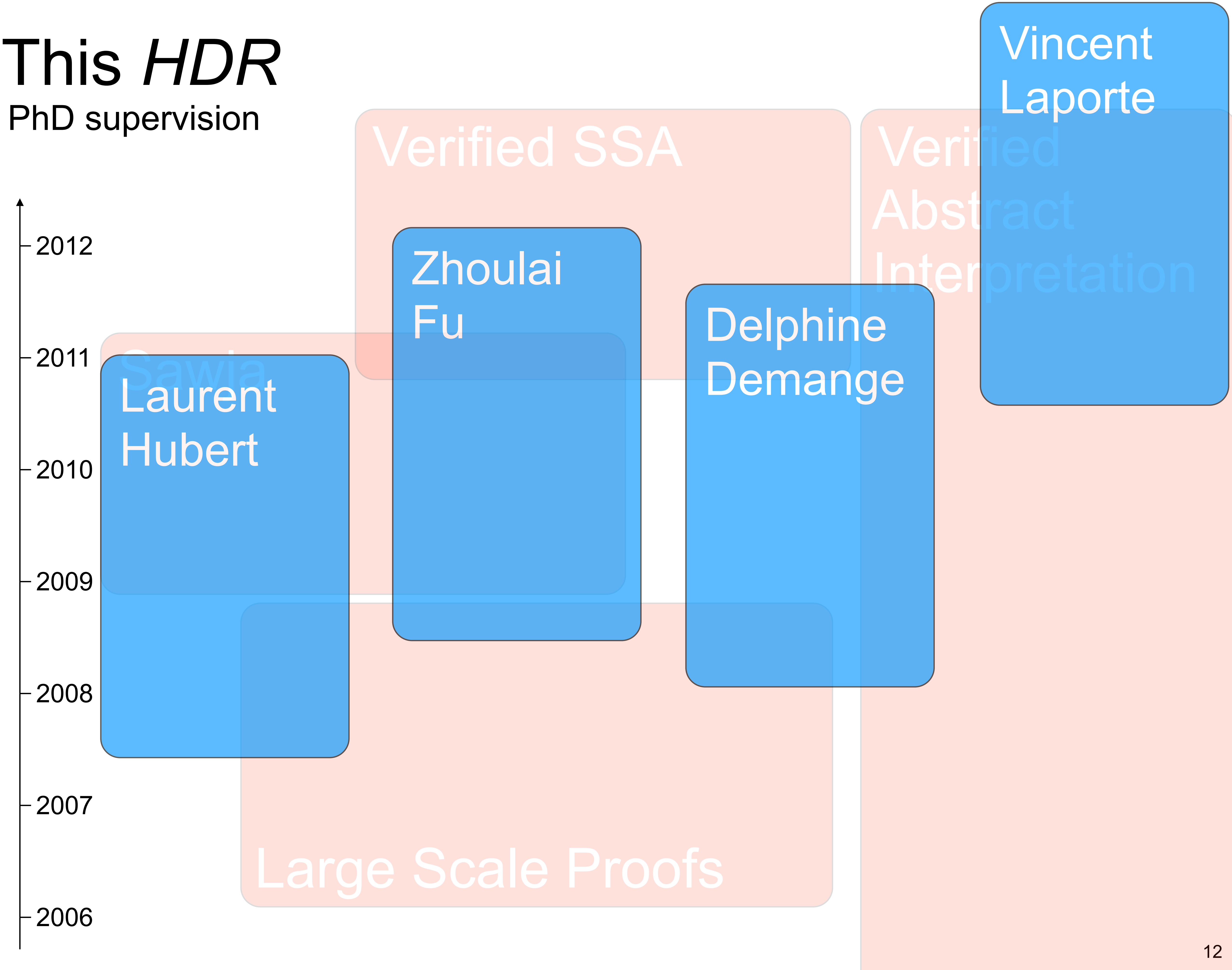
This *HDR*

PhD supervision

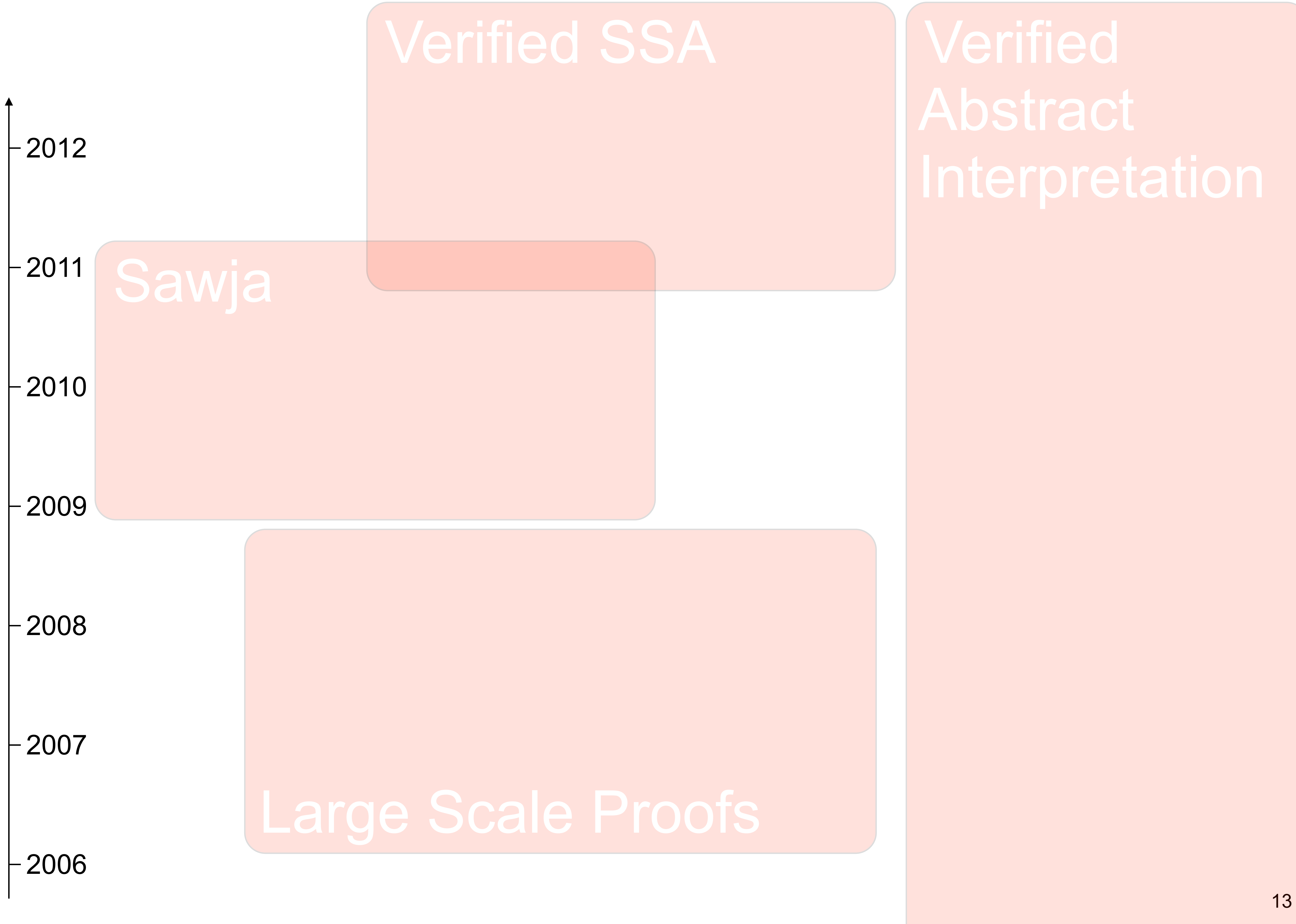


This *HDR*

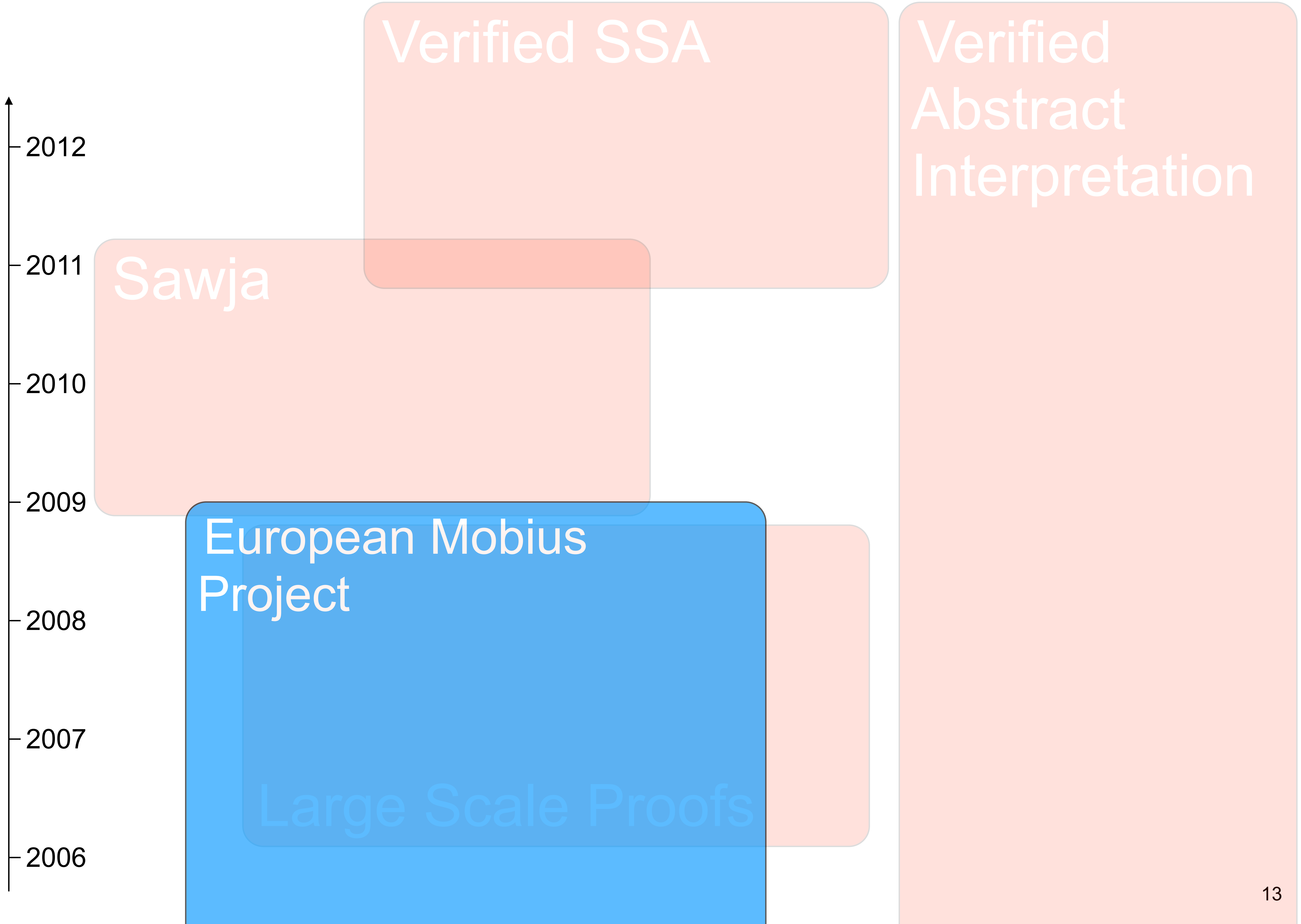
PhD supervision



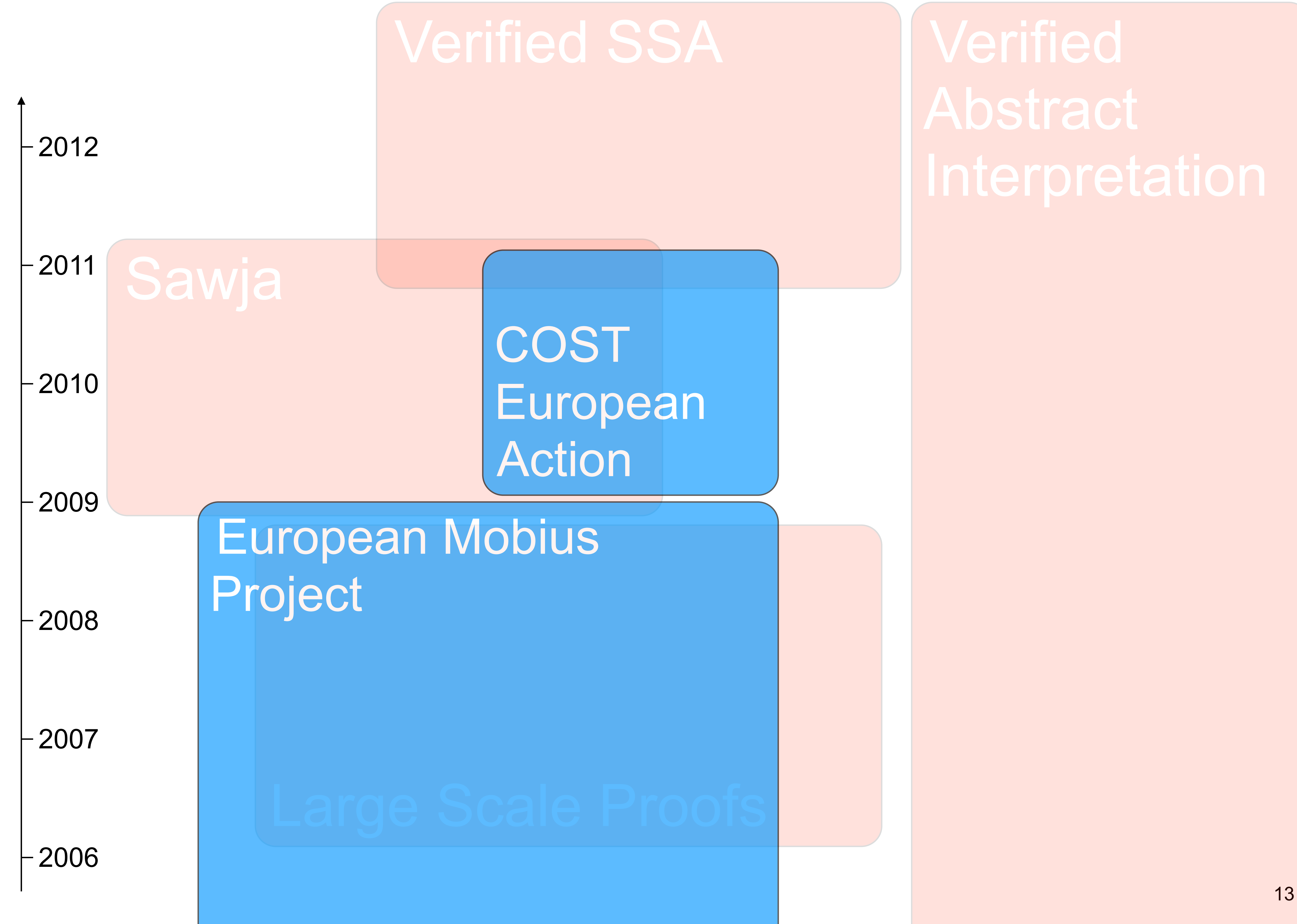
Collaborations



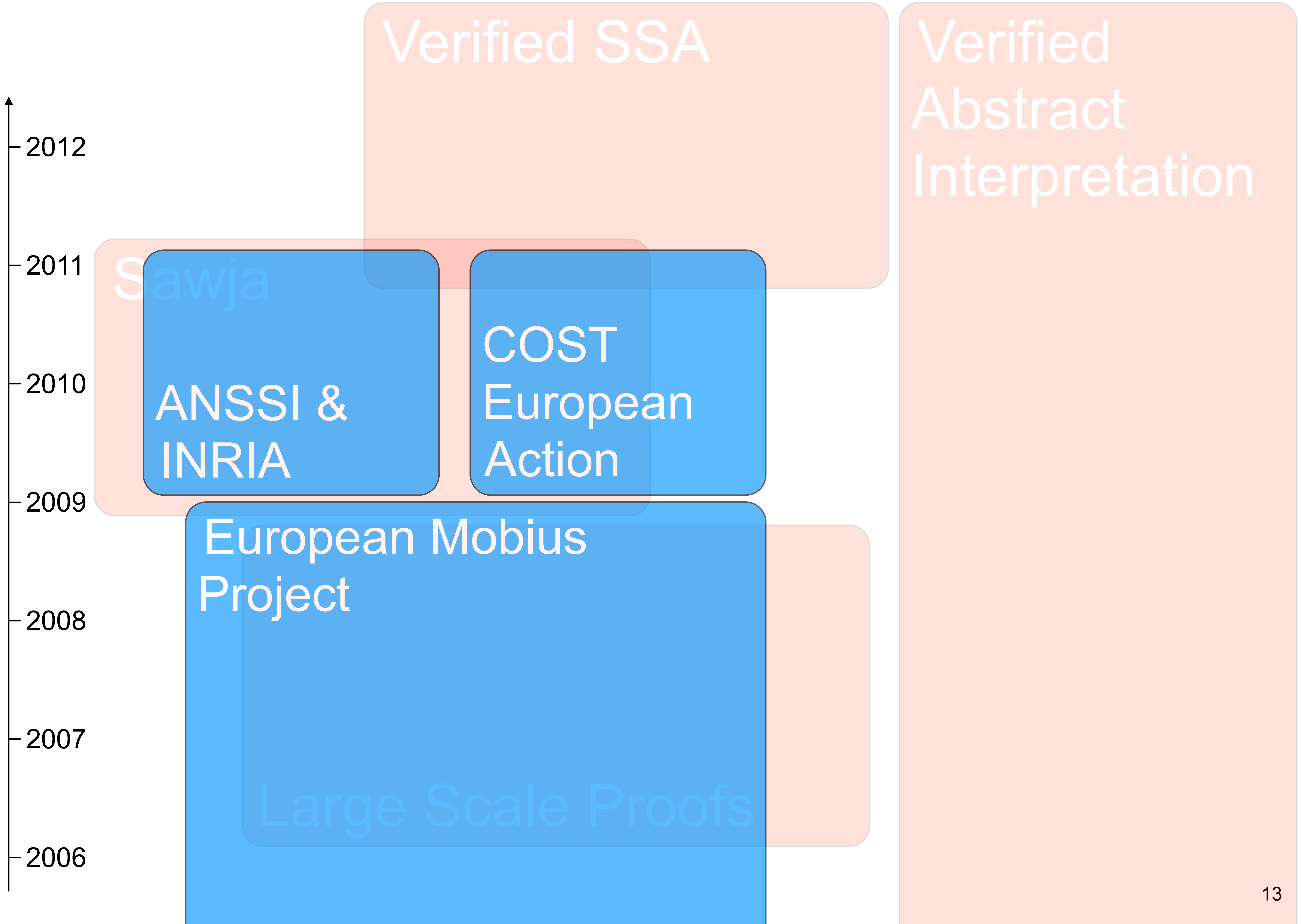
Collaborations



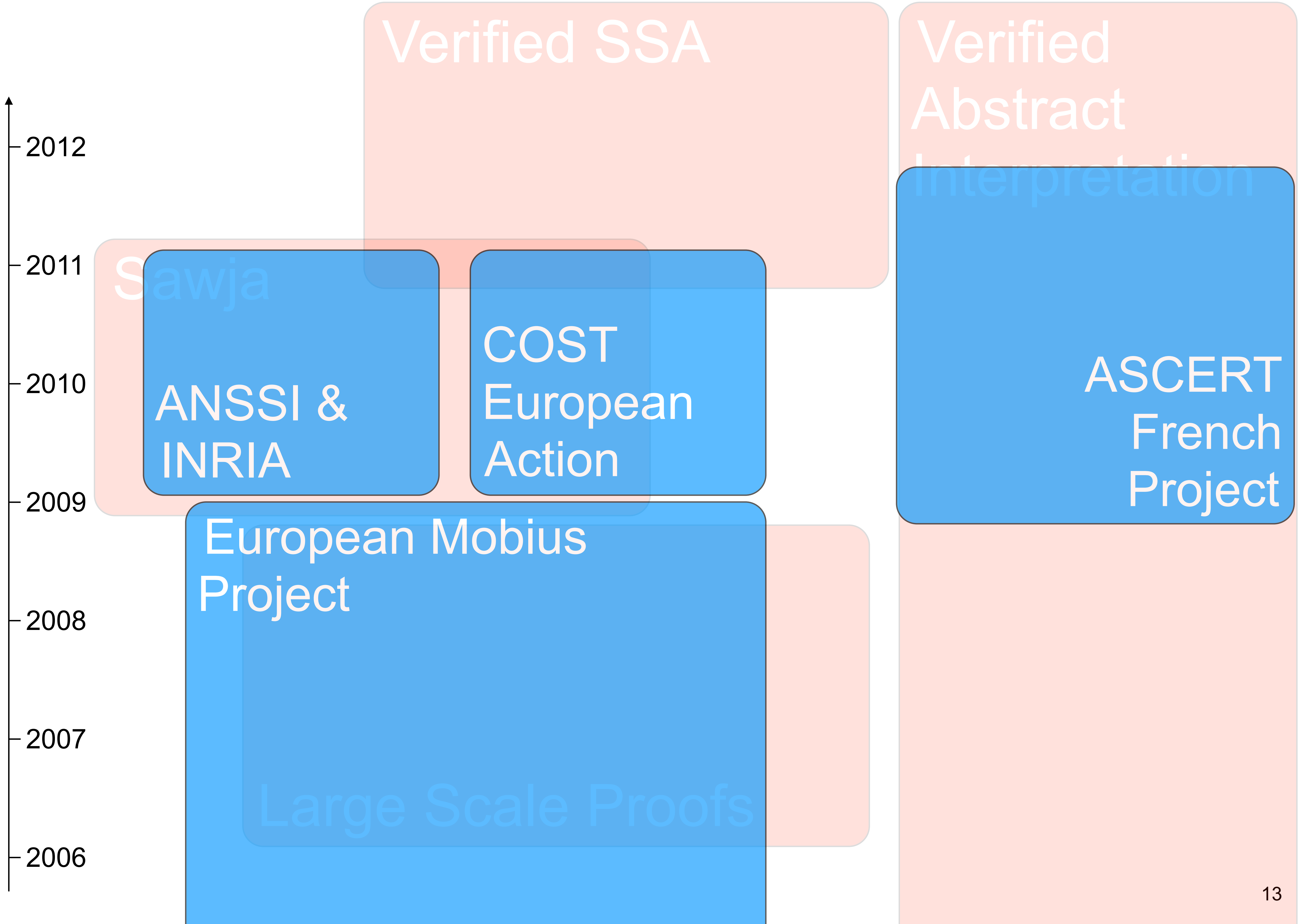
Collaborations



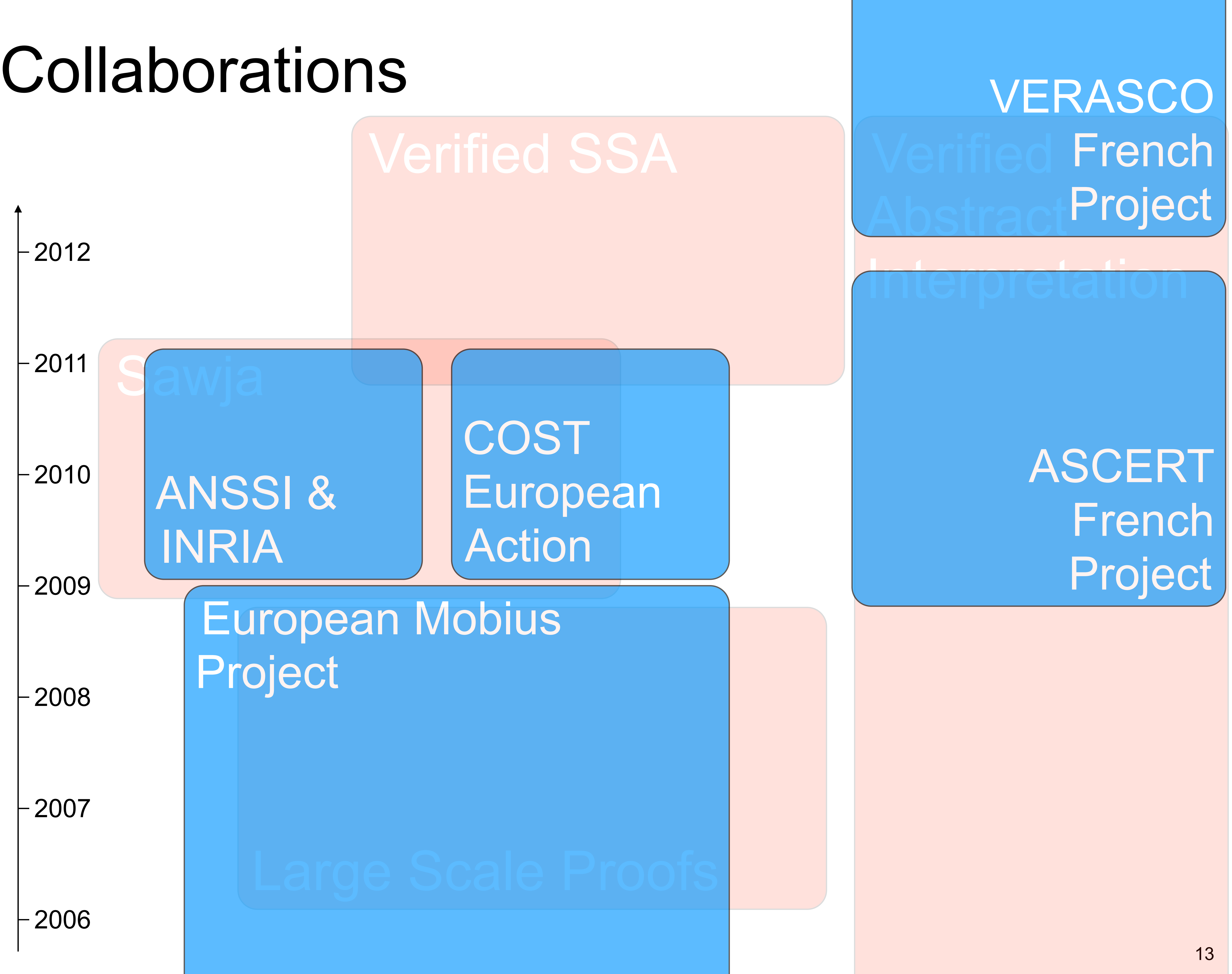
Collaborations



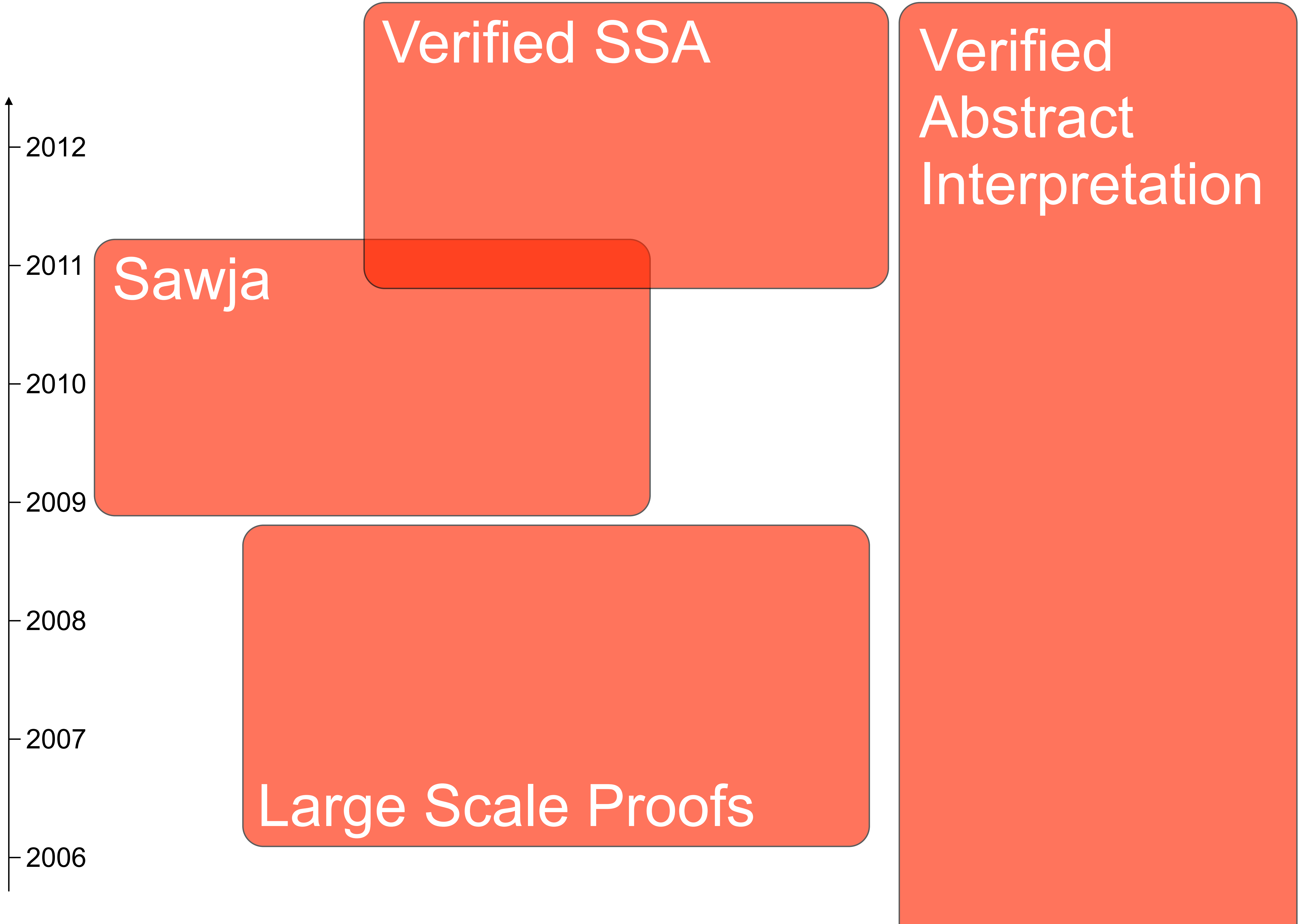
Collaborations



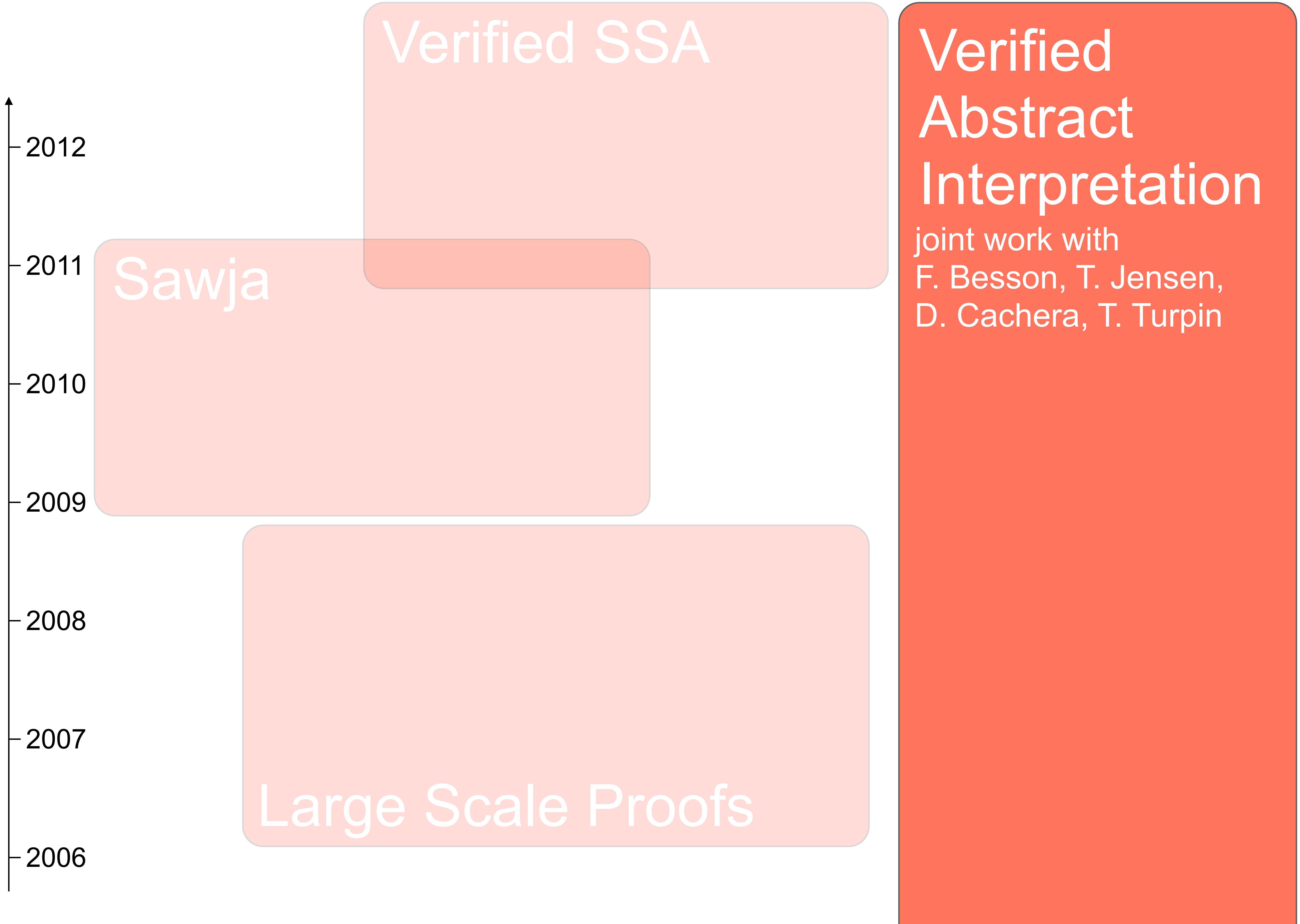
Collaborations



This Talk


Verified SSA
Verified Abstract Interpretation
2012
2011
Sawja
2010
2009
2008
2007
Large Scale Proofs
2006

This Talk


Verified SSA
Sawja
Large Scale Proofs
2012
2011
2010
2009
2008
2007
2006
Verified Abstract Interpretation
joint work with
F. Besson, T. Jensen,
D. Cachera, T. Turpin

Verified Abstract Interpretation

Extend my PhD work about mechanisation of Abstract Interpretation theory

Objectives : to embed more Abstraction Interpretation techniques inside mechanized proofs

Achievements

- *A posteriori* validation of relational abstract domains
- Advanced iteration strategies for widening/narrowing
- First mechanized proof that explicitly uses a collecting semantics
  - turn a standard operational semantics into a collecting interpreter
  - the interpreter is *aligned* with the static analyzer: easier soundness proof

Into the Depths of a Coq Development

```
Program Fixpoint Collect (i:stmt) (l:pp):
                         monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒


  | While p t i ⇒






  | [...]
end.
```

Into the Depths of a Coq Development

recursive function
with dependent
types

```
Program Fixpoint Collect (i:stmt) (l:pp):
                              monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒

  | While p t i ⇒

  | [...]
end.
```

Into the Depths of a Coq Development

recursive function with dependent types

statement of a simple imperative langage

```
Program Fixpoint Collect (i:stmt) (l:pp):
                            monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒

  | While p t i ⇒




  | [...]
end.
```

Into the Depths of a Coq Development

recursive function with dependent types

statement of a simple imperative langage

next program point after statement `i`

```
Program Fixpoint Collect (i:stmt) (l:pp):
                               monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒


  | While p t i ⇒






  | [...]
end.
```

Into the Depths of a Coq Development

recursive function with dependent types

statement of a simple imperative langage

next program point after statement `i`

monotone function from $\mathcal{P}$(`env`) to `pp` $\rightarrow$ $\mathcal{P}$(`env`)

```
Program Fixpoint Collect (i:stmt) (l:pp):
                         monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
   | Assign p x e ⇒


   | While p t i ⇒






   | [...]
end.
```

Into the Depths of a Coq Development

recursive function with dependent types

statement of a simple imperative langage

next program point after statement `i`

monotone function from $\mathcal{P}$(`env`) to `pp` $\rightarrow \mathcal{P}$(`env`)

```
Program Fixpoint Collect (i:stmt) (l:pp):
                         monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒

  | While p t i ⇒




  | [...]
end.
```

set of environments reachable before the statement

Into the Depths of a Coq Development

recursive function with dependent types

statement of a simple imperative langage

next program point after statement `i`

monotone function from $\mathcal{P}(\texttt{env})$ to $\texttt{pp} \to \mathcal{P}(\texttt{env})$

```
Program Fixpoint Collect (i:stmt) (l:pp):
                          monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒


  | While p t i ⇒






  | [...]
end.
```

set of environments reachable before the statement

set of environments reachable at each program point

Into the Depths of a Coq Development

recursive function with dependent types

statement of a simple imperative langage

next program point after statement `i`

monotone function from $\mathcal{P}$(`env`) to `pp` → $\mathcal{P}$(`env`)

```
Program Fixpoint Collect (i:stmt) (l:pp):
                          monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒

  | While p t i ⇒

  | [...]
end.
```

set of environments reachable before the statement

set of environments reachable at each program point

pattern matching on the statement `i`

Into the Depths of a Coq Development

```
Program Fixpoint Collect (i:stmt) (l:pp):
                          monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒
    Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _




  | [...]
end.
```

Into the Depths of a Coq Development

```
Program Fixpoint Collect (i:stmt) (l:pp):
                                monotone (𝒫(env)) (pp → 𝒫(env)) :=
                   x := e
match i with
  | Assign p x e ⇒
    Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _




  | [...]
end.
```

Into the Depths of a Coq Development

```
Program Fixpoint Collect (i:stmt) (l:pp):
                            monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒
    Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _
  | [...]
end.
```

x := e

constructor of
monotone functions

Into the Depths of a Coq Development

```
Program Fixpoint Collect (i:stmt) (l:pp):
                    monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒
    Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _
  | [...]
end.
```

x := e

constructor of
monotone functions

(Mono f π): monotone A B

iff

- f: A → B
- π is a proof term of « f monotone »

Into the Depths of a Coq Development

```
Program Fixpoint Collect (i:stmt) (l:pp):
                        monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒
    Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _
```

x := e

constructor of monotone functions

hole for the monotonicity proof (automatically filled)

```
  | [...]
end.
```

(Mono f π): monotone A B

iff

- f: A $\to$ B
- π is a proof term of « f monotone »

Into the Depths of a Coq Development

```
Program Fixpoint Collect (i:stmt) (l:pp):
                         monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒
    Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _
  | [...]
end.
```

x := e

constructor of monotone functions

the precondition is attached to p

hole for the monotonicity proof (automatically filled)

`(Mono f π): monotone A B`

iff

- `f: A → B`
- π is a proof term of « `f` monotone »

Into the Depths of a Coq Development

```
Program Fixpoint Collect (i:stmt) (l:pp):
                       monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒
    Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _
  | [...]
end.
```

x := e

constructor of monotone functions

the precondition is attached to p

the strongest postcondition of Env is attached to l

hole for the monotonicity proof (automatically filled)

(Mono f π): monotone A B

iff

- f: A $\to$ B
- π is a proof term of « f monotone »

Into the Depths of a Coq Development

```
Program Fixpoint Collect (i:stmt) (l:pp):
                            monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
   | Assign p x e ⇒
     Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

   | While p t i ⇒
     Mono (fun Env ⇒
          let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
            (Collect i p (assert t I))
                 +[p ↦ I] +[l ↦ assert (Not t) I]) _

   | [...]
end.
```

Into the Depths of a Coq Development

```
Program Fixpoint Collect (i:stmt) (l:pp):
                             monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
   | Assign p x e ⇒
     Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

   | While p t i ⇒
     Mono (fun Env ⇒
          let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
            (Collect i p (assert t I))
                 +[p ↦ I] +[l ↦ assert (Not t) I]) _

   | [...]
end.
```

while t do i

Into the Depths of a Coq Development

```
Program Fixpoint Collect (i:stmt) (l:pp):
                            monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
   | Assign p x e ⇒
     Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

   | While p t i ⇒
     Mono (fun Env ⇒
          let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
            (Collect i p (assert t I))
                 +[p ↦ I] +[l ↦ assert (Not t) I]) _

   | [...]
end.
```

while t do i

loop invariant

Into the Depths of a Coq Development

```
Program Fixpoint Collect (i:stmt) (l:pp):
                         monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒
    Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

  | While p t i ⇒
    Mono (fun Env ⇒
          let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
            (Collect i p (assert t I))
                 +[p ↦ I] +[l ↦ assert (Not t) I]) _

  | [...]
end.
```

while t do i

loop invariant

least fixpoint on
complete lattices

Into the Depths of a Coq Development

```
Program Fixpoint Collect (i:stmt) (l:pp):
                          monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e ⇒
    Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

  | While p t i ⇒
    Mono (fun Env ⇒
          let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
            (Collect i p (assert t I))
                 +[p ↦ I] +[l ↦ assert (Not t) I]) _

  | [...]
end.
```

while t do i

loop invariant

least fixpoint on complete lattices

builds the fixpoint equation
`I == Env ∪`
`    (Collect i p (assert t I) p)`

Verified Abstract Interpretation

Lessons learned

Abstract Interpretation is still marginally employed in PL mechanized proofs

- Not always the fastest path to prove an analysis
- Lack of good tutorials

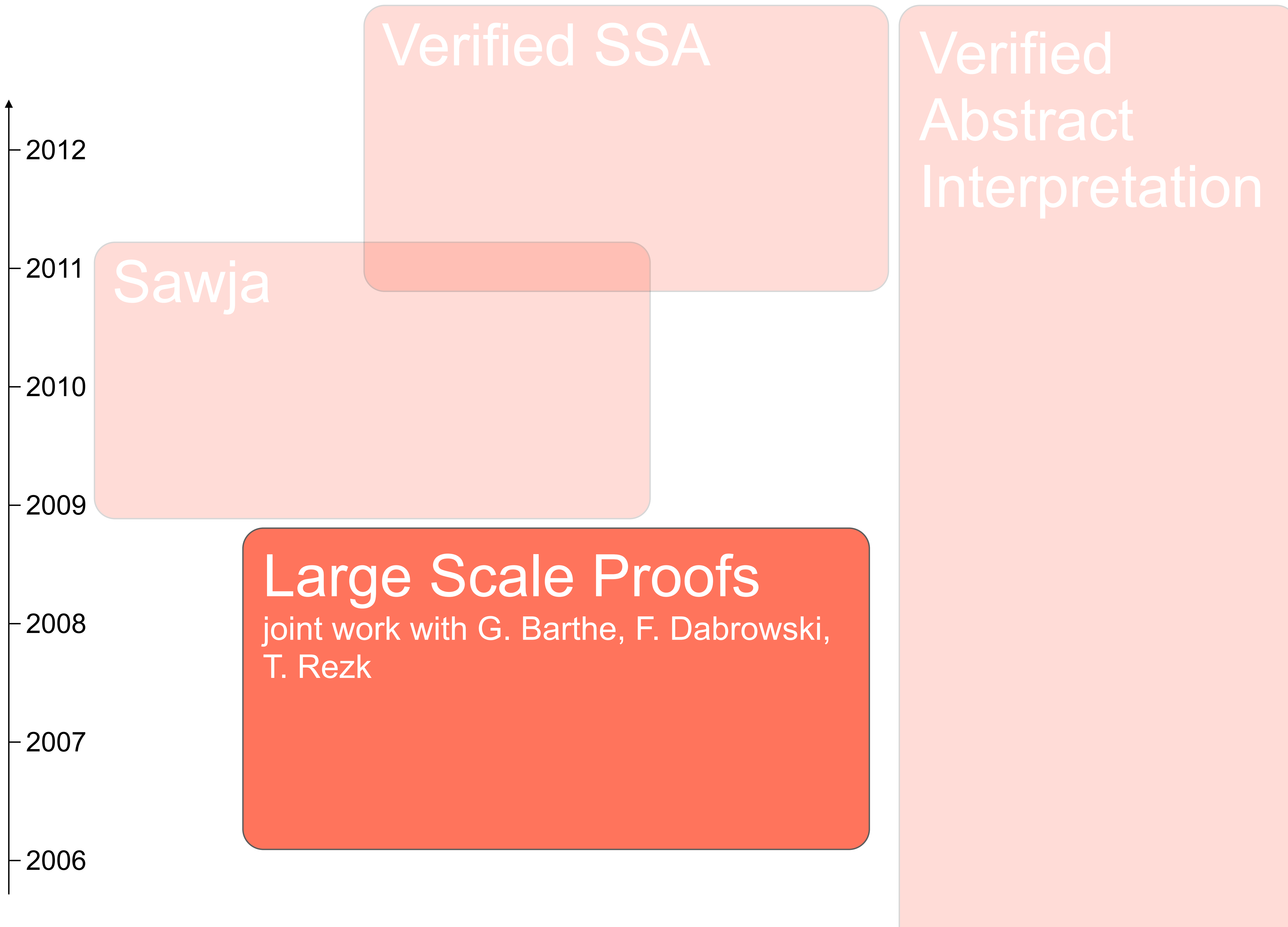
Still a rewarding proof technique

- It gives a better understanding of an analysis (collecting semantics)
- Putting everything in a same framework facilitates combination of analyses

Current limitations

- Still no symbolic derivation of a best abstract transformer
- Hard to draw definite conclusions about the scalability of the *a posteriori* approach on large programs: need to test this ideas on a more realistic langage

This Talk


Verified SSA
Verified Abstract Interpretation
Sawja
Large Scale Proofs
joint work with G. Barthe, F. Dabrowski, T. Rezk
2012
2011
2010
2009
2008
2007
2006

Large Scale Proofs

Objectives:

- Go beyond the state of the art in terms of the complexity of the mechanized static analysis
- Attempt to capture large fragments of the Java semantics

Achievements:

- Soundness of an information flow type checker for Java bytecode
- Soundness of a data race analysis for Java bytecode

Information Flow Type Checker

Information Flow Type Checker

Objectives

- Proving non-interference of Java bytecode programs
- Public outputs should not depend on secret inputs

Information Flow Type Checker

Objectives

- Proving non-interference of Java bytecode programs
- Public outputs should not depend on secret inputs
- Non-interference can be enforced by a type system [Volpano97]

D. Volpano and G. Smith, *A Type-Based Approach to Program Security*, Theory and Practice of Software Development, 1997.

$$\text{CONST}\,\frac{}{\vdash n : L} \qquad \text{VAR}\,\frac{x \in \mathbb{V}_\tau}{\vdash x : \tau} \qquad \text{BINOP}\,\frac{\vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash e_1\ o\ e_2 : \tau}$$

$$\text{EXP-SUBTYP}\,\frac{\vdash e : \tau_1 \quad \tau_1 \sqsubseteq \tau_2}{\vdash e : \tau_2}$$

$$\text{ASSIGN}\,\frac{x \in \mathbb{V}_\tau \quad \vdash e : \tau}{\vdash x := e : \tau} \qquad \text{SEQ}\,\frac{\vdash S_1 : \tau \quad \vdash S_2 : \tau}{\vdash S_1\ ;\ S_2 : \tau}$$

$$\text{IF}\,\frac{\vdash e : \tau \quad \vdash S_1 : \tau \quad \vdash S_2 : \tau}{\vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \tau} \qquad \text{WHILE}\,\frac{\vdash e : \tau \quad \vdash S : \tau}{\vdash \texttt{while}\ e\ \texttt{do}\ S : \tau}$$

$$\text{STM-SUBTYP}\,\frac{\vdash S : \tau_2 \quad \tau_1 \sqsubseteq \tau_2}{\vdash S : \tau_1}$$

Information Flow Type Checker

Objectives

- Proving non-interference of Java bytecode programs
- Public outputs should not depend on secret inputs
- Non-interference can be enforced by a type system [Volpano97]

Achievements: mechanized proof of a type-checker for a bytecode langage handling

- unstructured control flow
- operand stack
- exceptions
- objects and array dynamically allocated
- classes and virtual method calls

```
| vaload_np_caught : forall i te t k1 k2 ke st,
  (forall j, region i (Some np) j -> k2 <= se j) ->
  handler i np = Some te ->
  texec i (Vaload t) (Some np) (L.Simple k1::L.Array k2 ke::st) (Some (L.Simple k2::nil))
| vaload_np_uncaught : forall i t k1 k2 ke st,
  (forall j, region i (Some np) j -> k2 <= se j) ->
  k2 <= sgn.(resExceptionType) np ->
  handler i np = None ->
  texec i (Vaload t) (Some np) (L.Simple k1::L.Array k2 ke::st) None
| vaload_iob_caught : forall i te t k1 k2 ke st,
  (forall j, region i (Some iob) j -> k1 U k2 <= se j) ->
  handler i iob = Some te ->
  texec i (Vaload t) (Some iob) (L.Simple k1::L.Array k2 ke::st) (Some (L.Simple (k1 U k2)
| vaload_iob_uncaught : forall i t k1 k2 ke st,
  (forall j, region i (Some iob) j -> k1 U k2 <= se j) ->
  k1 U k2 <= sgn.(resExceptionType) iob ->
  handler i iob = None ->
  texec i (Vaload t) (Some iob) (L.Simple k1::L.Array k2 ke::st) None
| vastore : forall i t kv ki ka ke st,
  kv <=' ke ->
  ki <= ke ->
  ka <= ke ->
  (forall j, region i None j -> ka <= se j) ->
  (forall j, region i None j -> (L.join ki ka) <= (se j)) ->
  (forall j, region i None j -> ke <= (se j)) ->
  L.leql' (keepEffect sgn) ke ->
  texec i (Vastore t) None (kv::L.Simple ki::L.Array ka ke::st) (Some (elift m i ke st))
| vastore_np_caught : forall i te t kv ki ka ke st,
  (forall j, region i (Some np) j -> ka <= se j) ->
  handler i np = Some te ->
  texec i (Vastore t) (Some np) (kv::L.Simple ki::L.Array ka ke::st) (Some (L.Simple ka::n
| vastore_np_uncaught : forall i t kv ki ka ke st,
  (forall j, region i (Some np) j -> ka <= se j) ->
  ka <= sgn.(resExceptionType) np ->
  handler i np = None ->
  texec i (Vastore t) (Some np) (kv::L.Simple ki::L.Array ka ke::st) None
| vastore_ase_caught : forall i te t ki ka (kv ke:L.t') st,
  (forall j, region i (Some ase) j -> (L.join kv (L.join ki ka)) <= se j) ->
  handler i ase = Some te ->
  texec i (Vastore t) (Some ase) (kv::L.Simple ki::L.Array ka ke::st)
                                  (Some (L.Simple (L.join kv (L.join ki ka))::nil)
| vastore_ase_uncaught : forall i t ki ka (kv ke:L.t') st,
  (forall j, region i (Some ase) j -> (L.join kv (L.join ki ka)) <= se j) ->
  (L.join kv (L.join ki ka)) <= sgn.(resExceptionType) ase ->
  handler i ase = None ->
  texec i (Vastore t) (Some ase) (kv::L.Simple ki::L.Array ka ke::st) None
| vastore_iob_caught : forall i te t ki ka (kv ke:L.t') st,
  (forall j, region i (Some iob) j -> (L.join ki ka) <= se j) ->
  handler i iob = Some te ->
  texec i (Vastore t) (Some iob) (kv::L.Simple ki::L.Array ka ke::st) (Some (L.Simple (L.j
| vastore_iob_uncaught : forall i t ki ka (kv ke:L.t') st,
  (forall j, region i (Some iob) j -> (L.join ki ka) <= se j) ->
  (L.join ki ka) <= sgn.(resExceptionType) iob ->
  handler i iob = None ->
  texec i (Vastore t) (Some iob) (kv::L.Simple ki::L.Array ka ke::st) None
| vload : forall i t x st,
  texec i (Vload t x) None st (Some (L.join' (se i) (sgn.(lvt) x)::st))
| vstore : forall i t x k st,
  se i <= sgn.(lvt) x ->
  L.leql' k (sgn.(lvt) x) ->
  texec i (Vstore t x) None (k::st) (Some st)
| vreturn : forall i x k kv st,
  sgn.(resType) = Some kv ->
  L.leql' k kv ->
  texec i (Vreturn x) None (k::st) None.
```

66 typing rules...

Data Race Analysis

A Challenging Example

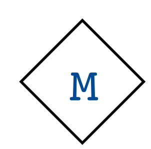
```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

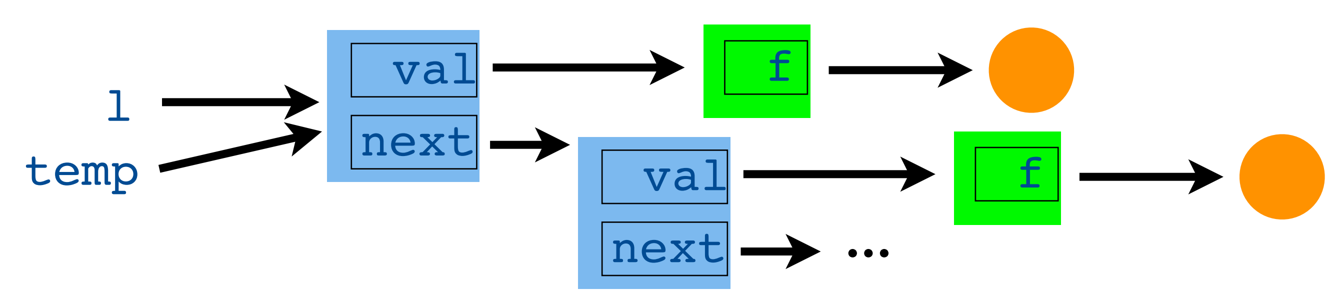
Data Race Analysis

A Challenging Example

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

1. We create a link list `l`

Threads: 

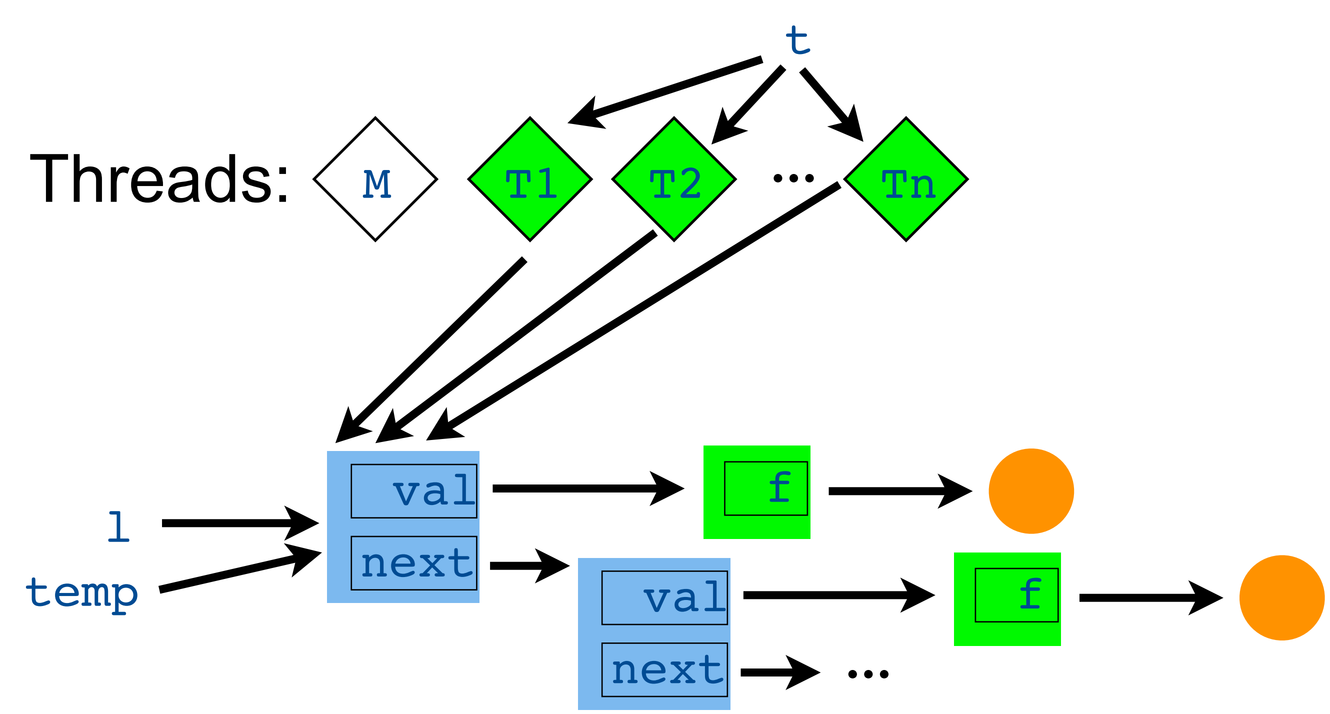


Data Race Analysis

A Challenging Example

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

1. We create a link list `l`
2. We create several threads that all share the list `l`

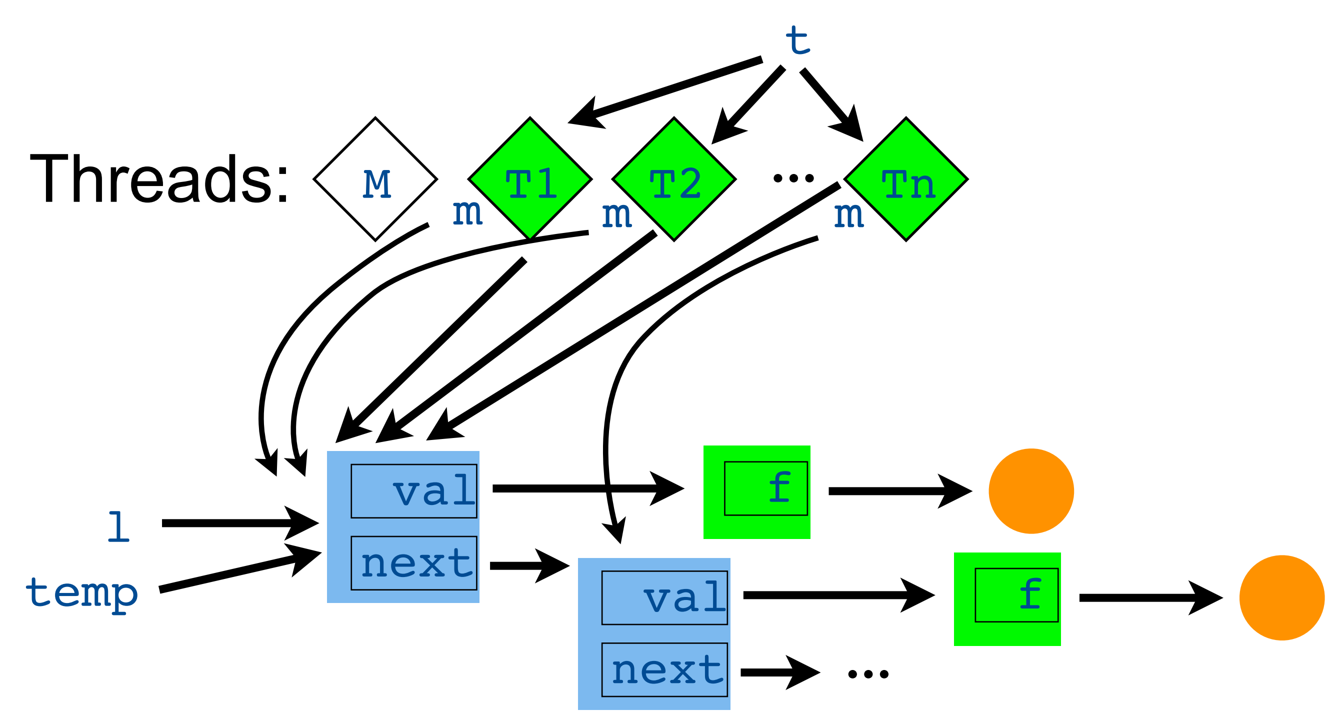


Data Race Analysis

A Challenging Example

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

1. We create a link list `l`
2. We create several threads that all share the list `l`
3. Each thread chooses a cell, takes a lock on it and updates it.

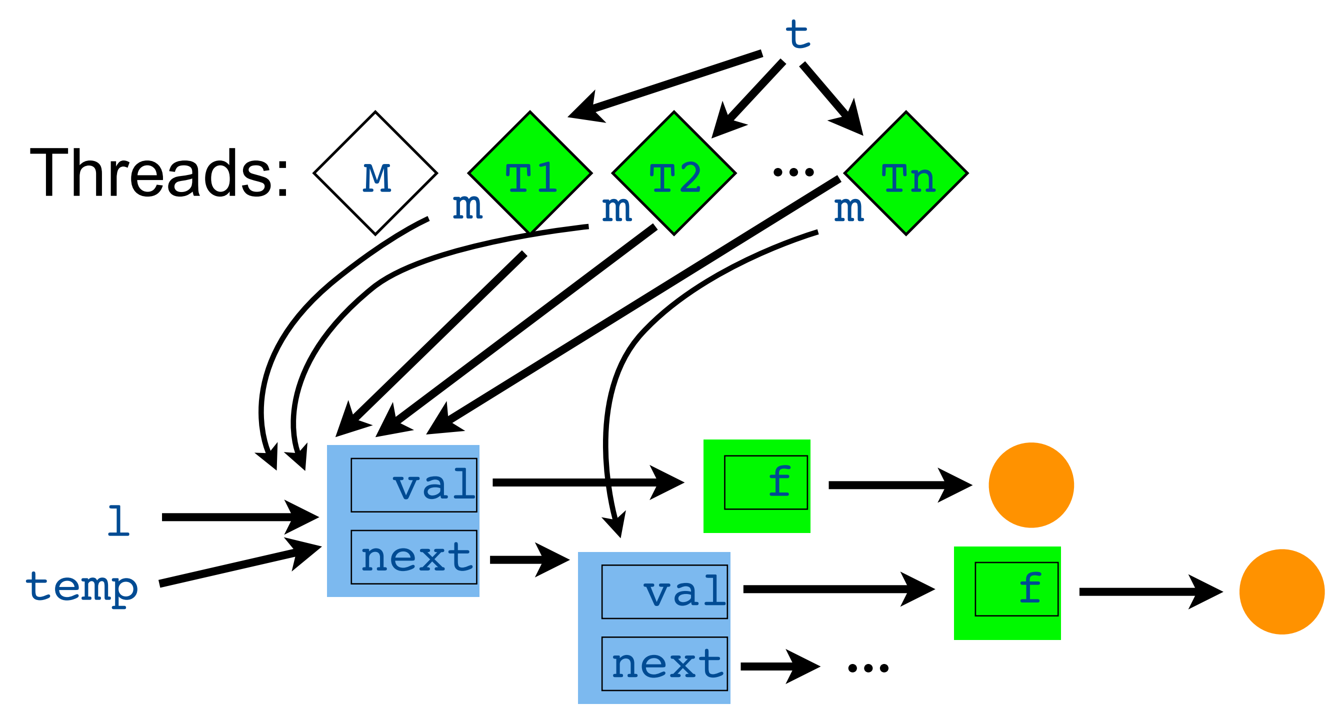


Data Race Analysis

A Challenging Example

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

1. We create a link list `l`
2. We create several threads that all share the list `l`
3. Each thread chooses a cell, takes a lock on it and updates it.



Data Race Analysis

A Challenging Example

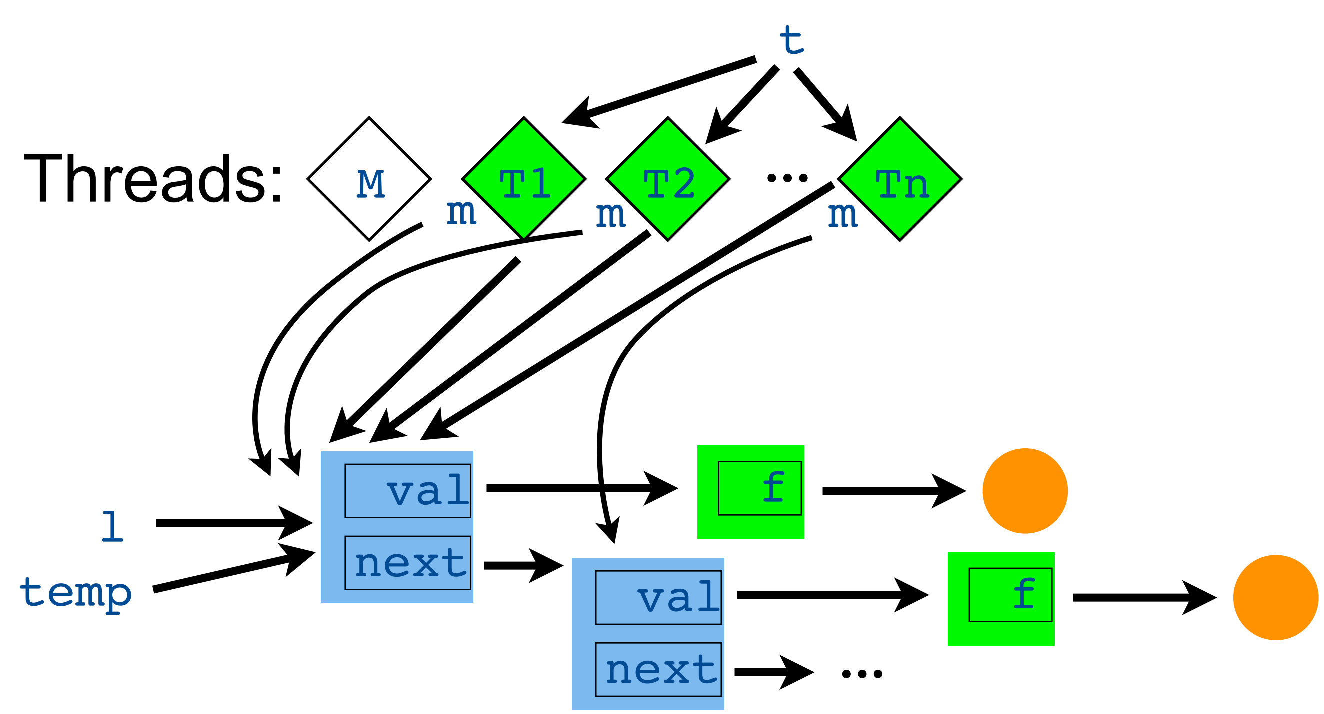
```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

Definition [Data Race]

- The situation where two different processes attempt to access to the same memory location and at least one access is a write.

Question

- Is this program data race free ?



Effective Static Data Race Detection for Java

Naik's PhD 2008

Original pairs

Reachable pairs

Aliasing pairs

Escaping pairs

Unlocked pairs

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

Effective Static Data Race Detection for Java

Naik's PhD 2008

- Original pairs
- Reachable pairs
- Aliasing pairs
- Escaping pairs
- Unlocked pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) (2,f,5) (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

The first set of potential races is based on field safety

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

Effective Static Data Race Detection for Java

Naik's PhD 2008

Original pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) (2,f,5) (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Reachable pairs

~~(1,val,1)~~ ~~(1,val,2)~~ ~~(2,f,2)~~ ~~(3,next,3)~~
~~(4,data,4)~~ ~~(5,f,5)~~ ~~(2,f,5)~~ (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

The pairs that may be reachable from the program entry and that may concern two distinct threads

Aliasing pairs

Escaping pairs

Unlocked pairs

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
  }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

Effective Static Data Race Detection for Java

Naik's PhD 2008

Original pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) (2,f,5) (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Reachable pairs

~~(1,val,1)~~ ~~(1,val,2)~~ ~~(2,f,2)~~ ~~(3,next,3)~~
~~(4,data,4)~~ ~~(5,f,5)~~ ~~(2,f,5)~~ (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Aliasing pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) ~~(2,f,5)~~ ~~(5,f,8)~~ (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Using a points-to abstraction we compute the pairs that may touch the same heap location

Escaping pairs

Unlocked pairs

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

Effective Static Data Race Detection for Java

Naik's PhD 2008

Original pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) (2,f,5) (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Reachable pairs

(1,~~val~~,1) (1,~~val~~,2) (2,~~f~~,2) (3,~~next~~,3)
(4,~~data~~,4) (5,~~f~~,5) (2,~~f~~,5) (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Aliasing pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) (2,~~f~~,5) (5,~~f~~,8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Escaping pairs

(1,~~val~~,1) (1,~~val~~,2) (2,~~f~~,2) (3,~~next~~,3)
(4,~~data~~,4) (5,f,5) (2,~~f~~,5) (5,f, 8) (4,~~data~~,6)
(3,~~next~~, 7) (1,~~val~~, 8) (2,~~f~~,8) (8,f, 8)

The pairs of program points where the target location may be shared at that points

Unlocked pairs

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

Effective Static Data Race Detection for Java

Naik's PhD 2008

Original pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) (2,f,5) (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Reachable pairs

~~(1,val,1)~~ ~~(1,val,2)~~ ~~(2,f,2)~~ ~~(3,next,3)~~
~~(4,data,4)~~ ~~(5,f,5)~~ ~~(2,f,5)~~ (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Aliasing pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) ~~(2,f,5)~~ ~~(5,f,8)~~ (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Escaping pairs

~~(1,val,1)~~ ~~(1,val,2)~~ ~~(2,f,2)~~ ~~(3,next,3)~~
~~(4,data,4)~~ (5,f,5) ~~(2,f,5)~~ (5,f, 8) ~~(4,data,6)~~
~~(3,next, 7)~~ ~~(1,val, 8)~~ ~~(2,f,8)~~ (8,f, 8)

Unlocked pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) (2,f,5) (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) ~~(8,f,8)~~

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
```

Pairs that may not be guarded by the same lock

Effective Static Data Race Detection for Java

Naik's PhD 2008

Original pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) (2,f,5) (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Reachable pairs

~~(1,val,1)~~ ~~(1,val,2)~~ ~~(2,f,2)~~ ~~(3,next,3)~~
~~(4,data,4)~~ ~~(5,f,5)~~ ~~(2,f,5)~~ (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Aliasing pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) ~~(2,f,5)~~ ~~(5,f,8)~~ (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Escaping pairs

~~(1,val,1)~~ ~~(1,val,2)~~ ~~(2,f,2)~~ ~~(3,next,3)~~
~~(4,data,4)~~ (5,f,5) ~~(2,f,5)~~ (5,f, 8) ~~(4,data,6)~~
~~(3,next, 7)~~ ~~(1,val, 8)~~ ~~(2,f,8)~~ (8,f, 8)

Unlocked pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) (2,f,5) (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) ~~(8,f,8)~~

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
```

If each original pair has been removed at least one time, the program is data-race free

```
5:      t.f = ...;}
    return;
  }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:      List m = this.data;
7:      while (*) { m = m.next; }
8:      synchronized(m){ m.val.f = ...;}}
    return;}}
```

Effective Static Data Race Detection for Java

Naik's PhD 2008

Original pairs

(1,~~val~~,1) (1,~~val~~,2) (2,~~f~~,2) (3,~~next~~,3)
(4,~~data~~,4) (5,~~f~~,5) (2,~~f~~,5) (5,~~f~~,8) (4,~~data~~,6)
(3,~~next~~, 7) (1,~~val~~, 8) (2,~~f~~, 8) (~~8,f~~, 8)

Reachable pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) (2,f,5) (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Aliasing pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) (2,f,5) (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Escaping pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) (2,f,5) (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

Unlocked pairs

(1,val,1) (1,val,2) (2,f,2) (3,next,3)
(4,data,4) (5,f,5) (2,f,5) (5,f, 8) (4,data,6)
(3,next, 7) (1,val, 8) (2,f, 8) (8,f, 8)

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
```

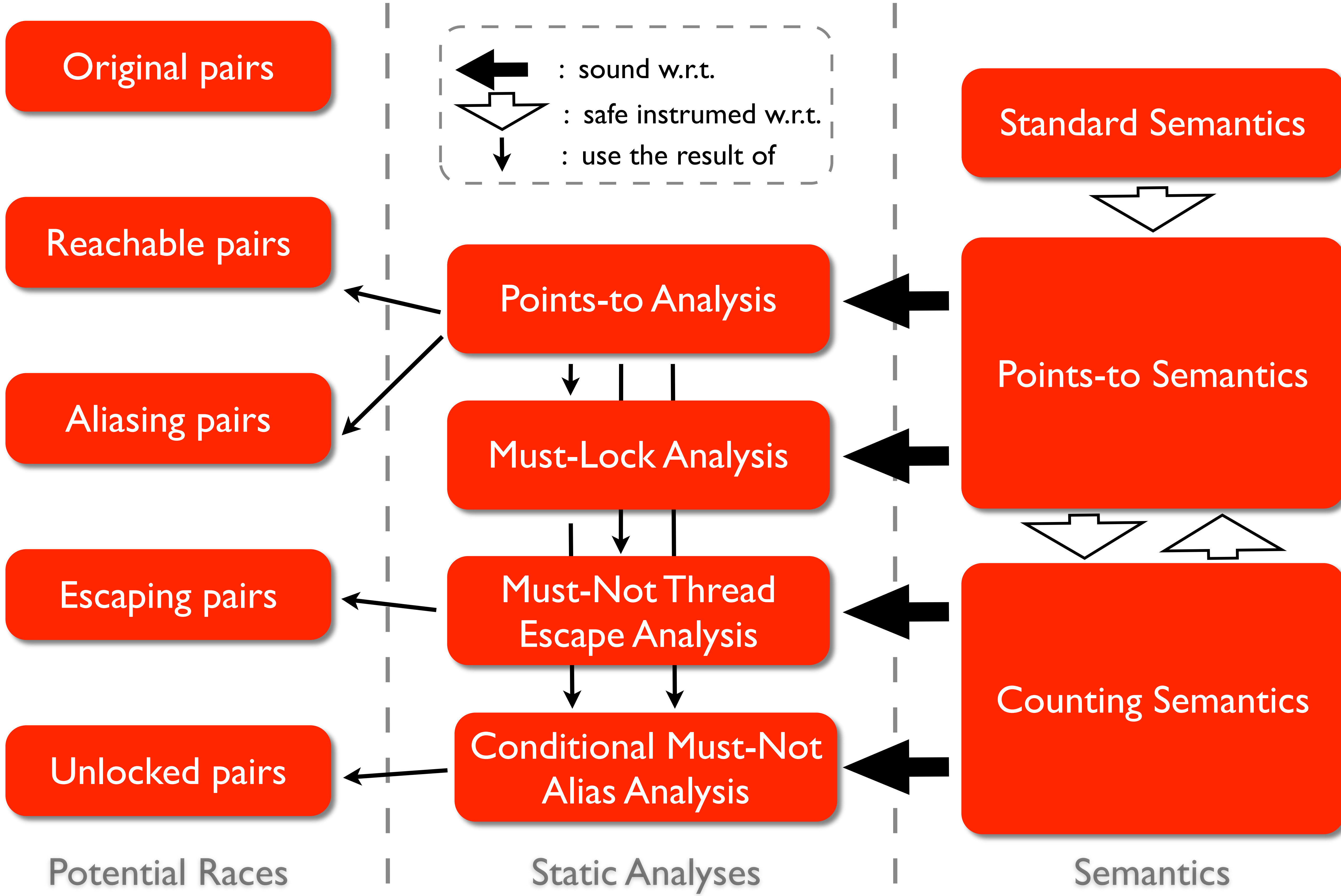
If each original pair has been removed at least one time, the program is data-race free

```
5:      t.f = ...;}
    return;
  }
```

This program is data-race free!

```
7:      while (*) { m = m.next; }
8:      synchronized(m){ m.val.f = ...;}}
    return;}}
```

Proof Architecture



Large Scale Proofs

Lessons learned

- Complex analyses are generally a composition of several sub-analyses.
  Mechanized proof make explicit the interactions between them.
- Directly analyzing bytecode programs is a methodological mistake.
  About 1.5 man year effort and 15K loc for each proof...

Conclusions

- Verified static analysis should learn from what others do with (traditional) static analysis platforms.

Traditional Static Analysis Platforms

Standard architecture of a (Java) static analysis platform

- A parser
- An IR (stackless, SSA)
- A generic fixpoint solver
- Some examples of analysis (control flow analysis is almost mandatory for Java)

Well known platforms: Soot, Wala

Problem: programmed in Java (mutable internal states, visitor patterns...)

Reboot in Coq?...

Traditional Static Analysis Platforms

Standard architecture of a (Java) static analysis platform

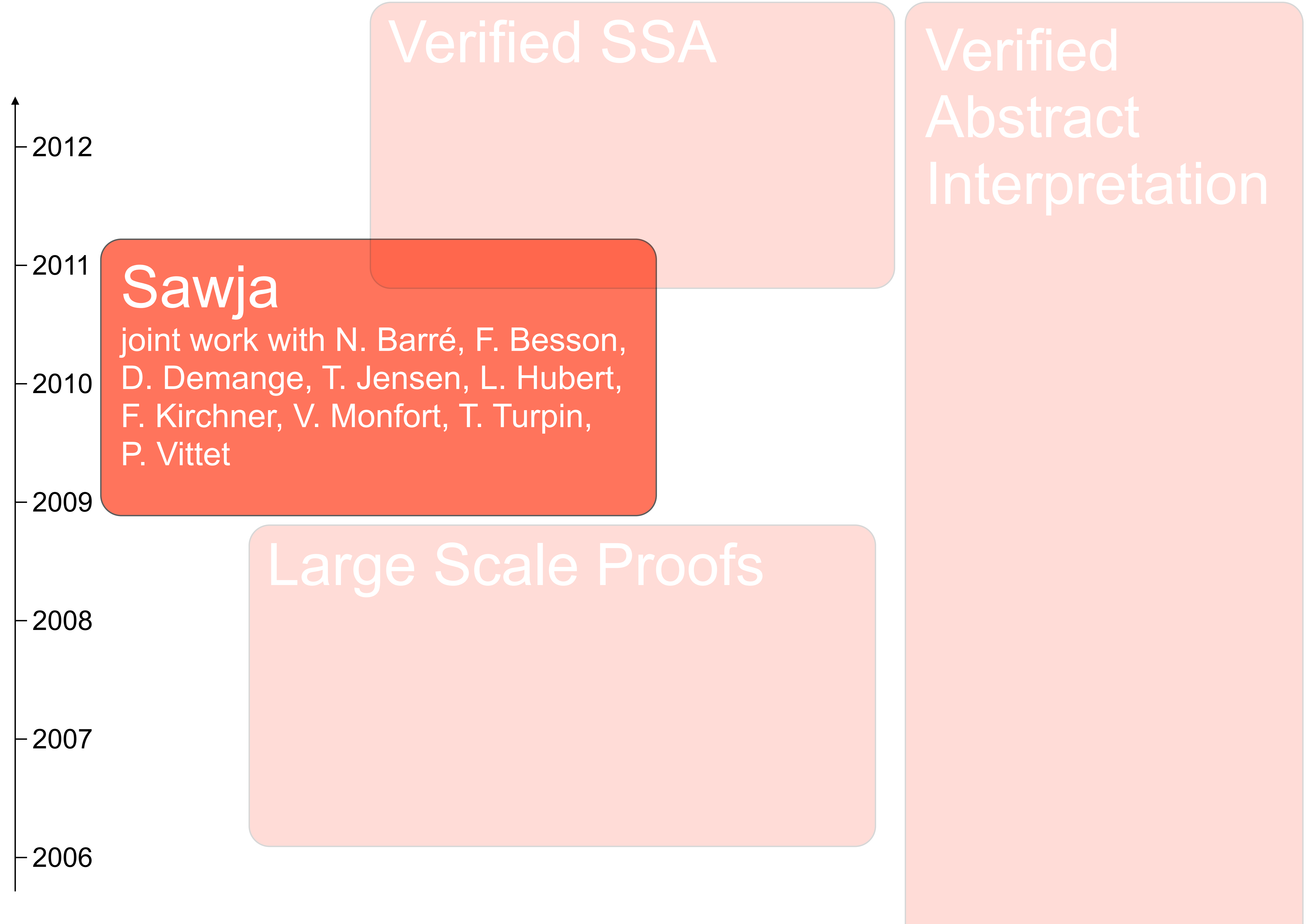
- A parser
- An IR (stackless, SSA)
- A generic fixpoint solver
- Some examples of analysis (control flow analysis is almost mandatory for Java)

Well known platforms: Soot, Wala

Problem: programmed in Java (mutable internal states, visitor patterns...)

Reboot in Coq?... or let's start with OCaml first!

This Talk



Verified SSA
Verified Abstract Interpretation
2012
2011
Sawja
joint work with N. Barré, F. Besson, D. Demange, T. Jensen, L. Hubert, F. Kirchner, V. Monfort, T. Turpin, P. Vittet
2010
2009
Large Scale Proofs
2008
2007
2006

Sawja (Static Analysis Workshop for JAva)

http://sawja.inria.fr

An efficient OCaml library to analyze Java bytecode programs

- A parser generates a hash-consed program representation
- BIR: stackless intermediate representation (with/without SSA)
- Fixpoint solvers (worklist, Bourdoncle iterations)
- Various control flow analyses

Applications: static analyses to strengthen the security of Java programs

- Secure initialisation of objects [ESORICS'10]
- Secure copying of objects [ESOP'11]

Each time a similar methodology

- We prove correct in Coq the analysis on a core language close to BIR
- We implement the analysis with Sawja and run large scale experiments on Java programs to check that our analyses are precise enough

Sawja

Sawja

Lessons learned

- OCaml + BIR + generic fixpoint solvers make the prototyping of static analysis really easy (3 man month)
- Proving a static analysis on a core language is relatively easy (2 man month)
- An OCaml static analysis platform is competitive with Java platform

From OCaml to Coq?

Sawja

Lessons learned

- OCaml + BIR + generic fixpoint solvers make the prototyping of static analysis really easy (3 man month)
- Proving a static analysis on a core language is relatively easy (2 man month)
- An OCaml static analysis platform is competitive with Java platform

From OCaml to Coq?

Most proof challenges rely in the generation of IR

- stackless representation of bytecode (paper proof) [APLAS'11]
- SSA form (Coq proof) [ESOP'12]

Sawja

Lessons learned

- OCaml + BIR + generic fixpoint solvers make the prototyping of static analysis really easy (3 man month)
- Proving a static analysis on a core language is relatively easy (2 man month)
- An OCaml static analysis platform is competitive with Java platform

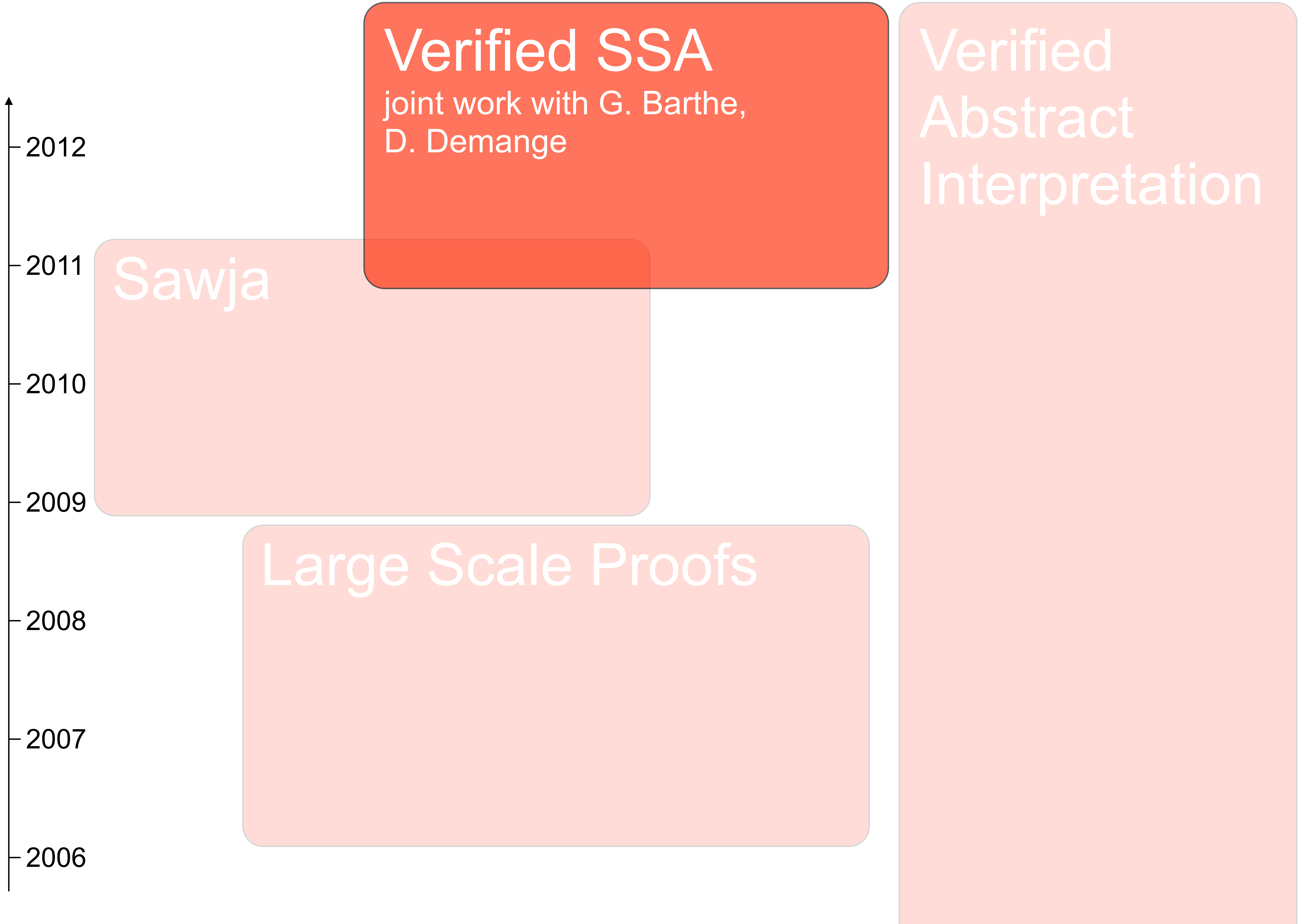
From OCaml to Coq?

Most proof challenges rely in the generation of IR

- stackless representation of bytecode (paper proof) [APLAS'11]
- SSA form (Coq proof) [ESOP'12]

The last challenge for Sawja is concurrency... (stay tuned)

This Talk


Verified SSA
joint work with G. Barthe,
D. Demange
Verified Abstract Interpretation
Sawja
Large Scale Proofs
2012
2011
2010
2009
2008
2007
2006

Verified SSA Representation

The Static Single Assignment (SSA) representation is widely used in modern compilation/analysis platforms but it has never been used in formal proofs.

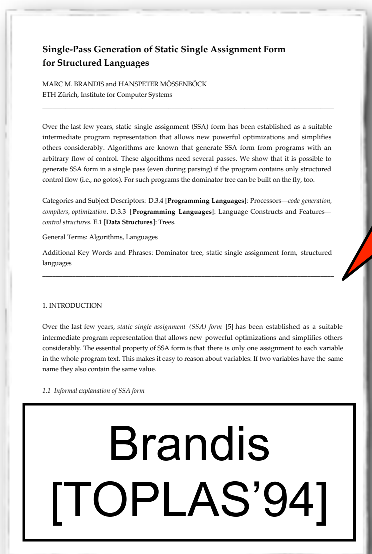
We provide

- a formal semantics of SSA that follows closely (but in a rigorous way) the informal semantics used in the literature
- a verified SSA generator, using the most efficient algorithm of the literature and an original *a posteriori* validator
- a mechanized proof of an emblematic SSA-based static analysis (Global Value Numbering)
- an integration into the CompCert C compiler

Verified SSA Representation

Lessons learned

- Compiler gurus are (almost) right. Our work make precise their informal explanations



Single-Pass Generation of Static Single Assignment Form for Structured Languages

MARC M. BRANDIS and HANSPETER MÖSSENBÖCK

ETH Zürich, Institute for Computer Systems

Brandis [TOPLAS'94]

In SSA,
«If two expressions are textually the same, they are sure to evaluate the same result»

```
Lemma equation_lemma : ∀ d op args x succ f m rs sp pc s,
   (fn_code f)!d = Some (Iop op args x succ) →
   sdom f d pc →
   reachable prog (State s f sp pc rs m) →
   eval_operation ge sp op (rs ## args) m = Some (rs # x).
```

Missing hypothesis about dominance

- As we did for abstract interpretation we try not to reinvent the wheel in our mechanized proofs to make formal proofs benefit from past research
- In return, we expect our work will help those who need to understand these theories

Conclusions

We have explored various virgin lands in the area of verified static analyses

- numerical relational abstract interpreter
- alias analysis
- concurrency
- information flow
- shape
- SSA-based

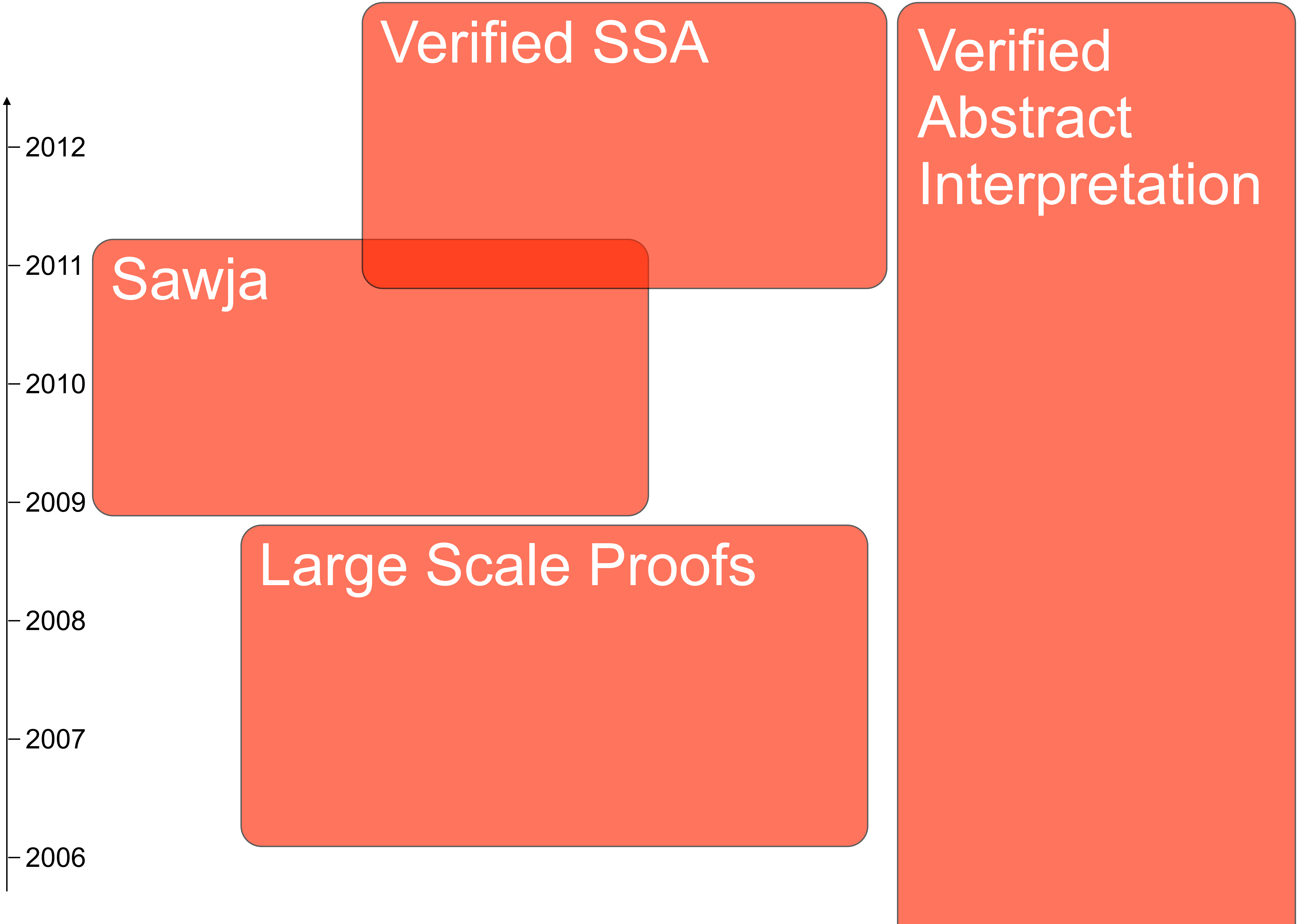
This variety gives a better understanding of

- the high potential of this approach,
- but also its limits (human effort)

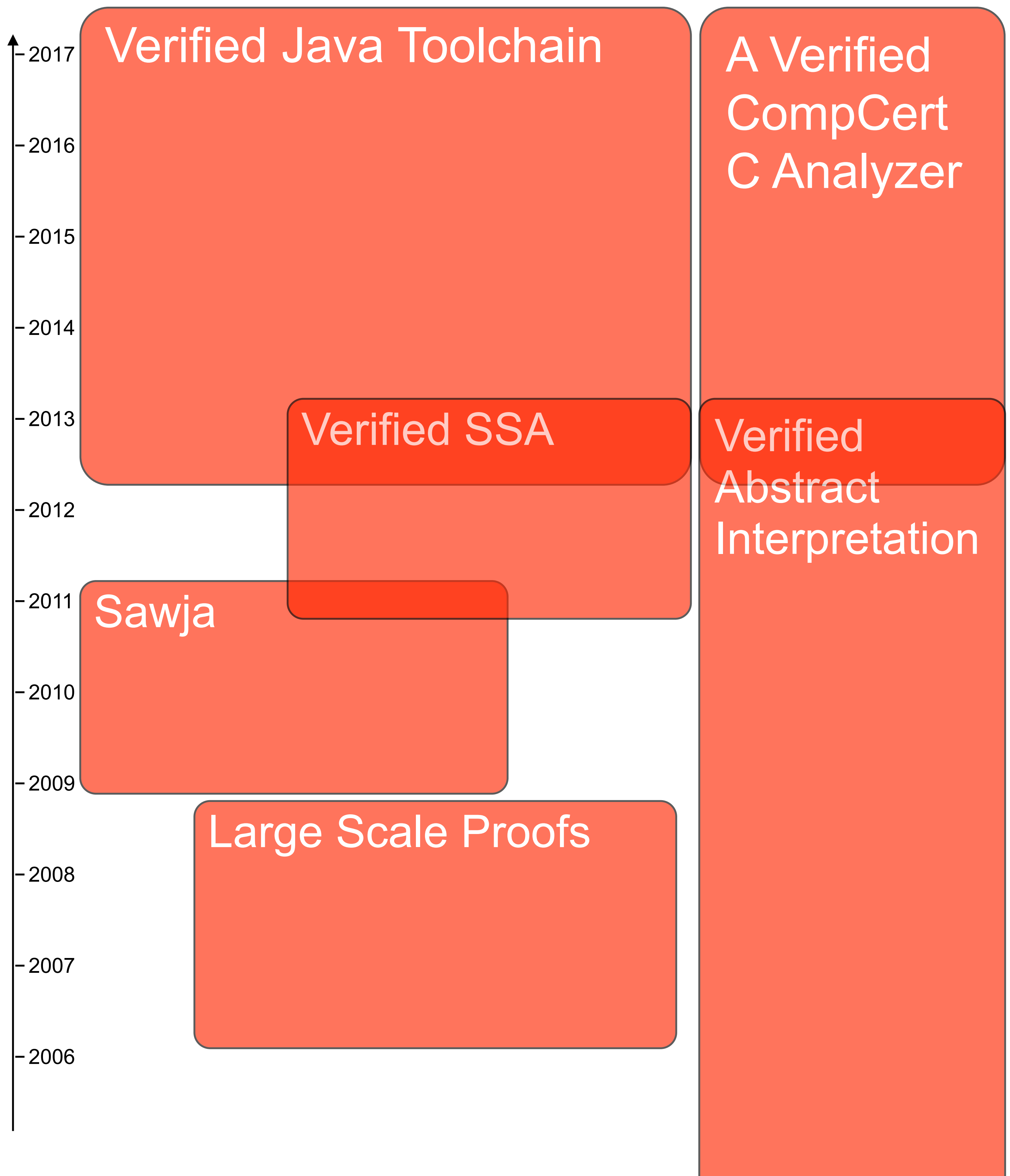
In reaction to these limits, we have built a platform that

- gives the foundations of a verified static analysis platform
- provides a semantically-grounded platform to the static analysis community
- has been applied to design new security mechanisms for Java

Perspectives



Perspectives



1. A Verified CompCert C Analyzer

2. A Verified Software Toolchain for Java

A Verified CompCert C Analyzer

The Verasco Project (Univ. Rennes, Inria Paris & Saclay, Verimag, Airbus)

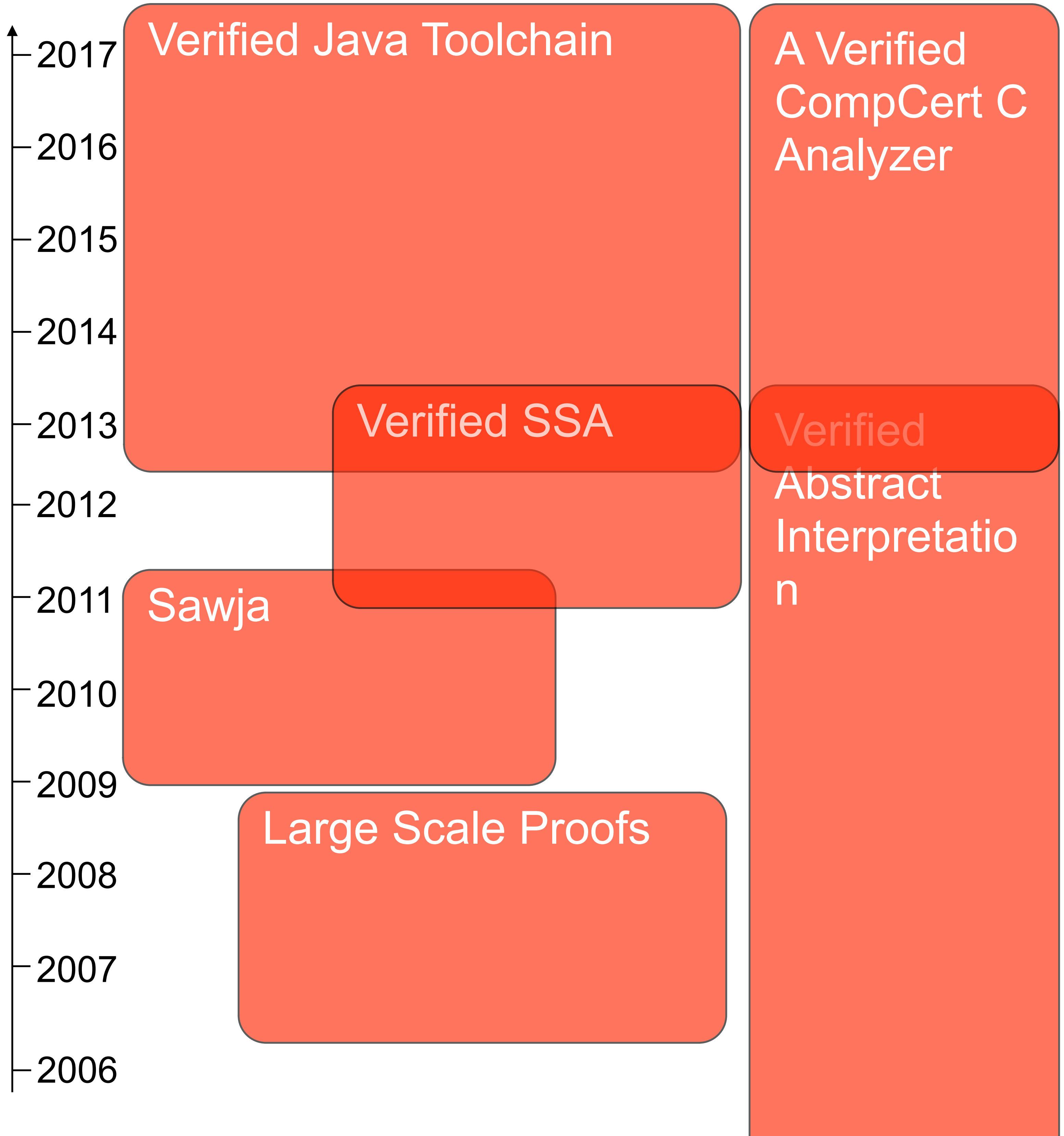
Verified abstract interpretation : it’s time to scale these techniques to a realistic tool

- static analysis inside CompCert
- targets the formal verification of some of the key components of the Astrée tool
  - interprocedural
  - abstraction of the memory
  - relational numerical abstraction both for machine integers and floats

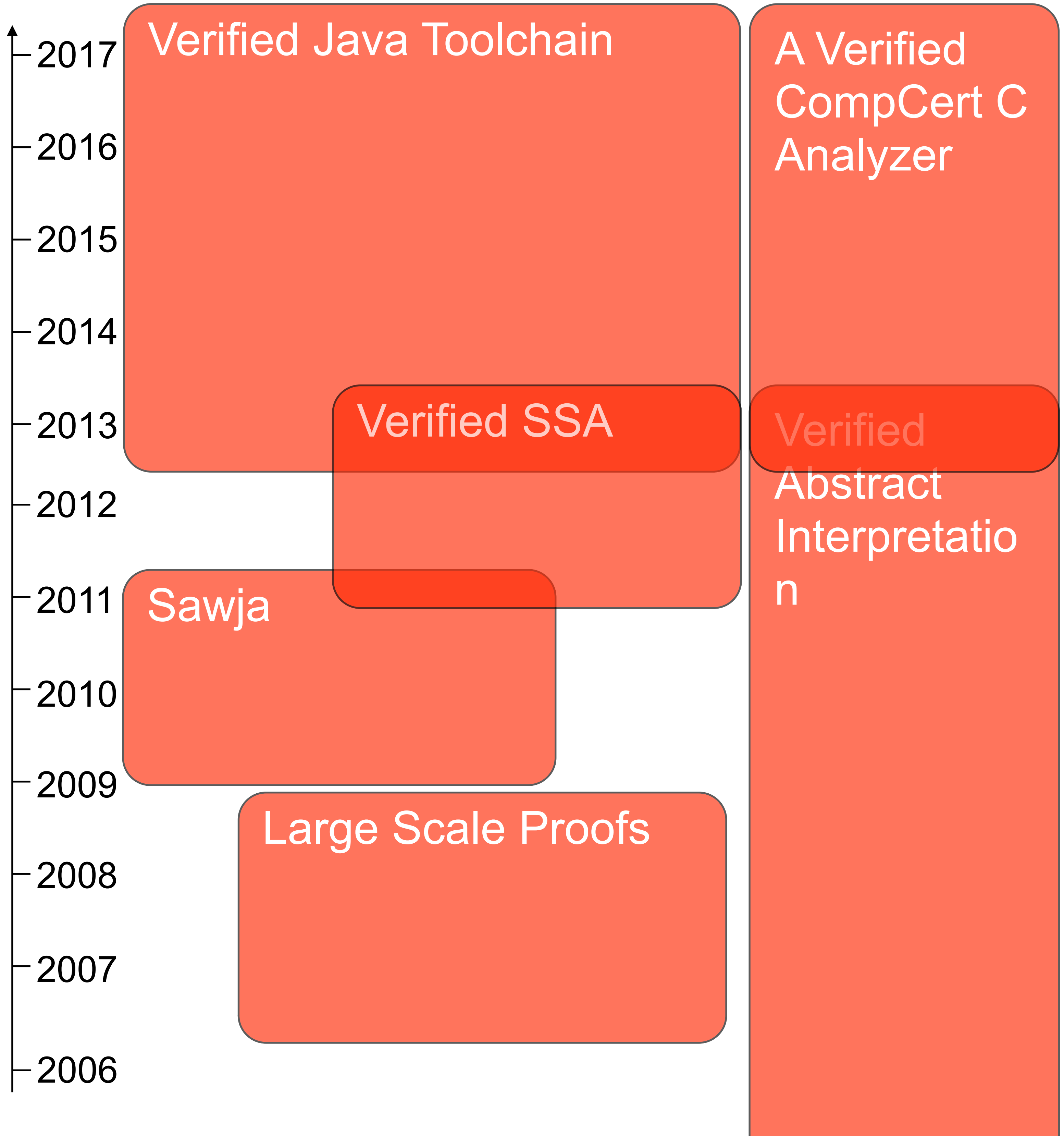
Preliminary results:

- an intra procedural interval analysis of integers on the RTL CompCert IR and its application for a WCET analysis (joint work with A. Maroneze and S. Blazy)

Perspectives



Perspectives



1. A Verified CompCert C Analyzer

2. A Verified Software Toolchain for Java

A Last Challenge: Concurrency

We need to give a semantics to the various Sawja IRs (bytecode, BIR)

- May seem a easy task if we build on top of the previous mechanized semantics (M6, Jinja, Bicolano)
- (Un)fortunately much research remains to be done when formalizing the semantics of concurrent Java programs...

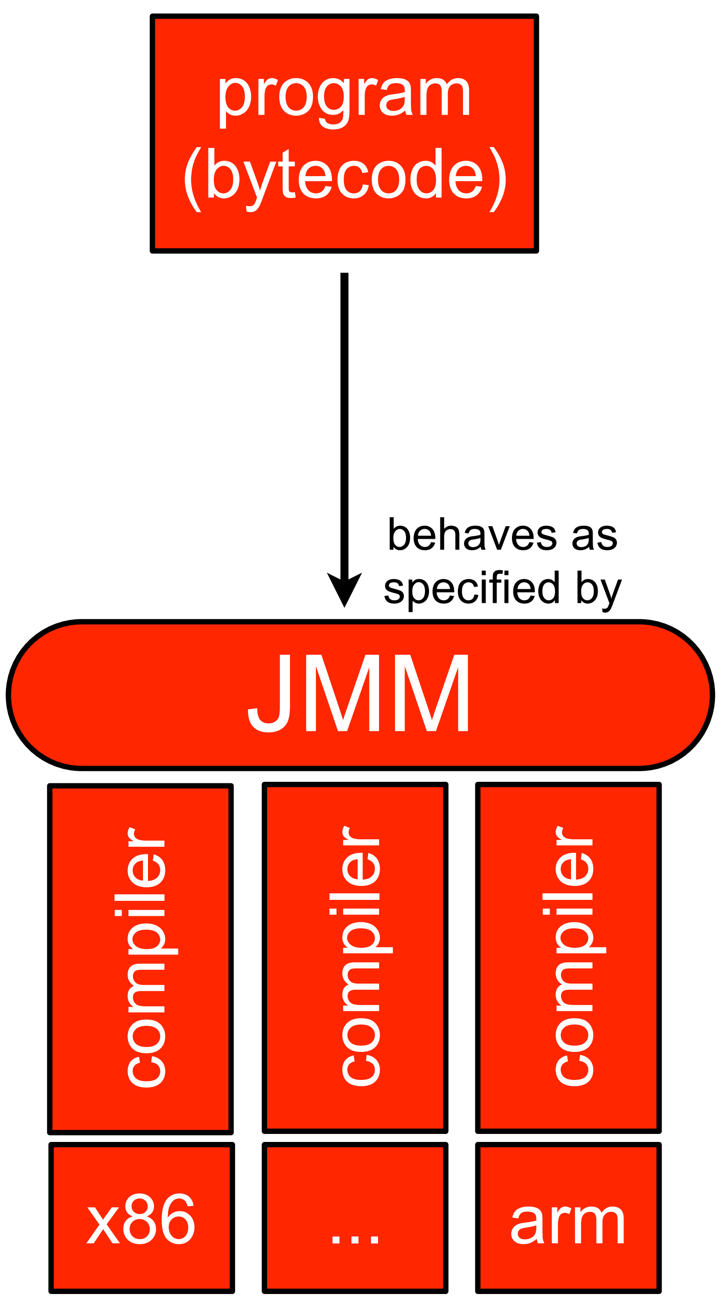
A Last Challenge: Concurrency

We need to give a semantics to the various Sawja IRs (bytecode, BIR)

- May seem a easy task if we build on top of the previous mechanized semantics (M6, Jinja, Bicolano)
- (Un)fortunately much research remains to be done when formalizing the semantics of concurrent Java programs...

Java Memory Model (JMM), the official specification

- Major revision in 2004 (JSR-133)
- Guarantees for programmers
  - A (safe) formal semantics for all Java programs (incl. those with races)
  - Data-race free programs execute like in a interleaving model (SC)
- Guarantees for optimizers
  - Allows all various hardware reordering semantics
  - Allows aggressive compiler optimizations



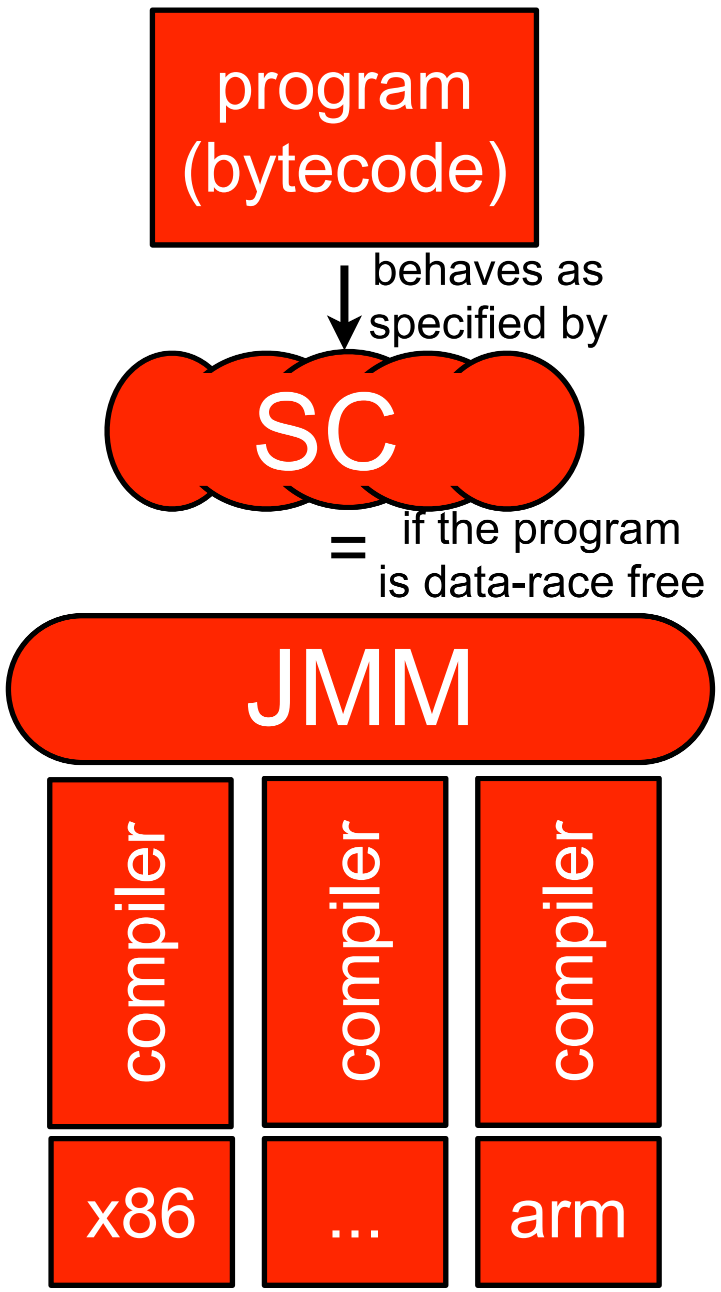
A Last Challenge: Concurrency

We need to give a semantics to the various Sawja IRs (bytecode, BIR)

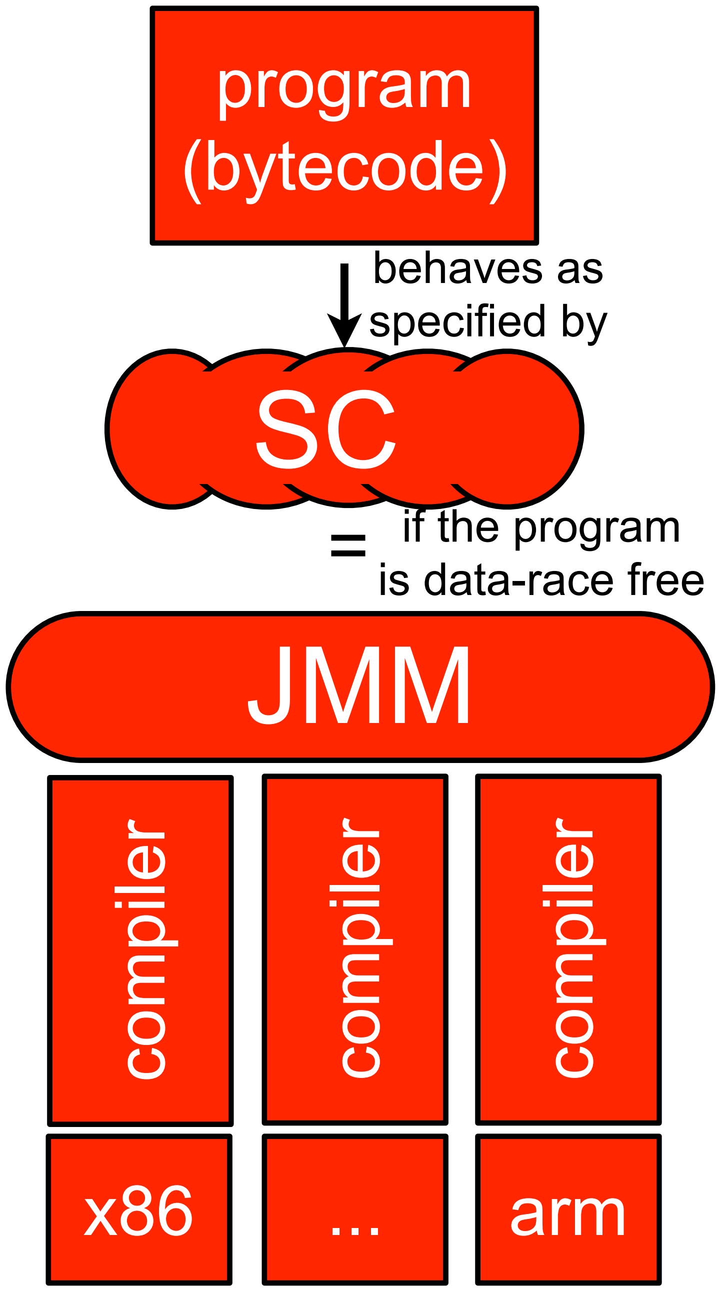
- May seem a easy task if we build on top of the previous mechanized semantics (M6, Jinja, Bicolano)
- (Un)fortunately much research remains to be done when formalizing the semantics of concurrent Java programs...

Java Memory Model (JMM), the official specification

- Major revision in 2004 (JSR-133)
- Guarantees for programmers
  - A (safe) formal semantics for all Java programs (incl. those with races)
  - Data-race free programs execute like in a interleaving model (SC)
- Guarantees for optimizers
  - Allows all various hardware reordering semantics
  - Allows aggressive compiler optimizations



Bad News

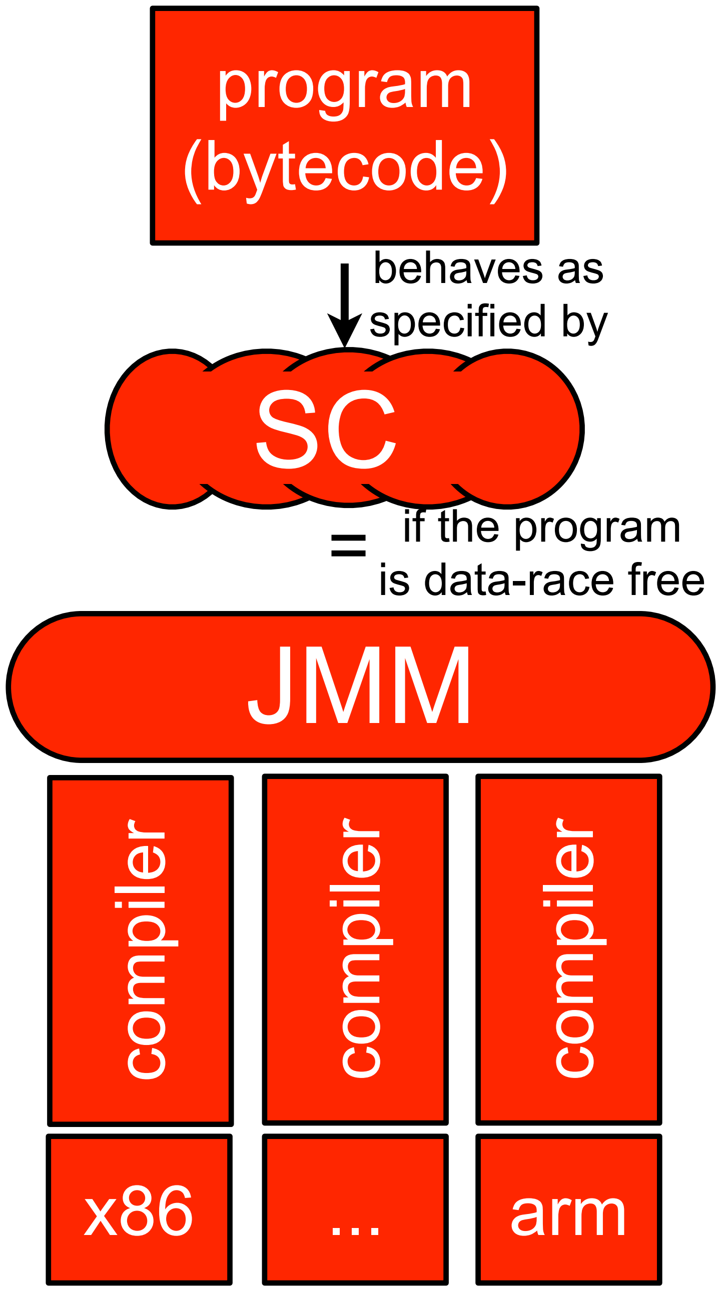


program
(bytecode)
behaves as
specified by
SC
=
if the program
is data-race free
JMM
compiler
compiler
compiler
x86
...
arm

Bad News

JMM is formally broken

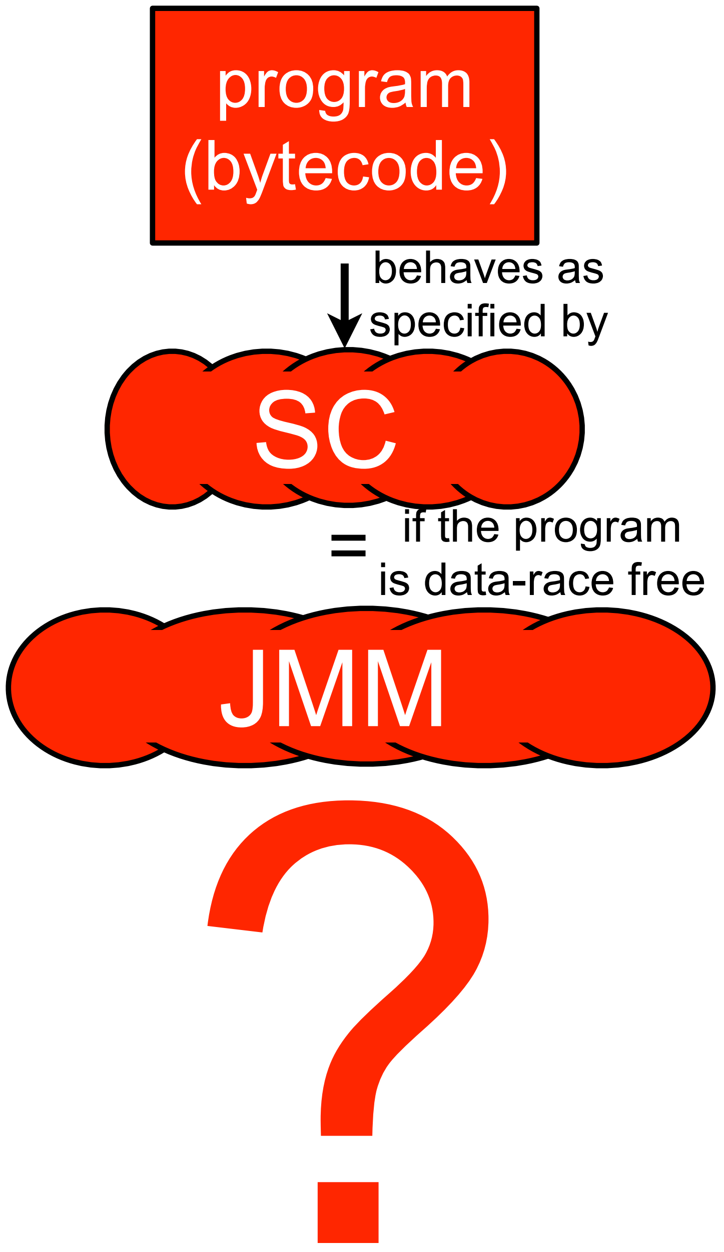
- The initial definitions and theorems were flawed
- Aspinall, Sevcik [ECOOP’08] and later Lochbihler [ESOP’12] provides formal patches in order to patch the guarantees for the programmers
- but nobody has fully patched/proved the guarantees for the optimizers



Bad News

JMM is formally broken

- The initial definitions and theorems were flawed
- Aspinall, Sevcik [ECOOP’08] and later Lochbihler [ESOP’12] provides formal patches in order to patch the guarantees for the programmers
- but nobody has fully patched/proved the guarantees for the optimizers



Bad News

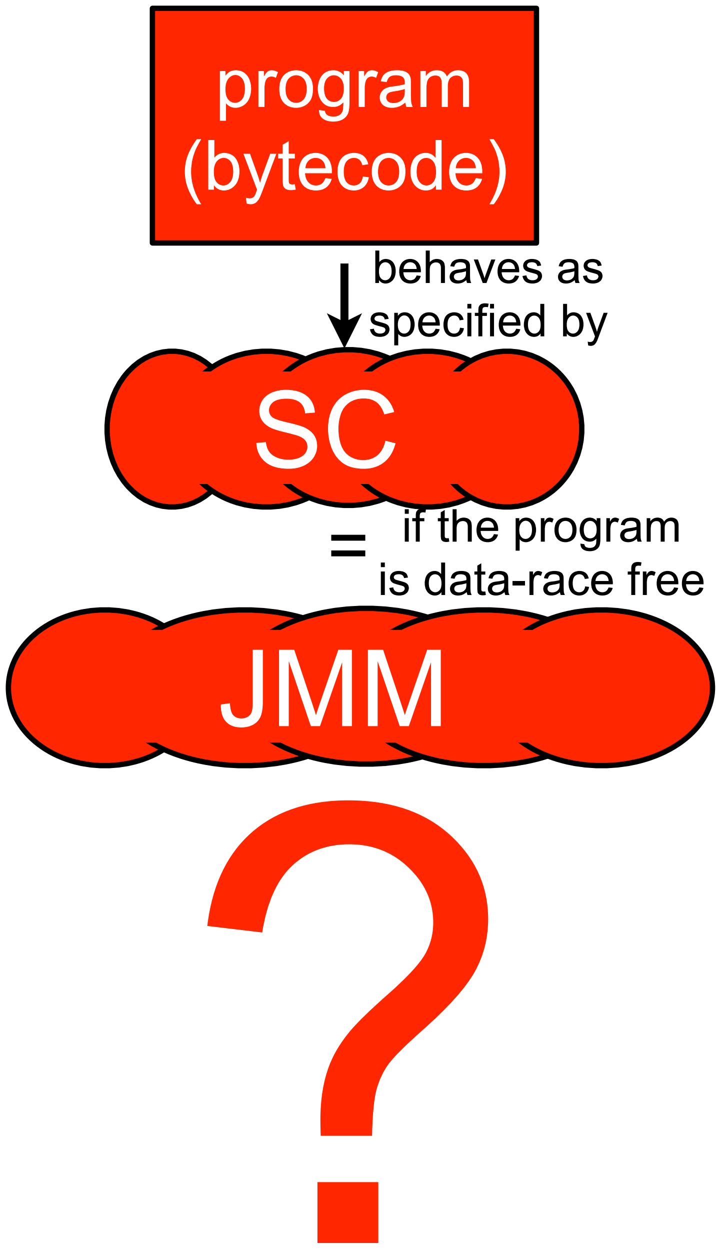
JMM is formally broken

- The initial definitions and theorems were flawed
- Aspinall, Sevcik [ECOOP'08] and later Lochbihler [ESOP'12] provides formal patches in order to patch the guarantees for the programmers
- but nobody has fully patched/proved the guarantees for the optimizers

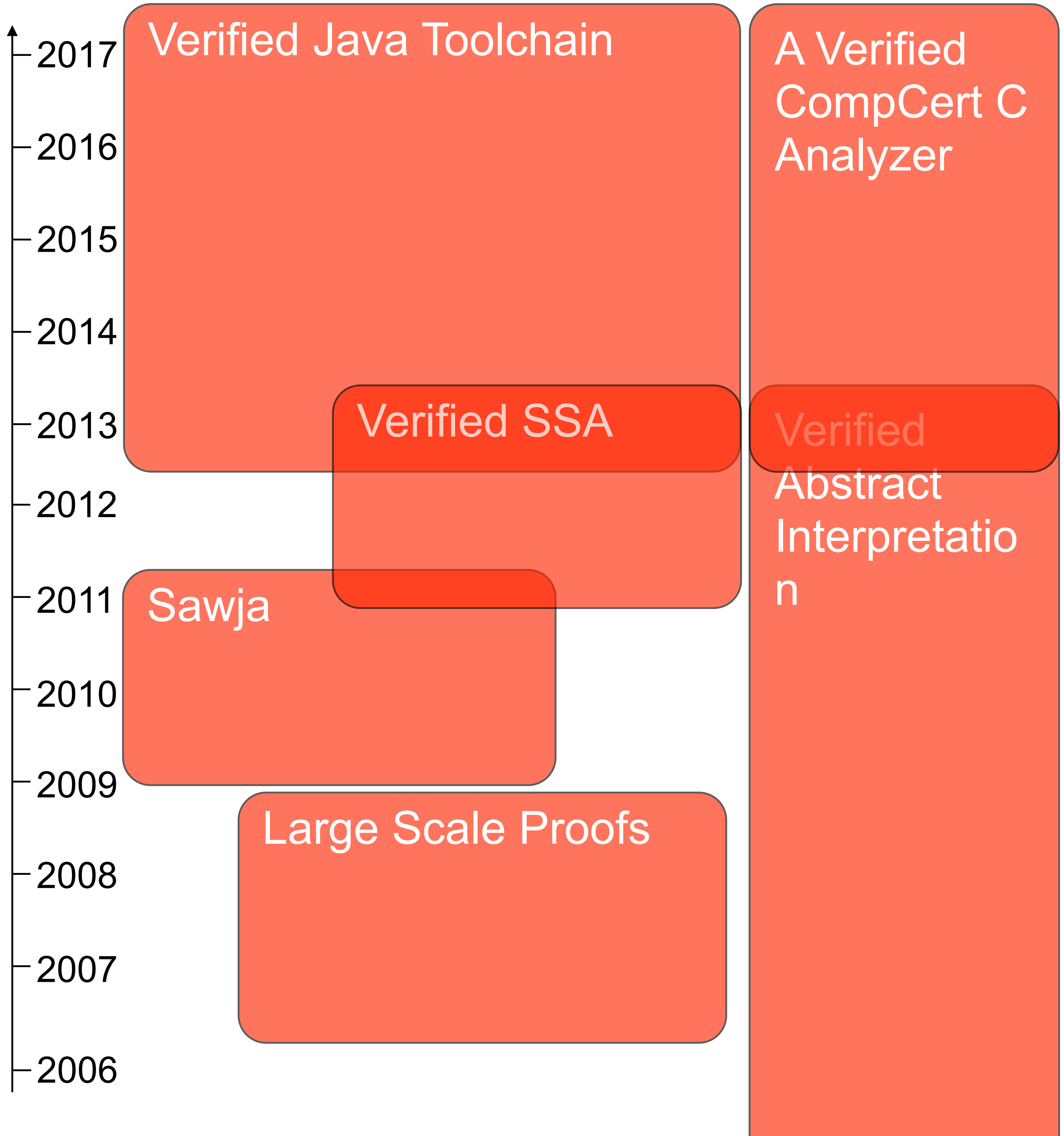
To patch the JMM properly, we need to link, in a same formal proof

- hardware semantics with weak memory models
- formalization of aggressive compiler optimizations

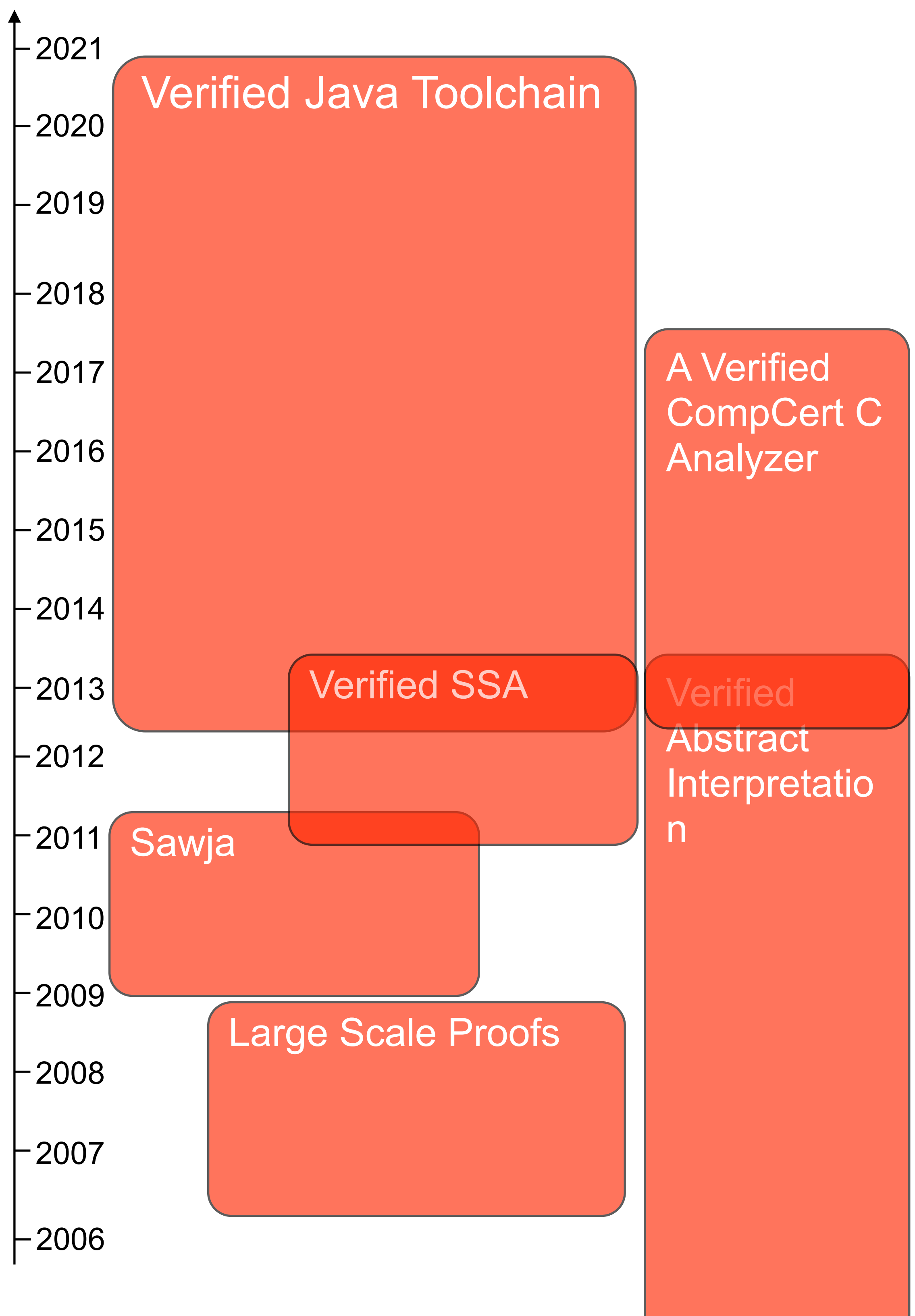
So we need to build a formally-verified Java compiler too!



Perspectives



Perspectives



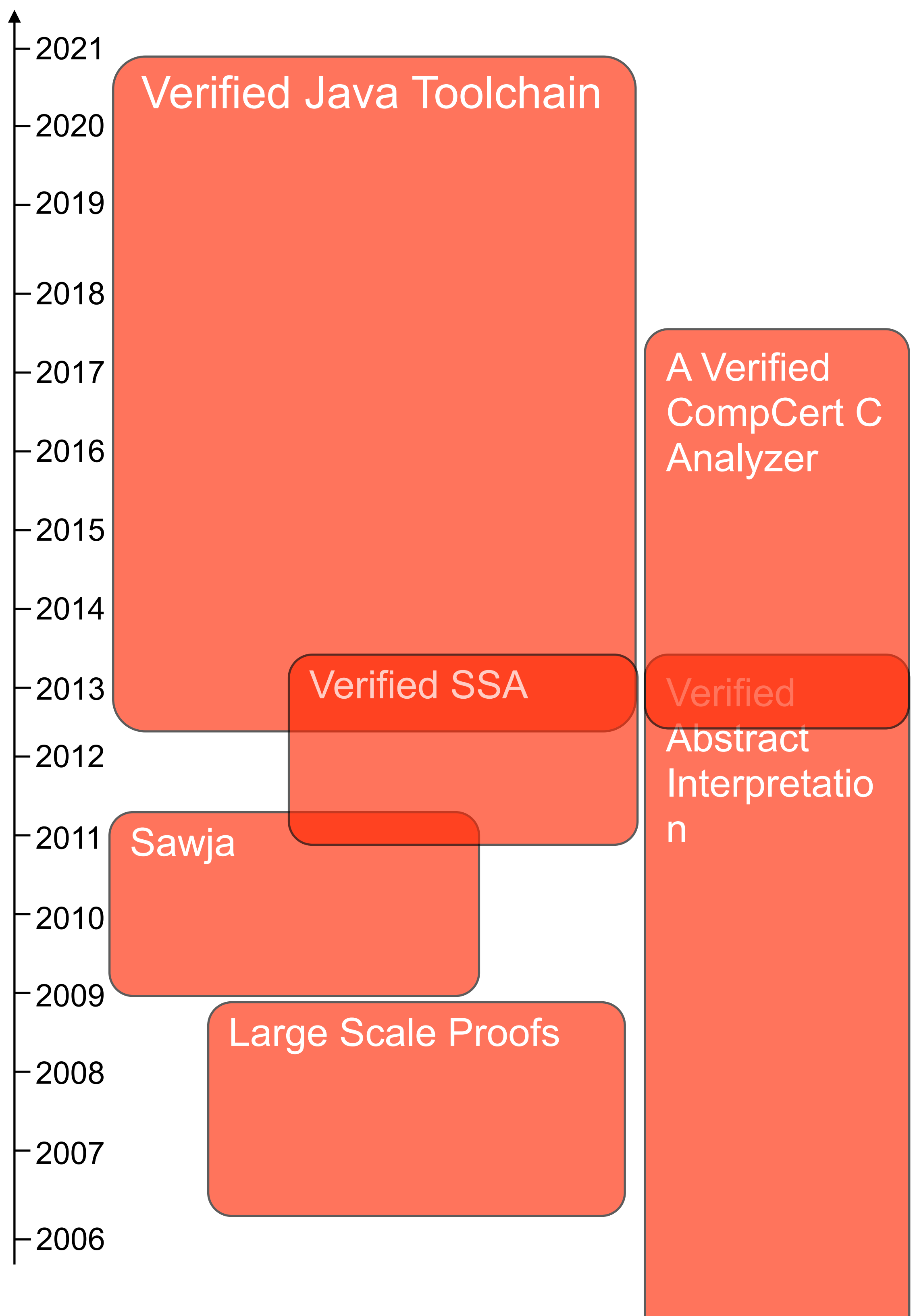
Verified Software Toolchain for Java

- a verified static analysis platform
- a formal definition of (a new) JMM
- a verified implementation of JMM

A verified Java compiler is full of challenges

- aggressive optimizations
- concurrent implementations of
  - monitors
  - data-structures
  - garbage collector
- Just In Time compiler

Perspectives



Verified Software Toolchain for Java

- a verified static analysis platform
- a formal definition of (a new) JMM
- a verified implementation of JMM

A verified Java compiler is full of challenges

- aggressive optimizations
- concurrent implementations of
  - monitors
  - data-structures
  - garbage collector
- Just In Time compiler

First step achieved during my stay at Purdue University:

- we provide an intermediate semantic step between JMM and x86