Proof Certificates for SMT-based Model Checkers

Alain Mebsout and **Cesare Tinelli**

SMT 2016

July 2nd, 2016

The University of Iowa

Motivation

- Model checkers return error traces but no evidence when they say *yes*
- Complex tools

Motivation

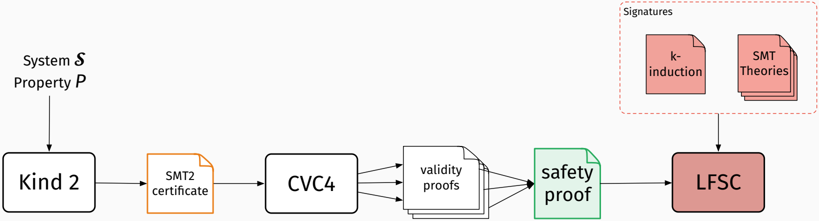
- Model checkers return error traces but no evidence when they say *yes*
- Complex tools
- **Goal:** improve trustworthiness of these tools
- **Approach:** produce proof certificates

Motivation

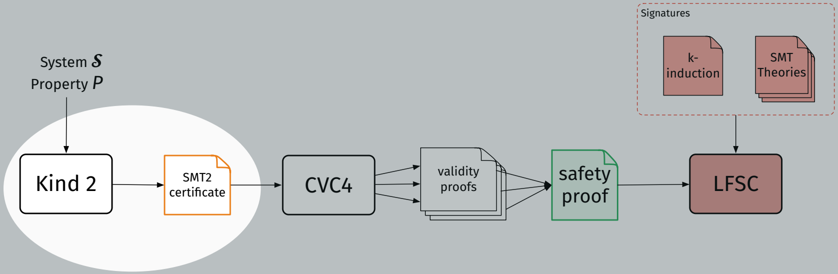
- Model checkers return error traces but no evidence when they say *yes*
- Complex tools
- **Goal:** improve trustworthiness of these tools
- **Approach:** produce proof certificates
- Implemented in Kind 2

Certificate generation and checking

Proof certificate production as a two-steps process



Intermediate certificates



Intermediate Certificates

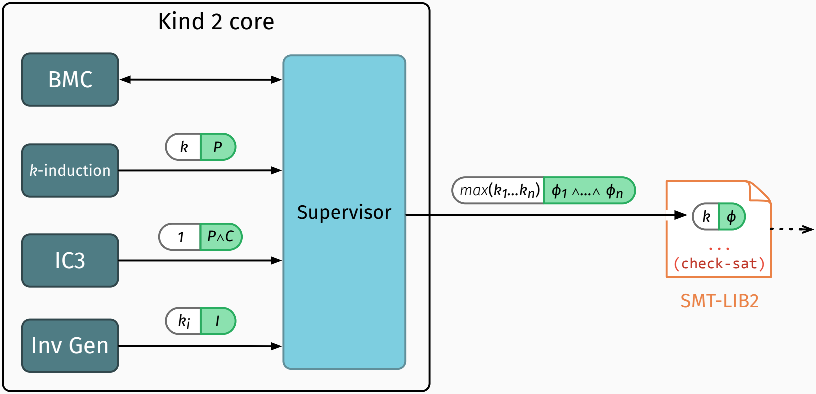
where $\boldsymbol{\phi}$ is $\boldsymbol{k}$-inductive and implies the property $\boldsymbol{P}$,

$\Rightarrow$ enough to prove that P holds in $\mathcal{S} = (\mathbf{x}, I, T)$

Intermediate Certificates

where ϕ is k-inductive and implies the property P,

$\Rightarrow$ enough to prove that P holds in $\mathcal{S} = (\mathbf{x}, I, T)$



Minimization of Intermediate (SMT-LIB 2) Certificates

Two dimensions:

- reduce k
- simplify inductive invariant
  - simplify with unsat cores
  - simplify with counter-examples to induction

Rationale: easier to check a smaller/simpler certificate

A taste of certificate minimization

(1) Trimming invariants certificate: $(1,\ \phi_1 \wedge \ldots \wedge \phi_n \wedge P)$

$$\underbrace{\phi_1 \wedge \ldots \wedge \phi_n}_{\text{invariants}} \wedge \underbrace{P}_{\text{property}} \wedge T \wedge \neg P' \models \bot$$

(1) Trimming invariants certificate: $(1,\ \phi_1 \wedge \ldots \wedge \phi_n \wedge P)$

$$\underbrace{\phi_1 \wedge \ldots \wedge \phi_n}_{\text{invariants}} \wedge \underbrace{P}_{\text{property}} \wedge T \wedge \neg P' \models \bot$$

from unsat core : $R \subseteq \{\phi_1 \wedge \ldots \wedge \phi_n\}$

(1) Trimming invariants certificate: $(1,\ \phi_1 \wedge \ldots \wedge \phi_n \wedge P)$

$$\underbrace{\phi_1 \wedge \ldots \wedge \phi_n}_{\text{invariants}} \wedge \underbrace{P}_{\text{property}} \wedge T \wedge \neg P' \models \bot$$

from unsat core : $R \subseteq \{\phi_1 \wedge \ldots \wedge \phi_n\}$

$$R \wedge P \wedge T \overset{?}{\models} R' \wedge P'$$

(1) Trimming invariants certificate: $(1, \phi_1 \wedge \ldots \wedge \phi_n \wedge P)$

$$\underbrace{\phi_1 \wedge \ldots \wedge \phi_n}_{\text{invariants}} \wedge \underbrace{P}_{\text{property}} \wedge T \wedge \neg P' \models \bot$$

from unsat core : $R \subseteq \{\phi_1 \wedge \ldots \wedge \phi_n\}$

$$R \wedge P \wedge T \overset{?}{\models} R' \wedge P'$$

- **yes**: keep R
- **no**: restart with $P := R \wedge P$

(2) Cherry-picking invariants certificate: $(1, \overbrace{\phi_1 \land \ldots \land \phi_n}^{R} \land P)$

$$P \land T \not\models P'$$

A taste of certificate minimization (cont.)

(2) Cherry-picking invariants certificate: $(1, \overbrace{\phi_1 \wedge \ldots \wedge \phi_n}^{R} \wedge P)$

$$P \wedge T \not\models P'$$

from model $\mathcal{M} : \phi \in R$ such that $\mathcal{M} \not\models \phi$

A taste of certificate minimization (cont.)

(2) Cherry-picking invariants certificate: $(1, \overbrace{\phi_1 \wedge \ldots \wedge \phi_n}^{R} \wedge P)$

$$P \wedge T \not\models P'$$

from model $\mathcal{M} : \phi \in R$ such that $\mathcal{M} \not\models \phi$

$$P := \phi \wedge P \qquad R := R \setminus \{\phi\}$$

Front End Certificates

Front end certificates in Kind 2

Translation from one formalism to another are sources of error

In Kind 2,

- several intermediate representations
- many simplifications (slicing, path compression, encodings, ...)

Front end certificates in Kind 2

Translation from one formalism to another are sources of error

In Kind 2,

- several intermediate representations
- many simplifications (slicing, path compression, encodings, ...)

How to trust the translation from input language to internal FOL representation ?

Front end certificates in Kind 2

Translation from one formalism to another are sources of error

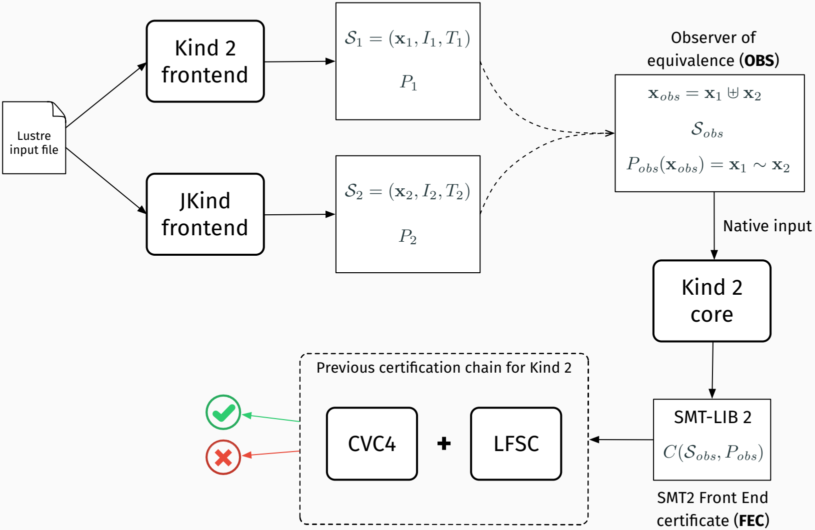
In Kind 2,

- several intermediate representations
- many simplifications (slicing, path compression, encodings, ...)

How to trust the translation from input language to internal FOL representation ?

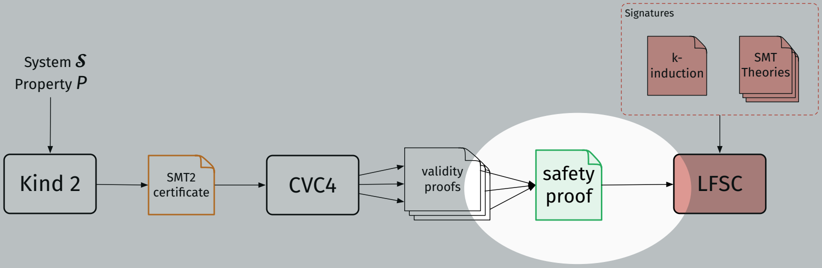
Lightweight verification akin to **Multiple-Version Dissimilar Software Verification** of DO-178C (12.3.2)

Front end certificates in Kind 2: approach



LFSC Proofs

Producing proofs



Producing proofs of invariance

$\mathcal{S} = (\mathbf{s}, I[\mathbf{s}], T[\mathbf{s}, \mathbf{s}'])$:	input system
$P[\mathbf{s}]$:	property proven invariant for $\mathcal{S}$
$(k, \phi[\mathbf{s}])$:	certificate produced by Kind 2

- We can formally check that ϕ
  1. is k-inductive
  2. implies P

- **Our goal:** produce a detailed, self-contained and independently machine-checkable proof

Proving invariance by k-induction

$$\mathcal{S} = (\mathbf{s}, I[\mathbf{s}], T[\mathbf{s}, \mathbf{s}']) : \quad \text{input system}$$

$$P[\mathbf{s}] : \quad \text{property proven invariant for } \mathcal{S}$$

$$(k, \phi[\mathbf{s}]) : \quad \text{certificate produced by Kind 2}$$

ϕ is a k-inductive strengthening of P:

$$I[\mathbf{s}_0] \;\wedge\; T[\mathbf{s}_0, \mathbf{s}_1] \;\wedge\; \ldots \;\wedge\; T[\mathbf{s}_{k-2}, \mathbf{s}_{k-1}] \;\models\; \phi[\mathbf{s}_0] \wedge \ldots \wedge \phi[\mathbf{s}_{k-1}] \qquad (base_k)$$

$$\phi[\mathbf{s}_0] \wedge T[\mathbf{s}_0, \mathbf{s}_1] \;\wedge\; \ldots \;\wedge\; \phi[\mathbf{s}_{k-1}] \wedge T[\mathbf{s}_{k-1}, \mathbf{s}_k] \;\models\; \phi[\mathbf{s}_k] \qquad (step_k)$$

$$\phi[\mathbf{s}] \;\models\; P[\mathbf{s}] \qquad (implication)$$

Proving invariance by k-induction

$$\mathcal{S} = (\mathbf{s}, I[\mathbf{s}], T[\mathbf{s}, \mathbf{s}']) : \quad \text{input system}$$

$$P[\mathbf{s}] : \quad \text{property proven invariant for } \mathcal{S}$$

$$(k, \phi[\mathbf{s}]) : \quad \text{certificate produced by Kind 2}$$

ϕ is a k-inductive strengthening of P:

$$I[\mathbf{s}_0] \;\wedge\; T[\mathbf{s}_0, \mathbf{s}_1] \;\wedge\; \ldots \;\wedge\; T[\mathbf{s}_{k-2}, \mathbf{s}_{k-1}] \;\models\; \phi[\mathbf{s}_0] \wedge \ldots \wedge \phi[\mathbf{s}_{k-1}] \qquad (base_k)$$

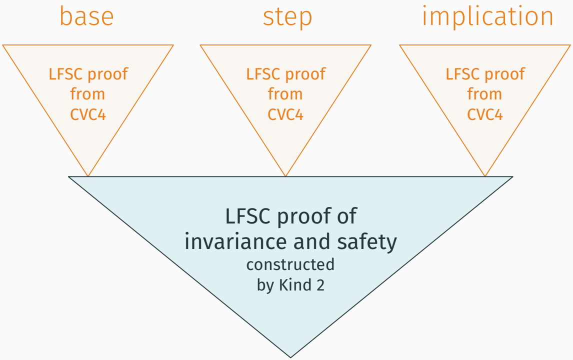
$$\phi[\mathbf{s}_0] \wedge T[\mathbf{s}_0, \mathbf{s}_1] \;\wedge\; \ldots \;\wedge\; \phi[\mathbf{s}_{k-1}] \wedge T[\mathbf{s}_{k-1}, \mathbf{s}_k] \;\models\; \phi[\mathbf{s}_k] \qquad (step_k)$$

$$\phi[\mathbf{s}] \;\models\; P[\mathbf{s}] \qquad (implication)$$

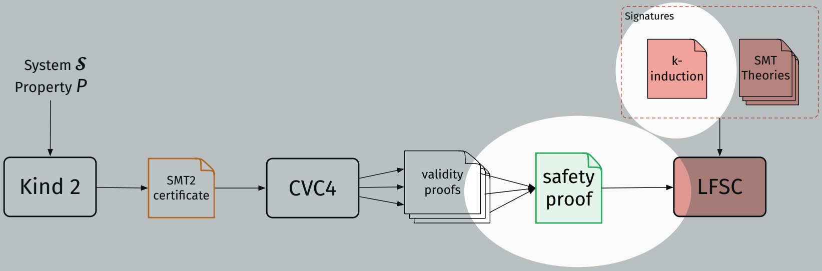
Approach

Use CVC4 to generate proofs for the validity of each sub-case

Kind 2 generates a proof of invariance by k-induction and reuses the proofs of CVC4



LFSC rules


Signatures
k-induction
SMT Theories
System $\mathcal{S}$
Property P
Kind 2
SMT2 certificate
CVC4
validity proofs
safety proof
LFSC

LFSC encodings

Encoding of Lustre variables as functions over naturals (indexes)

In Lustre

```
node main (a: bool) returns (OK: bool)
var b: bool;
...
```

In the LFSC signature:

```
(declare index sort)
(declare ind int → index)
```

In the LFSC proof:

```
(declare a (term (arrow index Bool)))
(declare b (term (arrow index Bool)))
(declare OK (term (arrow index Bool)))
...
```

LFSC encodings (cont.)

Predicates and relations over copies of the same state
$\rightsquigarrow$ predicates/relations over indexes

- $P(\mathbf{s}_i) \quad \rightsquigarrow \quad P_{\mathbf{s}}(i)$
- $R(\mathbf{s}_i, \mathbf{s}_j) \quad \rightsquigarrow \quad R_{\mathbf{s}}(i, j)$

Predicates and relations over copies of the same state
$\rightsquigarrow$ predicates/relations over indexes

- $P(\mathbf{s}_i) \rightsquigarrow P_{\mathbf{s}}(i)$
- $R(\mathbf{s}_i, \mathbf{s}_j) \rightsquigarrow R_{\mathbf{s}}(i, j)$

In the LFSC signature:

```
;; relations over indexes (used for transition relation)
(define rel int → int → formula)

;; sets over indexes (used for initial formula and properties)
(define set int → formula)

;; derivability judgment for invariance proofs
(declare invariant set → rel → set → type)
```

LFSC encodings (cont.)

Predicates and relations over copies of the same state
$\rightsquigarrow$ predicates/relations over indexes

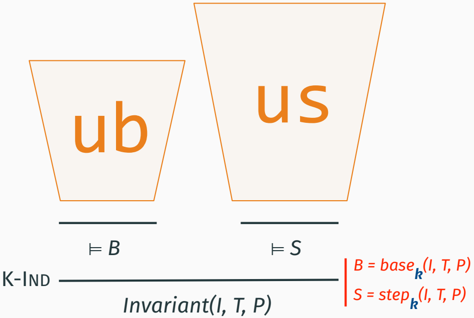
- $P(\mathbf{s}_i) \quad \rightsquigarrow \quad P_{\mathbf{s}}(i)$
- $R(\mathbf{s}_i, \mathbf{s}_j) \quad \rightsquigarrow \quad R_{\mathbf{s}}(i, j)$

In the LFSC proof:

```
;; encoding of property
(define P : set
 (λ i. (p_app (apply _ _ OK (int i)))))

;; encoding of transition relation
(define T : rel
 (λ i. λ j. ...))
```

LFSC rules – *k*-induction

```
(declare k-ind
  Π k: int. ; bound k
  Π I: set. ; initial states
  Π T: rel. ; transition relation
  Π P: set. ; k-inductive invariant

  ; formula for base case
  Π r1: B = (base I T P k).

  ; formula for step case
  Π r2: S = (step T P k).

  ; proof of base case
  Π ub : (th_holds B).

  ; proof of step case
  Π us : (th_holds S).

  ;--------------------------------
  invariant I T P
)
```



LFSC rules – implication

```
(declare inv-impl
  Π I: set.  Π T: rel.
  Π P1: set.  Π P2: set.

  ;; proof that P1 => P2
  Π u :
     Π k: int.
     th_holds ((P1 k) ⇒ (P2 k)).

  ;; proof that P1 is invariant
  Π i :
     invariant I T P1.

  ;----------------------------
  invariant I T P2
)
```

$$\textsc{Inv-Impl}\ \dfrac{\dfrac{}{\models P1 \Rightarrow P2} \qquad \dfrac{}{\mathit{Invariant}(I, T, P1)}}{\mathit{Invariant}(I, T, P2)}$$

LFSC proofs: Example

Small Lustre node: detection of rising edge:

```
node edge (x: bool) returns (y: bool);
var OK: bool;
let
  y = false -> x and not pre x;
  OK = not x => not y;
  --%PROPERTY OK;
tel
```

LFSC proof for rising edge node

```
;;------------------------------------------------------------------
;; LFSC proof produced by kind2 v0.8.0-425-g294ec4d and CVC4
;; from original problem ex.lus
;;------------------------------------------------------------------

;; Declarations and definitions
(declare edge.usr.x (term (arrow index Bool)))
(declare edge.usr.y (term (arrow index Bool)))
(declare edge.res.init_flag (term (arrow index Bool)))
(declare edge.impl.usr.OK (term (arrow index Bool)))

(define I (: (! _ int formula)
  (\ I%1 (@ let3 (ind I%1) (@ let4 (p_app (apply _ _ edge.usr.y (ind I%1))) (and (iff let4 false)
  (and (iff (p_app (apply _ _ edge.impl.usr.OK (ind I%1))) (impl (not (p_app (apply _ _ edge.usr.x (ind I%1)))) (not let4)))
  (and (p_app (apply _ _ edge.res.init_flag (ind I%1))) true))))))
))

(define T (: (! _ int (! _ int formula))
  (\ T%1 (\ T%2 (@ let22 (ind T%2) (@ let23 (p_app (apply _ _ edge.usr.y (ind T%2))) (@ let24 (p_app (apply _ _ edge.usr.x (ind T%2)))
  (and (iff let23 (and let24 (not (p_app (apply _ _ edge.usr.x (ind T%1)))))) (and (iff (p_app (apply _ _ edge.impl.usr.OK (ind T%2)))
  (impl (not let24) (not let23))) (and (not (p_app (apply _ _ edge.res.init_flag (ind T%2)))) true))))))))
))

(define P (: (! _ int formula) (\ P%1 (p_app (apply _ _ edge.impl.usr.OK (ind P%1))))))

(define PHI (: (! _ int formula) (\ PHI%1 (p_app (apply _ _ edge.impl.usr.OK (ind PHI%1))))))
```

LFSC proof for rising edge node (cont.)

```
(define base
  (: (! A0 (th_holds (@ let1 (ind 0) (@ let2 (p_app (apply _ _ edge.usr.y (ind 0))) (@ let5 (p_app (apply _ _ edge.impl.usr.OK (ind 0))) (and
  (and (iff let2 false) (and (iff let5 (impl (not (p_app (apply _ _ edge.usr.x (ind 0)))) (not let2))) (and (p_app (apply _ _
  edge.res.init_flag (ind 0))) true))) (not let5)))))) (holds cln)) (\ A0 (th_let_pf _ (trust_f false) (\ .PA193 (th_let_pf _ (trust_f (not
  false)) (\ .PA197 (decl_atom false (\ .v1 (\ .a1 (satlem _ _ (ast _ _ _ .a1 (\ .l3 (clausify_false (contra _ .l3 .PA197)))) (\ .pb3 (satlem
  _ _ (asf _ _ _ .a1 (\ .l2 (clausify_false (contra _ .PA193 .l2)))) (\ .pb4 (satlem_simplify _ _ _ (R _ _ .pb4 .pb3 .v1) (\empty
  empty)))))))))))))))
)

(define induction
  (: (! A0 (th_holds (@ let1 (ind 0) (@ let3 (ind 1) (@ let4 (p_app (apply _ _ edge.usr.y (ind 1))) (@ let5 (p_app (apply _ _ edge.usr.x (ind
  1))) (@ let10 (p_app (apply _ _ edge.impl.usr.OK (ind 1))) (and (and (p_app (apply _ _ edge.impl.usr.OK (ind 0))) (and (iff let4 (and let5
  (not (p_app (apply _ _ edge.usr.x (ind 0)))))) (and (iff let10 (impl (not let5) (not let4))) (and (not (p_app (apply _ _ edge.res.init_flag
  (ind 1)))) true)))) (not let10)))))))) (holds cln)) (\ A0 (th_let_pf _ (trust_f false) (\ .PA193 (th_let_pf _ (trust_f (not false)) (\
  .PA197 (decl_atom false (\ .v1 (\ .a1 (satlem _ _ (ast _ _ _ .a1 (\ .l3 (clausify_false (contra _ .l3 .PA197)))) (\ .pb3 (satlem _ _ (asf _
  _ _ .a1 (\ .l2 (clausify_false (contra _ .PA193 .l2)))) (\ .pb4 (satlem_simplify _ _ _ (R _ _ .pb4 .pb3 .v1) (\empty empty))))))))))))))
)

(define implication
  (: (! %%k int (! A0 (th_holds (@ let2 (p_app (apply _ _ edge.impl.usr.OK (ind %%k))) (not (impl let2 let2)))) (holds cln))) (\ %%k (\ A0
  (th_let_pf _ (trust_f false) (\ .PA193 (th_let_pf _ (trust_f (not false)) (\ .PA197 (decl_atom false (\ .v1 (\ .a1 (satlem _ _ (ast _ _ _
  .a1 (\ .l3 (clausify_false (contra _ .l3 .PA197)))) (\ .pb3 (satlem _ _ (asf _ _ _ .a1 (\ .l2 (clausify_false (contra _ .PA193 .l2)))) (\
  .pb4 (satlem_simplify _ _ _ (R _ _ .pb4 .pb3 .v1) (\empty empty)))))))))))))))
)

;; Proof of invariance by 1-induction
(define proof_inv
 (: (invariant I T P)
  (inv-impl I T PHI P implication
  (k-ind 1 I T PHI _ _ base induction))))

(check proof_inv)
```

LFSC proof for rising edge node (cont.)

```
;;----------------------------------------------------------------------
;; LFSC proof produced by kind2 v1.0.alpha1-208-gae70098 and
;; CVC4 version 1.5-prerelease [git proofs 7ba546df]
;; for frontend observational equivalence and safety
;; (depends on proof.lfsc)
;;----------------------------------------------------------------------

;; System generated by JKind
(declare JKind.$x$ (term (arrow index Bool)))
(declare JKind.$y$ (term (arrow index Bool)))
(declare f1 (term (arrow index Bool)))
(declare JKind.$OK$ (term (arrow index Bool)))

(define I2 (: (! _ int formula) ...))
(define T2 (: (! _ int (! _ int formula)) ...))
(define P2 (: (! _ int formula) ...))

;; System generated for Observer
(define same_inputs (: (! _ int formula)
 (\ same_inputs%1 (@ let73 (ind same_inputs%1)
   (iff (p_app (apply _ _ edge.usr.x let73))
        (p_app (apply _ _ JKind.$x$ let73)))))))

(define IO (: (! _ int formula) ...))
(define TO (: (! _ int (! _ int formula)) ...))
(define PO (: (! _ int formula) ...))
```

LFSC proof for rising edge node (cont.)

```
;; k-Inductive invariant for observer system
(define PHIO (: (! _ int formula) ...))

;; Proof of base case
(define base_proof_2 ...)

;; Proof of inductive case
(define induction_proof_2 ...)

;; Proof of implication
(define implication_proof_2 ...)

;; Proof of invariance by 1-induction
(define proof_obs (: (invariant IO TO PO)
  (inv-impl IO TO PHIO PO implication_proof_2
  (k-ind 1 IO TO PHIO _ _ base_proof_2 induction_proof_2))))

;; Proof of observational equivalence
(define proof_obs_eq
 (: (weak_obs_eq I T P I2 T2 P2)
  (obs_eq I T P I2 T2 P2 same_inputs proof_obs)))

;; Final proof of safety
(define proof_safe
 (: (safe I T P) (inv+obs I T P I2 T2 P2 proof_inv proof_obs_eq)))

(check proof_safe)
```

Checking the proof

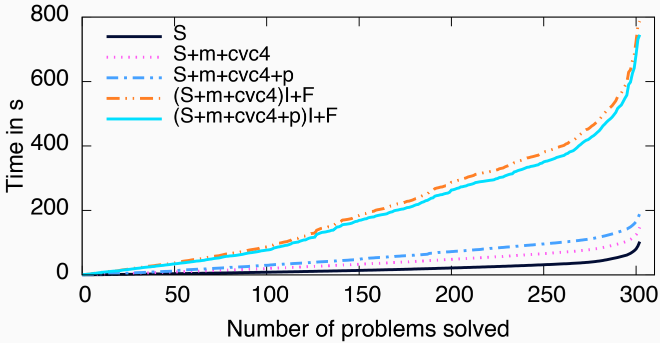
Proof checker

- proof checker generator: `lfsc-checker`
- proof rules: `sat.plf smt.plf th_base.plf th_int.plf th_real.plf kind.plf`
- proof: `proof.lfsc`

```
> lfsc-checker sat.plf smt.plf th_base.plf th_int.plf th_real.plf kind.plf proof.lfsc
```

- `sat.plf`: signature for SAT solving (resolution)
- `smt.plf`: signature for SMT (cnf + theory)
- `th_base.plf`: signature for EUF theory
- `th_int.plf`: symbols fot linear integer arithmetic
- `th_real.plf`: symbols fot linear real arithmetic
- `kind.plf`: signature for *k*-induction

Evaluation



- proved invariance (of encoded system) for 80%

  (rest is unsupported fragment of proofs for CVC4)

Trust base

The trusted core of our approach consists in:

1. LFSC checker (5300 lines of C++ code)

2. LFSC signatures comprising the overall proof system LFSC (for a total of 444 lines of LFSC code)

3. Assumption that Kind 2 and JKind do not have identical defects that could escape the observational equivalence check. (reasonable considering the differences between the two model checkers)

Current limitations

- Holes in proofs produced by CVC4 (`trust_f` rule):
    - pre-processing
    - arithmetic lemmas
- Doesn’t work with combination of both real and integer arithmetic for now

Conclusion

- Kind 2 generates machine checkable proofs of invariance and safety in LFSC
- Currently limited by CVC4 capabilities for proofs ...
- ... but ready for when CVC4 will produce proofs for more theories

Ongoing and future work

- Leverage proofs for tool qualification — DO-178C, DO-330 (ongoing, collaboration with Rockwell Collins and NASA)
- Tests for checker and side-conditions
- Prove correctness of rules and side-conditions in a proof assistant like Coq or Isabelle

Thank you