ISSUES, CHALLENGES AND OPPORTUNITIES IN THE QUALIFICATION OF FORMAL METHOD TOOLS

Proof-producing SMT Solvers and Model Checkers

Cesare Tinelli

May 6, 2015

The University of Iowa

ACKNOWLEDGMENTS

Main Collaborators

Tianyi Liang, Alain Mebsout, Andrew Reynolds, Aaron Stump (**U. Iowa**)

Clark Barrett, Morgan Deters, Tim King, Liana Hadarean (**NYU**)

Darren Cofer, Lucar Wagner (**Rockwell-Collins**)

Chantal Keller (**Inria**)

Funding

DARPA HACMS

NASA SSAT-AFCS

Formal Methods are increasingly used to produce high-assurance software

FM tools are themselves complex and so error-prone

Agencies such as the FAA require them to be qualified (DO-330, DO-333)

FM tools often rely on external, third-party FM tools

RESEARCH QUESTIONS

1. How to reduce the burden of FM tool qualification?
2. How to incorporate untrusted FM tools in highly trusted ones?

RESEARCH QUESTIONS

1. How to reduce the burden of FM tool qualification?
2. How to incorporate untrusted FM tools in highly trusted ones?

A possible answer: Proof-producing FM tools

TRADITIONAL QUALIFICATION: AN EXAMPLE

ALT-ERGO

SMT Solver

Université Paris-Sud, INRIA, CNRS, OCamlPro

- Project leaders: Sylvain Conchon, Évelyne Contejean
- Current developer: Mohamed Iguernlala
- Contributors: Stéphane Lescuyer, Alain Mebsout

(Almost) Purely functional, written in OCaml

Fairly small: $\sim$ 10kloc

Implementation close to formal description

Correctness proof of core in Coq

ALT-ERGO: FEATURES

Typed polymorphic first-order logic

Theories:

- equality over uninterpreted symbols (EUF)
- linear arithmetic (on $\mathbb{Q}$ LRA, and $\mathbb{Z}$ LIA)
- non-linear arithmetic (on $\mathbb{Q}$ NRA, and $\mathbb{Z}$ NIA)
- Records (and pairs)
- Bit Vectors (concat and extract)
- Associative/Commutative symbols (AC)
- Functional arrays
- Enumerated datatypes
- Quantifiers ($\forall$, $\exists$)
- ...

Context:

- Airbus, for use on A350
- Verification of C code (pre-flight inspection) with Frama-C

What was qualified:

- Version 0.94
- Propositional engine
- Equality modulo AC engine
- Polymorphic types component
- Quantifiers compoment
- Arithmetic sub-solver (real and integer)

ALT-ERGO: QUALIFICATION PACKAGE (INRIA)

Rigorous description of qualified modules (65 pp. French text + inference rules)

Each section gives precise functional tool requirements (TR)

Version of Alt-Ergo with traces for TRs

Disable all unqualified features

$\sim$ 600 tests for the TRs (correctness and coverage)

Challenges

- Non-trivial process
- A few (2-3) man months
- Identifying inference rules and relevant TRs to cover whole tool
- Needs to be redone for any future versions

Collateral Benefits

- Allowed to completely review (and sometimes improve) code

Now a fundamental component of several FM techniques and tools

Implement complex and sophisticated algorithms

The major ones are rather large
(100-250K lines of optimized C/C++ code)

Bugs are still discovered in mature solvers

Traditional qualification process is infeasible

High-assurance of other tools (e.g., Isabelle, Coq) is compromised when they use them

HOW TO TRUST SMT SOLVER ANSWERS?

Require an externally-checkable certificate of correctness:

sat answer	$\longrightarrow$	satisfying assignment
unsat answer	$\longrightarrow$	proof of unsatisfiability

Require an externally-checkable certificate of correctness:

sat answer	$\longrightarrow$	satisfying assignment
unsat answer	$\longrightarrow$	proof of unsatisfiability

Formal certificates

- increase trust in the results produced by the solver
- shift focus of trust to a much simpler certificate checker
- can supplement traditional qualification processes
- partially address objectives in DO-330 (6.1.3.1.i, ...)

PROOF-PRODUCING SMT SOLVERS

cvc4

Major SMT solver

- Project leaders: Clark Barrett (NYU), Cesare Tinelli (Iowa)
- Main developers: Kshitij Bansal, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Tianyi Liang, Andrew Reynolds

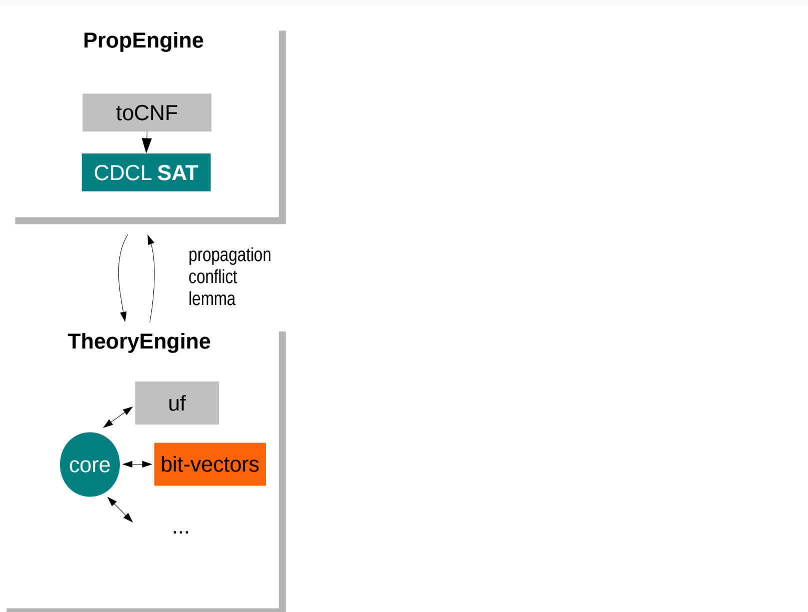
Rich set of theories and functionality

Started large instrumentation effort to make it proof-producing

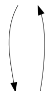
Effort underway, with many components now proof-producing

Proofs in LFSC format

CVC4 PROOF INFRASTRUCTURE

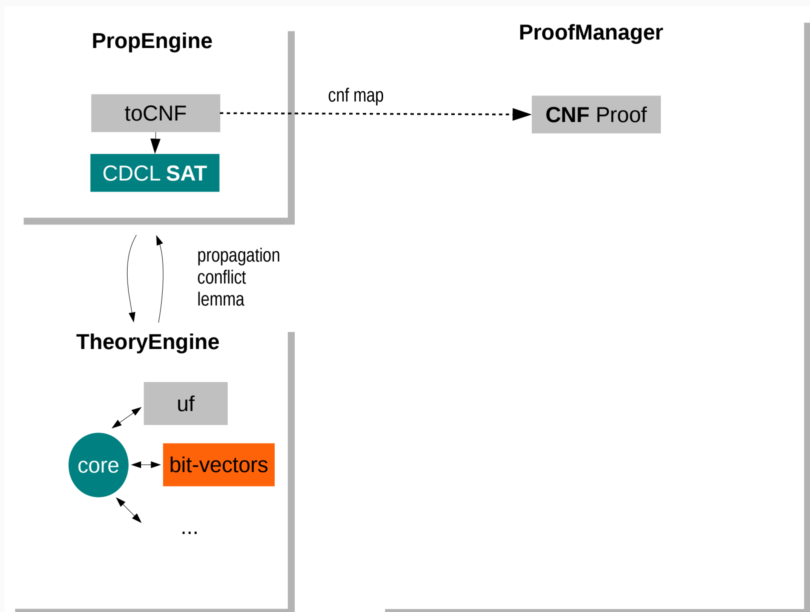


PropEngine
toCNF
CDCL SAT

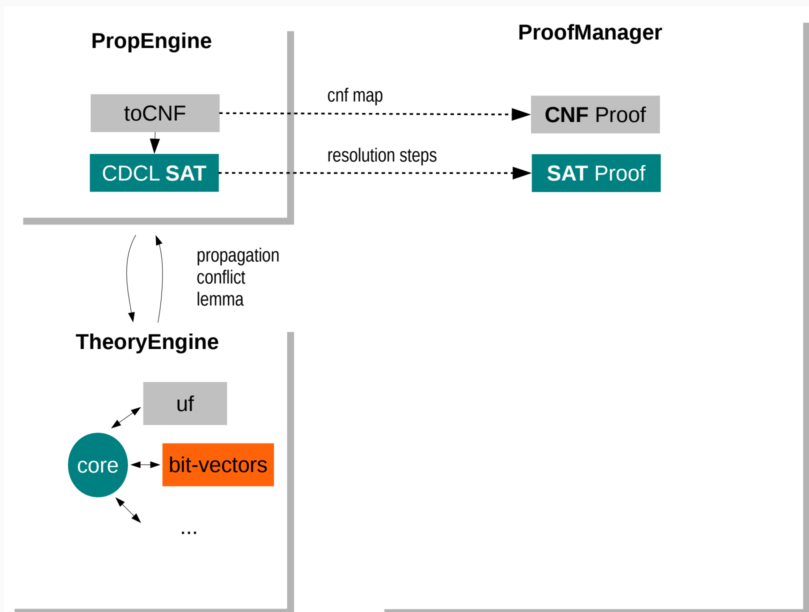


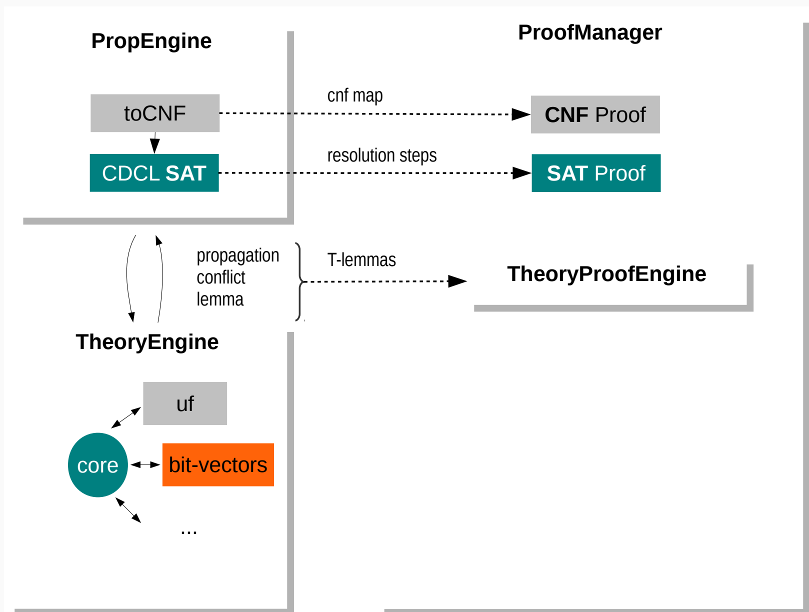
propagation
conflict
lemma
TheoryEngine
uf
core
bit-vectors
...
ProofManager

CVC4 PROOF INFRASTRUCTURE

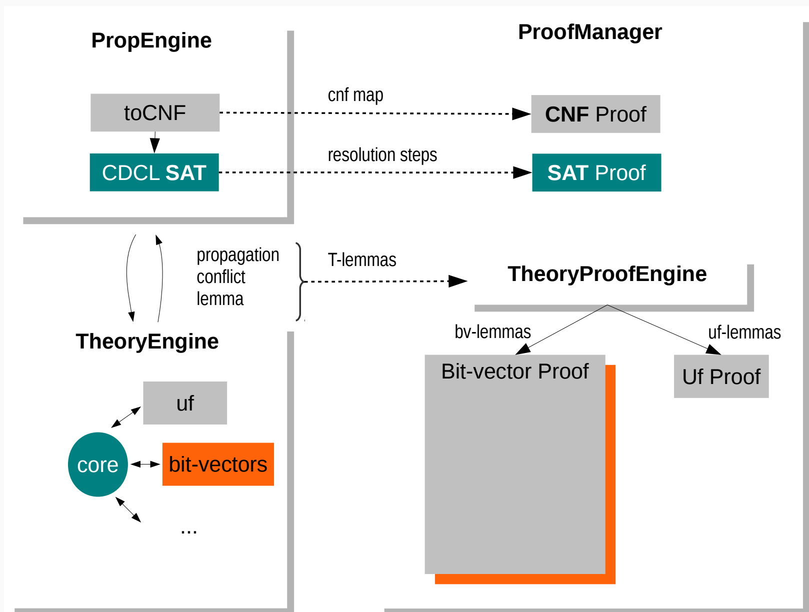


CVC4 PROOF INFRASTRUCTURE

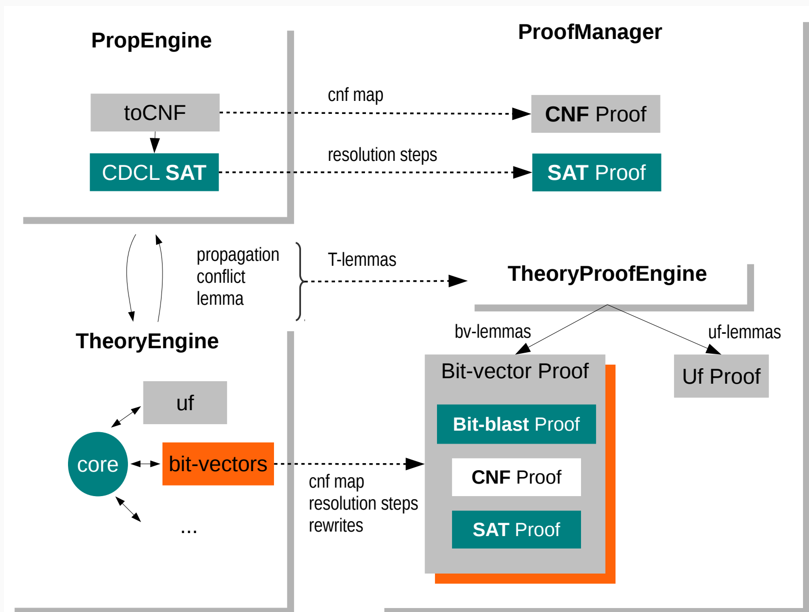



PropEngine
ProofManager
toCNF
cnf map
CNF Proof
CDCL SAT
resolution steps
SAT Proof
propagation
conflict
lemma
T-lemmas
TheoryProofEngine
TheoryEngine
uf
core
bit-vectors
...

CVC4 PROOF INFRASTRUCTURE



CVC4 PROOF INFRASTRUCTURE



LFSC CHECKER

Proof checker generator

- Project leaders: Aaron Stump (Iowa), Cesare Tinelli (Iowa)
- Main developers: Andrew Reynolds

Based on Edinburgh Logical Framework (LF)

Geared toward SMT solvers

Takes both a proof system and a proof object in it

Efficient (side conditions compiled)

Small (3.5kloc C++) and fast

Extension of LF with computational side-conditions

Support for dependent types

e.g. $\Pi x{:}\mathrm{Int}.\ \mathrm{BV}[x]$

Proof rules are typing declarations

Extension of LF with computational side-conditions

Support for dependent types

e.g. Πx:Int. BV[x]

Proof rules are typing declarations

$$\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3} \text{ eq_trans}$$

Extension of LF with computational side-conditions

Support for dependent types

e.g. $\Pi x{:}\mathrm{Int}.\ \mathrm{BV}[\mathrm{x}]$

Proof rules are typing declarations

$$\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}\ \text{eq_trans} \qquad \begin{array}{l} \Pi t_1{:}\mathrm{term}.\ t_2{:}\mathrm{term}.\ t_3{:}\mathrm{term}. \\ \quad \Pi u_1{:}\mathrm{holds}[t_1 = t_2]. \\ \quad\quad \Pi u_2{:}\mathrm{holds}[t_2 = t_3].\ \mathrm{holds}[t_1 = t_3] \end{array}$$

LFSC SYNTAX EXAMPLE: TYPES AND FUNCTION SYMBOLS

$\mathrm{Term} : \mathrm{Sort} \rightarrow \mathrm{Type}$

$\mathrm{BV} : \mathrm{Int} \rightarrow \mathrm{Type}$

$\mathrm{and} : \Pi n{:}\mathrm{Int}.\ \mathrm{Term}[\mathrm{BV}[n]] \rightarrow \mathrm{Term}[\mathrm{BV}[n]] \rightarrow \mathrm{Term}[\mathrm{BV}[n]]$

```
(declare Term (! t Sort Type))

(declare BV (! n Int Sort))

(declare and
   (! n Int
   (! x (Term (BV n))
   (! y (Term (BV n))
        (Term (BV n))))))
```

$$\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}\ \text{eq_trans}$$

```
(declare eq_trans
   (! s Sort
   (! t1 (Term s)
   (! t2 (Term s)
   (! t3 (Term s)
   (! u1 (Holds (= t1 t2))
   (! u2 (Holds (= t2 t3))
         (Holds (= t1 t3))))))))
```

LFSC SYNTAX EXAMPLE: SIDE CONDITIONS

concat : $\Pi m, n{:}\text{int}.\ \text{Term}[\text{BV}[n]] \to \text{Term}[\text{BV}[n]] \to \text{Term}[\text{BV}[n + m]]$

```
(declare concat
   (! m int
   (! n int
   (! p int
   (! t1 (Term (BV m))
   (! t2 (Term (BV n))  (! s (^ (plus m n) p))
         (Term (BV p)))))))))
```

$$\frac{v \lor l_1 \lor \ldots \lor l_n \quad \neg v \lor l'_1 \lor \ldots \lor l'_n}{l_1 \lor \ldots \lor l_n \lor l'_1 \lor \ldots \lor l'_n}\ \text{Res}$$

```
(declare Res
   (! c1 Clause
   (! c2 Clause
   (! c3 Clause
   (! v Var
   (! u1 (Holds c1)
   (! u2 (Holds c2)
   (! s (^ (resolve c1 c2 v) c3))
         (Holds c3))))))))
```

LFSC CHALLENGES

How to minimize

- instrumentation effort for proof generation
- proof-generation overhead
- proof-object size
- proof-checking time
- trusted core
  (proof checker code + proof rules + side condition code)

LFSC CHALLENGES

How to minimize

- instrumentation effort for proof generation
- proof-generation overhead
- proof-object size
- proof-checking time
- trusted core
  (proof checker code + proof rules + side condition code)

How to import LFSC proofs in tools such as interactive theorem provers

Possible approaches:

1. Write a dedicated certified checker in Coq
2. Produce a Coq proof script from LFSC proofs
3. Define an embedding of LFSC into Coq

$1 \to 3$ More ambitious, less efficient

IMPORTING LFSC PROOFS IN COQ

Our current approaches:

1. Extend dedicated certified checker SMTCoq (by C. Keller)

Challenges

- Convert LFSC proofs to SMTCoq proofs
- Add to SMTCoq support for LFSC side conditions

CERTIFYING MODEL CHECKERS

MODEL CHECKERS

Model checkers are major class of formal verification tools

They implement complex and sophisticated algorithms

Many rely on external reasoners such as SAT/SMT solvers

Traditional qualification process is expensive

Proof-certificates help here too

TRUSTING A MODEL CHECKER

1. **Trusted Logic** — The logics used by the model checker are trusted.
2. **Valid Model** — The model and properties accurately depict the system under scrutiny in the semantic given by the logic of the model checker.
3. **Correct Translation** — The input system is correctly translated to its internal representation.
4. **Correct Algorithms** — The model checking algorithms are sound for the models and properties expressible in the supported logic.
5. **Correct Implementation** — The model checking algorithms are correctly implemented.
6. **Trusted Components and Libraries** — If the model checker makes use of external libraries, their implementation is trusted, and if the model checker uses external tools, they are trusted to be correct.
7. **Correct Compilation** — The model checker and its components are correctly compiled to executable machine code.
8. **Correct Execution** — The machine correctly runs the executable.
9. **Trusted IO** — The parsing and output of the model checker are trusted and interpreted correctly.

TRUSTING A MODEL CHECKER

1. **Trusted Logic** — The logics used by the model checker are trusted.
2. **Valid Model** — The model and properties accurately depict the system under scrutiny in the semantic given by the logic of the model checker.
3. **Correct Translation** — The input system is correctly translated to its internal representation.
4. **Correct Algorithms** — The model checking algorithms are sound for the models and properties expressible in the supported logic.
5. **Correct Implementation** — The model checking algorithms are correctly implemented.
6. **Trusted Components and Libraries** — If the model checker makes use of external libraries, their implementation is trusted, and if the model checker uses external tools, they are trusted to be correct.
7. **Correct Compilation** — The model checker and its components are correctly compiled to executable machine code.
8. **Correct Execution** — The machine correctly runs the executable.
9. **Trusted IO** — The parsing and output of the model checker are trusted and interpreted correctly.

KIND 2

SMT-based safety checker for reactive systems

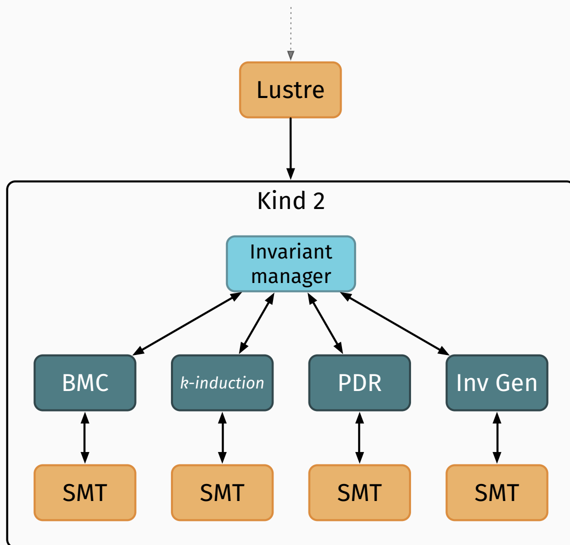
- Project leader: Cesare Tinelli
- Main developers: Adrien Champion, Alain Mebsout, Christoph Sticksel

Multiple and concurrent verification engines

Supports modular / compositional reasoning

Uses SMT solvers CVC4 and Z3

Models specified in an extension of Lustre language


Lustre
Kind 2
Invariant manager
BMC
k-induction
PDR
Inv Gen
SMT
SMT
SMT
SMT

PROVED CORRECT VS CERTIFICATE PRODUCING

Model checkers answers: yes / no

- **no** $\longrightarrow$ counterexample to safety property P of system $\mathcal{S}$
- **yes** $\longrightarrow$?

PROVED CORRECT VS CERTIFICATE PRODUCING

Model checkers answers: yes / no

- **no** $\longrightarrow$ counterexample to safety property P of system $\mathcal{S}$
- **yes** $\longrightarrow$?

Prove once and for all that

$$\forall \mathcal{S}, P.\ \ MC(\mathcal{S}, P) = \textbf{yes} \implies \mathcal{S} \models \Box P$$

PROVED CORRECT VS CERTIFICATE PRODUCING

Model checkers answers: yes / no

- **no** $\longrightarrow$ counterexample to safety property P of system $\mathcal{S}$
- **yes** $\longrightarrow$?

Prove once and for all that

$$\forall \mathcal{S}, P.\ \ MC(\mathcal{S}, P) = \textbf{yes} \implies \mathcal{S} \models \Box P$$

A formal proof of correctness of a model checker like Kind 2 is a hard task

- use of large external lib/tools like SMT solvers
- complex and heavily optimized
- concurrent architecture

Model checkers answers: yes / no

- **no** $\longrightarrow$ counterexample to safety property P of system $\mathcal{S}$
- **yes** $\longrightarrow$?

When the answer is **yes**, have the model checker return each run, a certificate $C(\mathcal{S}, P)$ such that

$$C(\mathcal{S}, P) \text{ valid} \implies \mathcal{S} \models \Box P$$

PROVED CORRECT VS CERTIFICATE PRODUCING

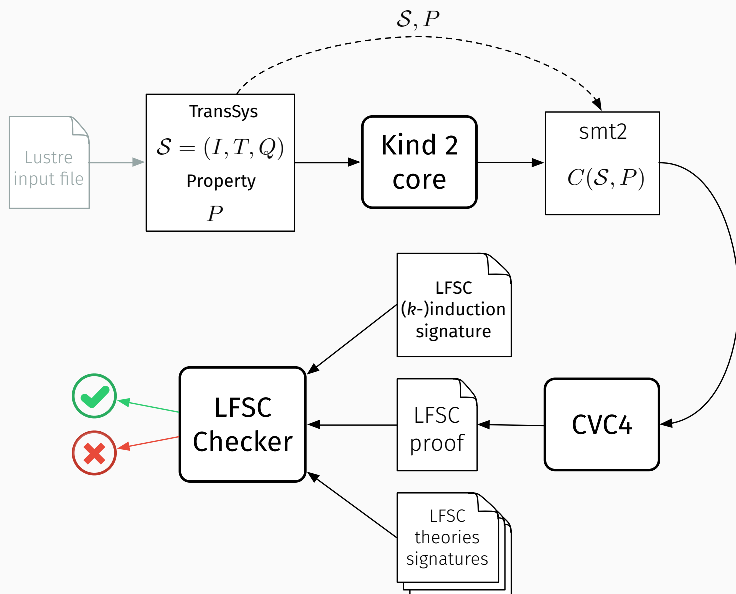
Model checkers answers: yes / no

- **no** $\longrightarrow$ counterexample to safety property P of system $\mathcal{S}$
- **yes** $\longrightarrow$?

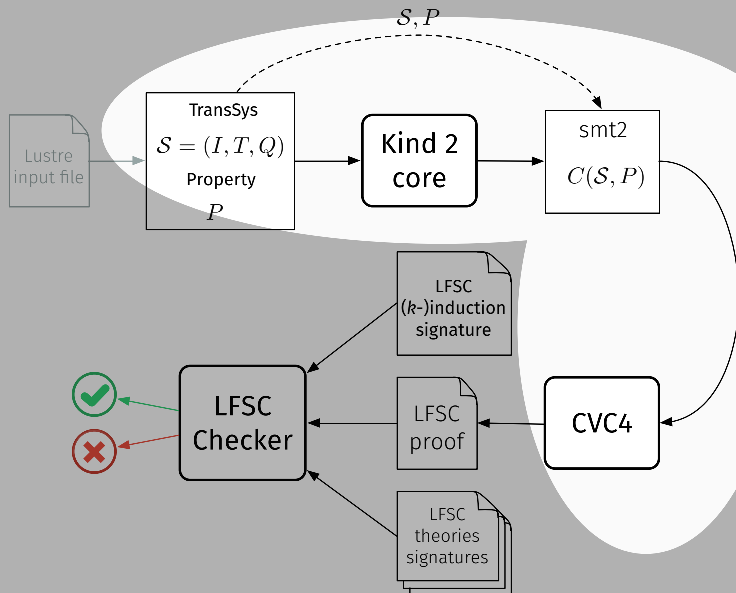
When the answer is **yes**, have the model checker return each run, a certificate $C(\mathcal{S}, P)$ such that

$$C(\mathcal{S}, P) \text{ valid} \implies \mathcal{S} \models \Box P$$

Works if the certificate is small and easily verifiable



$\mathcal{S}, P$
Lustre input file
TransSys
$\mathcal{S} = (I, T, Q)$
Property
P
Kind 2 core
smt2
$C(\mathcal{S}, P)$
LFSC (k-)induction signature
LFSC Checker
LFSC proof
CVC4
LFSC theories signatures



$\mathcal{S}, P$
TransSys
$\mathcal{S} = (I, T, Q)$
Property
P
Lustre input file
Kind 2 core
smt2
$C(\mathcal{S}, P)$
LFSC (k-)induction signature
LFSC Checker
LFSC proof
CVC4
LFSC theories signatures

$$C(\mathcal{S}, P) = (k, \phi)$$

where ϕ is k-inductive for $\mathcal{S}$ and entails the property P

- only engine = k-induction $\longrightarrow (k, P)$
- only engine = PDR $\longrightarrow (1, P \wedge \varphi)$
- works for (k-inductive) automatically generated invariants

$$(max(k_1, \ldots, k_n, k_\phi),\ \mathcal{I}_1 \wedge \ldots \wedge \mathcal{I}_n \wedge \phi)$$

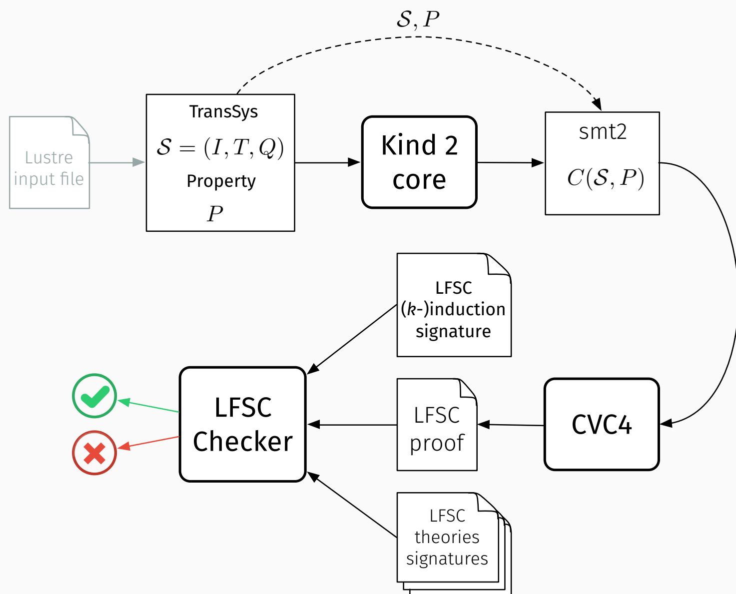
- simplification through *a posteriori* induction check with unsat cores extraction and fixpoint

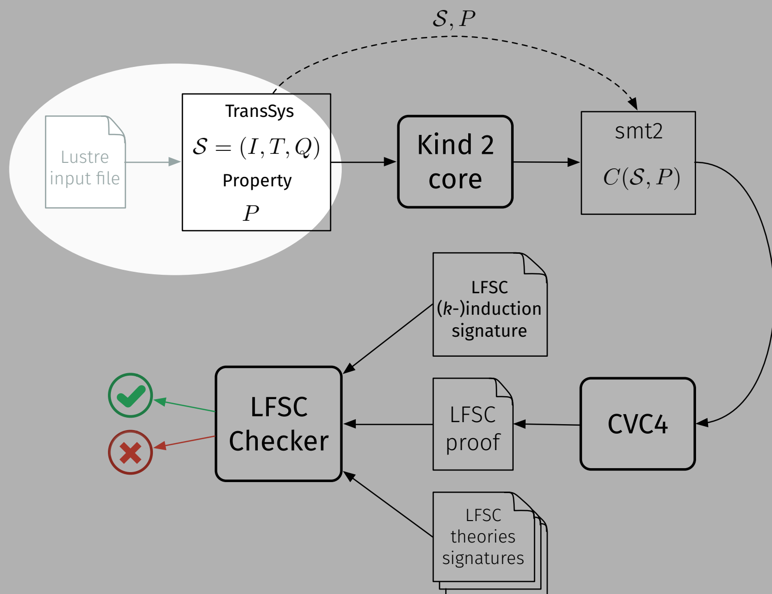
PRELIMINARY RESULTS

Benchmark	Kind 2 time	k	size	gen time	check (CVC 4)
simple_counter.lus	0.16	1	2	0.02	0.01
add_two.lus	0.15	1	1	0.02	0.01
dfa.lus	0.15	1	4	0.03	0.02
inv_gen.lus	0.21	1	2	0.03	0.01
triangle_peg_impossible.lus	7.47	9	1	20.17	5.70
bridge_and_torch.lus	3.50	3	23	0.31	0.21
pilot_flying.lus	13.96	1	50	0.20	0.22
pid.lus	8.73	24	1	4.69	3.64
microwave.kind.lus	2.32	2	23	0.21	0.32
active-standby.kind.lus	7.88	1	20	0.44	0.84
WBS.lus	806	26	91	866	2036 (Z3)
Leader_Selection.lus	1247	41	31	1026	4049 (Z3)
Pilot_Flying.lus	3342	52	83	5581	10628 (Z3)

times in s

FRONTEND CERTIFICATES IN KIND 2




$\mathcal{S}, P$
Lustre input file
TransSys
$\mathcal{S} = (I, T, Q)$
Property
P
Kind 2 core
smt2
$C(\mathcal{S}, P)$
LFSC (k-)induction signature
LFSC Checker
LFSC proof
CVC4
LFSC theories signatures

FRONTEND CERTIFICATES IN KIND 2 (CONT.)

Translation from one formalism to another are sources of error

In Kind 2,

- input models and properties are encoded as SMT formulas
- several intermediate representations
- many simplifications
  (slicing, path compression, encodings, ...)

FRONTEND CERTIFICATES IN KIND 2 (CONT.)

Translation from one formalism to another are sources of error

In Kind 2,

- input models and properties are encoded as SMT formulas
- several intermediate representations
- many simplifications
  (slicing, path compression, encodings, ...)

How to trust the translation from Lustre to SMT ?

FRONTEND CERTIFICATES IN KIND 2: A COMPARATIVE APPROACH

Use two different front ends

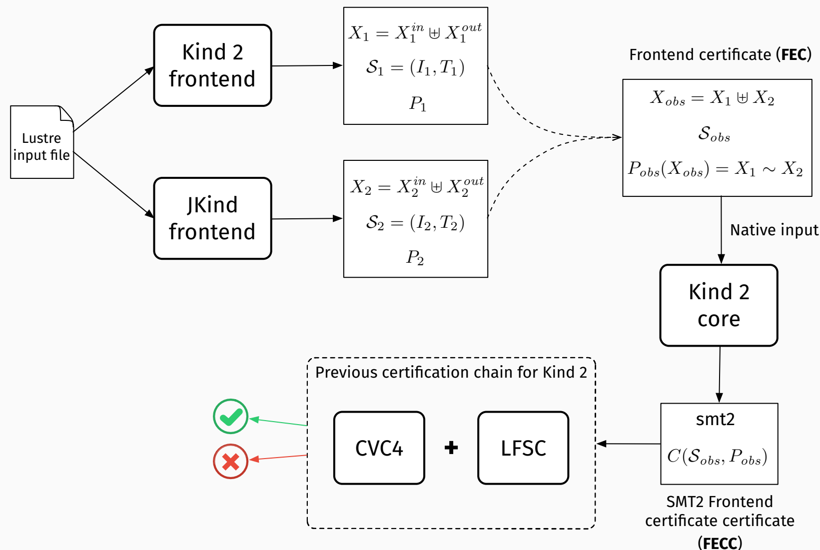
- One from Kind 2
- The other from a comparable tool: JKind (from Rockwell-Collins)

Compare their resulting transition systems and properties

Prove the two systems behaviorially equivalent

Produce a proof certificate

Check the proof certificate as usual



Lustre input file
Kind 2 frontend
JKind frontend
$X_1 = X_1^{in} \uplus X_1^{out}$
$\mathcal{S}_1 = (I_1, T_1)$
P_1
$X_2 = X_2^{in} \uplus X_2^{out}$
$\mathcal{S}_2 = (I_2, T_2)$
P_2
Frontend certificate (FEC)
$X_{obs} = X_1 \uplus X_2$
$\mathcal{S}_{obs}$
$P_{obs}(X_{obs}) = X_1 \sim X_2$
Native input
Kind 2 core
smt2
$C(\mathcal{S}_{obs}, P_{obs})$
SMT2 Frontend certificate certificate (FECC)
Previous certification chain for Kind 2
CVC4
+
LFSC

FEC: Frontend Certificate

- Observational equivalence between internal transition systems of

  Kind 2: $\mathcal{S}_1 = (\bar{x}_1\bar{y}_1, I_1, T_1)$

  JKind: $\mathcal{S}_2 = (\bar{x}_2\bar{y}_2, I_2, T_2)$

FECC: Frontend Certificate Certificate

- SMT2 certificate obtained from the execution of Kind 2 on the FEC $\mathcal{S}_{obs}$

PRELIMINARY RESULTS

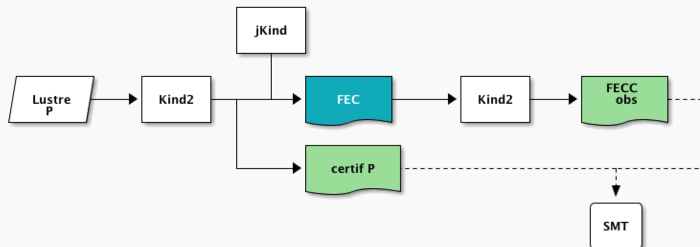
Benchmark	Kind 2 time	FEC				FECC			
		gen	size	MC	k	gen	k	size	check (z3)
simple_counter.lus	0.16	0.73	2	0.16	1	0.02	1	4	0.01
add_two.lus	0.15	0.74	4	0.18	1	0.02	1	4	0.01
dfa.lus	0.16	0.73	4	0.17	1	0.02	1	4	0.01
inv_gen.lus	0.22	0.72	3	0.16	1	0.02	1	4	0.01
triangle_peg_im...	7.90	1.08	158	1.53	1	0.42	1	158	0.44
bridge_and_torc...	1.04	0.80	62	0.24	1	0.12	1	62	0.05
pilot_flying.lus	19.83	1.05	107	0.38	1	0.24	1	107	0.12
pid.lus	8.81	0.80	15	0.18	1	0.04	1	15	0.02
microwave.kind.lus	2.57	1.07	46	35.7	6	1.07	3	79	0.81
active-...	6.54	1.19	75	3.50	1	1.31	1	56	0.67

times in s

SUMMARY OF CURRENT PROGRESS

Three certificates:

- the certificate of invariance of the property (SMT-LIB 2)
- the frontend certificate (FEC) of observational equivalence between Kind 2 and JKind (Kind 2 System)
- the corresponding frontend certificate certificate (FECC) (SMT-LIB 2)



TRUSTING A CERTIFYING MODEL CHECKER

1. **Trusted Logic** — The logics used by the model checker are trusted.
2. **Valid Model** — The model and properties accurately depict the system under scrutiny in the semantic given by the logic of the model checker.
3. **Correct Certificate** — The certificate produced by the model checker is sufficient to convince that the properties are true of the system under scrutiny.
4. **Correct Certificate Checker** — The program that checks the certificates is sound and correctly implemented (and compiled/executed).
5. **Trusted IO** — The parsing and output of the *certificate* checker are trusted.

Complete instrumentation of CVC4 to produce LFSC proofs

Complete extension of SMTCoq to CVC4 theories

Extend SMTCoq to process side conditions

Produce SMTCoq proofs from same instrumentation

Evaluate proof-producing module and proofs objects experimentally

Evaluate proof-producing approach for Kind 2 experimentally

LINKS

CVC4: http://cvc4.cs.nyu.edu/web/

LFSC: https://github.com/CVC4/CVC4/tree/master/proofs/lfsc_checker

Kind 2: http://kind.cs.uiowa.edu

Alt-Ergo: http://alt-ergo.lri.fr/

JKind: https://github.com/agacek/jkind

SMTCoq: https://github.com/smtcoq/smtcoq

Thank you