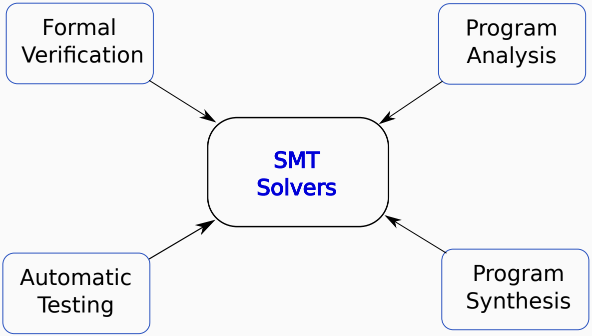
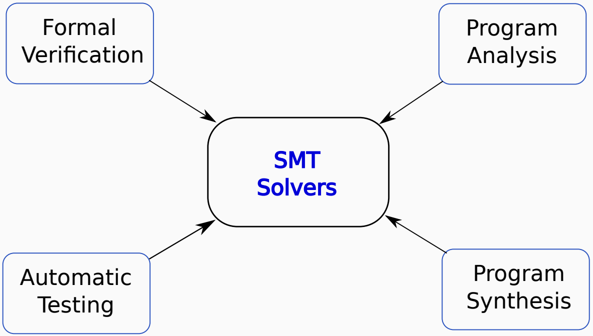
Flexible Proof Production in an Industrial-Strength SMT Solver

Haniel Barbosa,[1] Andrew Reynolds,[2] Gereon Kremer,[3] Hanna Lachnitt,[3] Aina Niemetz,[3] Andres Nötzli,[3] Alex Ozdemir,[3] Mathias Preiner,[3] Arjun Viswanathan,[2] Scott Viteri,[3] Yoni Zohar,[4] Cesare Tinelli,[2] Clark Barrett[3]

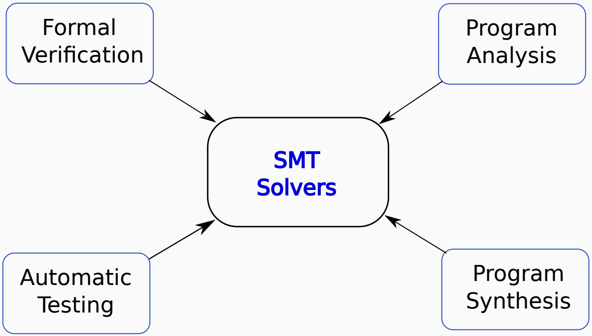
Special thanks: Vinícius Braga[1]

[1] Universidade Federal de Minas Gerais, [2] The University of Iowa, [3] Stanford University, [4] Bar-Ilan University


Formal Verification
Program Analysis
SMT Solvers
Automatic Testing
Program Synthesis



Called billions of times a day...



Called billions of times a day...

- Even a tiny fraction of wrong answers is bad

Bugs in SMT Solvers

- State-of-the-art solvers are large projects:
  - Bitwuzla: 90k LoC (C/C++)
  - cvc5: 300k LoC (C++)
  - z3: 500k LoC (C++)

Bugs in SMT Solvers

- State-of-the-art solvers are large projects:
  - Bitwuzla: 90k LoC (C/C++)
  - cvc5: 300k LoC (C++)
  - z3: 500k LoC (C++)
- How do developers try to avoid bugs?
  - Code reviews
  - Testing on benchmark sets
  - Random input testing

Bugs in SMT Solvers

- State-of-the-art solvers are large projects:
  - Bitwuzla: 90k LoC (C/C++)
  - cvc5: 300k LoC (C++)
  - z3: 500k LoC (C++)
- How do developers try to avoid bugs?
  - Code reviews
  - Testing on benchmark sets
  - Random input testing
- But bugs remain:
  - Every year SMT-COMP has disagreements between solvers
  - Fuzzing tools often find bugs in solvers

SMT Recap

// Input

Many-sorted, first-order logic formula:

$$\overbrace{\text{contains}(x, \texttt{"FLoC"})}^{\text{String constraint}} \overbrace{\land}^{\text{Boolean connective}} \overbrace{|x| \geq 5}^{\text{Integer constraint}}$$

- Does there exist a string x s.t. x contains $\texttt{"FLoC"}$ and is at least five characters long?

SMT Recap

// Input

Many-sorted, first-order logic formula:

$$\overbrace{\mathsf{contains}(x, \texttt{"FLoC"})}^{\text{String constraint}} \overbrace{\wedge}^{\text{Boolean connective}} \overbrace{|x| \geq 5}^{\text{Integer constraint}}$$

- ▶ Does there exist a string x s.t. x contains "FLoC" and is at least five characters long?

// Output

- Satisfiable: There are values for which the formula evaluates to true
- Unsatisfiable: No values exist that make the formula true

SMT Recap

// Input

Many-sorted, first-order logic formula:

$$\overbrace{\mathsf{contains}(x, \texttt{"FLoC"})}^{\text{String constraint}} \overbrace{\wedge}^{\text{Boolean connective}} \overbrace{|x| \geq 5}^{\text{Integer constraint}}$$

▶ Does there exist a string x s.t. x contains "FLoC" and is at least five characters long?

// Output

- Satisfiable: There are values for which the formula evaluates to true
- Unsatisfiable: No values exist that make the formula true

// Examples of Theories

- Integer/real arithmetic: $5 + x \geq y$
- Bit-vectors: $\mathsf{bvule}(x, \texttt{0xFF})$
- Strings: $\mathsf{substr}(x, 0, 3) = \texttt{"foo"}$

Can We Just Certify the Solvers?

- Large, complex code bases are too costly to certify
- A (simpler) certified system can be too slow [FBL18; Fle19]
- Certifying/qualifying a system freezes it, potentially blocking improvements
  - Working around adding new features slow and costly [BD18]

Alternative: Checking Outputs

For satisfiable inputs: Evaluate formula on values of model generated by solver

Alternative: Checking Outputs

For satisfiable inputs: Evaluate formula on values of model generated by solver

$$\text{contains}(x, \texttt{"FLoC"}) \quad \wedge \quad |x| \geq 5$$

Alternative: Checking Outputs

For satisfiable inputs: Evaluate formula on values of model generated by solver

$$\text{contains}(x, \texttt{"FLoC"}) \quad \wedge \quad |x| \geq 5$$

Model: $\mathcal{M} = \{x \mapsto \texttt{"FLoC-2022"}\}$

Alternative: Checking Outputs

For satisfiable inputs: Evaluate formula on values of model generated by solver

$$\text{contains}(x, \texttt{"FLoC"}) \quad \wedge \quad |x| \geq 5$$

Model: $\mathcal{M} = \{x \mapsto \texttt{"FLoC-2022"}\}$

What about unsatisfiable inputs?

Agenda

- Uses of proofs
- Challenges in producing proofs
- Making a solver proof producing:
  - Proofs through instrumentation: Primary approach in cvc5
  - Proofs through reconstruction: Detailed rewrite proofs in cvc5
- Current and future work

Proofs: A New Hope

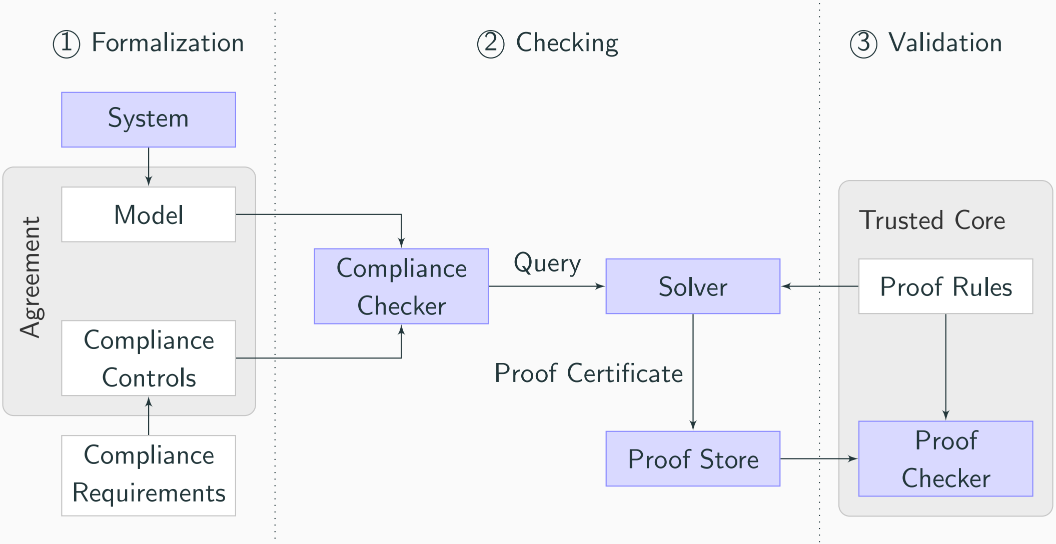
- Proofs are a justification of the logical reasoning the solver has performed to find a solution
- A proof can be checked *independently*
  - Smaller trusted base: LFSC 5.5k (C++) + 2k (signatures) LoC vs. CVC5 300k LoC
  - Proof checking is generally more efficiently than solving the problem
- Confidence in results is decoupled from the solver's implementation

Demo: A Simple Proof

Applications of SMT Proofs

- Strong correctness guarantees
    - High-quality proofs can be used to facilitate automated compliance
- Integrations with other systems
    - Automation in interactive theorem proving
    - External proof cecking can identify bugs in proof rules
- Valuable for debugging
- Formalization of proof rules improves code base
    - Uncovers existing issues
    - Forces modular and clean code design
    - Improves tool robustness
- A rich source of data that can be mined for various purposes (e.g., interpolation)

Applications of SMT Proofs: Compliance



Challenges for SMT proofs

- Collecting and storing proofs efficiently

  Many attempts, no silver bullet

  [SZS04; KBT+16; HBR+15; Mos08; MB08; Sch13; KV13; WDF+09; BODF09]
- Proofs for sophisticated preprocessing and rewriting techniques

  Initial progress but many challenges remain

  [BBFF20]
- Proofs for complex procedures in theory solvers (e.g., CAD, strings)

  Open
- Standardizing a proof format

  Open
- Scalable, trustworthy checking

  Many attempts, no silver bullet

  [BBP13; SOR+13; EMT+17; BBFF20; SFD21]

The Journey

- CVC4's old proof module struggled with many of those challenges
- For two years, we reimplemented its proof module *from scratch*

The Journey

- CVC4's old proof module struggled with many of those challenges
- For two years, we reimplemented its proof module *from scratch*
  - Producing proofs should not significantly change the behavior of the solver
    - Incorporate (almost) all relevant optimizations
    - Coarse-grained steps for non-supported inferences

The Journey

- CVC4's old proof module struggled with many of those challenges
- For two years, we reimplemented its proof module *from scratch*
  - Producing proofs should not significantly change the behavior of the solver
    - Incorporate (almost) all relevant optimizations
    - Coarse-grained steps for non-supported inferences
  - Modular infrastructure allowing fine-grained error localization
    - Independent proof components, combined in a trusted manner
    - Every rule associated with an internal proof checker

The Journey

- CVC4's old proof module struggled with many of those challenges
- For two years, we reimplemented its proof module *from scratch*
  - Producing proofs should not significantly change the behavior of the solver
    - Incorporate (almost) all relevant optimizations
    - Coarse-grained steps for non-supported inferences
  - Modular infrastructure allowing fine-grained error localization
    - Independent proof components, combined in a trusted manner
    - Every rule associated with an internal proof checker
  - Custom eager/lazy generation of proofs
    - Proof reconstruction (elaboration) via internal post-processing

The Journey

- CVC4's old proof module struggled with many of those challenges
- For two years, we reimplemented its proof module *from scratch*
  - Producing proofs should not significantly change the behavior of the solver
    - Incorporate (almost) all relevant optimizations
    - Coarse-grained steps for non-supported inferences
  - Modular infrastructure allowing fine-grained error localization
    - Independent proof components, combined in a trusted manner
    - Every rule associated with an internal proof checker
  - Custom eager/lazy generation of proofs
    - Proof reconstruction (elaboration) via internal post-processing
  - Support internal proof format and conversions to different proof formats
    - LFSC, Lean, Alethe

Proof module architecture for CDCL($\mathcal{T}$)

•

Proof module architecture for CDCL($\mathcal{T}$)

-

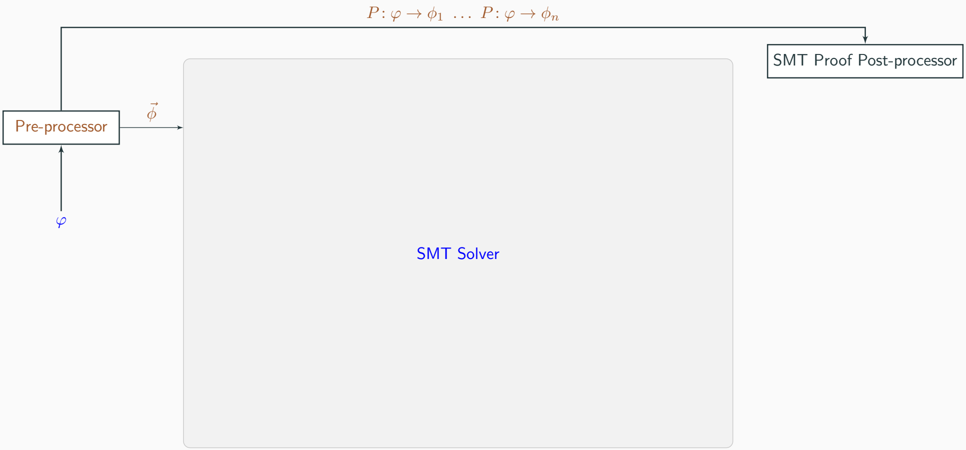
Proof module architecture for CDCL($\mathcal{T}$)

-

Proof module architecture for CDCL($\mathcal{T}$)

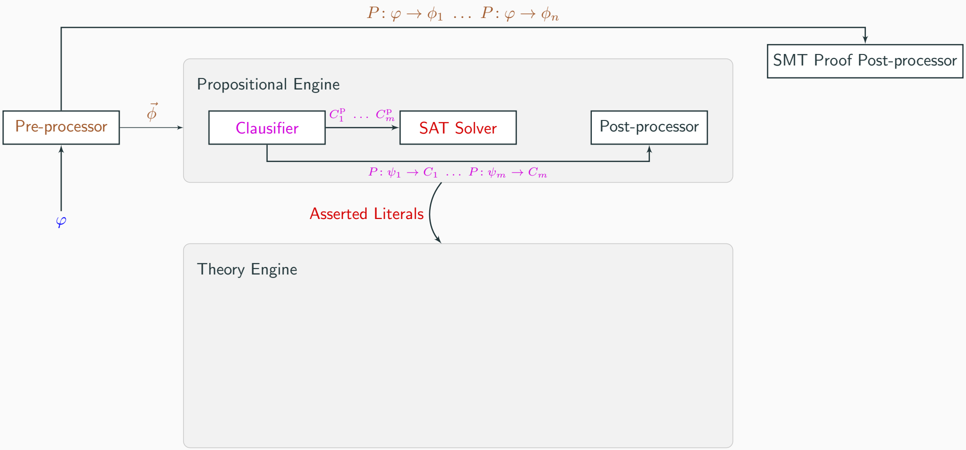
- Preprocessor simplifies formula globally:
  $x \simeq t \wedge F[x] \;\mapsto\; F[t] \qquad F[(ite\ P\ t_1\ t_2)] \;\mapsto\; F[t'] \wedge P \rightarrow t' \simeq t_1 \wedge \neg P \rightarrow t' \simeq t_2$

Proof module architecture for CDCL($\mathcal{T}$)



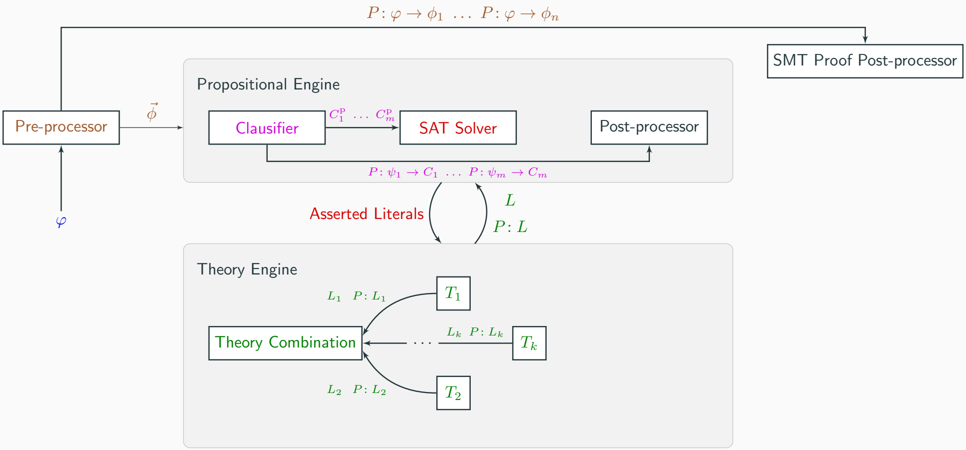
- Preprocessor simplifies formula globally:
  $x \simeq t \wedge F[x] \ \mapsto \ F[t] \qquad F[(ite\ P\ t_1\ t_2)] \ \mapsto \ F[t'] \wedge P \to t' \simeq t_1 \wedge \neg P \to t' \simeq t_2$

Proof module architecture for CDCL($\mathcal{T}$)



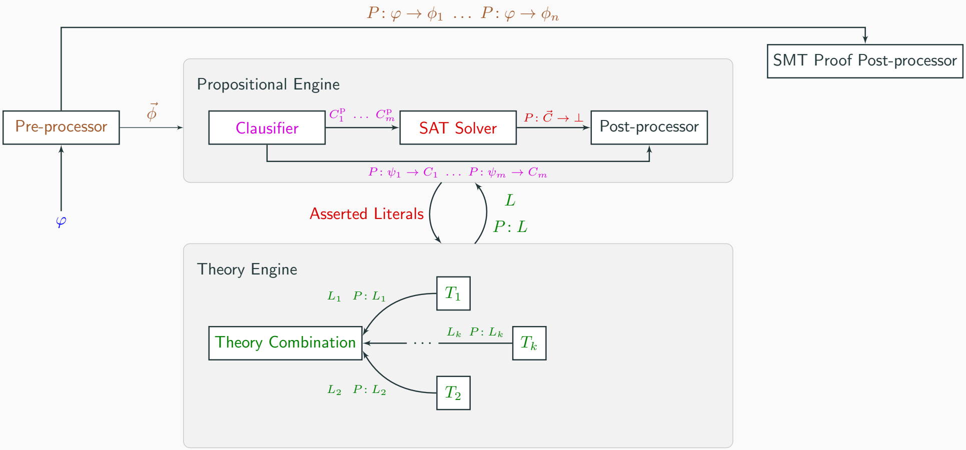
- Clausifier converts to Conjunctive Normal Form (CNF)
  SAT solver asserts literals that must hold based on Boolean abstraction

Proof module architecture for CDCL($\mathcal{T}$)



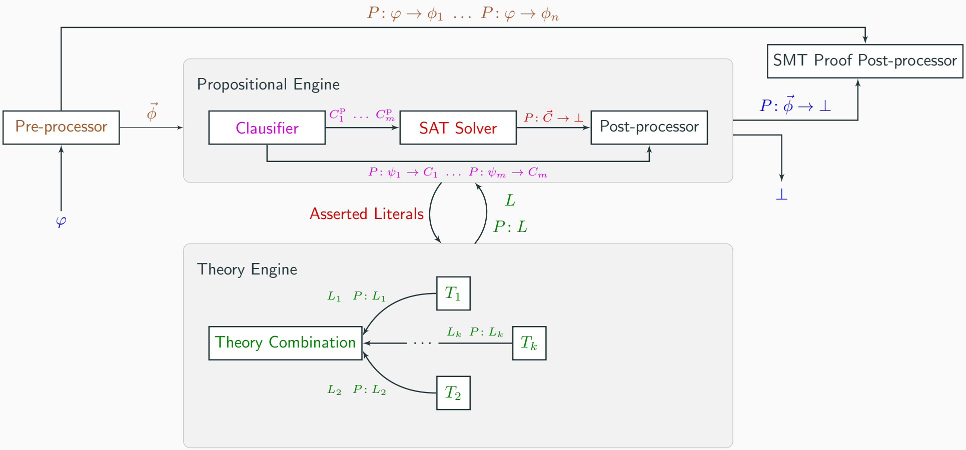
- Theory solvers check consistency in the theory

Proof module architecture for CDCL($\mathcal{T}$)



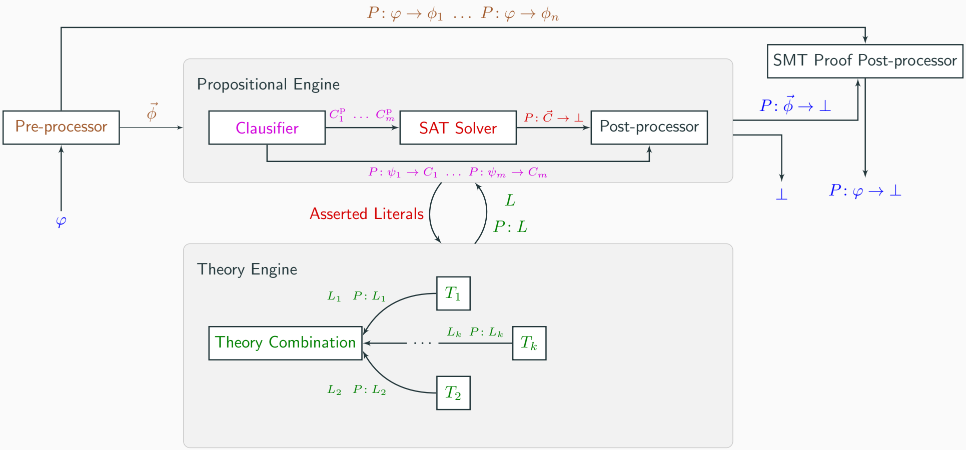
- Theory solvers check consistency in the theory

Proof module architecture for CDCL($\mathcal{T}$)



- Theory solvers check consistency in the theory

Proof module architecture for CDCL($\mathcal{T}$)



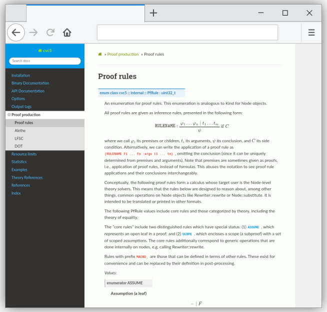
- Theory solvers check consistency in the theory

Main components: Internal proof calculus

- Rules for equality reasoning (congruence closure)
- Rules for rewriting, substitution
  - Coarse-grained rules for capturing multiple core utilities
- Rules for witness forms
  - Enable introduction and correct handling of new symbols
- Rules for scoped reasoning
  - Enable local reasoning, via assumptions and $\Rightarrow$-introduction
- Theory-specific rules
  - Boolean (clausification, resolution, ...)
  - Arithmetic (linear, non-linear, integer, rationals, transcendentals)
  - Arrays, Datatypes, Bit-vectors, Quantifiers, ...

Main components: Internal proof calculus

- Rules for equality reasoning (congruence closure)
- Rules for rewriting, substitution
  - Coarse-grained rules for capturing multiple core utilities
- Rules for witness forms
  - Enable introduction and correct handling of new symbols
- Rules for scoped reasoning
  - Enable local reasoning, via assumptions and $\Rightarrow$-introduction
- Theory-specific rules
  - Boolean (clausification, resolution, ...)
  - Arithmetic (linear, non-linear, integer, rationals, transcendentals)
  - Arrays, Datatypes, Bit-vectors, Quantifiers, ...

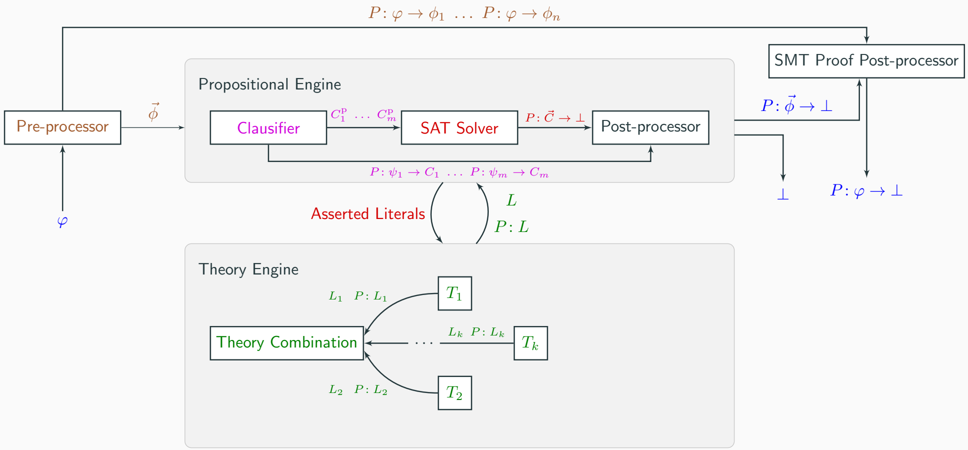


Documentation: `https://cvc5.github.io/docs/latest/proofs/proof_rules.html`

Main components: Library of proof generators

- Encapsulate common patterns for building proofs
- Solving components store information during solving
- Derived facts are distributed with associated proof generators
- When proof generator is requested for fact φ, its internal information is used to produce the proof $P : \varphi$.

Proof module architecture



- Actually, *proof generators* are transmitted between components
- Only at the post-processors are proofs requested (and fully computed)

Proof generation for substitution and rewriting

- Substitution and rewriting inferences recorded without further details
- No need to instrument utilities to track how terms are converted
  - Only macro steps and used rewrites rules are stored in generators

$$\dfrac{a \simeq 0 \qquad\qquad b \simeq 1}{(a > b \wedge F) \simeq \bot}\ \textsf{SR}$$

Proof generation for substitution and rewriting

- Substitution and rewriting inferences recorded without further details
- No need to instrument utilities to track how terms are converted
  - Only macro steps and used rewrites rules are stored in generators

$$\dfrac{a \simeq 0 \qquad b \simeq 1}{(a > b \wedge F) \simeq \bot}\ \textsf{SR}$$

$$\dfrac{\dfrac{a \simeq 0 \qquad b \simeq 1}{(a > b \wedge F) \simeq (0 > 1 \wedge F)}\ \textsf{SUBS} \qquad \dfrac{}{(0 > 1 \wedge F) \simeq \bot}\ \textsf{RW}}{(a > b \wedge F) \simeq \bot}$$

Proof generation for substitution and rewriting

- Substitution and rewriting inferences recorded without further details
- No need to instrument utilities to track how terms are converted
  - Only macro steps and used rewrites rules are stored in generators

$$\dfrac{a \simeq 0 \qquad b \simeq 1}{(a > b \wedge F) \simeq \bot}\ \text{SR}$$

$$\dfrac{\dfrac{a \simeq 0 \qquad b \simeq 1}{(a > b \wedge F) \simeq (0 > 1 \wedge F)}\ \text{SUBS} \qquad \dfrac{}{(0 > 1 \wedge F) \simeq \bot}\ \text{RW}}{(a > b \wedge F) \simeq \bot}$$

$$\dfrac{\dfrac{\dfrac{}{0 > 1 \simeq \bot}\ \text{arith\_rw} \qquad \dfrac{}{F \simeq F}\ \text{refl}}{(0 > 1 \wedge F) \simeq (\bot \wedge F)}\ \text{cong} \qquad \dfrac{}{(\bot \wedge F) \simeq \bot}\ \text{bool\_rw}}{(0 > 1 \wedge F) \simeq \bot}\ \text{trans}$$

Proof generation for substitution and rewriting

- Substitution and rewriting inferences recorded without further details
- No need to instrument utilities to track how terms are converted
  - Only macro steps and used rewrites rules are stored in generators

$$\dfrac{a \simeq 0 \qquad b \simeq 1}{(a > b \land F) \simeq \bot}\ \textsf{SR}$$

$$\dfrac{\dfrac{a \simeq 0 \qquad b \simeq 1}{(a > b \land F) \simeq (0 > 1 \land F)}\ \textsf{SUBS} \qquad \dfrac{}{(0 > 1 \land F) \simeq \bot}\ \textsf{RW}}{(a > b \land F) \simeq \bot}$$

$$\dfrac{\dfrac{\dfrac{}{0 > 1 \simeq \bot}\ \textsf{arith\_rw} \qquad \dfrac{}{F \simeq F}\ \textsf{refl}}{(0 > 1 \land F) \simeq (\bot \land F)}\ \textsf{cong} \qquad \dfrac{}{(\bot \land F) \simeq \bot}\ \textsf{bool\_rw}}{(0 > 1 \land F) \simeq \bot}\ \textsf{trans}$$

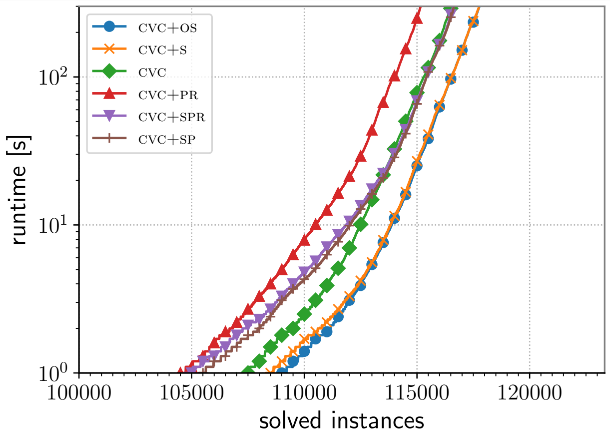
- Heavily used for strings, preprocessing, bitblasting, and so on.

Evaluation: proof production cost

- Techniques (currently) incompatible with proofs (O)
  - Variable and clause elimination (SAT solver), EUF symmetry breaking, off-the-shelf SAT solvers for BV bitblasted constraints
- Simplification under global assumptions (S)
- Producing proofs (P)
- Reconstructing fine-grained steps from coarse ones (R)

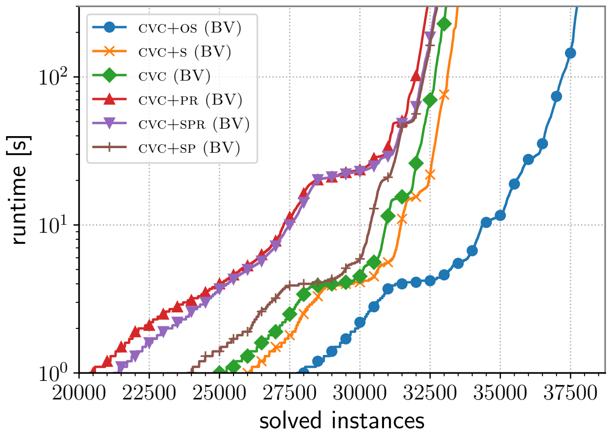
- Benchmarks
  - 123k NON-BVS benchmarks
  - 39k BVS benchmarks

Evaluation: non-BVs



- S is fundamental for performance
- P and R significant but not critical overhead
  - CVC+SP is in total 1.7× slower than CVC+S
  - CVC+SPR is in total 1.8× slower than CVC+S

Evaluation: BVs



- O is critical
- P and R significant but not critical overhead
  - CVC+SP is in total 2.6× slower than CVC+S
  - CVC+SPR is in total 3.9× slower than CVC+S

Evaluation: Proof Coverage

Perfect proofs are those without coarse-grained steps.

- 92% of perfect proofs in BVs
- 80% in non-BVs
  - Culprits are mostly yet-to-be-supported theory preprocessing passes
  - Also all non-linear arithmetic inferences from cylindric algebraic coverings
- 100% in QF_S, 80% in QF_SLIA

The Challenge with Rewrites

- Modern SMT solvers implement <u>hundreds</u> of rewriting rules for state-of-the-art performance
  - String solver in cvc5: Over 200 rules in $3{,}000$ lines of C++ code
  - Example:

$$\mathsf{substr}(\texttt{""}, m, n) \rightsquigarrow \texttt{""}$$

The Challenge with Rewrites

- Modern SMT solvers implement hundreds of rewriting rules for state-of-the-art performance
  - String solver in cvc5: Over 200 rules in $3{,}000$ lines of C++ code
  - Example:

$$\mathsf{substr}(\texttt{""}, m, n) \rightsquigarrow \texttt{""}$$

- Many proof applications require detailed proofs
  - Easier proof checking, better integration with interactive theorem provers
  - Required: Individual proof rules for rewrite rules

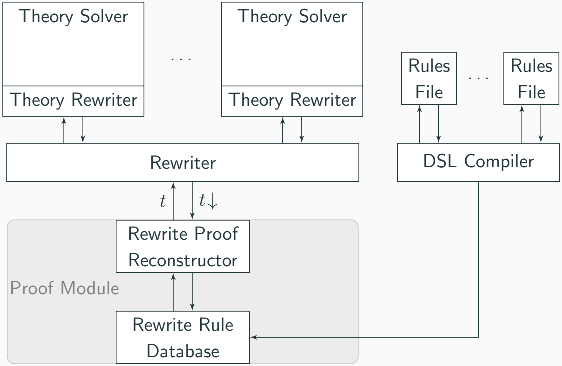
The Challenge with Rewrites

- Modern SMT solvers implement hundreds of rewriting rules for state-of-the-art performance
  - String solver in cvc5: Over 200 rules in $3{,}000$ lines of C++ code
  - Example:

$$\mathsf{substr}(\texttt{""}, m, n) \rightsquigarrow \texttt{""}$$

- Many proof applications require detailed proofs
  - Easier proof checking, better integration with interactive theorem provers
  - Required: Individual proof rules for rewrite rules
- Traditional approach: Instrumenting code
  - Difficult and tedious: Define proof rule and instrument code for every rewrite

Proofs for Rewrites: Our Approach



- Treat rewriter as black box and reconstruct proofs for rewrites externally
- A domain-specific language (DSL), RARE, to specify a database of rewrite rules
- A compiler for RARE that generates the C++ code that populates the rewrite rule database
- A general reconstruction algorithm, applied as a post-processor

Demo: Detailed Rewrite Proofs

Rare: Design Goals

- Succinct: Writing rewrite rules should be simple and concise.
- Expressive: Support for the majority of the rewrite rules in a state-of-the-art solver
- Accessible: Easy to parse and understand

Rare: Basic Rules

```
(define-rule substr-empty ((m Int) (n Int))
  (str.substr "" m n) "")
```

Rare: Matching

```
(define-rule eq-refl ((t ?)) (= t t) true)
```

Rare: Lists

```
(define-rule str-concat-flatten (
    (xs String :list) (s String)
    (ys String :list) (zs String :list))
  (str.++ xs (str.++ s ys) zs) ; match
  (str.++ xs s ys zs))         ; target
```

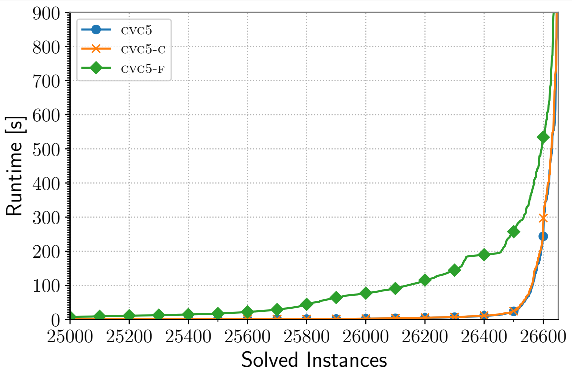
Rare: Conditional Rules

```
(define-cond-rule concat-clash (
    (s1 String) (s2 String :list)
    (t1 String) (t2 String :list))
  (and (= (str.len s1) (str.len t1)) ; precondition
    (not (= s1 t1)))
  (= (str.++ s1 s2) (str.++ t1 t2))  ; match
  false)                             ; target
```

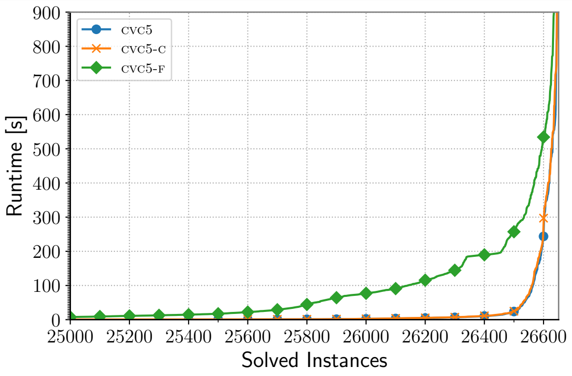
Rare: Fixed-Point Rules

```
(define-rule* str-len-concat-rec (
    (s1 String) (s2 String)
    (rest String :list))
  (str.len (str.++ s1 s2 rest)) ; match
  (str.len (str.++ s2 rest))     ; target
  (+ (str.len s1) _))            ; context
```

Rare: Evaluation



Rare: Evaluation



- Rewrites reconstructed: 95% for problems from the industrial set and of 87% for SMT-LIB
- Fully fine-grained: 20% of the proofs for industrial benchmarks, 23% of all proofs for SMT-LIB benchmarks with rewrite steps ($6{,}120$ out of $26{,}611$)

Current/Future Work

- Detailed proofs for remaining theories (e.g., non-linear arithmetic)
- Integration of DRAT proofs for propositional reasoning
- Integration with interactive theorem provers: Lean, Isabelle/HOL, Coq
- Proof components
  - Producing and checking fully detailed can be costly
  - Idea: Produce proofs that can be expanded <u>on-demand</u>
- More complete rules for rewriting
- Standardization of a proof format
  - Proof exhibition track at SMT-COMP 2022

Anecdotes

- Internal proof checker is highly valuable for development
- Error localization for proofs is important
- Formalization of proof rules uncovers existing issues
- Performance issues
  - In a few cases, proof checker indicated it could prove something stronger
- Soundness issues
  - Cannot write proper proof checker if the reasoning of the solver is wrong
- Proofs are also valuable for debugging
  - Soundness bug reported, proofs used to easily isolate the incorrect rewrite
- Combination of approaches for proof generation

Conclusion

- Proofs are integral for the trustworthiness SMT solvers (and have other applications)
- Fine-grained proofs are now available for most of cvc5's reasoning
  - Combination of instrumentation and reconstruction
  - Strings and simplification under global assumptions were special milestones
  - Detailed proofs for rewriting coming soon
- Multiple proof formats are supported
  - Integration into multiple proof checkers are ongoing
  - Formalization of new calculi in Lean, LFSC, Isabelle/HOL
  - DOT format and web-based proof visualizer

More information: `https://cvc5.github.io/`

SMT Proof Standardization Update today at 16:00

Flexible Proof Production in an Industrial-Strength SMT Solver

Haniel Barbosa,[1] Andrew Reynolds,[2] Gereon Kremer,[3] Hanna Lachnitt,[3] Aina Niemetz,[3] Andres Nötzli,[3] Alex Ozdemir,[3] Mathias Preiner,[3] Arjun Viswanathan,[2] Scott Viteri,[3] Yoni Zohar,[4] Cesare Tinelli,[2] Clark Barrett[3]

Special thanks: Vinícius Braga[1]

[1] Universidade Federal de Minas Gerais, [2] The University of Iowa, [3] Stanford University, [4] Bar-Ilan University

How some proofs look like

$$\frac{A \vee \ell \qquad B \vee \bar{\ell}}{A \vee B}$$

$$\neg(a \simeq b) \vee f(a) \simeq f(b)$$
$$\neg(y > 1) \vee \neg(x < 1) \vee y > x$$

$$\frac{\varphi_1 \wedge \cdots \wedge \varphi_n}{\varphi_i}$$

$$\neg(\varphi_1 \wedge \cdots \wedge \varphi_n) \vee \varphi_i$$

A particular challenge has been String solving

- Preprocessing
- Clausification
- SAT solving
- UF theory solver
- Linear Arithmetic solver
- Theory combination
- Quantifier instantiation
- Rewriting
    - Including complex string methods [RNBT19]
- Strings theory solver
    - Core calculus [LRT+14]
    - Extended function reductions [RWB+17]
    - Regular expression unfolding

References

[BBFF20] Haniel Barbosa, Jasmin Christian Blanchette, Mathias Fleury, et al. “Scalable Fine-Grained Proofs for Formula Processing”. In: Journal of Automated Reasoning 64.3 (2020), pp. 485–510.

[BBP13] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. “Extending Sledgehammer with SMT Solvers”. In: Journal of Automated Reasoning 51.1 (2013), pp. 109–128.

[BD18] Lilian Burdy and David Déharbe. “Teaching an Old Dog New Tricks - The Drudges of the Interactive Prover in Atelier B”. In: Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2 Ed. by Michael J. Butler, Alexander Raschke, Thai Son Hoang, et al. Vol. 10817. Lecture Notes in Computer Science. Springer, 2018, pp. 415–419.

[BODF09] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, et al. “veriT: An Open, Trustable and Efficient SMT-Solver”. In: Proc. Conference on Automated Deduction (CADE). Ed. by Renate A. Schmidt. Vol. 5663. Lecture Notes in Computer Science. Springer, 2009, pp. 151–156.

[EMT+17] Burak Ekici, Alain Mebsout, Cesare Tinelli, et al. "SMTCoq: A Plug-In for Integrating SMT Solvers into Coq". In: Computer Aided Verification (CAV). Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 126–133.

[FBL18] Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. "A verified SAT solver with watched literals using imperative HOL". In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, Ed. by June Andronick and Amy P. Felty. ACM, 2018, pp. 158–171.

[Fle19] Mathias Fleury. "Optimizing a Verified SAT Solver". In: NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings. Ed. by Julia M. Badger and Kristin Yvonne Rozier. Vol. 11460. Lecture Notes in Computer Science. Springer, 2019, pp. 148–165.

[HBR+15] Liana Hadarean, Clark W. Barrett, Andrew Reynolds, et al. "Fine Grained SMT Proofs for the Theory of Fixed-Width Bit-Vectors". In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). Ed. by Martin Davis, Ansgar Fehnker, Annabelle McIver, et al. Vol. 9450. Lecture Notes in Computer Science. Springer, 2015, pp. 340–355.

[KBT+16] Guy Katz, Clark W. Barrett, Cesare Tinelli, et al. "Lazy proofs for DPLL(T)-based SMT solvers". In: Formal Methods In Computer-Aided Design (FMCAD). Ed. by Ruzica Piskac and Muralidhar Talupur. IEEE, 2016, pp. 93–100.

[KV13] Laura Kovács and Andrei Voronkov. "First-Order Theorem Proving and Vampire". English. In: Computer Aided Verification (CAV). Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 1–35.

[LRT+14] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, et al. "A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions". In: Computer Aided Verification (CAV). Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 646–662.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Proofs and Refutations, and Z3". In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) Workshops. Ed. by Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, et al. Vol. 418. CEUR Workshop Proceedings. CEUR-WS.org, 2008.

[Mos08] Michał Moskal. "Rocket-Fast Proof Checking for SMT Solvers". In: Tools and Algorithms for Construction and Analysis of Systems (TACAS). Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 486–500.

[RNBT19] Andrew Reynolds, Andres Nötzli, Clark W. Barrett, et al. "High-Level Abstractions for Simplifying Extended String Constraints in SMT". In: Computer Aided Verification (CAV), Part II. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11562. Lecture Notes in Computer Science. Springer, 2019, pp. 23–42.

[RWB+17] Andrew Reynolds, Maverick Woo, Clark Barrett, et al. "Scaling Up DPLL(T) String Solvers Using Context-Dependent Simplification". In: Computer Aided Verification (CAV). Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 453–474.

[Sch13] Stephan Schulz. "System Description: E 1.8". English. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). Ed. by Ken McMillan, Aart Middeldorp, and Andrei Voronkov. Vol. 8312. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 735–743.

[SFD21] Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. "Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant". In: Proc. Conference on Automated Deduction (CADE). Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. Lecture Notes in Computer Science. Springer, 2021, pp. 450–467.

[SOR+13] Aaron Stump, Duckki Oe, Andrew Reynolds, et al. "SMT proof checking using a logical framework". In: Formal Methods in System Design 42.1 (2013), pp. 91–118.

[SZS04] Geoff Sutcliffe, Jürgen Zimmer, and Stephan Schulz. "TSTP Data-Exchange Formats for Automated Theorem Proving Tools". In: Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems. Ed. by Weixiong Zhang and Volker Sorge. Vol. 112. Frontiers in Artificial Intelligence and Applications. IOS Press, 2004, pp. 201–215.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, et al. "SPASS Version 3.5". English. In: Proc. Conference on Automated Deduction (CADE). Ed. by RenateA. Schmidt. Vol. 5663. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 140–145.