Ethos: A Fast Proof Checker for the Eunoia Logical Framework

Andrew Reynolds[1], Hans-Jörg Schurr[1,3,4], Mallku Soldevila[5], Haniel Barbosa[5], Clark Barrett[2], and Cesare Tinelli[1]

[1] The University of Iowa, Iowa City, USA
[2] Stanford University, Stanford, USA
[3] KU Leuven, Leuven, Belgium
[4] Vrije Universiteit Brussel, Brussels, Belgium
[5] Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

Abstract. SMT solvers are used in many safety-critical applications. To provide evidence of the correctness of their answers, some SMT solvers generate externally checkable proof certificates. We present a high-performance checker for SMT proof certificates called *Ethos*. In contrast with other dedicated SMT proof checkers, Ethos does not implement a fixed proof calculus. Instead, it allows users to provide their own proof calculi using the declarative language Eunoia, which extends the familiar SMT-LIB syntax to make this easy and convenient. We give a short overview of Eunoia and then focus on Ethos itself. We describe multiple optimization and implementation details which ensure that Ethos is fast and practical. We also evaluate Ethos on proofs generated by cvc5, showing that the flexibility of Ethos allows us to efficiently check fine-grained proofs, containing no proof holes, over all SMT-LIB logics without floating point arithmetic.

Keywords: SMT · proof checking · logical frameworks

1 Introduction

Satisfiability Modulo Theories (SMT) solvers reason about logical problems expressed in terms of a wide variety of theories (arithmetic, bit-vectors, strings, and so on.). They are used as back ends for software verification tools as well as a variety of other safety-critical applications (e.g., [8]). A key concern for these applications is that SMT solvers are large and complex systems. As such, they are at risk of having bugs which affect their correctness.

A pragmatic but rigorous approach for addressing this concern is for an SMT solver to produce a *proof certificate* (or just *proof* for short) that can be checked by an external proof checker [1, 14]. A proof accepted by a proof checker is a strong piece of evidence that the answer is correct. Since proof checking is in general considerably simpler than proof finding, which is ultimately what SMT solvers do, even a proof checker that is not formally verified can strengthen the user's trust in the SMT solver's answer. However, this approach requires a format

to be defined for SMT proofs, which poses a serious challenge, as different theory solvers may use vastly different calculi because of the differences in the theories they support and the preprocessing and reasoning methods they implement.

In this paper, we present *Ethos*, a generic proof checker for SMT. Instead of implementing a fixed calculus, Ethos supports the *Eunoia* logical framework, which is designed to make specifying proof calculi and proofs simple and convenient for SMT developers. Eunoia's syntax extends version 2.7 of the SMT-LIB standard [6] with commands for (*i*) defining theory *signatures*, i.e., types, function symbols, and proof rules, and (*ii*) expressing proofs themselves. Given a Eunoia file that provides both (a proof calculus and a proof), Ethos checks that the proof correctly applies the rules of the calculus. It can also verify that the assumptions used in a proof in fact correspond to the assertions in a specific SMT-LIB problem, thus ensuring that the proof is not only a correct proof, but is a proof of the correct theorem.

While Ethos is not a formally verified proof checker, it significantly reduces the trusted code base. For example, the source code of the cvc5 SMT solver [4] consists of over 300K lines of C++, while Ethos is less than 10K lines of C++. The proof signature for cvc5 is expressed in 6,397 lines of Eunoia code. Strong runtime performance was a main goal when implementing Ethos, with the aim of reducing the cost, and thus incentivizing the use, of proof checking. To this end, Ethos implements multiple optimization techniques, which we discuss below.

Related Work. The main inspiration for Ethos comes from two related systems. The first is Alethe [13], a proof format and calculus for SMT supported by the Carcara checker [1]. In contrast to Eunoia, Alethe defines a fixed calculus, described in English in a reference document [5]. The SMT solver SMTInterpol [11] uses a custom proof format similar to Alethe. The second inspiration is LFSC and its proof checker [14]. Like Eunoia, LFSC is a logical framework in which it is possible to define logics and proof rules. In contrast to Eunoia, its syntax does not resemble SMT-LIB and describes proofs as terms. SMT developers find it unintuitive and difficult to use, and as a consequence, it was not widely adopted.

Both LFSC and Eunoia are inspired by other logical frameworks, starting with Edinburgh LF [**?**]. Among them, Dedukti [2] stands out as explicitly designed for proof exchange. There are multiple Dedukti models of proof systems from SMT solvers and other automated reasoning systems [7]. RARE is a DSL for declaring proofs based on rewriting [**?**]. Eunoia is partially inspired by RARE and can be seen as a generalization of its functionality to arbitrary proof rules.

Beyond those mentioned so far, there a number of additional approaches for the definition of proof formats. The DRAT format [16] provides a unified proof format that enables SAT solvers to easily generate proofs for most state-of-the-art solving techniques in a manner that can be efficiently checked. As a result, most contemporary SAT solvers support DRAT. Unfortunately, due to the heterogeneous nature of SMT solving, it seems doubtful that a format with a fixed proof calculus could provide similar benefits for SMT. The eDRAT

format [10] uses DRAT together with unjustified theory lemmas, which must be checked separately. It does not support proofs for preprocessing.

The TPTP world [15] provides a proof format for automated theorem provers (ATPs). However, it does not prescribe a way to define proof rules, and theorem provers choose their own. A notable consequence of this design choice is that, since TPTP proofs rules can be quite high level, a trusted ATP, as opposed to a simpler checker, is needed to prove individual rule applications correct.

2 Eunoia

In this section, we provide an overview of Eunoia through an example containing a signature, proof rules, and a proof. The Ethos user manual [12] contains a a complete introduction to Eunoia.

Theory Signatures. A Eunoia signature starts with a declaration of relevant types and constants (in the HOL sense), as shown in the following Eunoia code. The symbols `Bool`, `true`, `false`, `->` are not declared because they are built in.

theory.eo

```
(declare-const Int Type) (declare-consts <numeral> Int)
(declare-const not (-> Bool Bool))
(declare-const and (-> Bool Bool Bool) :right-assoc-nil true)
(declare-parameterized-const = ((A Type :implicit)) (-> A A Bool))
(declare-const BitVec (-> Int Type))
(declare-parameterized-const concat ((n Int :implicit) (m Int :implicit))
  (-> (BitVec n) (BitVec m) (BitVec (eo::add n m))))
```

Eunoia contains several builtin syntactic categories. It is up to the user though to give them a type. In the example, the `declare-consts` command tells Ethos that every numeral (`0`, `1`, `-1`, ...) is an `Int`. This makes it possible to change the type of numerals depending on the SMT-LIB logic of interest. For example, in `QF_LIA`, numerals have type `Int`, while in `QF_LRA` they have type `Real`. The defined symbols do not carry any semantic information, beyond their type. They are simply the constructors of a term language on which the proof rules operate.

The `:right-assoc-nil` annotation for the constant `and` is an extension of SMT-LIB's `:right-assoc` annotation. It indicates that `and` can be used as a variadic operator with `true` as a neutral (or *nil*) element.[6] Like SMT-LIB's `:right-assoc`, it allows applications of the operator to more than two arguments as syntactic sugar for a nested application. However, it also allows applications of the form `(and a)`, treated as a shorthand for `(and a true)`. This implies that `(and a b c)` desugars to `(and a (and b (and c true)))` instead of `(and a (and b c))`, which would be the result with the `:right-assoc` annotation. One motivation for `:right-assoc-nil` is that when operators annotated with `:right-assoc` are indeed used as variadic operators, which is common in SMT solvers, ambiguity arises: both `(and a b c)` and `(and a (and b c))` desugar to the same term. In contrast, with `:right-assoc-nil`, the latter term desugars to `(and a (and (and b (and c true)) true))`. As we will see, Eunoia has additional features to facilitate working with such operators.

[6] Eunoia also supports the specular annotations `:left-assoc` and `:left-assoc-nil`.

In Eunoia, it is also possible to define dependent types and parametric constants. In the example above, we first declare the usual polymorphic equality constant with an implicit type argument `A`. Afterwards, we declare a size-indexed type for bit-vectors and the bit-vector concatenation constant. The return type of `concat` is defined by a computation performed by the builtin operator `eo::add` for adding two numbers. Eunoia has a large library of such operators. Some, like `eo::add`, are primitive operators that implement fundamental functionality; others are defined as Eunoia programs built from primitive operators.

Proof Rules. After declaring the *theory signature* we can declare a *proof signature*, which consists of proof rules and associated helper programs which can be used to implement side conditions. A file inclusion command allows us to isolate the theory signature in its own file. Proof rules follow the theory signature.

rules.eo

```
(include "theory.eo")
(declare-rule contra ((F Bool)) :premises (F (not F)) :conclusion false)
(program andE-rec ((F Bool) (Fs Bool :list) (n Int)) :signature (Bool Int) Bool
 (((andE-rec         F  0) F)
  ((andE-rec (and F Fs) n) (andE-rec Fs (eo::add n -1)))))
(declare-rule andE ((Fs Bool) (i Int))
    :premises (Fs) :args (i) :conclusion (andE-rec Fs i))
```

Every rule has a name (here `contra` and `andE`) and a list of parameters that appear in its declaration. The declaration then lists patters for the rule's premises. When checking a proof step that invokes rule `contra`, the proof checker applies pattern matching to two previously derived formulas to find a matching substitution for the parameter `F`. This substitution is then applied to the conclusion, which in general is also a pattern, although in this rule it is the `false` constant. Rule `andE` illustrates the use of computations in proofs. The rule implements the elimination of variadic applications of `and` using a recursive program, `andE-rec`, which returns the n-th argument of a conjunctive formula.

Programs have a name, and a list of *local* parameters that serve as pattern variables. They also have a type signature that declares the argument types (here `Bool` and `Int`) and the return type (`Bool`). As in many functional programming languages, a program is defined by a list of cases that are considered in order. Abstractly, each case is a pair (p, b), where p is a list of term patterns for the program inputs and b is the case's body. Each local parameter in b must appear at least once in p (non-linear patterns are supported). A program application is evaluated by a Eunoia checker by matching the application's arguments against each pattern list. When this succeeds, it applies the matching substitution to the body, evaluates it, and returns the resulting term. If no case matches, the checker raises an error and rejects the entire proof.

The example also demonstrates another feature of Eunoia's handling of variadic operators: the `:list` annotation. Annotating an operator with `:list` suppresses desugaring if it occurs in the tail position of a variadic operator. Without it, the pattern `(and F Fs)` would desugar to `(and F (and Fs true))` which, however, would match only with binary conjunctions—of the form `(and t1 t2)`.

Proofs. We can now express a proof using these rules. With the `reference` command a proof can be linked to an SMT-LIB problem (e.g., `problem.smt2`) to check that all declared constants and free assumptions appear in the problem.

```
(reference "problem.smt2") (include "rules.eo")
(declare-const F Bool)
(assume a1 (and F (not F)))
(step s1 (F) :rule andE :premises (a1) :args (0))
(step s2 (not F) :rule andE :premises (a1) :args (1))
(step s3 (false) :rule contra :premises (s1 s2))
```

The proof above has one assumption and three derivation steps. The first introduces the assumption `(and F (not F))` and names it `a1`. The next two apply rule `andE` to `a1`, deriving formulas `F` and `(not F)`, respectively. The final command concludes `false` from the formulas derived by the previous two steps. The second argument of `step` is its expected conclusion and is optional. If provided, as in this case, it is matched during proof checking against the actual conclusion, generating an error in case of a mismatch. This is useful for debugging.

3 Proof Checking with Ethos

In this section, we describe at a high level some of the low-level implementation details of Ethos. We first focus on a description of the general process Ethos uses to check proofs, and then discuss some low-level aspects.

Proof Checking Model. Ethos is designed to be, first and foremost, a *fast* proof checker. This means that it omits some checks that might be *convenient* for the signature programmer but are not *essential*, i.e., their absence cannot lead to Ethos accepting invalid proofs. For example, Ethos does not fully check the types of programs when they are defined. Instead, it checks the monomorphic types of the ground terms that appear when checking a proof.

For Ethos, checking a proof amounts to evaluating a proof term. Every rule is in fact a program with only one case. For example, the `contra` rule from above is internally translated to the following program.

```
(declare-const Proof Type)
(declare-const pf (-> Bool Proof))
(program contra ((T Bool)) :signature (Proof Proof) Proof
  ((contra (pf T) (pf (not T))) (pf false)) )
```

The earlier proof corresponds roughly to the term `(contra (andE A 0) (andE A 1))`, where `A` is the assumption `(and F (not F))`. This term is considered a correct proof if Ethos can evaluate it to `(pf false)` without error. Note that the symbols `Proof` and `pf` are *not* part of the front-end syntax of Eunoia.

Ethos implements a function $\texttt{eval}(t, \sigma)$, where t is a Eunoia term and σ is a grounding substitution over the parameters of t. It evaluates t bottom up, replacing each parameter x in it with $\sigma(x)$. When `eval` encounters a term $(c\ u_1\ \cdots\ u_n)$, where c is a program with cases $[(p_1, b_1), \cdots, (p_m, b_m)]$, it invokes

a helper `match`$(p_i, u_1 \ \cdots \ u_n)$ on each pattern p_i of c in order. If this function succeeds, it returns a substitution σ', and Ethos evaluates `eval`(b_i, σ').

To ensure the conclusion of proof steps are not mistyped, Ethos can compute the type of parameter-free terms. This task is easier than general type checking because for the theories considered by Eunoia, all literals and constants have monomorphic types. Hence, computing types of applications amounts to pattern matching between the domain types and the monomorphic argument types. This approach has the benefit of being easy to implement, and optimize, at the expense of somewhat reduced statically checked guarantees.

Implementation. Ethos is written in C++, and currently has 9,852 lines of code. Terms are stored using a data structure with hash-consing (each unique AST is allocated exactly once) and reference counting on expressions. Number-like SMT-LIB literals (numerals, rationals, bit-vector constants) are stored in the AST using GMP arbitrary precision datatypes. Most builtin Eunoia operators on these literals translate directly to GMP functions. Ethos also implements the SMT-LIB policy for Unicode strings.

To accelerate proof checking, Ethos caches evaluation results. As mentioned, the evaluator takes as arguments a term to evaluate and a substitution (i.e. a context) mapping its free parameters to ground terms. The evaluator is implemented as a directed-acyclic graph traversal over terms. Since the evaluation primitives in Eunoia are stateless, Ethos can cache intermediate results in this process. This means that the result of invoking a side condition on a particular set of arguments is only computed once. However, this caching is not global, meaning that invoking the same computation on two independent proof steps will result in computing the result again. Ethos also evaluates parameter-free terms in rule and program bodies during parsing.

Since term construction is often the bottleneck, the core evaluation primitives on lists are optimized at a low-level to minimize the number of terms that must be reallocated. For example, `eo::list_erase_all`, which removes all occurrences of a particular element from a list, will reuse the tail of the list after the last occurrence of the element. To ensure the correctness of the optimized implementation, we cross-check these optimizations against a simpler reference

	Verified			**Holey**					
	Search		**Gen.**	**Search**		**Gen.**	**no Gen.**	**TO**	**Other**
Alethe	19,502	90,908	13,869	867	7,172	1,192	23	26	522
LFSC	454	36	42	19,275	81,459	223,989	23	126	1,062
CPC	20,813	85,868	93,157	0	0	0	23	43	61
LFSC	705	36	221,911	136,482	229,300	221,911	468	460	2,133
CPC	138,912	657,911	219,267	0	0	0	473	617	247

Table 1. Evaluation on benchmark sets (1) and (2). Proofs are either fully checked (**Verified**), or have holes (**Holey**). The table shows the sum of the time in seconds to **Generate** a proof and to **Check** it. The last three columns count failures: no proof was generated, checking timed out, any other checking error occurred.

implementation of the list operators as ordinary Eunoia programs. The Ethos regression set contains tests that ensure that the builtin evaluation matches the results of these reference implementations.

Finally, Ethos can be extended with plugins. These are modules that register callbacks with the main loop of Ethos. The callbacks notify the plugin when an event occurs, e.g., when a proof step is parsed. The plugin system can be used to implement custom external proof checkers, or converters to other proof formats.

4 Evaluation

The cvc5 solver supports three proof checking pipelines. It can produce proofs in the Alethe format which can be checked by Carcara. It can produce proofs in a calculus modeled in LFSC and checkable by the LFSC checker. Finally, it can produce proofs in its native calculus, called CPC, which is modeled in Eunoia and can be checked by Ethos. However, the support for cvc5 features differs significantly in these three pipelines. The Alethe proof format only supports uninterpreted functions, linear arithmetic, and quantifiers, and the LFSC calculus omits rules for many steps performed by cvc5. In those cases, the LFSC proof contains a step marked as `trust`. The Eunoia-based pipeline is the most complete. If cvc5 is run in *safe* mode, intended for users that require high assurance, the generated CPC proofs contains no *holes* (unjustified steps). An important factor for making this possible was the ease of writing proof rules in Eunoia.

To provide a perspective on the performance of Ethos, we compared the different proof production and checking pipelines of cvc5, and show that the pipeline that uses Ethos is competitive with the other two, while also supporting more features (Table 1). We used all non-incremental SMT-LIB benchmarks [**?**] without floating-points that do not have status `sat` or have an `inferredStatus` of `sat` in the SMT-LIB catalog [**?**]. To get a collection of benchmarks cvc5 could solve, we ran it for 60s in *safe mode* with proofs turned off.[7] This gave us a set of 161,188 benchmarks. We split this benchmarks set in two: (1) benchmarks in logics supported by Alethe, and (2) the rest. Set (1) contains 20,940 benchmarks. We then ran cvc5 for 600s on these sets with proofs turned on, and afterwards each checker on the generated proofs for another 600s. For Eunoia and Alethe, we used fine-grained proof rules for term rewriting steps, which are not supported by LFSC. This is the main reason for the large number of proof holes in LFSC proofs. Support for such rewrite rules is also experimental in Alethe. On commonly solved problems, Ethos is 6.25 times slower than Carcara.

As we gain experience using Ethos in practice, we expect to find more opportunities to improve its performance and usability. We also plan to focus on the support infrastructure around Ethos to make Eunoia easier to use. This includes the development of a standardized unit testing framework for Eunoia signatures, a documentation generator for proof rules, and a language server. One of our goals is to make Eunoia and Ethos robust and flexible enough to be a useful proof checking infrastructure for other SMT solvers and automated theorem provers.

[7] We used machines with Intel Xeon E5-2620 v4 CPUs an an 8 GiB memory limit.

References

1. Andreotti, B., Lachnitt, H., Barbosa, H.: Carcara: An efficient proof checker and elaborator for SMT proofs in the alethe format. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for Construction and Analysis of Systems (TACAS), Part I. Lecture Notes in Computer Science, vol. 13993, pp. 367–386. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_19
2. Assaf, A., Burel, G., Cauderlier, R., Delahaye, D., Dowek, G., Dubois, C., Gilbert, F., Halmagrand, P., Hermant, O., Saillard, R.: Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory. CoRR **abs/2311.07185** (2023). https://doi.org/10.48550/ARXIV.2311.07185
3. Baek, S., Carneiro, M., Heule, M.J.H.: A flexible proof format for SAT solver-elaborator communication. CoRR **abs/2109.09665** (2021), https://arxiv.org/abs/2109.09665
4. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
5. Barbosa, H., Fleury, M., Fontaine, P., Schurr, H.J.: The alethe proof format: An evolving specification and reference (2026), https://verit.gitlabpages.uliege.be/alethe/specification.pdf
6. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
7. Blanqui, F.: Translating libraries of definitions and theorems between proof systems (invited talk). In: Bertot, Y., Kutsia, T., Norrish, M. (eds.) 15th International Conference on Interactive Theorem Proving, ITP 2024, Tbilisi, Georgia, September 9-14, 2024. LIPIcs, vol. 309, pp. 2:1–2:1. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024), https://doi.org/10.4230/LIPIcs.ITP.2024.2
8. Bouchet, M., Cook, B., Cutler, B., Druzkina, A., Gacek, A., Hadarean, L., Jhala, R., Marshall, B., Peebles, D., Rungta, N., Schlesinger, C., Stephens, C., Varming, C., Warfield, A.: Block public access: trust safety verification of access control policies. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (eds.) ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020. pp. 281–291. ACM (2020). https://doi.org/10.1145/3368089.3409728
9. Gocht, S.: Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning. Ph.D. thesis, Lund University, Lund, Sweden (Jun 2022), available at https://portal.research.lu.se/en/publications/certifying-correctness-for-combinatorial-algorithms-by-using-pseu
10. Hitarth, S., Codel, C.R., Lachnitt, H., Dutertre, B.: Extending DRAT to SMT. In: Narodytska, N., Rümmer, P. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024. pp. 1–11. IEEE (2024). https://doi.org/10.34727/2024/ISBN.978-3-85448-065-5_8
11. Hoenicke, J., Schindler, T.: A simple proof format for SMT. In: Déharbe, D., Hyvärinen, A.E.J. (eds.) Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022. CEUR Workshop Proceed-

ings, vol. 3185, pp. 54–70. CEUR-WS.org (2022), https://ceur-ws.org/Vol-3185/paper9527.pdf
12. Reynolds, A., Schurr, H.J., Soldevila, M., Tinelli, C.: The ethos user manual (2026), https://github.com/cvc5/ethos/blob/main/user_manual.md
13. Schurr, H., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: Towards a generic SMT proof format (extended abstract). In: Keller, C., Fleury, M. (eds.) Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021. EPTCS, vol. 336, pp. 49–54 (2021). https://doi.org/10.4204/EPTCS.336.6
14. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. Formal Methods Syst. Des. **42**(1), 91–118 (2013). https://doi.org/10.1007/S10703-012-0163-3
15. Sutcliffe, G.: Stepping stones in the TPTP world. In: Benzmüller, C., Heule, M.J.H., Schmidt, R.A. (eds.) Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14739, pp. 30–50. Springer (2024). https://doi.org/10.1007/978-3-031-63498-7_3
16. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing (SAT). Lecture Notes in Computer Science, vol. 8561, pp. 422–429. Springer (2014)