A light-weight proof checker for TSTP refutations

Melanie Taprogge[1,2,*,†], Happy Sariyanto[1,†] and Alexander Steen[1,†]

[1]*Institute of Mathematics and Computer Science, University of Greifswald, Greifswald, Germany*
[2]*Université Paris-Saclay, ENS Paris-Saclay, LMF, CNRS,* **INRIA**, *France*

Abstract

The heterogeneity of proof outputs produced by different state-of-the-art automated theorem proving systems presents a considerable challenge for their efficient and uniform verification. In this paper, a proof checker called Nörgler is introduced that builds upon and extends the established approach pioneered by GDV. It supports checking propositional, (untyped and typed) first-order, and higher-order refutations represented in TSTP. For increasing flexibility and performance, Nörgler can parallelize proof checking tasks and incorporate model finders (for rejecting proofs). We evaluate the efficiency of Nörgler's design and implementation, and show that it outperforms GDV in terms of success rate and verification time on a benchmark drawn from the TSTP solution library.

Keywords

automated theorem proving, proof checking, TSTP

1. Introduction

Automated Theorem Proving (ATP) systems are designed to fully automate the proof search within a specific underlying logic, most commonly *first-order logic* (FOL), with prominent examples for established provers being Vampire [1] and E [2] (among many others). More recently, systems dedicated to reasoning in *higher-order logic* (HOL) have received increased attention, among them Leo-III [3], Satallax [4] and extensions of established FOL systems like E and Vampire. While ATPs do not generally provide intrinsic correctness guarantees (despite being based on sound and complete calculi), many established ATP systems have earned the community's confidence through a long history of reliable performance.[1] However, such empirical trust is insufficient as these tools are under active development (so new bugs may be introduced at any time), and are generally very complex (so existing bugs may only show rarely).

Proof verification is hampered by the differences in base logics, calculi, strategies, and reporting formats across ATP systems. Machine-checkable proof formats for ATPs have been proposed, both in the shape of new formats (e.g., [6]), and as extensions of the existing standard output format *TSTP* [7]. While the latter aims to–in the long term–enforce more fine-grained proofs using a shared set of calculus rules [8], none of the current state-of-the-art provers follow such standards (at least in full), rendering uniform proof-replay approaches impossible (cf. related work below).

A strategy pioneered by the GDV system is so-called *semantic verification* [9]. In this setting, a single general-purpose checker is, in principle, capable of verifying all proofs expressed in a given format. The checker first validates global structural properties of the proof and then establishes correctness step by step by re-proving each inference using a trusted system. In contrast to proof replay, this process involves new proof search, and success is not guaranteed, as some inference steps may exceed the capabilities of the trusted prover.

Practical Aspects of Automated Reasoning (PAAR) 2026, as part of the Federated Logic Conference (FLoC) 2026, Lisbon, Portugal, July 25, 2026

*Corresponding author.
†These authors contributed equally.
melanie.taprogge@stud.uni-greifswald.de (M. Taprogge); happy.sariyanto@stud.uni-greifswald.de (H. Sariyanto); alexander.steen@uni-greifswald.de (A. Steen)
https://melanie-taprogge.github.io (M. Taprogge); https://www.alexandersteen.de (A. Steen)
0009-0008-1906-6831 (M. Taprogge); 0009-0008-0342-7022 (H. Sariyanto); 0000-0001-8781-9462 (A. Steen)

[1]A notable example being Otter [5], for which no bugs have been found yet. This is, of course, also due to the fact that Otter is not being developed anymore.

Following the semantic verification approach, we introduce Nörgler, a proof checker for TSTP refutations. Nörgler builds on established techniques from GDV, and extends them with additional features and improved implementation designs. In particular, Nörgler incorporates additional structural checks missing in GDV, it invokes countermodel search to detect incorrect proof steps, and improves efficiency through multi-core parallelization. Furthermore, it provides fine-grained control over the strictness with which formatting conventions are enforced, making it suitable both for scenarios where robustness against imperfect prover output is required, as well as for settings that demand stricter adherence to proof standards. Nörgler supports checking TSTP refutations in all major TPTP dialects (FOF, TFF, THF), though in this paper we focus on FOF refutations as representative.

The rest of the paper is structured as follows: Related work is discussed next. Sect. 2 summarizes standard preliminaries. Sect. 3 then describes general principles of proof verification and specific checks applicable to TSTP refutations. Sect. 4 introduces Nörgler's architecture, its checks and its differences to GDV. Important implementation details of Nörgler are detailed in Sect. 5. In Sect. 6, we evaluate the performance of Nörgler across various configurations, and compare its performance to that of GDV wrt. FOL proofs in the TPTP FOF dialect. Finally, Sect. 7 concludes and discusses future work.

Related work. Verification is tackled in various ways across different domains. In SAT solving, the community has converged on a concise, well-defined set of inference rules (e.g., DRAT/LRAT [10, 11]). Tools utilize these rules to generate fine-grained proof certificates, which enable verification via so-called *proof replay*, i.e. the precise repetition of each step by a trusted, minimal, and sometimes formally verified checker (e.g. [11]).

Verification in SMT is more challenging, as these systems integrate theory-specific solvers that require a broader variety of inference rules. Consequently, SMT solvers often produce proofs that take the form of execution traces rather than formal logical derivations. While shared proof formats with a common rule set have been proposed, such as the Alethe format [12], they have not yet achieved universal adoption. To bridge this gap, proof elaboration tools like Carcara [13] can process traces, decomposing them into known rules and making implicit steps explicit.

Another approach pursued within the ATP community is to extend systems with the capability of outputting proofs in a format that can be verified externally, for instance via translations to the Dedukti framework [14] (see, for example, ZenonModulo [15], or, more recently, Vampire [16]). While this yields fully formal proofs and thus supports a high degree of trust, the implementation effort is substantial and introduces overhead in proof generation, as the resulting certificates must be highly detailed [17, 16].

Hybrid approaches combine both semantic verification and logical framework techniques. Tools such as Extrakto [18] and the extension of GDV, GDV-LP [19], follow the re-proving paradigm but employ provers that can themselves produce verifiable proof certificates. This makes it possible to reconstruct a fully formal proof from the individually verified steps, providing an additional layer of assurance.

The emerging ProoVer[2] competition aims at evaluating and fostering the capabilities of general-purpose TSTP checkers.

2. Preliminaries

2.1. First-Order Logic

The language of first-order logic (FOL) is defined as usual (e.g., as in [20]). Given a signature Σ, an abstract syntax for FOL is given by:

$$\varphi, \psi ::= p(t_1, \ldots, t_n) \mid t_1 = t_2 \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \to \psi \mid \varphi \leftrightarrow \psi \mid \forall x.\, \varphi \mid \exists x.\, \varphi$$

where $p \in \Sigma$ is an n-ary predicate symbol, and all the t_i are well-formed terms. Notions such as logical consequence, interpretations, and models in FOL are defined the standard way.

[2] See https://proover-competition.github.io/competitions/2026/.

```
fof(main, conjecture, ~ ((? [X]: (~ ((~ (p1(X)) | p1(X)))))),
        file('LCL662+1.001.p',main)).
fof(c0, negated_conjecture, ~ (~ ((? [X]: (~ ((~ (p1(X)) | p1(X))))))),
        inference(assume_negation,[status(cth)],[main])).
fof(c1, negated_conjecture, ~ (~ ((? [X]: (~ ((~ (p1(X)) | p1(X))))))),
        inference(fof_simplification,[status(thm)],[c0])).
fof(c2, negated_conjecture, (? [X]: ((p1(X) & ~ (p1(X))))),
        inference(fof_nnf,[status(thm)],[c1])).
fof(c3, negated_conjecture, (? [X2]: ((p1(X2) & ~ (p1(X2))))),
        inference(variable_rename,[status(thm)],[c2])).
fof(c4, negated_conjecture, (p1(skolem0001) & ~ (p1(skolem0001))),
        inference(skolemize,[status(esa)],[c3])).
fof(c5, negated_conjecture, p1(skolem0001),
        inference(split_conjunct,[status(thm)],[c4])).
fof(c6, negated_conjecture, ~ (p1(skolem0001)),
        inference(split_conjunct,[status(thm)],[c4])).
fof(c7, plain, $false,
        inference(resolution,[status(thm)],[c6,c5])).
```

Figure 1: A correct TSTP FOF proof by refutation for TPTP problem LCL662+1.001 by PyRes.

During proof derivations, modifying steps such as Skolemization [21, 22, 23] may be applied. The aim of Skolemization is to eliminate existential quantifiers by replacing their bound variables by Skolem terms that depend on all universally quantified variables in their scope; a formal definition is left out.

2.2. TPTP and TSTP

The TPTP World is an infrastructure supporting ATP development and employment [24]. The TPTP problem library [25] documents logical problems while solutions found by different ATPs are collected in the TSTP solution library [7, 26]. Problems and solutions are written in TPTP language that is built on so-called annotated formulae of the form *language*(*name*, *role*, *formula*, *source*, *useful_info*). ending in a dot. The *language* denotes the logic that is being used, including `fof`, `tff`, and `thf` for TPTP's first-order form (FOF), typed first-order form (TFF), and typed higher-order form (THF), respectively. We only focus on FOF in this work. Each annotated formula has its own *name*. The *role* is the role of the *formula*, such as `axiom` or `conjecture`. The *formula* is a logical formula in TPTP syntax, following Prolog conventions. The predefined identifiers `$true` and `$false` represent verum and falsum, respectively. The *source* and *useful_info* are optional in general. For derivations, the *source* is either a file record, or an inference record, and refers to the origin of the given *formula*. If the given *formula* is from a *problem_file*, a file record of the form `file`(*problem_file*, *formula_name*) is given. The *formula_name* refers to the *name* of the formula used in the *problem_file*. If the given *formula* is inferred from previously used parent formulae in the derivation, an inference record of the form `inference`(*rule*, *inference_info*, *parent*) is given. The *rule* used during the inference is given in shorthand as well as a list of the *parent* formulae. The second argument is a list of useful *inference_info*, which can be either the semantic relation of the *formula* to its parent, indicator of special reasoning theories or types of inference. Semantic relationships are given in a status record of the form `status`(*relation*). For example, a *relation* can be (`cth`) `thm` if the (negated) *formula* is a logical consequence of the given *parent*s, or `esa` if the given formula is equisatisfiable with the given single *parent*. An example TSTP-compliant proof can be seen in Fig. 1. This proof was generated by PyRes [27] for the TPTP problem LCL662+1.001, and translated to FOF, and is part of the benchmark described in section 6.

To report the status of logical data in a standardized way, the TPTP offers the SZS ontology [28, 29, 30]. In particular, the SZS success ontology is used to report on the status of the relationship between axioms and conjecture. For example, given a problem with a set of axioms Φ and a conjecture φ, if all models of Φ are also models of φ, the tuple describes a `Theorem` (THM). If all models of Φ are models of $\neg\varphi$, then a `CounterTheorem` is given (CTH). If some models of Φ are also models of $\neg\varphi$, the given problem is `CounterSatisfiable`(CSA). If no interpretation is a model of Φ, the problem possesses

`ContradictoryAxioms` (CAX). If no SZS success status can be established, the SZS no-success ontology is used to report the reason. This can for example be due to a `Timeout` (TMO) or because the prover `GaveUp` (GUP).

3. Proof Verification

Most state-of-the-art ATPs employ proof by refutation, negating the conjecture to derive a contradiction [31]. We therefore focus on verifying proofs from such systems, specifically resolution-based proofs that terminate in `$false` in the TPTP syntax. In the following, we characterize the properties that can generally be checked in the verification of such proofs in the TSTP format.

3.1. Checks

In the following, $S = (s_1, ..., s_n)$ denotes a sequence of annotated formulae comprising a TSTP proof. Likewise, $P = (p_1, ..., p_m)$ denotes the problem proved by S. $G_S = (V, E)$ is the representation of S as a directed graph, where $V = \{v_1, ..., v_n\}$ is the set of nodes in G_S and $E = \{e = (v_i, v_j) : v_i \to v_j\}$ the set of directed edges. Each node $v_i \in V$ corresponds to an annotated formula $s_i \in S$ via the bijection $\pi : V \to S$. While *parent* in an annotated formula is a set of *names* of the annotated formulae, we use $pa_S(s_i)$ to signify the set of annotated formulae that correspond to the *name*s in the set *parent* of s_i. For a node v_i, $pa(v_i) = \{v_{i_1}, \ldots, v_{i_k}\}$ then denotes the set of its predecessors, i.e., the nodes v_j such that $(v_j, v_i) \in E$. By construction, the nodes in $pa(v_i)$ are exactly the nodes corresponding to the formulae in $pa_S(s_i)$. Lastly, we use $\mathrm{name}(s_i)$ to map the annotated formula s_i to its *name*. Within this framework, the proof verification checks are summarized in Table 1, and described in detail below.

Table 1
A summary of all named checks, their mnemonics are given in the left column.

Abbreviation	Performed check
(EIF)	proof **e**nds **i**n **f**alse
(IAA)	**i**nference parents **a**re **a**cyclic
(NUF)	**n**o **u**nused **f**ormulae
(NAU)	**n**ames **a**re **u**nique
(PAA)	**p**rovides **a**n **a**nnotation
(RIA)	**r**ole **i**s **a**dmissible
(IPE)	**i**nference **p**arents **e**xist
(NSC)	**n**o **s**tatus **c**th
(ONU)	**o**nly **n**egated conjecture **u**sed
(DCP)	**d**erivation of **c**ontradiction or **p**arent

(a) Global and local structural checks.

Abbreviation	Performed check
(ICN)	**i**s **c**orrectly **n**egated
(ICE)	**i**s **c**orrectly **e**ntailed
(ICS)	**i**s **c**orrectly **s**kolemized
(APS)	**a**xioms of **p**roblem are **s**atisfiable
(PIS)	**premises** of **i**nference are **s**atisfiable
(COO)	**c**opy **o**f the **o**riginal
(NAA)	**n**o **a**dditional **a**xioms

(b) Logical and faithfulness checks.

Global structure. Global structural checks verify properties of the proof tree as a whole.

Check **(EIF)** verifies that the derivation **e**nds **i**n **f**alse , i.e., that the *formula* of $\pi(v)$ is `$false`, where $\pi(v) = s_n$. Also, G_S must be an acyclic directed graph. This is checked by **i**nference parents **a**re **a**cyclic **(IAA)**, i.e., for all possible paths $(v_{i_0}, ..., v_{i_l})$ in G_S of all possible lengths $l \in \{1, ..., n-1\}$ it holds that $v_{i_0} \neq v_{i_l}$. It is also possible to demand that there are **n**o **u**nused **f**ormulae **(NUF)**. This means that for all $s_i \in S \setminus \{s_n\}$ there exists $s_j \in S$ for which $s_i \in pa_S(s_j)$. The **(NAU)** check ensures that formula **n**ames **a**re **u**nique, i.e., name(.) is an injection.

Local structure. Local structural checks are performed on the structure of each annotated formula $s_i \in S$ individually.

An annotated formula needs to **p**rovide **a**n **a**nnotation **(PAA)**, a **r**ole that **i**s **a**dmissable **(RIA)**, as well as a list of **i**nference **p**arents that actually **e**xist earlier in the proof **(IPE)**. For **(PAA)**, this means that for the given s_i an inference record of the form `inference(`*rule*`,status(`*relation*`),`*parent*`)` or a file record of the form `file(`*problem_file*`,`*formula_name*`)` must be given as *source*, with *relation* $\in \{$`thm`, `cth`, `esa`$\}$. To pass the **(RIA)** check, the *role* of s_i needs to be either `axiom`, `conjecture`, `negated_conjecture`, `plain`, `definition`, or `type`. **(IPE)** verifies that for each *name* n in the *parent* record of s_i, there exists a $s_j \in S$, $j < i$, for which $name(s_j) = n$ holds.

In refutational proofs, it is also necessary that, apart from the negation of the conjecture, **n**o step has the **s**tatus `cth` **(NSC)**, which holds iff any s_i with status record `status(cth)` has the *role* `negated_conjecture` and $pa_S(s_i) = \{s_j\}$ with the *role* of s_j being `conjecture`. Moreover, **o**nly the **n**egated version of the conjecture is **u**sed as a parent to any step except the negation of the conjecture **(ONU)**, i.e., if s_i has the *role* `conjecture` and $s_i \in pa_S(s_j)$ for any other $s_j \in S \setminus \{s_i\}$, then s_j must be of the *role* `negated_conjecture`.

Another possible check is to verify that the derived formula is either a **d**erivation of a **c**ontradiction (`$false`) or a **p**arent for another inference **(DCP)**. This means either the *formula* φ of s_i is `$false`, or $\exists(s_j \in S \setminus \{s_i\}).s_i \in pa_S(s_j)$.

Logical checks. Logical checks are performed to ensure the correctness of derived formulae in each step of the proof.

If s_i is the negation of the conjecture, it can be verified that the conjecture **i**s **c**orrectly **n**egated **(ICN)**. This means, if $pa_S(s_i) = \{s_j\}$ with $i \neq j$ and the *role* of s_j is `conjecture`, it holds for the *formula* φ_i of s_i and the *formula* φ_j of s_j that $\varphi_i = \neg\varphi_j$. If s_i is a `thm` inference, then it can be verified that the step **i**s **c**orrectly **e**ntailed **(ICE)**, i.e., for *formula* φ_s of s_i and the set of formulae Φ consisting of the *formula* of each parent in $pa_S(s_i)$, $\Phi \vdash \varphi_s$ holds.

It can also be checked whether a formula **i**s **c**orrectly **s**kolemized **(ICS)**. This applies for a formula s_i of `status(esa)` and with $pa_S(s_i) = \{s_j\}$, if the quantifiers in the *formula* φ_j of s_j can be successively eliminated with a fresh Skolem symbol [3] such that it yields the *formula* φ_i in s_i.

Satisfiability checks can be performed as well. This includes the verification that the **a**xioms of the original **p**roblem are **s**atisfiable **(APS)** and the **p**remises of the **i**nference given in s_i are **s**atisfiable **(PIS)**. Specifically, **(APS)** means that the set of axioms consisting of the *formula*e of each annotated formula in P with *role* `axiom` is satisfiable. Similarly, **(PIS)** means that for s_i the set of formulae consisting of all *formula*e of annotated formulae in $pa_S(s_i)$ is satisfiable.

Faithfulness checks. Faithfulness checks guarantee that S actually represents a proof of P. One possible check is to verify that each formula is a direct **c**opy **o**f the **o**riginal formula **(COO)**. That means, for each $s_i \in S$ that contains a file record `file(`*source*`,`*formula_name*`)` with problem P as *source*, there exists $p_j \in P$ such that (i) $name(p_j) =$*formula_name* and (ii) the *formulae* of s_i and p_j are syntactically equal. Another possible check is ensuring that **n**o **a**dditional **a**xioms were introduced **(NAA)**, i.e. for all $s_i \in S$ with *role* `axiom`, s_i provides a file record such that **(COO)** holds.

3.2. Correctness Criteria

A combination of the checks introduced above guarantees the refutational correctness of a proof: **(ICN)**, **(ONU)**, **(EIF)**, and **(NSC)**. The semantic correctness of the derivations is verified by **(ICE)** and **(ICS)**, while other structural correctness properties are ensured by **(IPE)** and **(IAA)**. Additionally, the checks **(NAU)**, **(PAA)**, and **(NAA)** guarantee that the information necessary for verification is included. A proof is considered *verifiably correct* if all of these checks succeed. If a problem file is provided and the checks **(COO)** and **(NAA)** succeed, the proof is considered a *verifiably correct proof of the given problem.*

[3] Note that the freshness of Skolem variables is not strictly necessary for the correctness of these steps (see, for instance [32]). In verification efforts, it is however a common requirement.

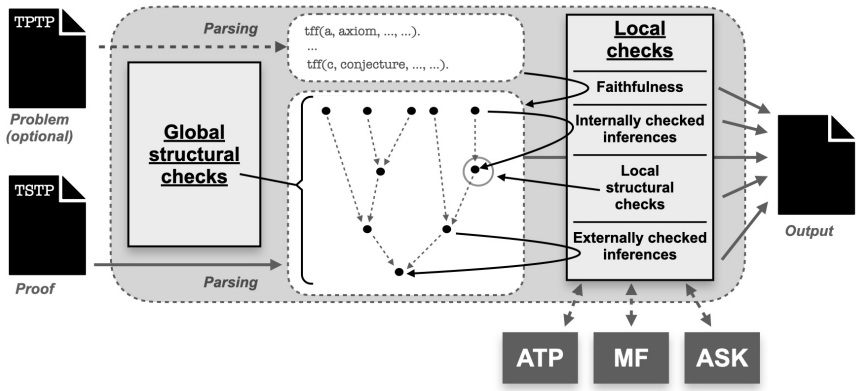

Figure 2: High-level system design of Nörgler. The light gray region denotes the Nörgler system boundary, and white boxes show representations of the processed inputs. External components are shown in dark gray. Gray arrows indicate the flow of information and results, solid lines denote mandatory paths, while dashed lines indicate optional components. Dark arrows visualize the application areas of specific verification checks.

The remaining checks ensure that no superfluous formulas are included in the proofs (**(NUF)** and **(DCP)**) and that axioms and premises are satisfiable–as derivations are otherwise trivially verifiable (**(APS)** and **(PIS)**). A proof that fulfills these checks is considered to be *informative*.

4. System Design

The high-level system design is visualized in figure 2. Nörgler takes a mandatory proof file and an optional problem file as input. After parsing, the verification proceeds in two phases, applying a selection of the checks introduced in Sec. 3. The first phase performs checks on the global structural properties of the proof. The second phase focuses on local properties, including local structure, faithfulness to the original problem, and the correctness of the entailment. In this phase, different verification approaches are applied for the various kinds of derived formulas, signified by the different SZS statuses. While the negation of the conjecture (status `cth`) is verified internally, other inferences crucially rely on the use of external tools: For Skolemization (status `esa`), Nörgler employs the trusted tool ASk [33], while `thm` steps are handled by producing a reasoning problem representing the individual inference and invoking external ATPs, a procedure detailed in Sec. 4.2. As a complementing approach aimed at identifying incorrect steps, we apply model finders to check for the existence of countermodels, which we discuss in Sec. 4.3. To improve performance, we offer a variety of different modes that run the checks carried out during this phase in parallel (see Sec. 5.2).

Nörgler's output includes logging information followed by the SZS status, and details concerning the incorrect proof step (if any). If all applied checks are passed, the SZS status `VerifiedGood` is returned. The SZS statuses `Timeout` and `Unknown` from the SZS ontology are used for proofs that could not be verified due to timeouts or inconclusive results from external systems, respectively. In both cases, the correctness of the proof remains unknown. However, if Nörgler identifies an incorrect proof due to a failed check, the SZS status `VerifiedBad` is returned instead. These SZS status values have been adopted from the design of the ProoVer competition.[4]

4.1. Checks implemented in Nörgler.

All checks for refutational correctness of a proof (**(ICN)**, **(ONU)**, **(EIF)** and **(NSC)**) are performed by Nörgler. In the case of **(ICN)**, this is achieved via a comparison of the formula given in the problem to

[4]Note that the use of `Timeout` as an SZS status for checkers was introduced only after this work was accepted for publication. Therefore, throughout the rest of the paper, `Unknown` is used for proofs whose correctness remains unclear due to timeouts and other inconclusive results by provers.

an internally produced negation of the original conjecture, both normalized to prenex normal form. **(EIF)** is verified by identifying the last annotated formula of the proof and testing if its *formula* field is indeed `$false`.

To verify the semantic correctness of the derivation, checks **(ICE)** (for `thm` inferences) and **(ICS)** (for `esa` steps) are performed. For **(ICE)**, a proof obligation is constructed and passed to external ATPs and countermodel finders (see Sect. 4.2 resp. Sect. 4.3 below).

Equisatisfiability of two formulae is significantly more challenging to establish than logical consequence, and the verification of procedures that are merely satisfiability-preserving requires specialized approaches. To the authors' knowledge, Skolemization is one of the most common of such procedures generally employed in ATP systems. In practice, many systems record clausification procedures including Skolemization steps as a single step in their certificates, without providing details on the specific Skolemization applied. Verifying such steps is currently beyond the capabilities of Nörgler, as well as other existing checkers like GDV. However, performing an **(ICS)** check becomes possible when the Skolemization is provided in isolation, and when the introduced Skolem-variables are indicated in the annotations. In these cases, Nörgler checks that the variables are fresh and employs the trusted tool ASk to independently perform the Skolemization, subsequently comparing the result with the formula given in the certificate. Nörgler also checks for **(NAU)**, **(IAA)**, **(IPE)**, **(PAA)**, and **(RIA)**.

Together, the checks performed by Nörgler ensure that checked proofs are *verifiably correct*, but not that they are *informative*.

Where applicable, faithfulness checks **(COO)** and **(NAA)**[5] are performed. In this case, Nörgler-verified proofs are *verifiably correct proofs of the given problem*.

4.2. External ATP Systems

Trusted ATPs can be used to verify a single proof step that records a logical entailment during the **(ICE)** check. While this approach principally allows the use of any TPTP compatible ATP, the provers Nörgler currently supports are E and Vampire.

To invoke external systems, a verification problem based on the given proof step is constructed: Given an annotated formula with an inference record `inference(`*rule*`,status(thm),`*parent*`)`, the set of *formula*e Φ of the *parent* annotation is used as the set of axioms, and the *formula* ν of the annotated formula recording the current proof step is taken as the conjecture. An inference is then considered to be verified if the external ATP can derive SZS status `THM` for this proof obligation. If the external ATP finds the constructed problem to be countersatisfiable, and returns the SZS status `CSA`, the inference–and as a result the proof as a whole–is considered to be incorrect. If the external ATP returns an inconclusive SZS status (`TMO` or a `GaveUp`), verification fails, and the validity of the proof remains unclear (but a model finder might provide more information, if used).

While this method provides a generic check for steps derived by arbitrary systems irrespective of their implemented calculi, it relies on proof search, and is therefore undecidable.

4.3. Countermodel Finding

Model finding aims to identify an interpretation that either demonstrates the satisfiability, or the countersatisfiability of a set of formulae. If the former holds for a set of assumptions and the negated conjecture of a problem, the conjecture cannot be a logical consequence of the axioms.

This approach can also be used in proof verification when a proof step falsely claims to be a logical consequence of a set of parent formulae. In such cases, a countermodel can be given to disprove the false claim. External finite model finders are employed in Nörgler in this fashion. While every TPTP-compatible model finder can be used for this task, we currently only support Mace4 [34][6].

[5]Note that some systems can introduce so-called theory-axioms to model underlying assumptions. The parameter `-allow-prover-axioms` can be used to exclude these axioms from the **(NAA)** check in Nörgler.

[6]https://prover9.org/

The generation of the countermodel finding problem is analogous to the generation of the verification problem detailed in the previous section. Following previous notation, if a countermodel is found, the claim that ν is a logical consequence of Φ is false, and, thus, the proof is rejected by Nörgler, which then returns `VerifiedBad`. In case no countermodel is found, no statement can be given about the correctness of this proof step.

4.4. Comparison to GDV

4.4.1. Differences in System Design

There are some fundamental differences in system design between GDV and Nörgler: GDV builds upon the TPTP infrastructure, and invokes its external provers via the SystemOnTPTP interface, which can be accessed via the internet. Although this enables the use of a wide range of ATPs without requiring local installations, it adds considerable overhead and latency. A local installation of SystemOnTPTP can also be used, this however entails significant configuration overhead and resource requirements. Furthermore, possible SZS statuses differ in GDV and Nörgler. Both tools use `VerifiedGood` and `Unknown` but GDV returns the latter for cases where no verification status can be established (e.g., due to timeouts), as well as those where a proof step actually has been established to be false. In contrast, Nörgler returns `VerifiedBad` in the latter case [7]. On top of that, Nörgler offers a more fine grained control over permissiveness wrt. format adherence during the proof verification process via a wide selection of options. This is necessary as, according to observations, strict checks may result in proofs being rejected despite not necessarily being inherently wrong.

Currently, GDV offers no support for parallelization, but there seems to be (partly undocumented) support for employing model finders in **(ICE)** entailment checks.

4.4.2. GDV Checks Omitted in Nörgler

Nörgler largely builds upon the strategies and verification checks established by GDV. However, Nörgler introduces additional checks, and deliberately omits others:

GDV performs the checks **(NUF)** and **(DCP)**. However, these are merely redundancy checks and as long as redundant derivations and formulae do not break any of our other correctness criteria, we do not consider them to render the overall proofs incorrect. We therefore omit these checks.

Another difference is found in the handling of `esa` steps. Like Nörgler, GDV uses the trusted Skolemization tool ASk to perform an **(ICS)**-check. For other `esa` steps, or those lacking necessary annotations, GDV employs bi-directional entailment as a best-effort fallback. For two annotated formulae s_i and s_j, where one is the singular parent of the other, GDV invokes external provers in an attempt to verify $(\{\varphi_i\} \vdash \varphi_j) \wedge (\{\varphi_j\} \vdash \varphi_i)$ (with φ_i and φ_j denoting the *formulae* of steps s_i and s_j, respectively). If this check succeeds, GDV considers the step to be verified, but the failure of the check does not result in GDV rejecting the proof. We do not consider this test to be an appropriate general-purpose mechanism to verify `esa` steps and hence omit it. Instead, we generally consider `esa` steps that can not be verified using **(ICS)** to be unverifiable. In these instances, the remainder of the proof can still be verified using the appropriate flags.

GDV optionally also performs satisfiability checks **(APS)** and **(PIS)** by invoking model finders. This enables GDV to identify inferences that trivially hold as they have unsatisfiable assumptions. This however does not affect the validity of the derivation, we therefore omit these checks in Nörgler.

[7] At the time this paper was written, GDV did not output distinct SZS statuses for proofs that were shown to be false and for those where verification was inconclusive. However, this distinction has been added since the paper was accepted. In Sec. 6, we manually post-process GDV output to differentiate between false proofs and those with an unknown verification status. This does not affect the results.

4.4.3. Nörgler-specific Checks

Two checks introduced in Nörgler to ensure correct handling of the negation of the conjecture using `cth` steps are **(ONU)** and **(NSC)**. Without the former check, ATPs can claim a proof to be a refutation, but actually derive the contradiction from the conjecture directly, or even use both the conjecture and its negation, which trivially allows the derivation of a contradiction. A straightforward example for an incorrect refutation proof verifiable in the absence of these checks is given in Fig. 3.

```
%--------------------------------------------------------------------------------
fof(p1, conjecture, $false, file('false1.p',c)).
fof(p3, plain, $false, inference(illegal_inference, [status(thm)], [p1])).
%--------------------------------------------------------------------------------
```

Figure 3: An incorrect proof verifiable in the absence of **(ONU)**.

(NSC) on the other hand ensures that the negation of the conjecture is the only inference with status `cth`. Without this check, proofs can just derive the negation of any given axiom, or even of the negated conjecture, and checkers will merely check that the resulting steps actually are a countertheorem of their parent. An example of an incorrect proof exploiting the omission of **(NSC)** is given in Fig. 4.

```
%--------------------------------------------------------------------------------
fof(p1, axiom, $true, file('false2.p',c)).
fof(p2, plain, (~ $true), inference(illegal_inference, [status(cth)], [p1])).
fof(p3, plain, $false, inference(illegal_inference, [status(thm)], [p1, p2])).
%--------------------------------------------------------------------------------
```

Figure 4: An incorrect proof verifiable in the absence of **(NSC)**.

GDV currently implements neither of these checks, and verifies both of the above proofs [8]. While GDV provides the necessary checks to ensure that proofs, in our classification, are *informative*, the lack of the two checks prevents it from guaranteeing that proofs are *verifiably correct*.

5. Implementation

Nörgler is implemented in Scala, and available as an open-source stand-alone executable via GitHub[9] and Zenodo [35] under an MIT license[10]. For parsing the open-source `scala-tptp-parser` library [36] is used, and the open-source `tptp-utils` library [37] is integrated for various auxiliary processing steps (such as syntax format transformations, normal form calculations, and extended parsing functionalities). Otherwise, no external dependencies are required or employed.

5.1. Architecture

Nörgler implements a layered architecture for separating the actual executable (and its specific business logic and set-up) from the underlying *Library* package containing the individual checks (package *Checks*) and their orchestration (via the `Controller`). This is visualized in Fig. 5.

Individual types of proof checks are factored out as objects resp. classes and, following the *single-responsibility principle*, only assess one specific correctness aspect (e.g., by implementing the check **(IPE)**, **(COO)**, etc.). The `Controller` then bundles these checks into a meaningful order of invocation,

[8] At the time of writing this paper, GDV did not implement either of these checks and verified both of the above proofs. Since acceptance, and following discussions with the maintainers, GDV has added both checks and now likewise recognizes both proofs as incorrect.

[9] `https://github.com/leoprover/noergler`.

[10] See `https://opensource.org/license/mit`.

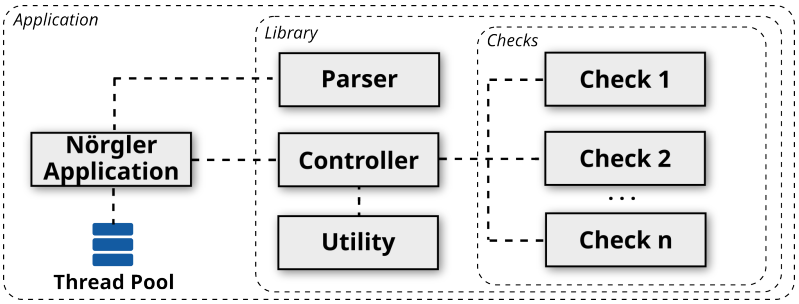

Figure 5: Top-level architecture of Nörgler. Layers (packages) are indicated by nested containers, rectangles are principal components, dashed lines indicate a used-by relationship.

it may decide to skip specific checks (e.g., if requested by the user using a command-line parameter), or run specific checks in parallel (see Sect. 5.2 for details).

An advantage of this design is that the individual checks are loosely coupled within the checker, so that checks can be altered or added with only marginal changes to the `Controller`. Moreover, every individual check and the complete check procedure of Nörgler can be re-used programmatically by other tools by using Nörgler as a library (i.e., without having to run the Nörgler executable itself via a command-line interface and reading/parsing its results). Of course, a drawback of such an approach is that the checking routine is likely to be less optimized as there is no information sharing between the individual checks, and some calculations may be done more than once. Framed positively, however, there is less surface for intricate bugs due to heavy optimizations, and the code is more understandable and maintainable.

Controller pseudo-code. A simplified version of the `Controller` is displayed in Listing 1.

```
1  proof := ...
2  problem := ...
3  checkGlobalProperties(proof)
4  foreach line in proof do:
5    checkLocalProperties(line)
6    switch annotation(line):
7      case inference(name,status):
8        checkInference(name, status, line, proof)
9      case file(filename,formulaname):
10       checkOrigin(line, problem, filename, formulaname)
11     otherwise: error
12   end
13 done
14
15 if some check fails: error
16 else if some check timed out: timeout
17 else: success
```

Listing 1: A simplified procedure implemented by Nörgler.

First, `checkGlobalProperties` assesses various global properties of the proof. Then, each line of the proof is checked sequentially via `checkLocalProperties` and, depending on its annotation, by additional inference-based checks (`checkInference`) or origin-based checks (`checkOrigin`). The concrete checks conducted in these subroutines are (referring to the verification check names from Sect. 3):

- `checkGlobalProperties`: **(EIF)**, **(NAU)**, **(IAA)**
- `checkLocalProperties`: **(RIA)**, **(IPE)**, **(PAA)**, **(ONU)**, **(NSC)**, **(NAA)**
- `checkInference`:
  - **(ICN)**, if the inference name is „*negate_conjecture*"

  - **(ICS)**, if the inference name is „*skolemize*"
  - **(ICE)**, if inference name is anything else and inference status is `thm` or `cth`
- `checkOrigin`: **(COO)**

Note that, by default, every method invocation is synchronous. In particular there is no parallelism involved and each check is executed sequentially. However, parallelism can easily be integrated in this procedure as described below. The procedure is designed to fail fast: If some check does not succeed, an exception is thrown so that the overall routine is stopped. Such exceptions are generally handled by reporting a failed verification attempt to the user (with a detailed message why the attempt failed).

5.2. Parallelization

In order to make use of multi-core parallelism, Nörgler uses the *Future* mechanism of the Scala runtime library. Futures are placeholder objects for a (future) computation that may not yet have been executed or completed. Futures can be chained in a non-blocking way using combinators, or one may wait (blockingly) for the Future to finish its computation, and to retrieve its result.

The employment of Futures allows for a simple extension of the `Controller` procedure from Listing 1 to parallel checks: A new set `openFutures` of Futures, initially empty, is introduced that holds any Future created during the checking. For each check, the `Controller` may decide to execute it sequentially, or to wrap its computation into a new Future which is started, added to `openFutures`, and scheduled for asynchronous execution. Currently, Nörgler is designed to only parallelize the check **(ICE)** that involves the (costly) invocation of external systems; every other check is executed sequentially. Finally, in line 14 of Listing 1, a line is added that waits for the completion of all Futures (while checking if some check has failed, so that the whole procedure can be aborted prematurely).

In total, there are four different parallelization modes that make use of the generic parallelization scheme described above:

1. **Sequential execution**: Do not parallelize at all.
2. **Step parallelism**: Parallelize **(ICE)** checks as described above.
3. **Prover parallelism**: Run all checks sequentially (including **(ICE)**), but run multiple ATP systems in parallel within the **(ICE)** check.
4. **Hybrid parallelism**: Combination of step and prover parallelism.

When incorporating model finders as well, their parallelization can be controlled independently using one of the three schemes:

1. **Naive parallelism**: Always run the model finder in parallel to the ATP system in **(ICE)**.
2. **Fallback parallelism**: Only run the model finder if the ATP system failed in **(ICE)**.
3. **Offset parallelism**: Start the model finder only after an offset of $1s$ after the ATP system(s) have been started.

6. Evaluation

Nörgler is evaluated using both proofs from the TSTP solution library [7], and a corresponding set of falsified proofs (see Sec. 6.1 for details). Sec. 6.2 compares Nörgler's parallelization and countermodel-finding configurations, while Sec. 6.3 compares Nörgler and GDV. The evaluation was carried out on an 8-core Intel Core i9 @ 2.3 GHz system with 32 GB DDR4 memory (2667 MHz). As external systems, E 3.1.0, Vampire 5.0.1 and Mace4 (64) version 2026-4F are used. The timeout is $10s$.

6.1. Benchmark Selection

To create a suitable benchmark set, the TPTP2T[11] interface was used to search for FOF refutations by ATP systems whose outputs are as similar as possible to those foreseen by the ProoVer competition.

[11] See `tptp.org/cgi-bin/TPTP2T`.

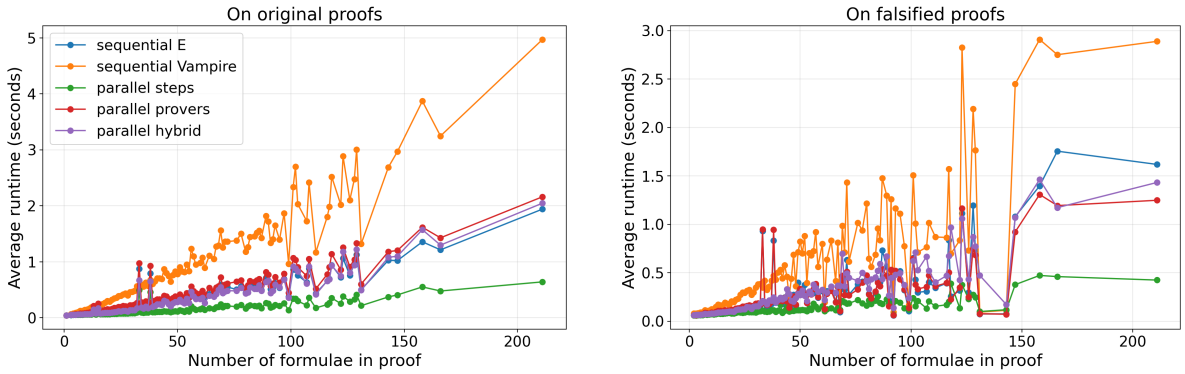

Figure 6: Average runtime per formula count for different parallelization configurations and external provers.

Many proofs in the library diverge from the TPTP standards in system-specific ways, so they were discarded. Otter [5] and PyRes [27] were identified to produce well-behaved proofs compatible with Nörgler and GDV, so their proofs were chosen for the evaluation. Their proofs were taken from the TSTP library and translated to FOF refutations using tptp-utils [37]. Also, the file records contained in the proof were corrected as a preprocessing step. The resulting benchmark consists of 1976 proofs: 1806 proofs from Otter and 170 from PyRes.

Falsification process. A simple automated falsification procedure was used to generate a set of incorrect proofs: Starting from the set of 1976 correct proofs, this procedure selects, for each proof, some annotated formulae that are to be mutated. For each such formula, it randomly applies one of several falsification methods (including replacing single connectives, negating sub-formulae, or replacing atoms with other existing atoms) in a way that preserves syntactical correctness. The original formula is replaced by this altered formula, and a comment detailing the specific change that was made is added to the proof.

In rare cases, random mutations still result in verifiable proofs. To exclude these from the benchmark, any candidate falsified proof was checked with GDV, regenerating falsified versions of the proof until an instance was obtained that could no longer be verified by GDV.

This process yielded a total of 1952 falsified proofs (1782 Otter and 170 PyRes). All benchmarks are archived and publicly available on Zenodo [38].

6.2. Nörgler Configurations

Parallelization and external provers. To compare the effects of the different parallelization modes discussed in Sec. 5, and the influence of different external provers, we ran Nörgler on our benchmark using the following configurations:[12]

1. **sequential E**: Sequential mode, using E as only external ATP system.
2. **sequential Vampire**: Likewise, with Vampire.
3. **parallel steps**: Step parallelization using E
4. **parallel provers**: Prover parallelization using both E and Vampire.
5. **parallel hybrid**: Hybrid parallelization using both E and Vampire.

Fig. 6 visualizes the average runtime in relation to the number of formulas in a proof, and Table 2 summarizes the results of all evaluations discussed in this section.

[12]The exact command line options used were: `--up-to-esa`, `--relax-annotation-format`, `--allow-prover-axioms`, `--relax-specified-inference-checks`, `--timeout 10`. They are required, e.g., because PyRes omits the information necessary for Skolemization verification, and Otter outputs additional proof lines. See Nörgler's README for details.

On the original proofs, all Nörgler configurations identified four proofs as incorrect, an assessment that is consistent with the results from GDV[13]. The remaining 1972 proofs were verified by all configurations except for **sequential Vampire**, which timed out on four problems. On the falsified benchmark, none of the configurations verifies any of the proofs, but the rate at which they are successfully rejected (instead of just producing a timeout) varies: Of the 1952 falsified proofs, **parallel hybrid** identifies 1907 incorrect proofs, while the individual systems only manage to find 1863 (**sequential Vampire**) and 1845 (**sequential E**); the latter being the configuration with the lowest success rate. In terms of performance, **sequential Vampire** exhibits the highest average runtime both on the original proofs ($0.3089s$) and the falsified ones ($0.2211s$), while **parallel steps** has the lowest average runtime on both benchmarks at $0.0842s$ on the original, and $0.0835s$ on the falsified ones.

Overall, the comparison of the average runtime on proofs with an increasing number of steps (Fig. 6) demonstrates the effectiveness of parallelization of proof steps, especially for larger proofs. Checking the same step with multiple provers in parallel, on the other hand, slightly decreased the performance on the original proofs. This is not surprising as E outperforms Vampire consistently on the verification tasks generated by Nörgler. On falsified proofs, the provers however complement each other in their capabilities of identifying proofs as incorrect, increasing the overall success rate when used in conjunction (1907 vs. 1845 for E resp. 1863 for Vampire).

Comparison of model finder configurations. To investigate the effectiveness of our various countermodel finding strategies, we use **parallel steps** as a baseline, and add the arguments invoking our various model finder modes, resulting in the following configurations:

1. **fallback**: Model finder fallback parallelization
2. **always**: Model finder naive parallelization
3. **offset**: Model finder offset parallelization

The results of the evaluation are summarized in Table 2. Like the **parallel steps** configuration, all tested configurations adding model finding still verify 1972 of the correct proofs. However, Nörgler's capability of identifying incorrect proofs is increased in all cases: In comparison to the 1888 identified by **parallel steps**, **fallback** identifies 1899 incorrect proofs while only increasing the average runtime slightly ($0.0839s$ vs. $0.0842s$ on the original proofs, and $0.0916s$ vs. $0.0835s$ on falsified proofs). **always** is the only tested configuration that identifies all 1952 falsified proofs as incorrect. However, the increased success rate comes at the expense of a decreased performance of $0.1173s$ on falsified proofs and–notably–$0.1307s$ on original ones. A compromise between performance and success rate is **offset**, which still increases the number of proofs successfully rejected to 1943, while displaying a similar average runtime to **always** on falsified proofs ($0.1203s$), and increasing the average runtime on correct proofs to only $0.0901s$.

6.3. Comparison to GDV

As detailed in Sec. 4, GDV relies on SystemOnTPTP to call external provers. To avoid the overhead of remotely using the online service (the default method), we used GDV's parameter `-T` invoking the use of a local SystemOnTPTP, and a script imitating the behavior of SystemOnTPTP calling provers locally.

As mentioned before, there are two different cases when GDV returns `Unknown`. To distinguish between them, we parsed the output of GDV, and instances where the external system returned `ContradictoryAxioms` were counted as `VerifiedBad`, while only those that failed due to inconclusive external results like timeouts were counted as `Unknown`.

The results can be found in Table 2 and are visualized in Fig. 7. The figure directly compares the number of completed instances for a given cumulative runtime of GDV against **sequential E**, which is the Nörgler configuration closest to GDV, as it uses the same external system without parallelization or model finding. Additionally, the figure includes comparisons for **parallel steps**, **always** and **offset**.

[13]The four proofs can not be verified as a result of a translation error from the native Otter output format to the TSTP syntax.

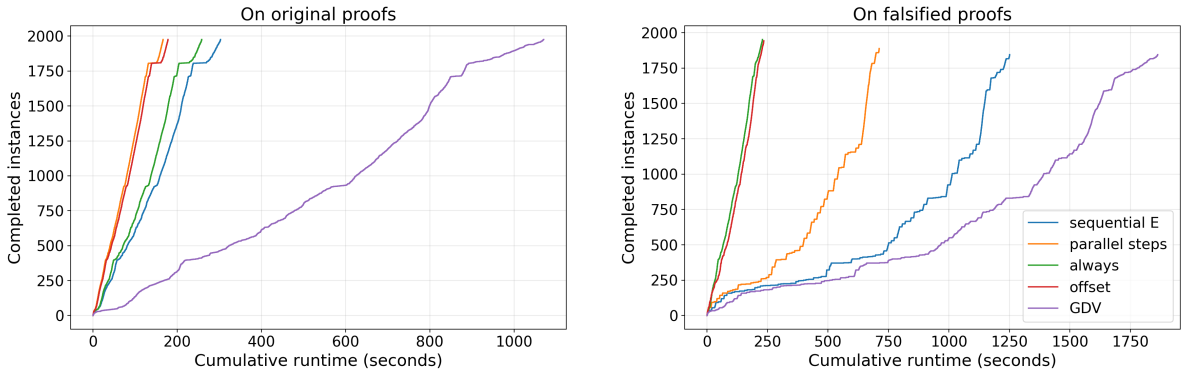

Figure 7: Completed instances over cumulative runtime for different Nörgler configurations and GDV.

Table 2
Results and average runtimes for different Nörgler configurations and GDV.

Original proofs

Configuration	Verified	Runtime (s)
Parallelization and external prover comparison		
sequential Vampire	1968	0.3089
sequential E	1972	0.1533
parallel steps	1972	0.0842
parallel provers	1972	0.1743
parallel hybrid	1972	0.1260
Countermodel finding comparison		
fallback	1972	0.0839
always	1972	0.1307
offset	1972	0.0901
GDV	1972	0.5418

Falsified proofs

Configuration	Shown false	Runtime (s)
Parallelization and external prover comparison		
sequential Vampire	1863	0.2211
sequential E	1845	0.1237
parallel steps	1888	0.0835
parallel provers	1869	0.1317
parallel hybrid	1907	0.1202
Countermodel finding comparison		
fallback	1899	0.0916
always	1952	0.1173
offset	1943	0.1203
GDV	1845	0.4323

As mentioned previously, GDV verifies the same 1972 original proofs as all Nörgler configurations, with the exception of **sequential Vampire**, which suffered from timeouts. Among the falsified proofs, GDV identifies 1845 as incorrect, the remaining instances were `Unknown`.

The average runtime of GDV is considerably higher than that of Nörgler, at $0.5418s$ for correct proofs and $0.4323s$ for falsified ones. This performance gap is already evident when comparing GDV to **sequential E**, which employs the same prover and relies on neither parallelization nor countermodel finding, and further pronounced in our other configurations. Regarding the falsified proofs, GDV rejects the same number of proofs as **sequential E** (1845), which had the lowest success rate among the tested Nörgler configurations.

The comparison in Fig. 7 underlines the merit of adding parallelization and countermodel finding. While the success rate on original proofs remained unchanged, **parallel steps** exhibits the fastest runtimes, improving the performance of the sequential version significantly. **always** is the most capable configuration for identifying incorrect proofs, although it introduces considerable runtime overhead, especially on verifiable proofs. **offset** improves the success rate on falsified proofs while increasing runtimes less noticeably, serving as a compromise between low runtime and a high success rate in disproving false proofs.

While the Otter proofs contain no `esa` steps, the 170 PyRes proofs in our benchmark do. Nörgler skips the verification of these steps, whereas GDV handles them by attempting to prove bi-directional implication. This feature can not be disabled in GDV, the introduced overhead can therefore not be avoided in our evaluation. However, such steps are just present in 84 of the proofs in our benchmarks.

Furthermore, PyRes introduces axioms from assumed theories in produced proofs. To handle these cases, we introduced the `--allow-prover-axioms` flag in Nörgler. GDV provides no such option and

therefore rejects PyRes proofs when requested to verify faithfulness to the problem. Consequently, we exclude the verification of problem faithfulness from our overall evaluation. We did, however, test this separately for Nörgler, and the check succeeded for all PyRes proofs.

7. Conclusion

In this work, we introduced the TSTP certificate checker Nörgler, which builds upon techniques established by GDV. We discussed the implemented verification checks, and evaluated the use of parallelization to enhance performance and the integration of countermodel finding to improve the identification of incorrect proofs. We evaluated several configurations of Nörgler on a benchmark drafted from the TSTP solution library and compared their performance to GDV. Our results demonstrate that: (i) Nörgler implements certain correctness-critical checks currently missing from GDV, (ii) the parallelization of proof step verification significantly enhances performance, and (iii) the use of countermodel finders allows for the detection of incorrect proof steps that would otherwise remain inconclusive. Overall, all tested Nörgler configurations outperformed GDV in terms of average verification time. Furthermore, all configurations incorporating countermodel finding had a higher success rate than GDV in identifying incorrect proofs.

Future work. The current implementation of Nörgler serves as a foundation for further development. Several features from GDV will be adapted in future versions. Specifically, we plan to extend support to other proof calculi and forms, such as tableaux calculi, and to currently unsupported TPTP formalisms, e.g., roles like *assumption* or procedures involving splitting. Implementing the ability to utilize external systems producing Lambdapi-conform proofs, which can be verified independently, is also planned, similarly to GDV-LP's approach [19]. This not only adds another layer of trust, but can also serve as a translation layer between ATPs and ITPs.

Handling specific provers may also require further flags to be added to increase permissibility regarding adherence to TSTP formatting, for example with regard to annotations. Dedicated procedures for system-specific mechanisms may also be necessary, for instance for handling Vampire's Avatar [39]. Other checks may also be adapted in future work to account for a broader range of sound derivations. For instance, **(NSC)** permits `cth` steps that introduce negations into formulae only during the negation of the conjecture. This restriction follows common conventions in automated reasoning, but it also rejects some sound derivations. For example, one could apply two successive `cth` steps that introduce negations into a formula. Should the need to verify such cases arise in the future, alternative checks more closely aligned with the semantics of SZS statuses could be implemented.

Additionally, we plan to implement features to further improve performance and to refine our parallelization techniques, for instance by developing strategies targeted at bigger proofs, e.g., parallelization of proof subtree verification as done in the context of SMT proof checking [40].

Extensions to the TSTP standard introducing standardized rule sets have already been proposed. Embracing this direction, we plan to enable Nörgler to verify rules in a deterministic, search-free way. Currently, two inferences commonly used by ATPs (the negation of the conjecture and Skolemization) can principally be verified without invoking an external system's proof search. Instead, they represent precisely defined procedures, enabling their verification following an approach akin to proof replay in SAT solving.

Finally, we intend to develop procedures similar to proof elaboration in SMT [13] to decompose more complex steps or make implicit modifications performed by ATPs, such as permutations, explicit.

Acknowledgments

The work was supported by the Academy of Sciences and Humanities in Hamburg within the Young Academy Fellows programme.

The authors would like to thank the anonymous reviewers for their constructive feedback.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] L. Kovács, A. Voronkov, First-order theorem proving and Vampire, in: N. Sharygina, H. Veith (Eds.), Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, volume 8044 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 1–35. doi:10.1007/978-3-642-39799-8_1.

[2] S. Schulz, E – a brainiac theorem prover, AI Commun. 15 (2002) 111–126. URL: http://content.iospress.com/articles/ai-communications/aic260.

[3] A. Steen, C. Benzmüller, Extensional higher-order paramodulation in leo-iii, J. Autom. Reason. 65 (2021) 775–807. URL: https://doi.org/10.1007/s10817-021-09588-x. doi:10.1007/S10817-021-09588-X.

[4] C. E. Brown, Satallax: An automatic higher-order prover, in: B. Gramlich, D. Miller, U. Sattler (Eds.), Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings, volume 7364 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 111–117. doi:10.1007/978-3-642-31365-3_11.

[5] W. McCune, OTTER 2.0, in: M. E. Stickel (Ed.), 10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings, Lecture Notes in Computer Science, Springer, 1990, pp. 663–664. URL: https://doi.org/10.1007/3-540-52885-7_131. doi:10.1007/3-540-52885-7_131.

[6] S. Baek, The tesc proof format for first-order atps, 2020.

[7] G. Sutcliffe, The TPTP World - Infrastructure for automated reasoning, in: E. Clarke, A. Voronkov (Eds.), Proceedings of the 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning, number 6355 in Lecture Notes in Artificial Intelligence, Springer-Verlag, 2010, pp. 1–12.

[8] S. Guilloud, J. Cailler, S. Gambhir, A. Poiroux, Y. Herklotz, T. Bourgeat, V. Kunčak, Interoperability of proof systems with sc-tptp, in: Automated Deduction – CADE 30: 30th International Conference on Automated Deduction, Stuttgart, Germany, July 28-31, 2025, Proceedings, Springer-Verlag, Berlin, Heidelberg, 2025, p. 325–340. URL: https://doi.org/10.1007/978-3-031-99984-0_18. doi:10.1007/978-3-031-99984-0_18.

[9] G. Sutcliffe, Semantic derivation verification: Techniques and implementation, Int. J. Artif. Intell. Tools 15 (2006) 1053–1070. doi:10.1142/S0218213006003119.

[10] N. Wetzler, M. Heule, W. A. H. Jr., Drat-trim: Efficient checking and trimming using expressive clausal proofs, in: C. Sinz, U. Egly (Eds.), Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings, Lecture Notes in Computer Science, Springer, 2014, pp. 422–429. URL: https://doi.org/10.1007/978-3-319-09284-3_31. doi:10.1007/978-3-319-09284-3_31.

[11] L. Cruz-Filipe, M. J. H. Heule, W. A. H. Jr., M. Kaufmann, P. Schneider-Kamp, Efficient certified RAT verification, in: L. de Moura (Ed.), Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings, volume 10395 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 220–236. URL: https://doi.org/10.1007/978-3-319-63046-5_14. doi:10.1007/978-3-319-63046-5_14.

[12] H. Schurr, M. Fleury, H. Barbosa, P. Fontaine, Alethe: Towards a generic SMT proof format (extended abstract), in: C. Keller, M. Fleury (Eds.), Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021, volume 336 of *EPTCS*, 2021, pp. 49–54. URL: https://doi.org/10.4204/EPTCS.336.6. doi:10.4204/EPTCS.336.6.

[13] B. Andreotti, H. Lachnitt, H. Barbosa, Carcara: An efficient proof checker and elaborator for SMT

proofs in the alethe format, in: S. Sankaranarayanan, N. Sharygina (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I, Lecture Notes in Computer Science, Springer, 2023, pp. 367–386. URL: https://doi.org/10.1007/978-3-031-30823-9_19. doi:10.1007/978-3-031-30823-9_19.

[14] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, R. Saillard, Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory, CoRR abs/2311.07185 (2023). doi:10.48550/ARXIV.2311.07185. arXiv:2311.07185.

[15] D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, O. Hermant, Zenon modulo: When Achilles outruns the tortoise using deduction modulo, in: Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings 19, Springer, 2013, pp. 274–290.

[16] A. P. Komel, M. Rawson, M. Suda, Case study: Verified Vampire proofs in the lambdapi-calculus modulo, arXiv preprint arXiv:2503.15541 (2025).

[17] M. Taprogge, Computer-assisted proof verification for higher-order automated reasoning within the Dedukti framework, Master's thesis, Universität Greifswald, 2024. URL: https://inria.hal.science/hal-04733263.

[18] M. Y. E. Haddad, G. Burel, F. Blanqui, EKSTRAKTO A tool to reconstruct Dedukti proofs from TSTP files (extended abstract), in: G. Reis, H. Barbosa (Eds.), Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August 26, 2019, volume 301 of *EPTCS*, 2019, pp. 27–35. doi:10.4204/EPTCS.301.5.

[19] G. Sutcliffe, F. Blanqui, G. Burel, Proof verification with GDV and LambdaPi - it's a matter of trust, in: D. A. Talbert, I. Biskri (Eds.), Proceedings of the 38th International Florida Artificial Intelligence Research Society Conference, FLAIRS 2025, Daytona Beach, FL, USA, May 20-23, 2025, Florida Online Journals, 2025. URL: https://doi.org/10.32473/flairs.38.1.138642. doi:10.32473/FLAIRS.38.1.138642.

[20] J. Barwise, An introduction to first-order logic, in: J. Barwise (Ed.), HANDBOOK OF MATHEMATICAL LOGIC, volume 90 of *Studies in Logic and the Foundations of Mathematics*, Elsevier, 1977, pp. 5–46. doi:https://doi.org/10.1016/S0049-237X(08)71097-8.

[21] T. Skolem, Logisch-kombinatorische untersuchungen über die erfüllbarkeit oder bewiesbarkeit mathematischer sätze nebst einem theorem über dichte mengen (1970).

[22] P. B. Andrews, Resolution in type theory, Journal of Symbolic Logic 36 (1971) 414–432. doi:10.2307/2269949.

[23] P. B. Andrews, Theorem proving via general matings, J. ACM 28 (1981) 193–214. URL: https://doi.org/10.1145/322248.322249. doi:10.1145/322248.322249.

[24] G. Sutcliffe, Stepping Stones in the TPTP World, in: C. Benzmüller, M. Heule, R. Schmidt (Eds.), Proceedings of the 12th International Joint Conference on Automated Reasoning, number 14739 in Lecture Notes in Artificial Intelligence, 2024, pp. 30–50.

[25] G. Sutcliffe, The TPTP problem library and associated infrastructure. From CNF to TH0, TPTP v6.4.0, Journal of Automated Reasoning 59 (2017) 483–502.

[26] G. Sutcliffe, TPTP, TSTP, CASC, etc., in: V. Diekert, M. Volkov, A. Voronkov (Eds.), Proceedings of the 2nd International Symposium on Computer Science in Russia, number 4649 in Lecture Notes in Computer Science, Springer-Verlag, 2007, pp. 6–22.

[27] S. Schulz, A. Pease, Teaching automated theorem proving by example: Pyres 1.2 - (system description), in: N. Peltier, V. Sofronie-Stokkermans (Eds.), Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II, Lecture Notes in Computer Science, Springer, 2020, pp. 158–166. URL: https://doi.org/10.1007/978-3-030-51054-1_9. doi:10.1007/978-3-030-51054-1_9.

[28] G. Sutcliffe, J. Zimmer, S. Schulz, Communication Formalisms for Automated Theorem Proving Tools, in: V. Sorge, S. Colton, M. Fisher, J. Gow (Eds.), Proceedings of the Workshop on Agents and Automated Reasoning, 2003, pp. 52–57.

[29] G. Sutcliffe, J. Zimmer, S. Schulz, TSTP Data-Exchange Formats for Automated Theorem Proving Tools, in: W. Zhang, V. Sorge (Eds.), Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems, number 112 in Frontiers in Artificial Intelligence and Applications, IOS Press, 2004, pp. 201–215.
[30] G. Sutcliffe, The SZS ontologies for automated reasoning software, in: P. Rudnicki, G. Sutcliffe, B. Konev, R. A. Schmidt, S. Schulz (Eds.), Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008, volume 418 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2008. URL: https://ceur-ws.org/Vol-418/paper3.pdf.
[31] J. A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning (in 2 volumes), Elsevier and MIT Press, 2001. URL: https://www.sciencedirect.com/book/9780444508133/handbook-of-automated-reasoning.
[32] M. Rawson, M. Suda, P. Hozzová, G. Reger, Reuse of introduced symbols in automatic theorem provers (short paper), in: B. Konev, C. Schon, A. Steen (Eds.), Proceedings of the Workshop on Practical Aspects of Automated Reasoning Co-located with the 11th International Joint Conference on Automated Reasoning (FLoC/IJCAR 2022), Haifa, Israel, August, 11 - 12, 2022, volume 3201 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022. URL: https://ceur-ws.org/Vol-3201/paper10.pdf.
[33] A. Steen, ask v0.2.2, 2024. doi:10.5281/zenodo.14181705.
[34] W. McCune, Mace4 reference manual and guide, CoRR cs.SC/0310055 (2003). URL: http://arxiv.org/abs/cs/0310055.
[35] A. Steen, M. Taprogge, H. Sariyanto, Nörgler v1.0, 2026. doi:10.5281/zenodo.19787118.
[36] A. Steen, scala-tptp-parser: Version 1.7.4, 2026. URL: https://github.com/leoprover/scala-tptp-parser. doi:10.5281/zenodo.19428194.
[37] A. Steen, tptp-utils 1.4.2, 2025. URL: https://github.com/leoprover/tptp-utils. doi:10.5281/zenodo.17205289.
[38] M. Taprogge, A. Steen, H. K. Sariyanto, Tstp fof proof benchmark for the evaluation of proof checkers, 2026. doi:10.5281/zenodo.19792604.
[39] A. Voronkov, AVATAR: the architecture for first-order theorem provers, in: A. Biere, R. Bloem (Eds.), Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, volume 8559 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 696–710. URL: https://doi.org/10.1007/978-3-319-08867-9_46. doi:10.1007/978-3-319-08867-9_46.
[40] A. Coltellacci, Reconstructing SMT Proofs in Lambdapi, Ph.D. thesis, University of Lorraine, 2025.