Generating and Exploiting Automated Reasoning Proof Certificates

Haniel Barbosa
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil

Clark Barrett
Stanford University
Stanford, USA

Byron Cook
Amazon
New York City, USA
University College London
London, UK

Bruno Dutertre
Amazon
USA

Gereon Kremer
Stanford University
Stanford, USA

Hanna Lachnitt
Stanford University
Stanford, USA

Aina Niemetz
Stanford University
Stanford, USA

Andres Nötzli
Stanford University
Stanford, USA

Alex Ozdemir
Stanford University
Stanford, USA

Mathias Preiner
Stanford University
Stanford, USA

Andrew Reynolds
The University of Iowa
Iowa City, USA

Cesare Tinelli
The University of Iowa
Iowa City, USA

Yoni Zohar
Bar-Ilan University
Ramat Gan, Israel

ACM Reference Format:
Haniel Barbosa, Clark Barrett, Byron Cook, Bruno Dutertre, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Cesare Tinelli, and Yoni Zohar. 2023. Generating and Exploiting Automated Reasoning Proof Certificates. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Automated reasoning refers to a set of tools and techniques for automatically proving or disproving formulas in mathematical logic [35]. It has many applications in computer science—for example, questions about the existence of bugs or security vulnerabilities in hardware or software systems can often be phrased as logical formulas, or *verification conditions*, whose validity can then be proved or disproved using automated reasoning techniques, a process known as *formal verification* [15, 26]. When successful, formal verification can guarantee freedom from certain kinds of design errors, an outcome that is otherwise extremely difficult to achieve. Driven by such potential benefits, the past couple of decades have seen a dramatic improvement in the performance and capabilities of automated reasoning tools, with a corresponding explosion of use cases, including formal verification, automated test case generation, program analysis, program synthesis, and many more [5, 37, 38].

These applications rely crucially on automated reasoning tools producing correct results. However, ensuring correctness is a significant challenge. To meet the perpetual demand for better performance, automated reasoning tools have large and complex code bases, are highly-optimized, and evolve rapidly, all of which puts their reliability at risk. While conventional software engineering best practices can help, they are insufficient, especially when the goal is to provide incontrovertible mathematical guarantees. Formal verification of automated reasoning tools themselves requires an enormous effort and would have to be revisited every time the tools change. Fortunately, it is possible to provide strong correctness guarantees for an automated reasoner without having to trust the tool at all. The idea is to separate proof *finding* from proof *checking*. In contrast to the complex tools for finding proofs, automated proof checkers can be relatively simple (i.e., a few thousand lines of code as opposed to a few hundred thousand lines of code) and need only be revisited when the proof format changes, a relatively rare

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

event. Additionally, because of their relative simplicity, proof checkers can be vetted by the community when made open-source. Even formal verification of simple proof checkers is not out of the question. It is also worth noting that proof checking can often be done offline, reducing the impact on performance.

The main challenge with this approach lies in the need to instrument automated reasoning tools to produce independently checkable proofs. First steps in this direction have already been taken for one simple class of tools, namely those that check the satisfiability of propositional (i.e., Boolean) formulas, or *SAT solvers*. Thanks to steady progress in this area, most modern SAT solvers can produce proofs in standard formats (e.g., [23]), which can then be independently checked by proof checkers for those formats [22].

Many crucial industrial applications require more reasoning power than is provided by SAT solvers (for an example, see Section 2.1) and often rely instead on solvers for *Satisfiability Modulo Theories (SMT)* [11]. SMT solvers accept a more expressive language and have specialized procedures for reasoning about data types that arise when modeling systems (see Section 2 for more details). Their additional power makes them suitable for a wider array of applications, but also makes producing proofs much more challenging.

In this article, we take the next step towards a full suite of proof-producing automated reasoning tools by demonstrating that SMT solvers can now produce full, independently-checkable proofs for real-world problems using a rich set of constraints. This requires solving a series of challenging problems, both theoretical and technical. We stress that, although we focus on SMT solvers here, many of the ideas and techniques are broadly applicable to automated reasoning tools more generally. We outline our approach, describe its implementation in the cvc5 SMT solver [6], and discuss several exciting applications unlocked by this new capability.

2 SMT Solvers

SMT solvers are *satisfiability checkers*: they take as input a logical formula and try to determine if the formula has a solution, i.e., is satisfiable. More specifically, they are constraint solvers: they determine if there is a valuation of the formula's variables that makes the formula true. Often, however, we are not interested in showing that a property, expressed as a formula F, is true in *some* cases; rather, we want to know if it is true in *all cases*. Technically, we are interested in proving that F is *valid* (i.e., unfalsifiable). This can be done with SMT solvers by checking if the *negation* of the formula, $\neg F$, is false in all cases, or *un*satisfiable. A proof of the unsatisfiability of $\neg F$ provides evidence that F is valid. Since valid formulas are often referred to as *theorems*, SMT solvers can then serve both as constraint solvers and as *theorem provers*.

What distinguishes SMT solvers from other automated reasoning tools is that they are specialized to reason about logical theories that formalize the main data types used in computer science, such as finite and infinite precision integer/rational numbers, floating point numbers, strings, arrays, sequences, records, finite sets, and more. In a precise sense, SMT solvers know about the salient algebraic properties of the types mentioned above. Their theories are *built-in*. The details of how this is done are beyond the scope of this article, but can be found in the literature (e.g., [10, 11]).

```
{"Statement": [{
   "Effect": "Allow",
   "Action": "*",
   "Resource": "bucket/private/*"
   "Condition": {"StringEquals": {"ec2:Vpc": "vpc-123"}}
}]}
```

(a) Boundary policy

```
{"Statement": [{
   "Effect": "Allow",
   "Action": "s3:GetObject",
   "Resource": "bucket/private/reports/*"
  },{
   "Effect": "Deny",
   "Action": "*",
   "Resource": "*"
   "Condition": {"StringNotEqualsIfExists":
     {"ec2:Vpc": "vpc-123"}}
  }]}
```

(b) S3 bucket policy

Figure 1. Example access policies

2.1 Motivating example

The security of data in the cloud relies on tightly controlling who has access to it. At Amazon Web Services (AWS), for example, the Identity and Access Management (IAM) policy language encodes access control information as a JSON document (a structured text format supporting hierarchy and key/value pairs). Each data request is evaluated against the control policy to determine if it is permitted. The power of conventional testing in this context is quite limited, both theoretically and practically, since the policy language allows the use of unbounded strings, making the number of possible requests essentially infinite. In contrast, it *is* possible to reason formally and symbolically about both policies and requests using SMT solvers [3], as we describe next.

For private data, a key property that one may want to check is that the data cannot be accessed from outside of a given organization. We can capture this requirement as a policy that defines an *upper bound* for allowable requests. Figure 1a shows an example of such a *boundary* policy. The policy allows access only to resources with the prefix `bucket/private/`, and only from within the VPN `vpc-123`. Consider now the specific policy shown in Figure 1b. To show that it complies with the boundary policy, we must prove that every request it accepts is also accepted by the boundary policy. We can do this by constructing a formula P defining the requests allowed by the boundary policy and a formula B defining the requests allowed by the bucket policy and then asking a solver whether the implication $B \Rightarrow P$ always holds. In this case, P can be encoded using the theory of strings as prefixof("bucket/private/", r) $\wedge$ *vpcExists* $\wedge$ *vpc* $\approx$ "vpc-123", where r and *vpc* are string variables that represent the resource and VPN values of the request, and *vpcExists* is a Boolean variable indicating whether the VPN is defined in the request or not. The semantics of prefixof is defined by the theory of strings and is true if r has the string "bucket/private" as a prefix. B can be encoded similarly.

If $B \Rightarrow P$ holds, then the policy is compliant, and a proof-producing SMT solver can generate a proof effectively demonstrating *why*. The example bucket policy is indeed compliant, because the first part only allows the retrieval of objects from `bucket/private/reports`, and the second part denies requests that do not match the VPN `vpc-123`.[1]

3 Producing Proofs in SMT Solvers

We now move to the question of how best to produce proofs in SMT solvers. We distinguish here a *proof*, which is generated internally by the solver using a suitable data structure, from a *proof certificate*, a representation of the proof emitted by the solver for external consumption. We discuss proof certificates in Section 3.5, below.

As mentioned above, instrumenting solvers to be proof-producing in a way that is comprehensive and has minimal impact on performance is a hard technical problem. It is no surprise then that even though several SMT solvers with proof-producing capabilities have been developed [13, 16, 19, 24], each targets a different proof format, uses a unique and insulated tool-chain, and, more importantly, has one or more of the following shortcomings:

- does not produce detailed enough proofs, thereby requiring a high-complexity proof checker;
- has non-proof-producing performance-critical components, meaning that the solver is unusable or unacceptably slow when producing proofs;
- emits proof certificates that are checkable only in a monolithic setup, rendering them unusable outside that setup.

In the following, we describe how to overcome these shortcomings and produce fast, comprehensive, and flexible proofs which are also simple to check.

3.1 Proof representation

In general, SMT solvers check the joint satisfiability of a *set* $\{F_1, \ldots, F_n\}$ of input formulas. The solver concludes that the input set is unsatisfiable when it is able to prove $\bot$, the universally unsatisfiable formula, from the assumptions $F_1, \ldots, F_n$.

A proof is a justification, via a series of *proof steps*, for deriving a conclusion from a set of assumptions. Each proof step is the application of a *proof rule*, which infers a formula, the *conclusion* of the rule, from zero or more *premises*. Premises must either be assumptions or previously inferred formulas. Rules *without* premises are typically used to introduce either valid formulas (e.g., $t = t$ for some term t) or assumptions in a proof. Proof rules *with* premises are applicable only when their premises match previously inferred formulas. Each rule must be *sound*, a technical notion guaranteeing that the rule's conclusion logically follows from its premises. A proof-producing solver fixes a set of sound proof rules and produces proofs based on them. Rule soundness is argued for externally to the proof-checking mechanism, and hence the rules are part of the solver's *trusted core* (i.e., they are a part of the system that must be trusted as opposed to a part of the system that is being checked).

A proof can be represented internally in the solver as a directed acyclic graph whose nodes represent individual proof steps. Each proof node stores a reference to the applied proof rule and the inferred conclusion. Edges in the graph connect a proof node to nodes representing its premises, if any. A proof graph is well-formed if it is acyclic and each node represents a proof step that applies its corresponding proof rule correctly. A well-formed proof of a formula F from some assumptions $F_1, \ldots, F_n$ has a single root node, with F as its conclusion. Its leaves are either valid formulas or assumptions taken from $F_1, \ldots, F_n$.

3.2 Producing modular proofs

Most SMT solvers are based on a common architecture (see, e.g., [10, 30]) with the following main components:

1. a *preprocessing module*, which simplifies the input formulas as much as possible;
2. a *clausifier*, which converts the preprocessed formulas into a formula F: a conjunction of *clauses*. Each clause is a disjunction of literals, where a literal is either a theory atom or the negation of a theory atom.
3. a *SAT engine*, which reasons about the Boolean abstraction of F (i.e., each theory atom is treated as a Boolean variable) and searches for a solution;
4. a *combination of theory solvers*, each specialized for a single theory; whenever, during its search, the SAT engine assigns a value to a variable, the corresponding literal is sent to the theory solvers as an *assertion*; the theory solvers cooperatively check the satisfiability of these assertions with respect to their theories and produce either a solution, when the assertions are jointly satisfiable, or an *explanation* (an unsatisfiable subset of the assertions) when they are not; theory solvers can also produce *lemmas*, formulas that are valid in the theory and that aid the SAT engine in its search.

[1]Note that the semantics of `StringNotEqualsIfExists` are such that if the request does not specify any VPN, the policy still denies access.

5. several *theory rewriters*, each implementing a set of theory-specific rewrite rules used to simplify terms in F.

For an SMT solver to produce proofs, each of the above components must be instrumented accordingly. Fortunately, this can be done modularly: each component produces proofs for its own reasoning steps, and these are then combined together to form the full proof. The process starts with a Boolean proof produced by the SAT engine, which justifies the (propositional) unsatisfiability of its clauses. These clauses either come from the clausifier or are lemmas or explanations from the theory solvers, and thus proofs of the clauses can be obtained from those modules. The clausifier, in turn, receives its input from the output of the preprocessing module, and both the preprocessing module and the theory solvers use proof steps that depend on the theory rewriters.

3.3 Producing well-formed proofs

It is of course possible to make mistakes when instrumenting a solver to produce proofs. Such mistakes manifest as *ill-formed* proofs. Proofs can be complex and very large, making proof debugging rather challenging. The ability to identify errors early on and pinpoint their source is crucial. There are the three kinds of errors that can occur in proof graphs:

1. **erroneous proof step**: a node's premises and conclusion are not a valid instance of its associated proof rule;
2. **cyclic proof**: the graph has a cycle, i.e., a proof node has itself as a descendant, making the proof meaningless; and
3. **open proof**: the proof relies on assumptions other than the allowed ones provided as input.

Errors of type 1 result when the code that creates a proof node is faulty. Errors of type 2 are possible as a consequence of proof transformations that change a proof node's children. Errors of type 3 are possible when a proof system contains rules that introduce local (temporary) assumptions. For instance, a rule may allow proving an implication of the form $P \Rightarrow Q$ by temporarily assuming P and then proving Q based on the assumption P. Once Q is proved, the assumption P is *discharged* to obtain the conclusion $P \Rightarrow Q$, with P no longer considered an assumption in the resulting proof. If the proof fails to discharge such an assumption, the proof is left *open*.

Errors of type 1 can be detected by employing an internal proof checker which, when enabled, is immediately applied to *each* proof node as it is generated. Errors are thus detected at proof node *creation time*, making them much easier to localize. The checker must implement a checking method for each supported proof rule. Errors of type 2 and 3 can be caught with more expensive checks which do a full traversal of the current proof graph. Such checks can be made available as debugging aids but should be disabled during normal use.

As mentioned above, proofs are *highly modular*, which also makes it possible to debug them modularly. Each time a component from Section 3.2 produces a proof, it can be checked for the three kinds of errors mentioned above. And each time two proofs are combined, error checking can be done again. This makes it possible to distinguish errors originating within a component from errors introduced during proof combination.

3.4 Producing proofs efficiently

SMT solver performance is critical, so it is important to ensure that it is minimally impacted by the proof infrastructure. One powerful technique for controlling proof overhead is *lazy proof generation*. The idea is to generate only a proof sketch at solving time—a high-level proof composed of simpler, unproven lemmas—and then to convert this to a full proof after the solver determines the unsatisfiability of the input. There are two main advantages to this approach.

- It is simpler and less invasive to generate proof sketches. This helps not only with performance but also with keeping the implementation effort low.
- During the search phase of solving, many directions are explored and lemmas generated that end up being irrelevant for the final proof. By delaying the generation of detailed proofs, only the lemmas actually required in the final proof need to be expanded.

Lazy proof generation is a key feature of the DRAT proof format [23] used by SAT solvers, and several ways of expanding (also known as *elaborating*) DRAT proofs have been proposed [17, 25, 27]. Lazy proof generation has also been explored in the context of SMT [24] to generate proofs in cvc5's predecessor, CVC4 [9].[2] In that work, the only information kept during solving is the list of derived lemmas and explanations. The elaboration process then consists of invoking a separate proof-producing instance of the component associated with each lemma or explanation.

We improve on this approach in several ways. First, we use a more general notion of a *lazy proof*. In particular, we introduce a new kind of proof node, a *lazy proof node*, containing the conclusion F as well as a reference to a *proof generator*.[3] The proof generator encapsulates the information necessary to compute a standard proof node N concluding F, possibly including references to other lazy proof nodes.[4] Proof generators can be configured to compute N eagerly, when the lazy proof node is first constructed, or later, during the elaboration of the lazy proof. As a general rule, a proof step should be generated eagerly only when the cost of doing so is less than or comparable to that of storing the information needed by the corresponding proof generator. A theory solver for equality and uninterpreted functions, for example, could produce proofs eagerly, since its explanation method

[2]Previous CVC systems, including CVC4, used capital letters. With the introduction of cvc5, we moved to lower-case letters.
[3]This is analogous to the notion of *futures* or *promises* in some programming languages.
[4]Similar methods for improving proof elaboration via the recording of key information during solving have recently been employed in the context of SAT solving as well [4].

typically already generates all the information necessary for proof-production (see, e.g., [29]).

Another important innovation is the use of *macro steps*. A macro step compresses the result of applying several proof rules into a single proof node. An example would be deriving $\neg A$ from $A \Rightarrow B$ and $\neg B$. This step can be justified in terms of basic proof steps as follows: locally assume A; derive B from $A \Rightarrow B$; derive a contradiction with $\neg B$; conclude $\neg A$ from the contradiction. Using macro rules reduces the size of proofs and eases the implementation burden when trying to capture multi-step reasoning. On the other hand, such rules complicate the job of the proof checker. To avoid this, we propose expanding macro steps into their more basic parts as a post-processing step. The post-processing is invoked after solving is complete and also includes invoking the proof generators of lazy proof nodes and connecting sub-proofs produced by different components. A useful byproduct of our approach is that it can support proofs at different levels of granularity (e.g., with or without macro steps).

3.5 Proof certificates and proof checking

A *proof certificate* is a textual representation of a proof. Checking a proof certificate for the unsatisfiability of a set $\mathcal{F}$ of formulas amounts to checking that it represents a well-formed proof of $\bot$ from $\mathcal{F}$. Whereas there is an extensive literature on proof procedures in SMT, as well as some consensus on how best to implement them, there is much less consensus on what specific form a proof certificate should take.

We can agree that a proof certificate for an unsatisfiable set $\mathcal{F}$ should be in a *formal language* and provide convincing evidence of $\mathcal{F}$'s unsatisfiability. This is a minimal requirement for the certificate to be checkable by a separate tool. Even so, the certificate could still take many forms, each differing in syntactic structure, or level of detail, or both. A pragmatic criterion for the acceptability of a proof certificate format, which we espouse and propose here, is that checking proof certificates must be *fundamentally simpler* than finding the proof in the first place. It is reasonable to expect, for instance, that certificates be checkable in time polynomial (ideally, linear) in their size. A proof checker should not need complex data structures or, worse, expensive search algorithms to check a certificate. Such needs would greatly increase the checker's complexity, thereby defeating the purpose of making the checker easy to trust.

An orthogonal challenge for identifying a suitable proof certificate format is that different SMT solvers rely on different solving techniques and proof procedures. Since a proof is a record of the arguments used internally by the solver, these differences mean that solvers may have highly diverse requirements for a proof certificate format.[5]

[5]This is a major reason it has been difficult to identify and establish a common format for SMT proof certificates.

Ultimately, proof certificates must be understood by a proof checker, so the design of the proof checker also influences design decisions about the format. One appealing approach for proof checking is to embed a proof checker within an existing trusted system such as a *proof assistant*. Proof assistants are powerful interactive theorem proving systems that rely on manual effort and expertise to produce proofs. So-called *skeptical* proof assistants are designed with a small *trusted kernel*, a designated part of the system whose correctness must be trusted. By design, all other parts of the system inherit their correctness from the correctness of the kernel. Widely-used skeptical proof assistants such as Coq [40] and Isabelle/HOL [31] have time-tested kernels generally regarded as highly trustworthy. The input languages of proof assistants are typically quite powerful (i.e., both Turing-complete and feature-rich) in order to facilitate the construction of proof scripts and tactics for guiding proof search tasks.[6] These features make skeptical proof assistants an attractive environment for developing a proof checker. Proof checkers embedded in proof assistants can be proved correct within the proof assistant itself. Alternatively, they can be used to translate proof certificates into trusted theorems based on the proof assistant's kernel. In either case, the only trusted component of the entire tool chain (including the proof-producing SMT solver) is the proof assistant's kernel. Alethe [36] is an emerging format for SMT proofs designed with proof assistant integration in mind. The Alethe format currently supports proofs for a subset of SMT theories and has been co-designed with proof checkers written within Coq and Isabelle/HOL. This means that for problems in the SMT fragment supported by Alethe, one can use proof checkers embedded in proof assistants to achieve the highest level of confidence currently possible for SMT solver results.

Alternatively, proof checking can be performed by stand-alone checkers whose trustworthiness must be established independently. Such checkers are more easily extensible, since the new extensions do not have to be formally verified in a proof assistant. Furthermore, stand-alone checkers are typically much more efficient than checkers built within proof assistants, especially when dealing with large proof certificates. The main reason for this is that the execution speed of programs in proof assistants can be quite slow.[7] An effort to address this issue is part of the motivation behind a more recent proof assistant called Lean [18].

A successful example of a format specifically co-designed with an accompanying, high-performance stand-alone checker is LFSC [39], a *logical framework with side conditions*.[8]

[6]The functional programming language ML was originally designed as the tactic language (or Meta Language) of an early proof assistant, LCF.

[7]Optimizing this has historically not been a priority, since proofs are typically constructed manually and interactively.

[8]Recently, a high-performance stand-alone checker for the Alethe format was also introduced [2].

Though the full LFSC checker must be trusted, it is nevertheless fairly small, comparable in size to a trusted kernel of a proof assistant. A distinguishing feature of the LFSC proof checker is that it takes as input not just a proof certificate but also a logical specification, referred to as a *signature*, of the *proof system* (essentially, the proof rules) used to build the proof. This feature makes it possible to specify *any* proof system used by a specific SMT solver without having to change the proof format or the checker. This great level of flexibility comes at the cost of having to trust also the signature provided to the checker, in addition to the checker itself.

4 Implementation

We have implemented our approach in cvc5, an efficient and full-featured open-source SMT solver [6]. In this section, we provide some insights gained by this implementation effort. Additional details can be found in Barbosa et al. [8].

4.1 Instrumenting modules to produce proofs

We instrumented each of the modules mentioned in Section 3.2 in cvc5 to produce proofs.

The preprocessing module in cvc5 consists of 34 distinct passes, each of which presents unique challenges. As an example, in one pass, an input of the form $x = t, F_1, \ldots, F_n$ (with x not occurring in t) is transformed into $F'_1, \ldots, F'_n$ where, for $i \in [1, n]$, F'_i is the result of replacing all occurrences of x in F_i by t. Notably, this preprocessing pass has resisted previous attempts at proof production. In particular, it is mentioned in Barbosa et al. [7, Section 4.6] as beyond the scope of their approach. cvc5 fully supports proofs for this pass with negligible overhead during solving by leveraging a macro step for substitution coupled with lazy proof generation.

cvc5 uses a modified version of MiniSat [20] for its SAT engine, which we have instrumented to be proof-producing. As mentioned earlier, modern SAT solvers have native support for the DRAT proof format. Integrating such a SAT engine in cvc5 is part of the roadmap for future work.

The theory solvers and rewriters in cvc5 implement theory-specific reasoning, and for non-trivial theories, they can be quite involved. Instrumenting these modules thus requires a commensurate amount of effort, which can be significant. To better understand what is required, we consider a specific theory, the theory of strings (which was highlighted in the example in Section 2.1).

The string solver in cvc5 comprises approximately 20,000 lines of C++ code. The theory solver takes as input a set of literals and deduces new literals from that set. For each deduced literal l, the solver is able to produce an explanation of the form $l_1 \wedge \cdots \wedge l_n \Rightarrow l$, where $\{l_1, \ldots, l_n\}$ is a subset of literals from the current input set. Internally, the solver consists of several different layers, each providing a solver for a different fragment of the theory [14]. The layers are invoked in sequence, with more costly layers used only if the cheaper ones fail to make new deductions. This is important for performance because the worst-case computational complexity of constraint solving in different fragments of the same theory can vary greatly, as shown in Figure 2 for the theory of strings over a finite alphabet.

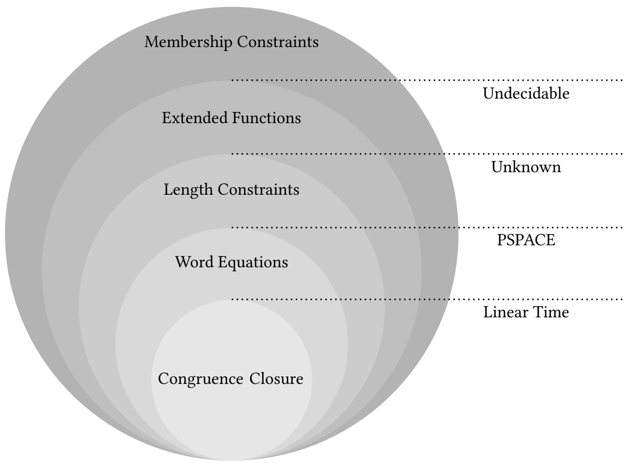

Figure 2. An example of layering string constraints and the computational complexity of each layer.

At the core layer of cvc5's solver for the theory of strings is a procedure that infers consequences of the basic axioms of equality. For instance, from the literals $x = x_2$, $x_1 = x_2$, and $x_1 \cdot y_1 \neq x_2 \cdot y_2$ (here, $\_ \cdot \_$ denotes string concatenation), it is able to infer $x \cdot y_1 \neq x \cdot y_2$. The next layer performs a basic form of string-specific reasoning based on *word equations*, i.e., (dis)equalities between concatenations of string variables and string literals [28]. This layer is able to infer $y_1 \neq y_2$ from $x \cdot y_1 \neq x \cdot y_2$, or $\bot$ from $\texttt{"abc"} \cdot y_1 = \texttt{"b"} \cdot y_2$. Reasoning about string length constraints is done in cooperation with a theory solver for linear integer arithmetic. Another layer processes constraints containing string operators other than concatenation by simplifying them based on the current set of literals [34] and lazily reducing them to word equations, length constraints, and quantified constraints [33]. Across the various layers in the string solver, we currently distinguish between 72 string-specific inferences, each captured by a proof rule. Additional non-string-specific rules are used to capture inferences for the equality, arithmetic, and quantifier reasoning proof steps the string solver relies on.

The theory rewriter for strings in cvc5 consists of 183 individual rewrite rules. A detailed proof requires proof steps for each of the relevant applications of these rewrite rules. We do this lazily using a rewrite rule DSL coupled with a proof-reconstruction algorithm [32].

4.2 Proof certificate formats

Given the existence of various proof certificate formats, as discussed in Section 3.5, each with their own trade-offs, we built cvc5's proof infrastructure to be flexible enough to support multiple formats. We achieved this by applying proof post-processing mechanisms similar to those described for lazy proofs. The final proof produced by cvc5 in its internal proof system is converted, step by step, to a proof tree data structure that captures the abstract syntax of proofs in the target proof certificate format. Since the level of granularity, the term representation, and the allowed proof rules may vary significantly between the internal and the target formats, the conversion is more than just a syntactic translation. In fact, it often requires elaborate transformations.[9] Once the proof has been converted, a pretty printer is used to produce a proof certificate in the concrete syntax of the target format.

We have implemented proof converters in cvc5 for both Alethe and LFSC. For the LFSC format, we have also developed signatures that capture cvc5's own proof system, making the conversion fairly direct. Note that these signatures are more general than the ones developed for CVC4, which also generated LFSC proof certificates [24, 39]. At the same time, we are collaborating with the developers of Alethe, Isabelle/HOL, and Coq to improve and extend both the Alethe format itself, as well as Isabelle's and Coq's support for it, in order to increase the range and expressiveness of SMT formulas that can be checked with these tools.

Finally, we have taken promising initial steps towards a third proof converter for the Lean proof assistant. Our Lean converter is similar in spirit to the LFSC one in that it relies on a formalization in Lean of cvc5's proof system. The goal in this case is to achieve the flexibility and proof-checking performance afforded by LFSC's stand-alone checker while also being able to prove the correctness of cvc5's proof system in Lean itself.

5 Applications

The most obvious benefit of proof production is the ability to reduce the size of the trusted code base. However, there are many other potential advantages. Our experience with cvc5 suggests that instrumenting a solver to produce proofs improves the quality of its code. This is because proof instrumentation requires the code to be clear and modular and often uncovers bugs and other issues with it. Additionally, proof infrastructure is a valuable debugging aid not only for proofs but also for the SMT solver itself. For example, if the solver incorrectly finds an input formula unsatisfiable, then either the attempt to produce a proof will fail or an incorrect proof will be generated. In the first case, the place in the solver's code where proof generation fails provides a good indication of where the problem is; in the second, a proof checker (either internal or external) can identify the problem, serving effectively as a test oracle for the solver.

[9]A simple, and very frequent, example of the need for such transformations is the conversion of proof steps with multiple premises into a proof using rules with at most two premises.

Beyond these benefits to SMT solver developers, proofs open the door to many potential novel applications. Here, we mention just two: automation for proof assistants; and regulatory compliance automation.

5.1 Automation for proof assistants

As discussed in Section 3.5, proof assistants can be used to construct highly trustworthy proof checkers for SMT proofs. However, the ability to communicate between SMT solvers and proof assistants is also useful in the other direction, as a way to bring more automation to proof assistants.

Proof assistants allow users to formulate conjectures and then prove them using a sequence of steps. Each step consists of the application of a proof rule built into the assistant or the application of a previously proved lemma. A limited degree of automation is provided by *tactics*, small programs that attempt to prove conjectures by systematically applying different rules and lemmas. However, these tactics fall short of the degree of automation provided by an SMT solver.

This gap can be addressed with a proof-producing SMT solver and a proof checker embedded in the proof assistant. When a proof step requires only reasoning in a logical fragment understood by SMT solvers, a lemma required to complete the proof step is automatically extracted and translated into an SMT formula. The formula is then sent to the SMT solver (used as a theorem prover as described in Section 2) which produces a proof certificate for it. This certificate can then be fed to the proof checker in the proof assistant. If the check is successful, the checker will have produced precisely the lemma needed to complete the original proof step.

The SMTCoq [21] and Sledgehammer [12] tools implement this workflow (for fragments of the full SMT language) for the Coq [40] and Isabelle/HOL [31] proof assistants, respectively. Ongoing efforts by the authors and others aim to extend these tools and to provide similar functionality for the Lean [18] proof assistant.

5.2 Regulatory compliance automation

Government regulation and industry standards are designed to give us confidence that the products and infrastructure we use are safe. For companies that build, operate, or use information technology (IT), various regulations apply, depending on which technologies they use, which industries they are a part of, and which countries they operate in. The Payment Card Industry Security Standards Council, for example, defines the "Payment Card Industry Data Security Standard" (PCI-DSS) for organizations that handle branded credit cards from the major card schemes. As the world increasingly relies on information technology, more government regulation and industry standards will be common.

Regulation comes at a cost, and organizations today take on time-consuming and expensive processes to ensure compliance with these regulations. Moreover, since irrefutable evidence is hard to construct, compliance audits often need

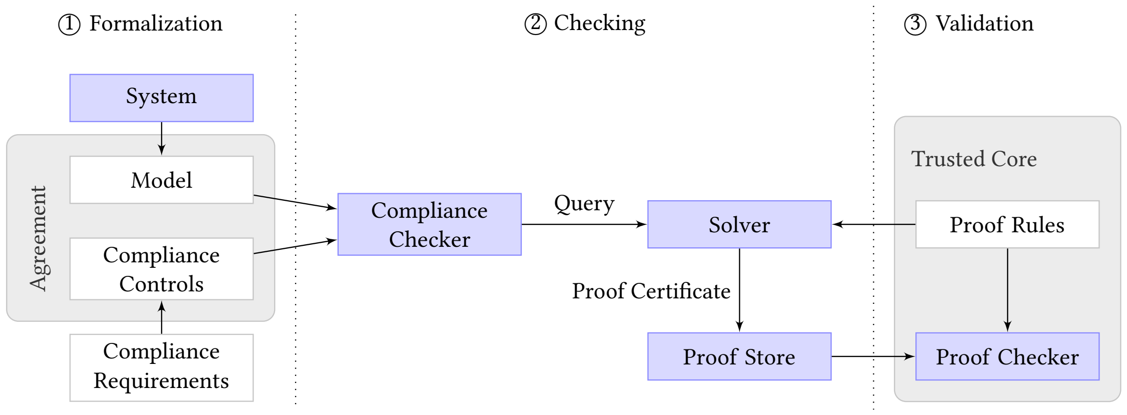

Figure 3. Overview of automated compliance checking using proof certificates: ① auditors and auditee agree on a model of the system and the controls for ensuring a compliance requirement; ② a solver proves that the model complies with the controls and produces a proof certificate; ③ an independent proof checker ensures the validity of the proof certificate.

to be performed by independent and industry-trusted third-party organizations. AWS, for example, works with independent third-party auditors on thousands of certifications, frameworks, and requirements worldwide. These audits are typically manual and slow. The result, both for AWS and other organizations that use IT, is that their products and services are more costly to launch, operate, and maintain.

Compliance automation has the potential to provide safety guarantees that are comparable to—or even stronger than—those provided by current techniques while dramatically lowering the cost and time required. The challenges for compliance automation in this context are usually computational in nature. In particular, showing compliance typically requires reasoning about the reachability of states in a computer system. PCI-DSS 1.3.1, for example, asks if there exists an execution that would allow unauthorized access into the defined DMZ.[10] If we attempt to check this with exhaustive testing, we must test all possible interactions of the branching behaviors of the underlying programs and hardware systems, and thus the size of the set of scenarios quickly grows intractably large.

Fortunately, automated reasoning tools can solve many of these kinds of problems efficiently in practice. Collins Aerospace, for example, uses them in the certification of airborne systems and air traffic management systems [41]. In AWS's spring 2021 audit for PCI-DSS, automated reasoning solvers were used to automatically check PCI controls on ingress and egress (PCI-DSS requirements 1.3.x), default deny-all for logical access (requirement 7.2), network security (requirement 4.1) and required logging metadata (requirements 10.x). The solver-based approach decreased the time for evidence collection and audit for relevant controls by a factor of 10.8. Furthermore, the solver-based approach provided exhaustive coverage, whereas the previous audit techniques were based on sampling.

[10]In a computer network, a DMZ, or demilitarized zone, is a subnet that separates a local network from external networks.

While these initial efforts are extremely promising, a remaining challenge is to convince auditors to accept the solver results as evidence of compliance. This is where proof certificates play a key role. If the solver produces a result that can be independently confirmed by a trusted proof checker, this provides the assurance required for the result to be accepted as evidence by auditors. Indeed, in the aforementioned AWS audit, the 3rd-party auditor extended its criteria for evidence to include the output of automated reasoning solvers that produce auditable proofs [1]. Figure 3 shows the envisioned scheme for automated regulatory compliance based on proof production and proof checking. Currently, cvc5 is the only SMT solver compatible with this workflow, and as a result, it is being used by AWS to produce evidence for compliance whenever possible (in particular, it is being used for the PCI-DSS requirements mentioned above).

6 Conclusion

Independently checkable proofs are an exciting new emerging capability in automated reasoning tools. In addition to vastly improving the trustworthiness of these tools, they have the potential to enable a host of new directions and applications, including better integration of tools and automatic generation of evidence for IT regulatory compliance.

Implementing proof production is a significant challenge. We have presented several key ideas to help address the challenge, including modular proof design, online error-checking, and the use of lazy proofs and macro steps. We have implemented these ideas in the cvc5 SMT solver, and our initial efforts have already confirmed its usefulness and viability for improving proof assistant automation and for automating regulatory compliance in an industrial setting.

References

[1] Karthik Amrutesh and Byron Cook. How I learned to stop worrying and start applying automated reasoning, 2021. https://ucl-pplv.github.io/CAV21/poster_facc_10/.

[2] Bruno Andreotti, Hanna Lachnitt, and Haniel Barbosa. Carcara: An efficient proof checker and elaborator for SMT proofs in the Alethe format. In *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 2023, Paris, France, April 24 - April 27, 2023, Proceedings*, April 2023. Paris, France (to appear).

[3] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. Semantic-based automated reasoning for AWS access policies using SMT. In Nikolaj S. Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018.

[4] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I*, volume 12651 of *Lecture Notes in Computer Science*, pages 59–75. Springer, 2021.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, 2018.

[6] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[7] Haniel Barbosa, Jasmin Christian Blanchette, Mathias Fleury, and Pascal Fontaine. Scalable fine-grained proofs for formula processing. *J. Autom. Reason.*, 64(3):485–510, 2020.

[8] Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, Yoni Zohar, Cesare Tinelli, and Clark W. Barrett. Flexible proof production in an industrial-strength SMT solver. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 15–35. Springer, 2022.

[9] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[10] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021.

[11] Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[12] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *J. Autom. Reason.*, 51(1):109–128, 2013.

[13] Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. verit: An open, trustable and efficient smt-solver. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[14] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 317–333. Springer, 2005.

[15] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification.* Springer, 2007.

[16] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Smtinterpol: An interpolating SMT solver. In Alastair F. Donaldson and David Parker, editors, *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, volume 7385 of *Lecture Notes in Computer Science*, pages 248–254. Springer, 2012.

[17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *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*, pages 220–236. Springer, 2017.

[18] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021.

[19] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Proofs and refutations, and Z3. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *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.

[20] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[21] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. Smtcoq: A plug-in for integrating SMT solvers into coq. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2017.

[22] Marijn Heule, Warren A. Hunt Jr., Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*,

pages 269–284. Springer, 2017.

[23] Marijn J. H. Heule. The DRAT format and drat-trim checker. *CoRR*, abs/1610.06229, 2016.

[24] Guy Katz, Clark W. Barrett, Cesare Tinelli, Andrew Reynolds, and Liana Hadarean. Lazy proofs for DPLL(T)-based SMT solvers. In Ruzica Piskac and Muralidhar Talupur, editors, *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 93–100. IEEE, 2016.

[25] Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn J. H. Heule. Extended resolution simulates DRAT. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 516–531. Springer, 2018.

[26] Igor Konnov. Edmund m. clarke, thomas a. henzinger, helmut veith, and roderick bloem (eds): Handbook of model checking - springer international publishing ag, cham, switzerland, 2018. *Formal Aspects Comput.*, 31(4):455–456, 2019.

[27] Peter Lammich. Efficient verified (UN)SAT certificate checking. In Leonardo de Moura, editor, *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*, pages 237–254. Springer, 2017.

[28] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In Armin Biere and Roderick Bloem, editors, *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*, pages 646–662. Springer, 2014.

[29] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In Jürgen Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005.

[30] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

[31] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[32] Andres Nötzli, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. Reconstructing fine-grained proofs of rewrites using a domain-specific language. In *2022 Formal Methods in Computer Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, pages 65–74. IEEE, 2022.

[33] Andrew Reynolds, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. Reductions for strings and regular expressions revisited. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*, pages 225–235. IEEE, 2020.

[34] Andrew Reynolds, Maverick Woo, Clark W. Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. Scaling up DPLL(T) string solvers using context-dependent simplification. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 453–474. Springer, 2017.

[35] John Alan Robinson and Andrei Voronkov. Preface. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages v–vii. Elsevier and MIT Press, 2001.

[36] Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). 336:49–54, 2021.

[37] Natarajan Shankar. Automated deduction for verification. *ACM Comput. Surv.*, 41(4):20:1–20:56, 2009.

[38] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 313–326. ACM, 2010.

[39] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. SMT proof checking using a logical framework. *Formal Methods Syst. Des.*, 42(1):91–118, 2013.

[40] The Coq development team. The coq proof assistant reference manual version 8.9, 2019.

[41] Lucas G. Wagner, Alain Mebsout, Cesare Tinelli, Darren D. Cofer, and Konrad Slind. Qualification of a model checker for avionics software verification. In Clark W. Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, volume 10227 of *Lecture Notes in Computer Science*, pages 404–419, 2017.