The TLA+ Model Checker Apalache

Rodrigo Otoni[1]✉, Shon Feder[2], Jure Kukovec[3], Andrey Kupriyanov[3], Gabriela Moreira[4], Philip Offtermatt[5], Thomas Pani[3], Thanh-Hai Tran[6], and Igor Konnov[3]

[1] University of Groningen, Groningen, Netherlands
r.b.otoni@rug.nl
[2] Tarides, Toronto, Canada
shon@tarides.com
[3] Independent Researcher, Vienna, Austria
jure.kukovec@gmail.com, andrey@kuprum.xyz, thomas@thpani.net, igor@konnov.phd
[4] Informal Systems, Joinville, Brazil
gabriela@informal.systems
[5] Informal Systems, Munich, Germany
philip.offtermatt@informal.systems
[6] Université Paris-Saclay, CEA, List, Palaiseau, France
thanh-hai.tran@cea.fr

To appear in the proceedings of CAV'26 (https://conferences.i-cav.org/2026)

Abstract. The TLA$^+$ language has been widely used, both in academia and industry, to specify and reason about distributed systems. This paper presents Apalache, an efficient and flexible symbolic model checker for TLA$^+$. Apalache's engine is based on bounded model checking, with symbolic transitions being extracted from TLA$^+$ specifications and verification conditions suitable for satisfiability modulo theories (SMT) solvers being generated from them. Reasoning can be done in terms of safety and liveness properties, with liveness checking realised via a liveness-to-safety reduction. Apalache's flexibility lies in its three complementary functionalities: bounded exhaustive verification, for bounded guarantees, randomised symbolic execution, for prototyping and bug detection, and inductiveness checking, for unbounded guarantees. The paper describes Apalache's architecture and features, including its support for PlusCal and Quint, two languages that share the same semantic foundation as TLA$^+$. Industrial usage of Apalache is also presented, together with a case study which illustrates how Apalache can be used to verify the agreement property of a consensus protocol.

Keywords: model checking · SMT solving · distributed systems

1 Introduction

Distribution has become an integral part of a large number of computer systems, providing them with essential improvements to aspects such as performance and fault-tolerance. These improvements, however, often come with the need for complex coordination strategies not required for sequential and concurrent systems.

Due to the critical nature of many of the distributed systems underpinning our modern digital infrastructure, their correctness is of crucial importance.

One way to establish guarantees about the behaviour of distributed systems is through the writing of, and reasoning about, TLA$^+$ specifications [31], an approach that has been adopted both in academia and in industry, e.g., at Amazon Web Services [40]. TLA$^+$ is a specification language based on a variant of linear-time temporal logic called temporal logic of actions (TLA) [30], which allows for the behaviour of a system to be described in a way that abstracts implementation details that do not impact high-level properties, such as memory management. Having specified a system in TLA$^+$, model checking [9] can then be employed to ensure correctness in an automated way. In addition to its adoption by Amazon [7], TLA$^+$ has many other success stories, e.g., in the context of Microsoft's Cosmos DB [16] and of MongoDB [48].

Reasoning about distributed systems is a notoriously difficult task, due to the numerous interleavings admitted by distributed algorithms, with state spaces generally growing exponentially with the number of network nodes. With the aim of aiding in the practical verification of distributed systems, this paper presents Apalache[7], an efficient and flexible symbolic model checker for TLA$^+$. At its core, Apalache extracts symbolic transitions from TLA$^+$ specifications and generates verification conditions that are checked by satisfiability modulo theories (SMT) solvers [26]. Verification can be done in terms of both safety and liveness properties. Apalache follows a bounded model checking approach and has two quantifier-free SMT encodings, based on the combination of nonlinear integer arithmetic with the theories of uninterpreted functions [22] and arrays [44]. Notably, Apalache exploits its engine to enable not only bounded exhaustive verification, but also randomised symbolic execution and inductiveness checking. By providing three complementary functionalities, Apalache allows users to obtain the guarantees they need with the computational resources they have.

The development of Apalache started at TU Wien in 2016 and has since spanned various institutions. Many successes have been achieved over the years, with Apalache being used to verify protocols for blockchain synchronization [6] and consensus [51,21,14], and algorithms for the detection of failures [49] and termination [34], as well as to aid in test case generation [28,37]. Apalache has also been used at NVIDIA for validation of safety-critical software development tooling, in the context of the ISO 26262 standard [27]. Furthermore, Apalache usage extends to other TLA-based languages, such as PlusCal [32]. Regarding terms of use, Apalache is developed under the open source Apache 2.0 license.

The remainder of this paper presents Apalache's architecture and features, some of which have not been described in any prior academic publication, e.g., randomised symbolic execution, and is structured as follows. Section 2 introduces a running example and the basics of TLA$^+$. Section 3 provides an overview of Apalache and its capabilities. Section 4 describes a case study which showcases how Apalache can be used in practice. Section 5 discusses industrial adoption and related work. Finally, Section 6 presents closing remarks and future work.

[7] Apalache is available at https://github.com/apalache-mc/apalache.

```
1  ──────────────────────── MODULE Ben_Or ────────────────────────
2  EXTENDS FiniteSets, Integers, typedefs
3  CONSTANTS N, T, F, CORRECT, FAULTY, ROUNDS
4  VARIABLES value, decision, round, step, msgs1, msgs2
5  NO_DECISION ≜ −1  alias indicating that no decision has yet been taken
6  Init ≜  predicate describing the set of initial states
7      ∧ value ∈ [CORRECT → VALUES]  nondeterministically choose the initial values
8      ∧ decision = [r ∈ CORRECT ↦ NO_DECISION]
9      ∧ round = [r ∈ CORRECT ↦ 1]
10     ∧ step = [r ∈ CORRECT ↦ S1]  correct participants start on S1, standing for step 1
11     ∧ msgs1 = [r ∈ ROUNDS ↦ {}]
12     ∧ msgs2 = [r ∈ ROUNDS ↦ {}]
13 Step1(id) ≜  send proposal message to all participants
14     LET r ≜ round[id] IN  update msgs1 and step, with M1 defining a record
15         ∧ step[id] = S1
16         ∧ msgs1′ = [msgs1 EXCEPT ![r] = @ ∪ {M1(id, r, value[id])}]
17         ∧ step′ = [step EXCEPT ![id] = S2]
18         ∧ UNCHANGED ⟨value, decision, round, msgs2⟩
19 Step2(id) ≜ ...  receive proposals and react to them (omitted for brevity)
20 Step3(id) ≜ ...  consider if a decision can be made (omitted for brevity)
21 CorrectStep ≜  predicate describing the operation of correct participants
22     ∃ id ∈ CORRECT :  correct participants execute the steps of the protocol
23         Step1(id) ∨ Step2(id) ∨ Step3(id)
24 FaultyStep ≜  predicate describing the operation of faulty participants
25     ∃ r ∈ ROUNDS :  faulty participants collectively inject messages for a single round
26         ...  omitted for brevity
27 Next ≜  predicate describing the transition relation
28     ∨ CorrectStep
29     ∨ FaultyStep
30 Agreement ≜  predicate describing the agreement property
31     ∀ id1, id2 ∈ CORRECT :  no two correct participants decide on different values
32         ∨ decision[id1] = NO_DECISION
33         ∨ decision[id2] = NO_DECISION
34         ∨ decision[id1] = decision[id2]
35 ──────────────────────────────────────────────────────────────
```

Fig. 1: Condensed specification of Ben-Or's Byzantine consensus protocol [2].

2 Running Example

To illustrate Apalache's capabilities, we use a TLA$^+$ specification of Ben-Or's Byzantine consensus protocol [2]. The protocol's relative simplicity, combined with its use of primitives present in many state-of-the-art protocols, makes it a suitable running example. A condensed version of the specification is shown in Figure 1, with the complete specification being available online[8]. In summary, the protocol operates by having the correct participants perform three steps per round, in which specific messages are exchanged in order, with faulty participants

[8] Detailed protocol specification at https://github.com/konnov/apalache-examples.

sending messages in an arbitrary manner; it is assumed that less than 1/5 of the participants are faulty.

TLA$^+$ specifications represent state machines, with the *Init* predicate describing the set of initial states and the *Next* predicate describing the transition relation. Predicates are also used to describe the actions possible, e.g., *CorrectStep* and *FaultyStep*, and the properties to be checked, e.g., *Agreement*. Notably, specifications are parametrised, meaning that constants dictating the total number of participants, N, the tolerated number of faulty participants, T, and the actual number of faulty participants, F, can be instantiated with different values. Further details on TLA$^+$ can be found in Lamport's book [31].

3 Overview

A description of Apalache's main components is given in Section 3.1. The type system enforced is discussed in Section 3.2. The core of Apalache, its SMT-based model checking engine for handling safety properties, is detailed in Section 3.3. Liveness checking is covered in Section 3.4. Lastly, Apalache's support for other TLA-based languages is described in Section 3.5.

3.1 Architecture

The high-level architecture of Apalache is given in Figure 2. Apalache works via a command-line interface and receives as input a TLA$^+$ specification. In TLA$^+$ the properties are traditionally written as part of the specification, with the user being able to select the property to be checked via a command-line option; deadlock absence is checked if no property is provided. The output is either a statement that no errors were found during the search, together with a trace which illustrates a valid protocol execution, or the summary of a violation, together with a concrete counterexample (CEX). By default, exhaustive search is conducted for 10 unrollings of the transition relation of the specified protocol; this and other parameters can be changed via command-line options.

Apalache uses SANY [15] for parsing the TLA$^+$ input and its own type checker, called Snowcat, to ensure type correctness. The preprocessor performs

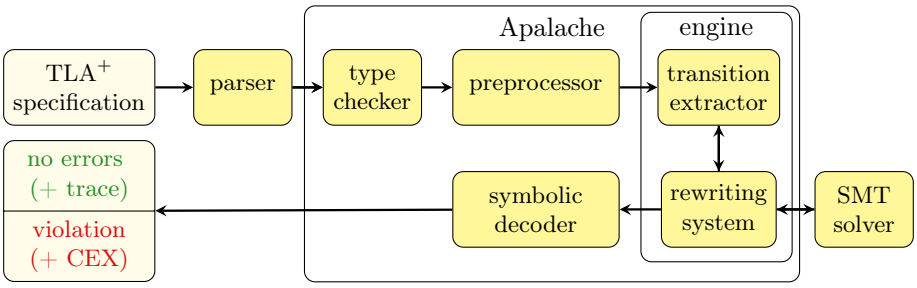

Fig. 2: Architecture of Apalache.

$$
\begin{aligned}
\langle \mathrm{T} \rangle ::= & \texttt{Bool} \mid \texttt{Int} \mid \texttt{Str} \mid \langle \mathrm{T} \rangle \to \langle \mathrm{T} \rangle \mid \texttt{Set}(\langle \mathrm{T} \rangle) \mid \texttt{Seq}(\langle \mathrm{T} \rangle) \mid \\
& \ll \langle \mathrm{T} \rangle, \ldots, \langle \mathrm{T} \rangle \gg \mid (\langle \mathrm{T} \rangle, \ldots, \langle \mathrm{T} \rangle) \Rightarrow \langle \mathrm{T} \rangle \mid \langle \text{typeConst} \rangle \mid \\
& \langle \text{typeVar} \rangle \mid \texttt{Variant}(\langle \text{typeVar} \rangle) \mid (\langle \mathrm{T} \rangle) \mid \\
& \langle \text{variantOption} \rangle \mid \ldots \mid \langle \text{variantOption} \rangle \mid \\
& \{ \langle \text{field} \rangle : \langle \mathrm{T} \rangle, \ldots, \langle \text{field} \rangle : \langle \mathrm{T} \rangle \} \\
\langle \text{field} \rangle ::= & \textit{ an identifier that matches } \texttt{[a-zA-Z\_][a-zA-Z0-9\_]}^* \\
\langle \text{typeConst} \rangle ::= & \textit{ an identifier that matches } \texttt{[A-Z\_][A-Z0-9\_]}^* \\
\langle \text{variantOption} \rangle ::= & \textit{ an identifier that matches } \texttt{[A-Z\_][A-Z0-9\_]}^* \\
\langle \text{typeVar} \rangle ::= & \textit{ a single letter from } \texttt{[a-z]}
\end{aligned}
$$

Fig. 3: Grammar of Apalache's type system.

a number of simplifications to improve checking efficiency, including inlining operator calls and normalising expressions to be in negation normal form. The model checking engine consists of two modules: the transition extractor, which generates a symbolic transition system from the specification, and the rewriting system, which generates verification conditions in terms of SMT queries. An off-the-shelf solver is used to solve the SMT queries, with Apalache currently employing Z3 [38]. The symbolic decoder produces, in case of a violation, a concrete CEX trace that reaches a state violating the property being checked.

3.2 Type System

TLA$^+$ was not designed with automated reasoning in mind, with the language being untyped as a result. Typing, however, is very useful for verification, especially in regards to the generation of SMT queries. As a way to benefit from typing while not altering the language, Apalache makes use of annotations to add types to TLA$^+$ specifications. These annotations, whose grammar can be seen in Figure 3, are lightweight and automatically checked by Snowcat; concretely, all definitions are checked for type correctness w.r.t. the type system. Usually, the user simply has to add types to constants and variables, since Apalache can infer the types of most operators [25]; in our experience AI tools can produce such annotations effortlessly. Annotations for the constants and variables in the running example can be seen in Figure 4. Further details on both Snowcat and the type system can found in Apalache's documentation[9].

3.3 Model Checking Engine

Apalache's engine generates a symbolic transition system [26] and then uses its rewriting system to generate SMT formulas in the theory of quantifier-free

[9]Typing details at https://apalache-mc.org/docs/tutorials/snowcat-tutorial.html.

CONSTANTS

@type: *Int*;
N, total number of participants
@type: *Int*;
T, bound on the number of faulty participants
@type: *Int*;
F, actual number of faulty participants
@type: *Set*(*REPLICA*);
CORRECT, set of the correct participants
@type: *Set*(*REPLICA*);
FAULTY, set of the faulty participants
@type: *Set*(*Int*);
ROUNDS set of rounds

VARIABLES

@type: *REPLICA* → *Int*;
value, current value of a participant
@type: *REPLICA* → *Int*;
decision, decision of a participant
@type: *REPLICA* → *Int*;
round, round number of a participant
@type: *REPLICA* → *STEP*;
step, participant step: *S*1, *S*2, *S*3
@type: *Int* → *Set*(*$msgOne*);
*msgs*1, type–1 messages sent, mapped by rounds
@type: *Int* → *Set*(*$msgTwo*);
*msgs*2 type–2 messages sent, mapped by rounds

Fig. 4: Types of the running example; $*msgOne* and $*msgTwo* are type aliases, standing for the record $\{src : REPLICA,\ r : Int,\ v : Int\}$ and for the variant $Q(\{src : REPLICA,\ r : Int\}) \mid D(\{src : REPLICA,\ r : Int,\ v : Int\})$.

uninterpreted functions and nonlinear integer arithmetic (QF_UFNIA) [22] or quantifier-free arrays, uninterpreted functions, and nonlinear integer arithmetic (QF_AUFNIA) [44]. The use of arrays tends to be beneficial in specifications that make ample use of sets and functions, but can lead to a slowdown if mixed data structures are used, e.g., records of sets, or in case of nesting, e.g., sets of sets, since Apalache relies on the extensionality axiom. Note that arithmetic nonlinearity is not essential, being only present if the TLA$^+$ specification has nonlinear constraints, with the solver being free to select the suitable theory.

In its default mode, the engine incrementally generates SMT formulas to check if a state violating the property can be reached in 0, 1, 2, ..., K unrollings, with K being a user-defined bound. This is illustrated on the right, with $\cdot$ representing the unrolling of the transition relation and Inv being a given property. In addition to bounding the number of unrollings, the data structures used in the specification need to be finite. Apalache can thus check for bounded safety in a fully automated manner. The engine's default mode can also be used for unbounded verification, if the user is able to provide it with a suitable inductive invariant, as described in Section 4.3.

$$
\begin{aligned}
&0 : Init \wedge \neg Inv \\
&1 : (Init \cdot Next) \wedge \neg Inv \\
&2 : (Init \cdot Next \cdot Next) \wedge \neg Inv \\
&\qquad \ldots \\
&K : (Init \cdot Next \ldots Next) \wedge \neg Inv
\end{aligned}
$$

In addition to exhaustive search, the engine is also used to perform randomised symbolic execution, called *simulation* in Apalache. In simulation mode the engine picks a symbolic trace, by randomly selecting enabled protocol actions at each unrolling, and then checks if the property holds on all the executions covered by this trace. A simulation of length 3 is illustrated on the right, in which three partici-

$$
\begin{aligned}
&0 : Init \wedge \neg Inv \\
&1 : (Init \cdot Step1(A)) \wedge \neg Inv \\
&2 : (Init \cdot Step1(A) \cdot Step1(B)) \wedge \neg Inv \\
&3 : (Init \cdot Step1(A) \cdot Step1(B) \cdot Step1(C)) \wedge \neg Inv
\end{aligned}
$$

pants execute the first protocol step; ids A, B, and C are left for the solver to instantiate, with different simulations potentially picking different symbolic traces. Although not capable of providing correctness guarantees, due to the loss of completeness, simulation is much faster and can be used for prototyping and bug detection. Since the exploration of a single trace yields limited information by itself, Apalache allows users to define how many simulations are desired, with hundreds normally possible in a timely manner.

3.4 Liveness Checking

For the checking of liveness properties Apalache employs a liveness-to-safety reduction [41], inspired by the work of Biere et al. [3]. This reductions allows users to reason about liveness in both model checking and simulation modes. It works by taking a TLA$^+$ specification S and a temporal property P and generating an instrumented version of S, called S_P, with a desired invariant I_P, such that P holds in S if and only if I_P is an invariant in S_P. Importantly, liveness checking is complete only for finite-state systems currently. Both the reduction and the checking of I_P are automated by Apalache; the user can also request Apalache to emit the instrumented specification into a file, so that it can be provided to other tools.

3.5 Support Beyond TLA+

Besides TLA$^+$, Apalache also provides support for two other languages whose semantics are based on TLA, namely PlusCal and Quint.

PlusCal. Positioned as an algorithm language that can be used to replace pseudo-code, PlusCal was inspired by TLA$^+$ from its inception [32]. Due to this close relation, the machinery of Apalache can easily be used to reason about PlusCal code. The only caveat is that, similar to how it is done for TLA$^+$, users are required to add type annotations to the code; further details can be found on Apalache's documentation[10].

Quint. Envisioned as a version of TLA$^+$ more approachable to engineers, Quint is a language that provides an alternative surface syntax to TLA. Apalache is integrated into Quint tooling[11], which makes its use seamless, and it has been employed to verify Quint specifications of the ChonkyBFT protocol for Byzantine fault-tolerant consensus [14] and the Aztec governance smart contracts [24].

4 Case Study

Our running example is a consensus protocol, and as the name suggests, it aims to ensure that all (correct) participants reach the same decision, e.g., all replicas

[10] PlusCal details at https://apalache-mc.org/docs/tutorials/pluscal-tutorial.html.
[11] Quint details at https://quint.sh/docs/model-checkers#apalache.

of a database decide to update a given entry with the same value. A central part of consensus is the *agreement* property, which dictates that no two correct participants may decide differently. The usage of Apalache's three functionalities is showcased in this context, as follows: bounded verification is illustrated in Section 4.1, symbolic execution is covered in Section 4.2, and inductiveness checking is detailed in Section 4.3. All experiments were done using Apalache v0.52.1 on a Linux laptop with an Intel Core i7-8650U CPU and 16 GB of RAM. For an extensive evaluation of Apalache's performance on a range of protocols the reader is referred to [22,44], whose results are summarised in Section 4.4.

4.1 Bounded Exhaustive Verification

To verify that agreement holds up to a bound we use the `check` command, with Apalache expecting a user-defined instantiation of a given protocol. The `n6t1f1.tla` file contains a protocol instance with 6 participants, `n6`, a tolerated limit of 1 faulty participant, `t1`, and 1 actual faulty participant, `f1`.

```
$ apalache-mc check --inv=Agreement --length=10 n6t1f1.tla
...
Checker reports no error up to computation length 10
It took me 0 days 0 hours 26 min 43 sec
```

As can be seen, the guarantees provided by `check` can be costly to achieve, due to the combinatorial explosion. Clever abstractions are often needed to make effective use of the model checking engine. As shown in Figure 1, *Next* allows faulty participants to inject messages at every protocol step, so Apalache has to consider all possible points in which this can happen. Since the protocol is asynchronous, and thus sent messages can be received at any point, an equivalent scenario can be achieved by injecting the messages from the faulty participants in the *Init* predicate. An alternative to *Init* is shown on the left. Since the faulty messages have been introduced in advance, the transition relation can now become the predicate *CorrectStep*.

$$
\begin{array}{l}
\mathit{InitWithFaults} \triangleq \\
\quad \text{msgs1 and msgs2 are nondeterministically initialised with faulty messages} \\
\quad \land \exists\, F1 \in \text{SUBSET}\ [\mathit{src} : \mathit{FAULTY},\ r : \mathit{ROUNDS},\ v : \mathit{VALUES}] : \\
\qquad \mathit{msgs}1 = [r \in \mathit{ROUNDS} \mapsto \{m \in F1 : m.r = r\}] \\
\quad \land \exists\, F1D \in \text{SUBSET}\ [\mathit{src} : \mathit{FAULTY},\ r : \mathit{ROUNDS},\ v : \mathit{VALUES}], \\
\qquad F1Q \in \text{SUBSET}\ [\mathit{src} : \mathit{FAULTY},\ r : \mathit{ROUNDS}] : \\
\qquad \mathit{msgs}2 = [r \in \mathit{ROUNDS} \mapsto \\
\qquad\quad \{D2(mm.\mathit{src},\ r,\ mm.v) : mm \in \{m \in F1D : m.r = r\}\} \\
\qquad\qquad \cup\, \{Q2(mm.\mathit{src},\ r) : mm \in \{m \in F1Q : m.r = r\}\}] \\
\quad \land \ \ldots \quad \textit{value}, \textit{decision}, \textit{round}, \text{ and } \textit{step} \text{ are as per } \textit{Init}
\end{array}
$$

```
$ apalache-mc check --init=InitWithFaults --next=CorrectStep
  --inv=Agreement --length=10 n6t1f1.tla
...
Checker reports no error up to computation length 10
It took me 0 days 0 hours 1 min 24 sec
```

With the improved specification, it is also possible to efficiently verify that agreement does not hold if there are more faulty participants than tolerated. Checking the `n6t1f2.tla` file, which instantiates 2 faulty participants, leads to a violation being found and a CEX returned. The CEX includes a state in which two participants decide on different values, as captured by the value of *decision*, i.e., $decision = SetAsFun(\{\ll \text{“0”}, -1 \gg, \ll \text{“1”}, -1 \gg, \ll \text{“2”}, 0 \gg, \ll \text{“3”}, 1 \gg\})$; -1 represents that no decision has yet been made.

4.2 Randomised Symbolic Execution

Exhaustive verification tends to become increasingly expensive as protocols become more intricate, and as more participants and protocol rounds are considered. For instance, while checking `n6t1f1.tla` with `--length=10` takes less than two minutes, checking it with `--length=15` requires ∼47 hours. This is explained by the presence in the specification of elements such as power sets, written in TLA$^+$ as SUBSET S for a given set S, and the cardinality function, which computes the number of elements of a finite set. These can be seen in *Step*2 on the right. To be able to perform fast reasoning during prototyping and potentially detect vulnerabilities cheaply, we can make use of the `simulate` command; a batch of 100 independent simulations of the file `n6t1f1.tla` with `--length=15`, via option `--max-run=100`, takes ∼17 minutes. Note that, although relatively slow in comparison to simulators for other formalisms, Apalache's simulation mode has to handle the richness of TLA$^+$ and its exploration is done in terms of symbolic traces rather than explicit ones.

```
Step2(id) ≜  receive proposals and react to them
  LET r ≜ round[id] IN
  ∧ ∃ received ∈ SUBSET msgs1[r] :
      wait until type–1 messages are received from N − T processes
      ∧ Cardinality(Senders1(received)) ≥ N − T
      ∧ LET Weights ≜ [v ∈ VALUES ↦
          Cardinality(Senders1({m ∈ received : m.v = v}))]
        IN
        ... omitted for brevity
```

4.3 Inductiveness Checking

To ensure that agreement holds for an arbitrary number of rounds, Apalache requires the user to provide an inductive invariant that implies agreement. Designing, or rather discovering, such an invariant is a highly nontrivial task, whose details are outside the scope of this paper. As a measure of the human effort needed, it took ∼15 hours of work for a co-author experienced in protocol specification to discover a suitable collection of lemmas, 13 in total, for our example.

Once the inductive invariant is at hand, Apalache can provide unbounded guarantees via three `check` calls. The first one establishes the base case, i.e., that the invariant holds in the initial states, the second one checks that the invariant implies the property, and the third one verifies the inductive case, i.e., that if the invariant holds in one state it keeps holding after one transition. Note that unboundedness is considered here only in terms of unrollings, i.e., number of rounds, with parameters such as number of nodes still being bounded.

Inductiveness checking of our example can be achieved via the the calls below. The first two checks of `n6t1f1_inductive.tla`, which contains all the needed lemmas, are relatively cheap, but checking the inductive case is more expensive. The *IndInit* predicate is an extension of *IndInv* that gives shape to the variable domains, so that it can describe initial states.

```
$ apalache-mc check --init=InitWithFaults --next=CorrectStep
  --inv=IndInv --length=0 n6t1f1_inductive.tla
...
Checker reports no error up to computation length 0
It took me 0 days 0 hours 0 min 7 sec
```

```
$ apalache-mc check --init=IndInit --next=CorrectStep
  --inv=Agreement --length=0 n6t1f1_inductive.tla
...
Checker reports no error up to computation length 0
It took me 0 days 0 hours 12 min 28 sec
```

```
$ apalache-mc check --init=IndInit --next=CorrectStep
  --inv=IndInv --length=1 n6t1f1_inductive.tla
...
Checker reports no error up to computation length 1
It took me 0 days 19 hours 26 min 15 sec
```

4.4 Evaluation on Other Protocols

Detailed quantitative evaluations of Apalache, including comparisons with the well-established explicit-state model checker TLC [52], are reported in [22,44].

In [22], ten protocols of varying complexity were considered, from the relatively simple Bakery algorithm for mutual exclusion to the much more complex Paxos and Raft consensus algorithms. Experiments were carried out for both bounded model checking and inductiveness checking, considering different properties and numbers of participants, with 40 experiment instances conducted in total. The results, obtained using Apalache's QF_UFNIA encoding, show that Apalache can outperform TLC in terms of runtime and memory use in many cases. Notably, Apalache could provide results in 8 cases in which TLC could not, given the available resources. In some scenarios, however, primarily involving smaller benchmarks or those that made extensive use of nondeterminism, TLC performed better than Apalache w.r.t. runtime, memory use, or both.

In [44], three protocols were considered, for Byzantine agreement, consensus with Byzantine faults, and non-blocking atomic commitment. Experiments were carried out for bounded model checking, considering the agreement property and different numbers of participants, with with 36 individual experiments conducted in total. A four-fold comparison was conducted between Apalache's

QF_UFNIA encoding, two variations of Apalache's QF_AUFNIA encoding, and TLC. The results obtained indicate that Apalache scales better than TLC, particularly the QF_AUFNIA encoding, with TLC having better performance on the scenarios with smaller state spaces; in 21 cases Apalache could provide a result with the available resources but TLC could not.

It is important to note that Apalache and TLC complement each other, and we do not necessarily see Apalache as a replacement of TLC. Apalache is more suited to scenarios with large data structures. In particular, the symbolic nature of Apalache enables it to handle Byzantine fault-tolerant algorithms, such as Ben-Or's consensus, whereas TLC normally explodes when facing such algorithms. On the other hand, TLC is more performant when dealing with benchmarks with compact state spaces, since Apalache has the initial overhead of generating a relatively large SMT formula; for large benchmarks model checking time is dominated by solving time though. Furthermore, TLC tends to perform well with specifications that rely heavily on control nondeterminism.

5 Industrial Adoption and Related Work

There are reported uses of Apalache by researchers and engineers of many organisations, including Informal Systems [6,49,28], the Stellar Foundation [51,34], Matter Labs [14,20], and NVIDIA [27], with interest also arising from institutions such as the Ethereum Foundation [21], Aztec Labs [24], and Commonware [10]. These examples showcase Apalache's maturity and impact, and its ability to provide value in industrial settings.

Besides Apalache, two other tools are prevalent in the TLA$^+$ ecosystem [23]: the explicit-state model checker TLC [52] and the interactive proof system TLAPS [8]. TLC and Apalache have different strengths, as detailed in Section 4.4, and TLAPS focuses on semi-automated proofs, also being powered by SMT solving. A translator from TLA$^+$ to B is also available [18], which allows for the use of ProB [33] for TLA$^+$ verification. When it comes to inductive invariant inference, approaches have been proposed for both TLA$^+$ [47,11] and other formalisms [17,50].

In regards to TLA$^+$ itself, related formalisms include Ivy [35,36], Veil [46], Alloy [19], and P [12], each with their own strengths, weaknesses, and tooling support. Ivy operates at the level of first-order relations, a different abstraction level than TLA$^+$, and is associated with a verification tool of the same name which supports both model checking and deductive verification; Ivy's abstractions are more amenable for automated reasoning, but in our experience distributed systems engineers in industry have a harder time understanding first-order relations than the state machine representation that underpins TLA$^+$. Veil is a framework that operates on a language closely related to Ivy, supporting both automated and interactive verification, with the distinct feature of being embedded into the Lean 4 proof assistant [39]. Alloy is similar to TLA$^+$ in its focus on transition systems, but is better suited to describe data relations than state progress; bounded verification is possible via the Alloy Analyzer. P is a state machine-

based programming language centred on asynchronous events, associated with a framework grounded on assume-guarantee reasoning [13]; initially restricted to systematic testing, supporting tooling is now capable of formal verification, and a security-focused variant of the language is also available [29].

6 Conclusion

This paper presented Apalache, an efficient and flexible TLA$^+$ model checker. Apalache makes use of bounded model checking and SMT solving to provide users with three complementary functionalities: bounded exhaustive verification, randomised symbolic execution, and inductiveness checking. Its symbolic engine is capable of reasoning about safety and liveness properties, and has also been employed in the context of two other TLA-based languages, PlusCal and Quint. Industrial usage of Apalache has led to many successes over the years, including the verification of state-of-the-art Byzantine fault tolerant consensus protocols powering modern blockchain platforms.

Future work is envisioned in three directions: (1) add support for unbounded model checking in a native and fully automated way, (2) lift the finite-state restriction from liveness checking, and (3) improve the tool to enable further flexibility. The first direction could consider unboundedness in terms of unrollings or data structure size. It can be pursued via a quantified SMT encoding or one based on Horn clauses, with the latter allowing for novel algorithms such as (SPLIT-)TPA [5,4] to be exploited. Furthermore, certification of solving results [42,43] could be incorporated into Apalache to enhance the strength of its guarantees. The second direction could be centred around the design of an approach based on the technique proposed by Padon et al. [45]. Lastly, the third direction could involve adding bindings for alternative solvers, such as cvc5 [1].

Acknowledgments. The authors are grateful for the institutional support received over the years from TU Wien (2016-2020), Inria Nancy and LORIA (2018-2019), and Informal Systems (2020-2024), as well as for all the past funding that was received from the Vienna Science and Technology Fund (project APALACHE ICT 15-103 in particular), Vienna Business Agency, Austrian Science Fund, Interchain Foundation, and Swiss National Science Foundation.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Data-Availability Statement. A self-contained artifact consisting of Apalache's source code (v0.52.1), built binary, and documentation, together with all files relating to our running example and instructions on how to reproduce all reported experiments, is available at https://doi.org/10.5281/zenodo.19696472.

References

1. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M.,

Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'22). pp. 415–442 (2022). https://doi.org/10.1007/978-3-030-99524-9_24

2. Ben-Or, M.: Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols. In: Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing (PODC'83). p. 27–30 (1983). https://doi.org/10.1145/800221.806707
3. Biere, A., Heljanko, K., Junttila, T., Latvala, T., Schuppan, V.: Linear Encodings of Bounded LTL Model Checking. Logical Methods in Computer Science **2**(5), 1–63 (2006). https://doi.org/10.2168/LMCS-2(5:5)2006
4. Blicha, M., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Split Transition Power Abstraction for Unbounded Safety. In: Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design (FMCAD'22). pp. 349–358 (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_42
5. Blicha, M., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Transition Power Abstractions for Deep Counterexample Detection. In: Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'22). pp. 524–542 (2022). https://doi.org/10.1007/978-3-030-99524-9_29
6. Braithwaite, S., Buchman, E., Konnov, I., Milosevic, Z., Stoilkovska, I., Widder, J., Zamfir, A.: Tendermint Blockchain Synchronization: Formal Specification and Model Checking. In: Proceedings of the 9th International Symposium on Leveraging Applications of Formal Methods (ISoLA'20). pp. 471–488 (2020). https://doi.org/10.1007/978-3-030-61362-4_27
7. Brooker, M., Desai, A.: Systems Correctness Practices at Amazon Web Services. Communications of the ACM **68**(6), 38–42 (2025). https://doi.org/10.1145/3729175
8. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: The TLA+ Proof System: Building a Heterogeneous Verification Platform. In: Proceedings of the 7th International Colloquium on the Theoretical Aspects of Computing (ICTAC'10). pp. 44–44 (2010). https://doi.org/10.1007/978-3-642-14808-8_3
9. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-10575-8
10. Commonware Inc.: Minimmit Formal Specification (2025), https://github.com/commonwarexyz/monorepo/tree/main/pipeline/minimmit/quint
11. Dardik, I., Kang, E.: Compositional Inductive Invariant Inference via Assume-Guarantee Reasoning (2025). https://doi.org/10.48550/arXiv.2509.06250
12. Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S., Zufferey, D.: P: Safe Asynchronous Event-Driven Programming. ACM SIGPLAN Notices **48**(6), 321–332 (2013). https://doi.org/10.1145/2499370.2462184
13. Desai, A., Phanishayee, A., Qadeer, S., Seshia, S.A.: Compositional Programming and Testing of Dynamic Distributed Systems. Proceedings of the ACM on Programming Languages **2**(OOPSLA), 1–30 (2018). https://doi.org/10.1145/3276529
14. França, B., Kolegov, D., Konnov, I., Prusak, G.: ChonkyBFT: Consensus Protocol of ZKsync (2025). https://doi.org/10.48550/arXiv.2503.15380
15. Grégoire, J.C., Jefferson, D., Yu, Y.: SANY Syntactic Analyzer, https://lamport.azurewebsites.net/tla/tools.html

16. Hackett, F., Rowe, J., Kuppe, M.A.: Understanding Inconsistency in Azure Cosmos DB with TLA+. In: Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'23). p. 1–12 (2023). https://doi.org/10.1109/ICSE-SEIP58684.2023.00006
17. Hance, T., Heule, M., Martins, R., Parno, B.: Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In: Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21). pp. 115–131 (2021), https://www.usenix.org/conference/nsdi21/presentation/hance
18. Hansen, D., Leuschel, M.: Translating TLA+ to B for Validation with ProB. In: Proceedings of the 9th International Conference on Integrated Formal Methods (IFM'12). pp. 24–38 (2012). https://doi.org/10.1007/978-3-642-30729-4_3
19. Jackson, D.: Alloy: a Language and Tool for Exploring Software Designs. Communications of the ACM **62**(9), 66–76 (2019). https://doi.org/10.1145/3338843
20. Kolegov, D., Konnov, I.: ZKsync Governance's Quint Specification (2024), https://github.com/zksync-association/zk-governance/tree/master/spec
21. Konnov, I., Kukovec, J., Pani, T., Saltini, R., Tran, T.H.: Exploring Automatic Model-Checking of the Ethereum Specification (2025). https://doi.org/10.48550/arXiv.2501.07958
22. Konnov, I., Kukovec, J., Tran, T.H.: TLA+ Model Checking Made Symbolic. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–30 (2019). https://doi.org/10.1145/3360549
23. Konnov, I., Kuppe, M., Merz, S.: Specification and Verification with the TLA+ Trifecta: TLC, Apalache, and TLAPS. In: Proceedings of the 11th International Symposium On Leveraging Applications of Formal Methods (ISoLA'22). pp. 88–105 (2022). https://doi.org/10.1007/978-3-031-19849-6_6
24. Konnov, I., Pani, T.: Aztec Governance Formal Specification and Verification (2025), https://github.com/konnov/aztec-governance-formal-verification-2025q3
25. Kukovec, J., Konnov, I.: Type Inference for TLA+ in Apalache. In: Summaries of the 2020 TLA+ Community Event (2020), https://conf.tlapl.us/2020/07-Kukovec_and_Konnov-Type_Inference_for_TLA_+_in_Apalache.pdf
26. Kukovec, J., Tran, T.H., Konnov, I.: Extracting Symbolic Transitions from TLA+ Specifications. Science of Computer Programming **187**, 102361 (2020). https://doi.org/10.1016/j.scico.2019.102361
27. Kulagin, D.: Validation Test Suite for TLA+ (2025), https://github.com/tlaplus/ValidationTestSuite
28. Kupriyanov, A., Konnov, I.: Model-Based Testing with TLA+ and Apalache. In: Summaries of the 2020 TLA+ Community Event (2020), https://conf.tlapl.us/2020/09-Kuprianov_and_Konnov-Model-based_testing_with_TLA_+_and_Apalache.pdf
29. Kushwah, S., Desai, A., Subramanyan, P., Seshia, S.A.: PSec: Programming Secure Distributed Systems using Enclaves. In: Proceedings of the 16th ACM Asia Conference on Computer and Communications Security (ASIA CCS'21). p. 802–816 (2021). https://doi.org/10.1145/3433210.3453113
30. Lamport, L.: The Temporal Logic of Actions. ACM Transactions on Programming Languages and Systems **16**(3), 872–923 (1994). https://doi.org/10.1145/177492.177726
31. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Professional (2002), https://lamport.azurewebsites.net/tla/book.html

32. Lamport, L.: The PlusCal Algorithm Language. In: Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing (ICTAC'09). pp. 36–60 (2009). https://doi.org/10.1007/978-3-642-03466-4_2
33. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Proceedings of the 12th International Symposium of Formal Methods Europe (FME'03). pp. 855–874 (2003). https://doi.org/10.1007/978-3-540-45236-2_46
34. Losa, G., Merz, S.: Distributed Termination Detection - Verified with Apalache, Isabelle/HOL, and TLAPS (2022), https://github.com/nano-o/distributed-termination-detection
35. McMillan, K.L., Padon, O.: Deductive Verification in Decidable Fragments with Ivy. In: Proceedings of the 25th International Static Analysis Symposium (SAS'18). pp. 43–55 (2018). https://doi.org/10.1007/978-3-319-99725-4_4
36. McMillan, K.L., Padon, O.: Ivy: A Multi-Modal Verification Tool for Distributed Algorithms. In: Proceedings of the 32nd International Conference on Computer Aided Verification (CAV'20). p. 190–202 (2020). https://doi.org/10.1007/978-3-030-53291-8_12
37. Moreira, G., Vasconcellos, C., Kniess, J.: Fully-Tested Code Generation from TLA+ Specifications. In: Proceedings of the 7th Brazilian Symposium on Systematic and Automated Software Testing (SAST'22). p. 19–28 (2022). https://doi.org/10.1145/3559744.3559747
38. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08). pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
39. de Moura, L., Ullrich, S.: The Lean 4 Theorem Prover and Programming Language. In: Proceedings of the 28th International Conference on Automated Deduction (CADE'21). pp. 625–635 (2021). https://doi.org/10.1007/978-3-030-79876-5_37
40. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses Formal Methods. Communications of the ACM **58**(4), 66–73 (2015). https://doi.org/10.1145/2699417
41. Offtermatt, P., Kukovec, J., Konnov, I.: Extending Apalache to Symbolically Reason about Temporal Properties of TLA+. In: Summaries of the 2022 TLA+ Conference (2022), https://conf.tlapl.us/2022/sub3.pdf
42. Otoni, R., Blicha, M., Eugster, P., Sharygina, N.: Validation of CHC Satisfiability with ATHENA. Formal Aspects of Computing **37**(4), 1–20 (2025). https://doi.org/10.1145/3716505
43. Otoni, R., Blicha, M., Rivera, M.B., Eugster, P., Kofroň, J., Sharygina, N.: Unsatisfiability Proofs for Horn Solving. In: Proceedings of the 31st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'25). pp. 67–87 (2025). https://doi.org/10.1007/978-3-031-90653-4_4
44. Otoni, R., Konnov, I., Kukovec, J., Eugster, P., Sharygina, N.: Symbolic Model Checking for TLA+ Made Faster. In: Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'23). pp. 126–144 (2023). https://doi.org/10.1007/978-3-031-30823-9_7
45. Padon, O., Hoenicke, J., Losa, G., Podelski, A., Sagiv, M., Shoham, S.: Reducing Liveness to Safety in First-Order Logic. Proceedings of the ACM on Programming Languages **2**(POPL), 1–33 (2017). https://doi.org/10.1145/3158114
46. Pîrlea, G., Gladshtein, V., Kinsbruner, E., Zhao, Q., Sergey, I.: Veil: A Framework for Automated and Interactive Verification of Transition Systems. In: Proceedings of the 37th International Conference on Computer Aided Verification (CAV'25). pp. 26–41 (2025). https://doi.org/10.1007/978-3-031-98682-6_2

47. Schultz, W., Dardik, I., Tripakis, S.: Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA+. In: Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design. pp. 273–283 (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_34
48. Schultz, W., Demirbas, M.: Design and Modular Verification of Distributed Transactions in MongoDB. Proceedings of the VLDB Endowment **18**(12), 5045–5058 (2025). https://doi.org/10.14778/3750601.3750626
49. Tran, T.H., Konnov, I., Widder, J.: A Case Study on Parametric Verification of Failure Detectors. In: Proceedings of the 41st International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE'21). pp. 138–156 (2021). https://doi.org/10.1007/978-3-030-78089-0_8
50. Yao, J., Tao, R., Gu, R., Nieh, J.: DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In: Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22). pp. 485–501 (2022), https://www.usenix.org/conference/osdi22/presentation/yao
51. Yu, Q., Losa, G., Wang, X.: TetraBFT: Reducing Latency of Unauthenticated, Responsive BFT Consensus. In: Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing (PODC'24). p. 257–267 (2024). https://doi.org/10.1145/3662158.3662783
52. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA+ Specifications. In: Proceedings of the 10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99). pp. 54–66 (1999). https://doi.org/10.1007/3-540-48153-2_6