Proof Certificates for SMT-based Model Checkers for Infinite-state Systems

Alain Mebsout Cesare Tinelli
The University of Iowa, Iowa City, IA, USA

Abstract—We present a dual technique for generating and verifying proof certificates in SMT-based model checkers, focusing on proofs of invariant properties. Certificates for two major model checking algorithms are extracted as \( k \)-inductive invariants, minimized and then reduced to a formal proof term with the help of an independent proof-producing SMT solver. SMT-based model checkers typically translate input problems into an internal first-order logic representation. In our approach, the correctness of translation from the model checker's input to the internal representation is verified in a lightweight manner by proving the observational equivalence between the results of two independent translations. This second proof is done by the model checker itself and generates in turn its own proof certificate. Our experimental evaluation show that, at the price of minimal instrumentation in the model checker, the approach allows one to efficiently generate and verify proof certificates for non-trivial transition systems and invariance queries.

I. Introduction

Model checkers are perhaps among the most successful formal methods tools in term of industrial use, particularly for the development of safety-critical systems. In addition to traditional applications in hardware design, they are increasingly used in model-based software development to analyze, for instance, models of embedded systems in the aerospace or automotive industry. One clear strength of model checkers, as opposed to proof assistants, is their ability to return precise error traces witnessing the violation of a given safety property. In addition to being invaluable to help identify and correct bugs, error traces also represent a checkable unsafety certificate. In contrast, most model checkers are currently unable to return any form of corroborating evidence when they declare a safety property to be satisfied by a system under analysis. This is unsatisfactory in general since model checker are complex tools, based on a variety of sophisticated algorithms and search heuristics, and so are not immune to errors.

To mitigate this problem, a possible approach is to use a model checker whose correctness has been formally verified \(^\text{[10]}\). An alternative is to instrument the model checker so that it is certifying, i.e. it accompanies its safety claims with a proof certificate, an artifact embodying a proof of the claim \(^\text{[16]}\). The certificate can then be validated by a trusted certificate checker. While the former approach may seem better at first, based on the fact that the model checker is verified once and for all, it has a number of disadvantages. To start, the effort is normally enormous since there are no general frameworks for verifying modern model checkers. Moreover, any modifications to the originally verified tool requires proofs to be redone. In more extreme cases (e.g., an in-depth modification) one may have to invest the same amount of effort as for the original correctness proof. The main advantage of the second approach is that it requires a much smaller human effort. A disadvantage of course is that every safety claim made by the model checker incurs the cost of generating and then checking the corresponding certificate. This is feasible in general only if such certificates are small and/or simple enough to be checkable by a target certificate checker in a reasonable amount of time (say, with at most an order of magnitude slowdown).

By reducing the trusted core to the certificate checker, certifying model checking facilitates the integration of formal method tools into safety critical processes such as those endorsed by the DO-178C guidelines for avionics software. In the spirit of the de Bruijn criterion \(^\text{[4]}\), traditionally applied to theorem provers, it redirects tool qualification requirements from a complex tool, the model checker, to a much simpler one, the proof checker.

We present an approach for generating and verifying proof certificates for SMT-based model checkers. These tools use a variety of model checking techniques and some of them even employ a portfolio approach by running several engines in parallel. Input models are typically represented internally as transition systems encoded in some fragment of first-order logic. Safety properties are expressed as invariant properties and reasoning about invariance is reduced to checking the satisfiability of formulas in certain logical theories such as integer or real linear arithmetic. The latter problem is then delegated to off-the-shelf SMT solvers.

We describe how to generate intermediate certificates that show that a given safety property is satisfied the internal transition system. These certificates are designed to be checkable by an SMT solver. Since SMT solvers themselves are complex artifacts, we also show how to reduce the validity of these certificates to proof objects obtained by a proof-producing SMT solver. This reduction capitalizes specifically on the recent proof production capabilities of the SMT solver CVC4 \(^\text{[5]}\) and the availability of an efficient proof checker for its proofs, which are generated in LFSC format \(^\text{[24]}\). Most model checkers do allow users to specify system models directly in this relatively low-level logical representation. Instead, they support some pre-existing modeling language (such as Simulink, Lustre, Promela, SMV, or even just C). To account for possible problems in the translation from the input modeling language to the

This work is partially supported by the National Aeronautics and Space Administration under NASA contract number NNL14AA06C.
internal logical representation, we include a second phase which produces an additional proof certificate providing some level of confidence in the correctness of the translation.

While the techniques we have developed are general enough to be applicable to arbitrary SMT-based model checkers, we have implemented them in a specific one: Kind 2 [7], an SMT-based, multi-engine, symbolic model checker that can prove or disprove safety properties of synchronous reactive systems expressed in the Lustre language [11]. As a consequence, we will describe our work in terms of Lustre and Kind 2, but with the assumption that a knowledgeable reader will be able to see how it generalizes to other SMT-based model checkers. In more detail, this work contains the following specific contributions:

(1) A technique for generating proof certificates for safety properties of transition systems. We show how to extract and simplify k-inductive invariants that are sufficient to summarize proofs generated by the different kinds of SMT-based model checking methods (in Section II) and how proofs can be reconstructed (in Section IV).

(2) An approach to increase trust in the translation from the external modeling language to an internal representation language, described in Section III. A translation certificate is generated in the form of observational equivalence between two internal representations generated by independently developed front ends. Their equivalence is recast as an invariant property; checking that yields itself a second proof certificate from which a global notion of safety can be derived and incorporated in the LFSC proof. We improve on similar previous approaches [19], [20] by adopting a weaker, property-based notion of observational equivalence, which is enough for our purposes.

(3) An implementation of these techniques in Kind 2. The first certificate summarizes the work of its different engines: bounded model checking (BMC), k-induction, IC3, as well as additional invariant generation strategies. The certification of the translation is applied to the Lustre language. The intermediate certificates are SMT-LIB 2 scripts checked by CVC4. CVC4’s own proof objects are used to construct an LFSC proof term providing an overall proof of safety.

The full certification process for Kind 2 is depicted in Figure 1. Kind 2 generates two sorts of safety certificates, in the form of SMT-LIB 2 scripts: one certifying the faithfulness of the translation from the Lustre input model to the internal encoding, and one certifying the invariance of the input properties for the internal encoding of the input system. These certificates are checked by CVC4, then turned into LFSC proof objects by collecting CVC4’s own proofs and assembling them to form an overall proof that can be efficiently verified by the LFSC proof checker. Our initial experimental evaluation indicates that, at the price of minimal instrumentation in the model checker, this approach allows one to efficiently generate and check proofs for non-trivial transition systems and invariance queries.

To illustrate our different techniques, we will rely on the toy model in Figure 2. In Lustre, reactive components are modeled as nodes. The node add_two in the figure encodes a component that initially outputs 1.0, in variable c, and at each execution step afterwards outputs the maximum between the previous value of c and the sum of the current values of input variables a and b. The model is annotated with an invariance property stating that the output c is positive whenever both inputs are.

A. Technical Preliminaries

We define a transition system as a tuple $S = (x, I, T)$ where x is a tuple of distinct (typed) variables; I is a formula of typed first-order logic with free variables from x, which characterizes the initial states of the system; and T is a formula with free variables from x and a renamed copy x' of x, which describes the system’s transition relation. If F is a formula with free variables from x, we write $F[y]$ to denote the instance of $F$ obtained by replacing its free variables by the corresponding ones in y. We write $T[y, y']$ similarly for T. We adopt the usual notions and notations of first-order logic. In particular, for an interpretation $M$ and a formula $\varphi$, we write $M \models \varphi$ to mean $M$ satisfies the formula $\varphi$. We also write $\models$ for the logical entailment in a theory (such as integer and real arithmetic) that encodes the data types used in the transition system. A state of the system $S = (x, I, T)$ is a model that gives an interpretation to the variables of x. A state $M$ of a system $S = (x, I, T)$ is said to be reachable iff there exists an $i \in \mathbb{N}$ such that, $M \models \exists x_0 \ldots x_{i-1}. I[x_0] \land T[x_0, x_1] \land \ldots \land T[x_{i-2}, x_{i-1}] \models P[x_{i-1}]$ for all $i = 1, \ldots, k$, and (ii) $T[x_0, x_1] \land \ldots \land T[x_{k-2}, x_{k-1}] \land P[x_0] \land \ldots \land P[x_{k-1}] \models P[x_k]$. A k-inductive strengthening $Q$ of $P$ is a k-inductive formula $Q[x]$ such that $Q[x] \models P[x]$. One can show that k-inductive

\begin{verbatim}
node add_two (a, b : real) returns (c : real) ;
var v : real;
let
v = a + b ;
c = 1.0 -> if (pre c) > v then (pre c) else v ;
~ ~ PROPERTY (a > 0.0 and b > 0.0) => c > 0.0 ;
tel
\end{verbatim}

Figure 2: Lustre model of running example.
state properties are invariant. It follows that every state property having a \( k \)-inductive strengthening is invariant.

II. \( k \)-inductive Safety Certificates

In this section, we focus on transition systems and present a certificate generation approach general enough to capture the information produced by different SMT-based model checking engines while proving invariance properties of a system \( S = (X, I, T) \). We show that \( k \)-inductive strengthenings of original properties are an adequate summary of the reasoning resulting from the combination of these engines. We also show how to combine and simplify them with the aim of generating the most easily verifiable objects.

A. Extracting and Verifying Certificates

Kind 2 converts internally input models and properties, expressed in Lustre, to a transition system that captures the same input/output behavior. The translation is relatively straightforward for single-node models, and is based on having state variables corresponding to the node’s input and output variables as well as any terms of the form \( \text{pre } t \). For multi-node models, the translation systems for the individual nodes are combined according to Lustre’s synchronous parallel composition semantics.

Certificate extraction. In Kind 2, an input property \( P \) can be proved invariant by one of two main model checking methods: \( k \)-induction \[22\] and IC3 \[6\], each implemented in an independent engine. The job of either engine is facilitated by a number of auxiliary invariant generation engines, which discover and pass along auxiliary invariants that might be helpful in proving the main property. Often these are local invariants, for instance specific to a sub-component of the input system. All of these engines, which run concurrently, generate safety certificates of the form \( (k, \phi) \) where \( k \) is a positive number and \( \phi \) is a \( k \)-inductive strengthening of some state property. The content of the certificate depends on the engine:

- The \( k \)-induction engine tries to prove that the input property \( P \) is invariant by proving that it is \( k \)-inductive for some \( k > 0 \). When this succeeds, \( P \) is its own \( k \)-inductive strengthening and a possible certificate is the pair \((k, P)\).
- The IC3 engine also tries to prove that an input property \( P \) is invariant. It succeeds when it is able to construct a conjunction \( \phi \) of formulas such that \( \phi \land P \) is \( 1 \)-inductive. In this case, a possible certificate is \((1, \phi \land P)\).
- The invariant generation engines are based on variations of the previous techniques. Every auxiliary invariant used in the proof of an input property \( P \) is provided with its own certificate, also of the form \((k, \phi)\).

Certificate combination. Kind 2 accepts as input multiple properties for a given model, and attempts to verify them individually. This means that it normally produces individual certificates for a collection of user-specified and internally generated properties. These safety certificates are combined together thanks to the following easily provable result.

Proposition 1. If \((k_i, \phi_i)\) is a \( k_i \)-inductive strengthening of property \( P_i[x] \) for \( i = 1, 2 \), then \((k, \phi)\) with \( k = \max(k_1, k_2) \) is a \( k \)-inductive strengthening of \( P_1[x] \land P_2[x] \).

Verifying Certificates. Checking a (combined) certificate \((k, \phi)\) for a (conjunctive) property \( P \) reduces to verifying that \( \phi \) is indeed a \( k \)-inductive strengthening of \( P \). This can be done using any tool that can prove the following entailments:

\[
I[x_0] \land T[x_i, x_{i-1}] \land \ldots \land T[x_{i-k}, x_{i-k+1}] \vdash \phi[x_{i-k+1}]
\]

for \( i \in [1, k] \) (base),

\[
T[x_i, x_{i-1}] \land \ldots \land T[x_{i-k}, x_{i-k+1}] \land \phi[x_i] \land \phi[x_{i-k}] \vdash \phi[x_{i-k+1}]
\]

for \( i \in [1, k] \) (step \( k \)).

Using an SMT solver to prove \((\text{base}_k)\), \((\text{step}_k)\), and \((\text{implication})\), effectively moves the burden of trust from the model checker to the solver. As we describe in Section IV, the latter can in turn be removed from the trusted core if it can provide an LFSC proof of the three entailments.

B. Simplifying Certificates

Good certificates need to be simple and easily checkable by an independent tool or method. In particular, there is an expectation that checking a certificate should not take more time than proving the original property. A common approach in the certificate production literature is to simplify and/or reduce the certificate \text{a posteriori} \([2], [8], [25]\). This extra effort at construction time can pay large dividends at checking time. In our case, a safety certificate \((k, \phi)\) can be simplified by reducing the value of \( k \) or the size/complexity of \( \phi \), or both. Currently, Kind 2 tries to reduce \( k \) before simplifying \( \phi \). Empirical evaluation, discussed in Section V, suggests that this sort of post-processing is always worth the overhead.

Reducing \( k \). Referring back to the entailments \((\text{base}_k)\) and \((\text{step}_k)\) from the previous section, because of the \( k \) checks in \((\text{base}_k)\), checking a certificate \((k, \phi)\) requires a number of sub-checks proportional to \( k \). Each of sub-checks in turn take time proportional to \( k \), making the whole process quadratic in \( k \). Due to the concurrent nature of Kind 2, proofs obtained by its \( k \)-induction engines are not guaranteed to have a minimal \( k \). Consequently, lowering \( k \) can often be the most effective way of simplifying a certificate. To do that, after it constructs an initial combined certificate \((k, \phi)\), Kind 2 will replay the inductive step \((\text{step}_k)\) for \( \phi \) for values \( k' \) smaller than \( k \), following one of three different strategies, chosen heuristically:

- \text{forward enumeration}: progressively try all values of \( k' \) from 1 to \( k \) and stop at the first where \( k' \)-inductiveness holds;
- \text{backward enumeration}: try values of \( k' \) from \( k \) down to 1, stopping as soon as \( k' \)-inductiveness is lost;
- \text{binary search}: partition \([1, k]\) into subintervals \([1, k']\) and \([k' + 1, k]\) of similar size and recursively consider the first or the second interval depending on whether \( \phi \) is \( k' \)-inductive or not.

Simplifying \( \phi \). Because of how combined certificates \((k, \phi)\) are generated, the invariant \( \phi \), which is a conjunction \( \psi_1 \land \ldots \land \psi_n \)
**Algorithm 1. Two-phase simplification of invariants**

**Input:** $R = \{\phi_1, \ldots, \phi_n\}$; invariant set to be reduced, $P$: input property set, $T$: Transition relation

**Function trim(R, P)**

- If $R(0..k-1) \land P(0..k-1) \land T(0..k) \models P(k)$
  - // $P$ is $k$-inductive wrt $R$
  - $U = \text{get-unsat-core}()$
  - $R' = \{\phi \in R \mid \phi \text{ occurs in } U\}$
  - If $R'(0..k-1) \land P(0..k-1) \land T(0..k) \models P(k)$
    - // $R' \land P$ is $k$-inductive
    - return $R' \cup P$
  - Else // $R'$ is not strong enough
    - trim($R \setminus R'$, $R' \cup P$)
- Else error "Not $k$-inductive";

**Function cherry-pick(R, P)**

- If $P(0..k-1) \land T(0..k) \models P(k)$
  - // $P$ is $k$-inductive
  - return $P$
- Else // Find cex to induction
  - $M = \text{get-cex}()$
  - // And a blocking invariant
  - $\psi = \text{choose}(\{\phi \in R \mid M \not\models \phi\};$
  - cherry-pick($R|\psi\}, P \cup \{\psi\}$)

In our experiments, it was always beneficial to apply the coarse reduction performed by trim before calling cherry-pick.

We observe that the effect of trim is similar to one of the reduction steps proposed by Irivii et al. [14] for invariants produced by SAT-based IC3-like model checkers. While potentially increasing precision, many of their other steps require a number of satisfiability checks linear in the size of $\phi$, which is already prohibitive for the SMT case.

It could be useful to try to reduce $k$ and simplify $\phi$ at the same time in the hope of getting closer to a minimal $k$ than we do currently with our algorithm. This, however, would be more expensive, so further empirical evidences would be needed to assess the practical effectiveness of more sophisticated approaches in practice.

### III. Front End Certification

The certificates discussed in the previous section are produced for Kind 2’s internal FOL representation of the input system and properties. Although the translation to this internal representation from the Lustre input is fairly direct, Kind 2’s front end also applies a number of optimizations and simplifications to the input, such as slicing, constant propagation, and so on. This raises the question of whether the front end can be trusted to be correct. We rule out the option of formally proving its correctness for the reason we gave in Section II. In alternative, we have the translation phase generate certificates of its own.

**Comparing independent translations.** Our goal is to keep the whole certification process lightweight and entirely automatic. As a consequence, instead of proving a semantic preservation between the input Lustre model and its internal representation as a transition system, we prove the observational equivalence of two internal representations obtained independently from the same input. This technique for certifying translations has already been employed in the SAT-based toolchain of Prover Technologies [20] and in the Systerel Smart Solver [19]. In our case, instead of developing another front end for Kind 2 we can rely on a pre-existing third-party tool: JKind, a Lustre model checker inspired by Kind but developed independently at Rockwell Collins [21]. JKind too converts input models to an FOL representation. It is a good candidate because it is sufficiently different from Kind 2: it has a completely different code base (it is written in Java whereas Kind 2 is written in OCaml) and was developed independently by a different team. While our approach does not actually guarantee the correctness of the Kind 2 translation, it provides some formal evidence of its trustworthiness.

Our certificate encodes the claim that the transition relations constructed by the two independent front ends are behaviorally equivalent over a set of relevant state variables. In essence, the certificate consists of a transition system that observes the internal states of the two systems generated by each front end. This observer system feeds its two subsystems the same inputs and verifies that their externally visible behavior is the same.

For $i = 1, 2$, let $S_i = (x_i, I_i[x_i], T_i[x_i, x_i])$ be the internal transition system, and $P_i$ the property, respectively generated.
We construct an observer system \( S_{\text{obs}} \) and a safety property \( P_{\text{obs}} = (S_1, P_1) \sim (S_2, P_2) \) expressing a suitable notion of observational equivalence \((\sim)\) between the two systems. Then we check the correctness of this observer in the same way as we would check the correctness of \( S_2 \) with respect to the original safety property. This process is illustrated as part of Figure 1, where \( \text{Obs Eq} \) is the observer described below and the module Kind 2 Core is the core part of Kind 2, which works directly with the internal FOL representation of a transition system.

**Observational equivalence.** A standard definition of observational equivalence would require the two systems \( S_1 \) and \( S_2 \) to produce the same outputs when given the same inputs at each step. This is, however, is unnecessarily stringent for our purposes and, depending on how different the two translations are, it might not even be the case. A better notion of equivalence is property-based: we consider \( S_1 \) and \( S_2 \) equivalent if, for the same input, agree at each step on the truth value they assign to their respective version of the original input property in the Lustre model. For \( j = 1, 2 \), let \( i_j \) be the subsuite of \( \mathbf{x}_j \) that corresponds to the input variables of the Lustre model. Then \( P_{\text{obs}} \) and \( S_{\text{obs}} = (\mathbf{x}_{\text{obs}}, \mathbf{I}_{\text{obs}}, T_{\text{obs}}) \) are defined as follows:

\[
P_{\text{obs}} = (P_1[\mathbf{x}_1] \equiv P_2[\mathbf{x}_2]) \quad I_{\text{obs}} = I_1 \equiv I_2 \quad T_{\text{obs}} = T_1 \equiv T_2
\]

where for two tuples \( \mathbf{a} = (a_1, \ldots, a_n) \) and \( \mathbf{b} = (b_1, \ldots, b_n) \), the expression \( \mathbf{a} \equiv \mathbf{b} \) denotes the formula \( \bigwedge_{i=1}^n a_i = b_i \).

The equivalence observer \( S_{\text{obs}} \) is defined by

\[
x_{\text{obs}} = \mathbf{x}_1, \mathbf{x}_2 \quad I_{\text{obs}} = (a_1 = a_2 \land b_1 = b_2 \land I_1 \land I_2) \quad T_{\text{obs}} = (a_1' = a_2' \land b_1' = b_2' \land T_1 \land T_2)
\]

Suggested auxiliary invariants in this case will be the equalities

\[
\begin{align*}
I_1 &= a_1 > 0 \land b_1 > 0 \Rightarrow c_1 > 0 \\
I_2 &= a_2 > 0 \land b_2 > 0 \Rightarrow c_2 > 0
\end{align*}
\]

**Example 1.** Consider again the Lustre model and property of Figure 2. The systems \( S_1 \) and \( S_2 \) respectively generated by JKind 2.4 and Kind 2 from that model are the following, in abstract syntax and modulo variable renaming:

<table>
<thead>
<tr>
<th>( S_1 )</th>
<th>( S_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 = [a_1, b_1, e_1, v_1] )</td>
<td>( x_2 = {a, c, v_2} )</td>
</tr>
<tr>
<td>( I_1 = R[x_1, x_1'] )</td>
<td>( I_2 = (i \land v_2 = a_2 + b_2 \land c_2 = 1) )</td>
</tr>
<tr>
<td>( T_1 = R[x_1, x_1'] )</td>
<td>( T_2 = (\neg \ite(c_1 &gt; v_1', c_1, v_1')) )</td>
</tr>
<tr>
<td>( R[R[x_1, x_1'], (v_1' = a_1' + b_1' \land c_1' = \ite(c_1 = \text{true}, c_1, v_1'))] )</td>
<td>( P_1 = a_1 &gt; 0 \land b_1 &gt; 0 \Rightarrow c_1 &gt; 0 )</td>
</tr>
<tr>
<td>( x_{\text{obs}} = x_1, x_2 )</td>
<td>( P_{\text{obs}} = (P_1 \equiv P_2) )</td>
</tr>
<tr>
<td>( T_{\text{obs}} = (a_1' = a_2', b_1' = b_2', c_1' = \ite(c_1 = \text{true}, c_1, v_1')) )</td>
<td>( T_{\text{obs}} = (a_1' = a_2', b_1' = b_2', c_1' = \ite(c_1 = \text{true}, c_1, v_1')) )</td>
</tr>
</tbody>
</table>

\[\square\]

**IV. FROM CERTIFICATES TO LFSC PROOFS**

The last step of our approach, once the various safety certificates have been produced and checked, is to gather the proofs of the various entailment checks performed by the SMT solver and assemble them into a self-contained overall proof of safety for the original system.

**LFSC proofs.** The entailment proofs are obtained specifically from CVC4 as proof terms in LFSC, an extension of the Edinburgh Logical Framework (LF) [12] with side conditions [25]. In LFSC, which is in essence a dependently typed \( \lambda \)-calculus, proof systems are encoded as type systems. Proof checking then reduces to type checking, performed by a highly optimized checker developed by Stump et al. [24]. This particular LFSC checker takes as input a type system \( S \) and a term \( t \) in that system, and checks whether \( t \) is well typed in \( S \). The efficiency of this framework for proof checking lies in the use of side-conditions, defined as small functional programs, which can be pre-compiled by the checker. Using proof rules with side conditions generally leads to both smaller proof sizes and faster proof checking times.

A proof system is formally defined in LFSC through signatures, which contain a definition of the system’s language together with axioms and proof rules. The proof system used by CVC4 is defined over a number of signatures, which are included in its source code distribution. Those relevant to this work include signatures for propositional logic and resolution (sat.plf); first-order terms and formulas, with rules for CNF conversion and abstraction to propositional logic (smt.plf); equality over uninterpreted functions (th_base.plf); and real and integer linear arithmetic (th_int.plf and th_real.plf).

**Extending CVC4’s proof system.** We have extended CVC4’s proof system with an additional signature (kind.plf) for \( k \)-inductive reasoning, invariance and safety\(^2\). This signature also specifies the encoding for state variables, initial states, transition

\(^2\)We produce \( S_k \) by having JKind 2.1 write a dump file from which we can extract its internal representation.

\(^3\)The LFSC checker with all the necessary signatures are distributed with Kind 2 and publicly available.
relations, and property predicates. State variables are encoded as functions from natural numbers to values. This way, the unrolling of the transition relation done in \([\text{base}_k]\) and \([\text{step}_k]\) does not need the creation of several copies of the state variable tuple \(x\). For example, for the state vector \(x = (y, z)\) with \(y\) of type real and \(z\) of type integer, the LFSC encoding will make \(y\) and \(z\) functions from naturals to reals and integers, respectively. So we will use the tuples \((y(0), z(0))\), \((y(1), z(1))\), \ldots instead of \((y_0, z_0), (y_1, z_1), \ldots\) where \(y_0, y_1, \ldots, z_0, z_1, \ldots\) are (distinct) variables. Correspondingly, our LFSC encoding of a transition relation formula \(T[x, x']\) is parametrized by two natural variables, the index of the pre-state and that of the post-state, instead of two tuples of state variables. Similarly, \(I\), \(P\) and \(\phi\) are parametrized by a single natural variable.

The signature defines several derivability judgments, including one for proofs of invariance, which has the following type:

\[
\text{invariant : } \Pi \ I : \mathbb{N} \rightarrow \text{formula. } \Pi \ T : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{formula. } \Pi \ P : \mathbb{N} \rightarrow \text{formula. Type}
\]

It also contains rules to build proofs of invariance by \(k\)-induction, as illustrated in Figure 4 in abstract syntax. There, proof rule \(\text{InvImp}l\) states that weakenings of invariants are invariants. Rule \(K\text{-IND}\) encodes the \(k\)-induction principle as presented in Section 5. It has two side-conditions that compute formulas for the subgoals of \(k\)-induction. As an example, we provide the definition of \(\text{step}\), which uses an auxiliary function to compute unrollings of the transition relation.

This signature also specifies how to encapsulate proofs for the front-end certificates by providing a additional judgment, \(\text{safe}(I, T, P, I', T', P')\), which can be derived only when \(\text{invariant}(I, T, P)\) is derivable and the observational equivalence between \((I, T, P)\) and \((I', T', P')\) is provable (judgment \(\text{woe}\)).

Figure 3: Sketch of derivation tree for LFSC proofs of safety produced by Kind 2

![Figure 3](image-url)

Figure 4: A sample of LFSC rules for \(k\)-induction proofs
without any simplification (S+cvc4) we can check a lot less
certificates and take much more time than with simplification.
We can also see that, even if the full simplification process
is more expensive (S+m vs. S+mE), it yields a larger number
of checked certificates within the time limit (S+m+cvc4 vs.
S+mE+cvc4). The superiority of full simplification is confirmed
by an analysis of the full results. It reduces the size of
the invariants on average by 74% (removing on average 19
invariants per certificate) for 42% of the benchmarks. For
one benchmark, it removes 236 invariants out of 243. The
value of \( k \) is reduced in 11% of the benchmarks, by 10 on
average, the maximum being a reduction from 36 down to
2. The bump at 428 is due to the simplification overhead for
a single benchmark, which is larger than the solving time.
However, even with this outlier, the cumulative benefit of full
simplification on certificates is clear.

**Checking full certificates.** The plot in Figure 6 refers to the
complete proof certification chain. The measurements show the
time necessary up to produce the proofs (S+m+cvc4) (which
involve an intermediate checking phase, cvc4, with CVC4) and
to check them with the LFSC proof checker (p). The second
and third curves are for the invariance property while the last
two also include the overhead for the front end proof (I+F).
The latter includes the time to: prove the input property; fully
minimize its safety certificate and generate the corresponding
proof; construct the equivalence observer, including the time
to call JKind and extract its transition system; model check the
observer with Kind 2; minimize and produce the proof for the
front end certificate; and finally check the combined resulting
proof with LFSC.

We are able to generate and check the proof of invariance for
around 80% of benchmarks that Kind 2 succeeds in verifying;
we produce and check a complete proof including the front end
for 60% of them. Most of the cases where we fail to generate
the proof are due to CVC4’s current limitations in its proof
producing capabilities. The biggest bottlenecks are the model
checking of the equivalence observer and the simplification of
certificates. Despite that, the time cost of the full certification
chain is overall within one order of magnitude of the cost of
just proving the input property. We find the overall level of
performance, which we think we could improve further, already
rather good, especially considering that a lot of the benchmarks
we used are non-trivial.

VI. Related Work

**Formally verified model checkers.** A natural approach to the
certification of verification tools consists in proving the program
(here the model checker) correct once and for all. This is
possible to a large extent for programs written in programming
languages with (largely automated) verification toolsets such as
ESC Java 2, Frama-C, VCC, F* etc. Proving full functional
correctness of a model checker, however, is currently a very
challenging job because these tools are often rather complex and
tend to evolve quickly with the ongoing advances in the field.
When feasible, one great advantage of this approach of course
is that the performances of the model checker is minimally
impacted by the verification process. One example of this kind
of certification effort is the modern SAT solver *versat*, which
was developed and verified using the programming language
Guru [17]. We are, however, not aware of similar results for
model checkers.

Another possibility is to prove the underlying algorithms of a model checker correct in a descriptive language of
interactive proof assistants such as Coq or Isabelle, and obtain
an executable program from these tools through a refinement
process or code extraction mechanism. Although the first
formal verification of a model checker in Coq for the modal
\( \mu \)-calculus [23] goes back to 1998, only recently have *certified
verification tools* started to emerge. Amjad [1] shows how to
embed BDD-based symbolic model checking algorithms in the
HOL theorem prover so that results are returned as theorems.
This approach relies on the correctness of the backend BDD
implementation. Esparza et al. [10] have fully verified an
automata-based model checker for finite state systems with
the Isabelle theorem prover. Using successive refinements, they
built a correct by construction model checker for finite level
specifications down to functional (SML) code.

A recent approach for the certification of SAT and SMT
solvers [2] consists in having the solver produce a detailed
certificate in which each rule is read and verified by a
combination of several small certified checkers, written and
proved correct in Coq. This approach also allows one to import
inside Coq proof terms from these solvers [3].

**Certifying model checkers.** A number of techniques have
been proposed to produce certifying model checkers. Earlier
solutions (e.g., [15], [16], [18]) were limited to finite-state
systems. The first certifying model checker for infinite-state

---

**Figure 5:** Overhead and improvements of minimization.

**Figure 6:** Evaluation of proof certification chain.
systems was perhaps the C model checker BLAST [13], which produced certificates for a control flow automaton internally generated from an input C program. BLAST provided proof certificates in the Edinburgh Logical Framework (LF) [12], which limits the scalability of certificate checking when proofs involve reasoning modulo the theory of C’s data types.

A more recent certifying model checker is SLAB [9], which produces certificates in the form of inductive verification diagrams to be checked by SMT solvers. We go one step further by relying on SMT solvers that are in turn proof producing. Also, we address the issue of certifying the translation from the input model to the internal representation.

For model checking of parameterized systems, the model checker Cubicle generates certificates as Why3 files that can be independently checked by several SMT solvers and automated theorem provers [8], where trust is claimed through the redundant use of multiple solvers.

VII. Conclusion and Future Work

We have presented a dual technique for generating and checking proof certificates for SMT-based model checkers, and applied it to the model checker Kind 2. Given a Lustre model and one or more invariance properties for it, Kind 2 generates LFSC proofs for the properties it can verify. These proofs have two parts. The first attests that the model and the properties are encoded correctly in Kind 2’s internal representation format. It does that by proving the observational equivalence, with respect to the properties, between the internal system and another one produced from the same Lustre input by an independent, third-party tool. The second part attests that the encoded properties are invariants of the internal transition system encoding the Lustre model. Initial certificates, which we call safety certificates, are generated as (possibly combined) $k$-inductive invariants, and simplified before being verified by the CVC4 SMT solver. The eventual proof certificates, in LFSC format, are assembled from the proofs generated by CVC4 after verifying these safety certificates.

The trusted core of our approach consists in:
1) The LFSC checker (5300 lines of C++ code).
2) The LFSC signatures comprising the overall proof system in LFSC (CVC4’s sat.plf, smt.plf, th_int.plf, th_real.plf and our own kind.plf, for $k$-induction and safety), for a total of 444 lines of LFSC code.
3) The assumption that Kind 2 and JKind do not have identical defects that could escape the observational equivalence check.

A current but temporary limitation of our certificate generation process is that LFSC proofs may contain an unsound proof rule, trust_f, which derives any formula. This rule is used by the current version of CVC4 to fill in present gaps in its proof generation code. However, it will be progressively phased out as the instrumentation of CVC4 to produce full proofs is completed.

Kind 2 has the ability to do compositional and modular analyses of Lustre models extended with assume-guarantee-style contracts. A possible line of future research is to extend the work described here to apply to such analyses by incorporating their underlying abstraction mechanisms.

Kind 2’s proof certificate generation is being leveraged in an ongoing project funded by NASA and the FAA as an innovative way to reduce the cost of tool qualification with respect to DO-178C requirements.

Acknowledgments. We would like to thank Lucas Wagner and Konrad Slind for their feedback on this work and Andrew Gacek for his assistance with JKind.

References