Proof Certificates for SMT-based Model Checkers*
(Presentation only)

Alain Mebsout and Cesare Tinelli

The University of Iowa, Iowa City, IA, USA

Abstract

We present a two-fold technique for generating and verifying 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 verified by 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 certificates for non-trivial transition systems and invariance queries.

1 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 [4, 14] they are increasingly used in model-based software development to analyze, for instance, models of embedded systems in the aerospace or automotive industry. Remarkable industrial verification efforts have been carried out successfully at NASA [17], Rockwell Collins [22], SRI [32], Airbus [8], *etc.* One clear strength of model checkers, as opposed to proof assistants, say, is their ability to return precise *error traces* witnessing the violation of a given safety property. Such traces not only are invaluable for designers to correct bugs, they also constitute a checkable certificate. In most cases, it is possible to use a counter-example for a safety property to direct the execution of the system under analysis to a state that falsifies that property. 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 the system. This is unsatisfactory in general since these 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 [13]. An alternative is to instrument a model checker so that it is *certifying*, *i.e.* it accompanies its safety claims with a *certificate*, an artifact embodying a proof of the claim [23]. The certificate can then be checked by a *proof checker*, with the effect of shifting the focus of trust to a much simpler tool. 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 also a number of disadvantages. To start, the effort is normally enourmous since there are no general frameworks for verifying modern model checkers, heavily optimized programs with a large number of components. Moreover, any modifications to the 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 advantage of the second

*This work is partially supported by the National Aeronautics and Space Administration under NASA grant number NNL14AA06C.

approach is that it requires a much lower human effort. Also, by reducing the trusted core to the proof checker, certifying model checking facilitates the integration of formal method tools into safety critical processes such as those endorsed by the DO-178C [30] guidelines for avionics software. In the spirit of the *de Bruijn criterion* [5], which is traditionally applied to theorem provers, it redirects tool qualification requirements [31] from a complex tool, the model checker, to a much simpler one, the proof checker. A disadvantage of course is that every safety claim made by the model checker incurs the cost of generating and then checking the corresponding proof certificate. This is feasible in general only if the proof certificates are small and/or simple enough to be checkable by a target proof checker in a reasonable amount of time (say, with at most a one order of magnitude slowdown).

We present an approach for generating and verifying certificates for SMT-based model checkers, focusing on proofs of invariant properties. These tools use a variety of model checking techniques and some of them even employ a portfolio approach by running several engines in parallel. Models are typically represented internally as transition systems encoded in some fragment of first-order logic. Reasoning about property 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, state-of-the-art SMT solvers. We generate a certificate, in the form of a k-inductive invariant, that shows that the input property is an invariant of the internal transition system. For model checkers that allow users to specify systems directly in this relatively low-level logical representation, this certificate is sufficient to guarantee that the produced result is correct. However, most model checkers 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 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, an SMT-based, multi-engine, symbolic model checker[1] that can prove or disprove safety properties of synchronous reactive systems expressed in the Lustre language [15]. 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. This work makes the following specific contributions:

1. A technique for generating proof certificates for safety properties of transition systems. This technique is described in Section 2 where 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.

2. An approach to ensure trustworthiness of the translation from the external modeling language to an internal representation language, described in Section 3. A translation certificate is generated in the form of observational equivalence between two internal representations generated by independently developed front ends. The equivalence proof yields itself a second proof certificate. We improve on similar previous approaches [26, 27] 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 certificates produced by Kind 2 are designed to be checkable by an SMT solver. Since SMT solvers themselves are complex artifacts, we also worked on a reduction of the validity of Kind 2 certificates to proof objects obtained by a proof-producing SMT solver. This reduction capitalizes in

[1] Available at `http://kind.cs.uiowa.edu`.

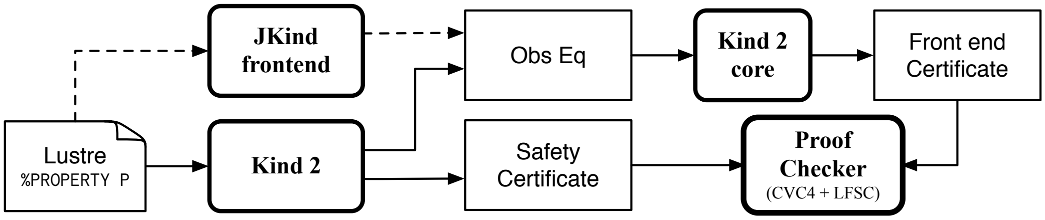

Figure 1: Process for proof certificates generation and verification in Kind 2.

```
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
```

Figure 2: Lustre model of running example.

particular on the recent proof production capabilities of the SMT solver CVC4 [6] and the availability of an efficient proof checker for its proofs, which are generated in LFSC format [35].

Our 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 another 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.

In this paper, we will not discuss the additional level of certification provided by LFSC proofs, and focus instead on certificate generation at the model checker level. As we show in Section 4, 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 certificates for non-trivial transition systems and invariance queries.

To illustrate our different techniques, we will rely on the toy model in Figure 2. The model encodes in Lustre a synchronous reactive component, `add_two`, that at each round of execution other than the first, outputs the maximum between the previous value of its output variable `c` and the sum of the current values of input variables `a` and `b`. The value of `c` is initially 1.0. The model is annotated with a safety property stating that, at each round, the output `c` is positive whenever both inputs are.

1.0.1 Technical Preliminaries.

A *transition system* is a tuple $\mathcal{S} = (\mathbf{x}, I, T)$ where $\mathbf{x}$ is a tuple of distinct (typed) variables, I is a formula of typed first-order logic with free variables from $\mathbf{x}$, which characterizes the initial states of the system, and T is a formula with free variables from $\mathbf{x}$ and a renamed copy $\mathbf{x}'$ of $\mathbf{x}$ which characterizes the system's transition relation. If F is a formula with free variables from $\mathbf{x}$ we write $F[\mathbf{y}]$ to denote the instance of F obtained by replacing its free variables by the corresponding ones in $\mathbf{y}$. We write $T[\mathbf{y}, \mathbf{y}']$ in a similar way for T. We adopt the usual notions and notation of first-order logic. In particular, for an intepretation $\mathcal{M}$ and a formula φ, we write $\mathcal{M} \models \varphi$ to mean $\mathcal{M}$ satisfies the formula φ. 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 $\mathcal{S} = (\mathbf{x}, I, T)$ is a model that gives an interpretation to the variables of $\mathbf{x}$. A state $\mathcal{M}$ of a system $\mathcal{S} = (\mathbf{x}, I, T)$ is said to be *reachable* iff there exists an $i \in \mathbb{N}$ such that, $\mathcal{M} \models \exists \mathbf{x}_0 \ldots \mathbf{x}_{i-1}.\ I[\mathbf{x}_0] \wedge T[\mathbf{x}_0, \mathbf{x}_1] \wedge \ldots \wedge T[\mathbf{x}_{i-1}, \mathbf{x}_i]$. *State properties* for a system

$\mathcal{S}$ are described by first-order formulas whose free variables are from $\mathbf{x}$. Let P be a state property for $\mathcal{S} = (\mathbf{x}, I, T)$. *$P$ holds* in, or is an *invariant* of $\mathcal{S}$, if every reachable state $\mathcal{M}$ of $\mathcal{S}$ is a model of P. Property P is k-inductive for some $k > 0$ if $I[\mathbf{x}_0] \wedge T[\mathbf{x}_0, \mathbf{x}_1] \wedge \ldots \wedge T[\mathbf{x}_{i-2}, \mathbf{x}_{i-1}] \models P[\mathbf{x}_{i-1}]$ for all $i = 1, \ldots, k$, and $T[\mathbf{x}_0, \mathbf{x}_1] \wedge \ldots \wedge T[\mathbf{x}_{k-1}, \mathbf{x}_k] \wedge P[\mathbf{x}_0] \wedge \ldots \wedge P[\mathbf{x}_{k-1}] \models P[\mathbf{x}_k]$. A *$k$-inductive strengthening Q* of P is a k-inductive formula $Q[\mathbf{x}]$ such that $Q[\mathbf{x}] \models P[\mathbf{x}]$. One can show that k-inductive state properties are invariant. It follows that every state property having a k-inductive strengthening is invariant.

2 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 properties of a system $\mathcal{S} = (\mathbf{x}, I, T)$. 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.

2.0.1 Internal logical representation.

Recall that we consider input models and properties expressed in Lustre. Kind 2 converts them internally 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 `(pre t)`.[2] For multi-node models, the transition systems for the individual nodes are combined according to Lustre's synchronous parallel composition semantics.

2.0.2 Extraction of Certificates.

In Kind 2, an input property P can be proved invariant by one of two main model checking methods: k-induction [33] and IC3 [9], 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 invariance of the input property. All of these engines generate *safety certificates* of the form (k, ϕ) where k is a positive number and ϕ is a k-inductive strengthening of some state property. The specific content of the proof certificate depends on the specific engine, as detailed below.

k-induction. 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 so the most immediate form of certificate is the pair (k, P).

IC3. The IC3 engine also tries to prove, concurrently, that an input property P is invariant. It succeeds when it is able to construct a conjunction ϕ of formulas such that $\phi \wedge P$ is 1-inductive. In this case then, the produced certificate can be $(1, \phi \wedge P)$.

Invariant Generation. Kind 2 has a number of engines that produce auxiliary invariants used to strengthen the transition relation, to help the main engines prove the input properties. Often these are local invariants, for instance specific to a sub-component of the input system. Every auxiliary invariant used in the proof of an input property P is provided with its own certificate, also of the form (k, ϕ).

[2] For each execution round but the first, `(pre t)` denotes the value of `t` in the previous round.

2.0.3 Combining Certificates.

In addition to producing auxiliary invariants, 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 several (user-specified and internally generated) properties. These safety certificates can be combined together thanks to the following easily provable result.

Proposition 1. *If* (k_i, ϕ_i) *is a* k_i*-inductive strengthening of property* $P_i[\mathbf{x}]$ *for* $i = 1, 2$*, then* $(k, \phi_1 \wedge \phi_2)$ *with* $k = max(k_1, k_2)$ *is a* k*-inductive strengthening of* $P_1[\mathbf{x}] \wedge P_2[\mathbf{x}]$.

2.0.4 Verifying Certificates.

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

$$I[\mathbf{x}_0] \wedge T[\mathbf{x}_0, \mathbf{x}_1] \wedge \ldots \wedge T[\mathbf{x}_{i-2}, \mathbf{x}_{i-1}] \models \phi[\mathbf{x}_{i-1}] \quad \text{for all } i = 1, \ldots, k \qquad (base_k)$$

$$T[\mathbf{x}_0, \mathbf{x}_1] \wedge \ldots \wedge T[\mathbf{x}_{k-1}, \mathbf{x}_k] \ \wedge \ \phi[\mathbf{x}_0] \wedge \ldots \wedge \phi[\mathbf{x}_{k-1}] \models \phi[\mathbf{x}_k] \qquad (step_k)$$

$$\phi[\mathbf{x}] \models P[\mathbf{x}] \qquad (implication)$$

A success in proving $(base_k)$, $(step_k)$, and (*implication*) reduces the trusted core from the model checker to the prover, specifically, an SMT solver in our case. Since SMT solvers are complex tools in their own right, a further level of confidence can be achieved by using a *proof-producing* SMT solver, like CVC4. CVC4 generates proofs for the entailments above in LFSC format. In a second phase, not described in this paper, these proofs can be automatically assembled and fed to a *trusted* LFSC proof checker.

2.0.5 External syntax for certificates.

Since certificate checking is done by an external tool, we produce a textual representation of the transitions system, the property, and its certificate needed for the entailment tests $(base_k)$, $(step_k)$, and (*implication*). For simplicity's sake this representation encodes state variables as functions from the integer numbers to values. This way, the unrolling of the transition relation done in $(base_k)$ and $(step_k)$ does not need the creation of several copies of the state variable tuple $\mathbf{x}$. For example, if internally $\mathbf{x} = (y, z)$ with y of type real and z of type integer, externally y and z will be 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. This way, the formula T is parametrized by two integer variables, the index of the pre-state and of the post-state, instead of two tuples of state variables. Similarly, I, P and ϕ are parametrized by a single integer variable.

The concrete external syntax for these formulas is that of the SMT-LIB 2 language, a standard command language for interacting with SMT solvers through a textual interface [7]. Our implementation in Kind 2 outputs them within an SMT-LIB 2 script that defines I, T, P and ϕ and then checks the validity of the entailments $(base_k)$, $(step_k)$, and (*implication*). An example of such a script for the system and property in Figure 2 is sketched in Figure 3. The script contains a header with some meta-level information to identify the predicates that represent the input system (`I`, `T` and `P`) and the certificate (`Inv`). Then come actual commands for the SMT solver that define constants, state variables and node symbols as function symbols in first-order logic. Finally, the script contains checks to verify the certificate. The script can be executed by any SMT-LIB 2-compliant solver that supports the logic the various formulas belong to. The logic used in Kind 2's case is a combination of quantifier-free linear real and integer arithmetic with uninterpreted function symbols.

```
(set-info :origin "Certificate generated by kind2
      v0.8.0-133-g7004523")
(set-info :input "add_two.lus")
(set-info :status unsat)
(set-info :init  I)
(set-info :trans T)
(set-info :prop  P)
(set-info :certif "(1 , Inv)")

;; Constants:
(declare-fun main.__nondet_0 () Bool)
...
;; State variables:
(declare-fun main.usr.b (Int) Real)
...
;; Function symbols:
(define-fun __node_init_add_two ...)
...

;; Initial states:
(define-fun I ((i Int)) Bool ...)
;; Transition_relation :
(define-fun T ((i Int) (j Int)) Bool ...)
;; Original property:
(define-fun P ((i Int)) Bool ...)
;; 1-Inductive invariant
(define-fun Inv ((i Int)) Bool ...)
```

```
;; CERTIFICATE CHECKER

(echo "Checking base case")

(push 1)
(assert (and (I 0) (not (Inv 0))))
(check-sat)
(pop 1)

(echo "Checking 1-inductive case")

(push 1)
(assert (and (Inv 0) (T 0 1)))
(assert (not (Inv 1)))
(check-sat)
(pop 1)

(echo "Checking property implication")


(push 1)
(declare-const n Int)
(assert (not (=> (Inv n) (P n))))
(check-sat)
(pop 1)

(exit)
```

Figure 3: A sketch of an SMT-LIB 2 intermediate certificate for our running example.

2.0.6 Minimization of Certificates.

Good certificates need to be simple and easily verifiable by an independent tool or method. In particular, there is an expectation that verifying 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 *a posteriori* [2, 11, 35]. This extra effort at construction time can pay large dividends at checking time. Our certificates, of the form (k, ϕ), can be simplified by reducing either the value of k or the size/complexity of ϕ (or both). In its current implementation, Kind 2 tries to minimize k before simplifying ϕ. Empirical evaluation, discussed in Section 4, suggests that this sort of *a posteriori* minimization is always worth the overhead.

Minimizing k. As one can see from Section 2.0.4, because of the k checks in ($base_k$), verifying a certificate (k, ϕ) requires a number of sub-checks proportional to k. The individual sub-checks for ($base_k$) and ($step_k$) are themselves of size proportional to k. This makes the whole process quadratic in k, so it is important for k to be as small as possible. Due to the concurrent nature of Kind 2, however, proofs obtained by its k-induction engine 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 certificate (k, ϕ), Kind 2 will replay the inductive step ($step_k$) for ϕ for values k' smaller than k, following one of three different strategies, chosen heuristically:

- *forward*: progressively try all values of k' from at 1 to k and stop at the first where k'-inductiveness holds;

- *backward*: try values of k' from k down to 1, stopping as soon as k'-inductiveness is lost;

- *binary search:* partition interval $[1; k]$ into subintervals $[1; k']$ and $[k' + 1; k]$ of similar size and recursively consider the first or the second interval depending on whether k'-inductiveness holds or not.

Algorithm 1. Two-phases simplification of k-inductive invariant.

Input: $R = \psi_1, \ldots, \psi_n$: invariant to simplify, P: input property, T: Transition relation

```
Function trim($R$, $P$)
    if $R(0..k-1) \wedge P(0..k-1) \wedge T(0..k) \models P(k)$ then // $P$ is $k$-inductive w.r.t. $R$
        $\mathcal{U}$ = get-unsat-core();
        $R_1 = \mathcal{U} \cap R$;
        if $R_1(0..k-1) \wedge P(0..k-1) \wedge T(0..k) \models R_1(k) \wedge P(k)$ then
            // $R_1$ $k$-inductive with P
            return $R_1$
        else // $R_1$ is not strong enough
            trim($R$, $R_1 \wedge P$)
    else error "Not $k$-inductive";
```

```
Function cherry-pick($R$, $P$)
    if $P(0..k-1) \wedge T(0..k) \models P(k)$ then
        // $P$ is $k$-inductive
        return $P$
    else
        // Find counter-example to induction
        $\mathcal{M}$ = get-model();
        // . . . and an invariant that blocks it
        $\varphi$ = choose($\{\varphi \in R \mid \mathcal{M} \not\models \varphi\}$);
        cherry-pick($R \setminus \{\varphi\}$, $P \cup \{\varphi\}$);

cherry-pick( trim($\{\psi_1, \ldots, \psi_n\}$, $P$), $P$);
```

Simplifying ϕ. Because of how combined certificates (k, ϕ) are generated, the invariant ϕ, which is a conjunction $\varphi_1 \wedge \ldots \wedge \varphi_n$ of formulas, can have some redundancy in it. For instance, it can contain auxiliary invariants that are actually not needed to prove the original property. We discuss now how ϕ is tightened to remove redundant or unnecessary information from it. The core of this process consists in two fixpoint computations described in Algorithm 1. There, we use the notation $f(0..k)$ to abbreviate a conjunction of the form $f(0) \wedge \ldots \wedge f(k)$, and $T(0..k)$ to abbreviate $T(0, 1) \wedge \ldots \wedge T(k-1, k)$. Also, we treat finite sets of formulas as the conjunction of their elements. For entailment checks, we assume the availability of a function `get-unsat-core` that returns a minimal unsatisfiable core of the premises and the negated conclusion of the entailment when the entailment holds, and a function `get-model` that returns a counter-model when the entailment does not hold. Both of these functionalities are provided by most SMT solvers.

The simplification is performed by two independent techniques, applied in sequence. The first one, implemented by function `trim` of Algorithm 1, aims at identifying and discarding chunks of invariants that are not useful for the certificate. It relies on unsat cores in order to iteratively refine the set R of auxiliary invariants as long as $P \wedge R$ remains k-inductive. The second technique, implemented by function `cherry-pick`, recursively checks that P is k-inductive and, if it is not, adds to it any invariant from R that eliminates the k-induction counter-example found by the SMT solver. The whole process is terminating since the set R is finite. It is also sound in the sense that its returned formula is a k-inductive strengthening of P whenever the input $\phi = \psi_1 \wedge \cdots \wedge \psi_n \wedge P$ is; but it is not guaranteed to yield the smallest k-inductive strengthening of P contained in ϕ. We forgo this strong minimization requirement intentionally, for practical efficiency.

We observe that in principle applying `trim` is computationally expensive because of the cost of its entailment checks. In practice though, `trim` terminates after a very small number of iterations (generally less than three on our benchmarks). Moreover, it is very effective at removing large unnecessary parts of the certificate. Considering that certificates with hundreds of conjuncts are common, the cost of running `cherry-pick` on the original certificate can become prohibitive. 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 Irvii *et al.* for invariants produced by SAT-based IC3-like model checkers [19]. While potentially more precise, many of their other steps require a number of satisfiability checks linear in the size of ϕ which is already prohibitive for the SMT case.

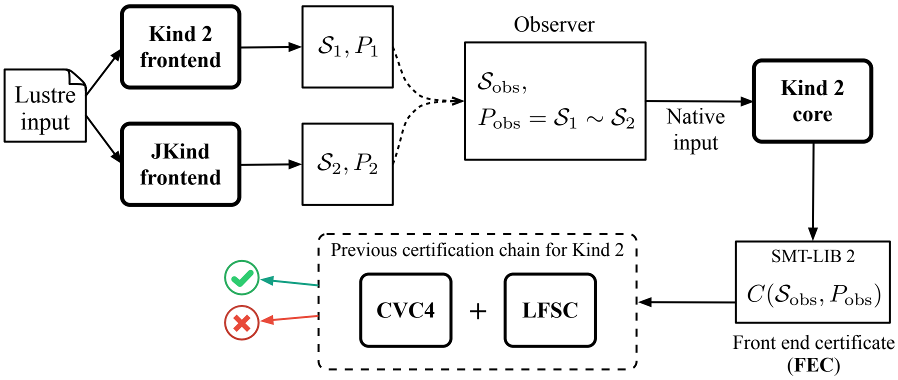
Lustre input
Kind 2 frontend
JKind frontend
$\mathcal{S}_1, P_1$
$\mathcal{S}_2, P_2$
Observer
$\mathcal{S}_{\text{obs}}$,
$P_{\text{obs}} = \mathcal{S}_1 \sim \mathcal{S}_2$
Native input
Kind 2 core
SMT-LIB 2
$C(\mathcal{S}_{\text{obs}}, P_{\text{obs}})$
Front end certificate
(FEC)
Previous certification chain for Kind 2
CVC4
+
LFSC

Figure 4: Generation and verification of front end certificates.

3 Front End Certificates

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 preliminarily applies a number of optimizations and simplifications to the input, such as slicing, constant propagation, and so on. As a consequence, a careful certification process needs to argue that these optimizations and translations are correct. One possibility would be to prove once and for all that Kind 2's front end is correct, using for instance a proof assistant like Coq. This would require one to formalize the semantic of the input language and show that it is preserved by all simplifications and translations. This approach is currently rather onerous in terms of human effort and makes future changes to the front end quite burdensome. Another possibility, which we consider here instead, is to have this translation phase generate certificates of its own.

Our certification process for the front end of the model checker is lightweight and designed with the goal of keeping the whole chain entirely *automatic*. Instead of proving semantic preservation between the input Lustre model and its internal translation as an FOL representation of 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 [27,28] and in the Systerel Smart Solver [26]. 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 [29]. JKind too converts input models to an FOL representation based on a two-state transition relation. 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.

3.0.1 Equivalence Observer.

Our certificate encodes the claim that the transition relations constructed by the two independent front ends are observationally equivalent over a set of *relevant* state variables. Intuitively, the main idea for such certificates is to define them as a transition system that observes the internal states of the two systems generated by each front end. The *observer* system feeds its two subsystems the same inputs and verifies that their *externally visible* behavior is the same.

For $i = 1, 2$, let $\mathcal{S}_i = (\mathbf{x}_i, I_i[\mathbf{x}_i], T_1[\mathbf{x}_i, \mathbf{x}'_i])$ be the internal transition system respectively generated by JKind and Kind 2, with $\mathbf{x}_1$ and $\mathbf{x}_2$ sharing no components. We construct an observer system $\mathcal{S}_{\text{obs}}$ and a safety property $P_{\text{obs}} = \mathcal{S}_1 \sim \mathcal{S}_2$ expressing a suitable notion of observational equivalence ($\sim$) between $\mathcal{S}_1$ and $\mathcal{S}_2$. Then we could check the correctness of this observer system in *the same way* as we would check the correctness of $\mathcal{S}_2$ with respect to the original safety property. The full process is illustrated in Figure 4. There, the module **Kind 2 Core** is the core part of Kind 2, which works directly with the internal FOL representation of a transition system.

One way to define observational equivalence would be to say that the two systems $\mathcal{S}_1$ and $\mathcal{S}_2$ produce the same outputs when given the same inputs. This, however, is unnecessarily strong for our purposes and in fact, depending on how different the two translations are, such equivalence might not even hold or may be very hard to prove. This is the case, for instance, if the two model checkers employ different simplification strategies, especially if they have different ways to apply property-directed slicing on the input model. A workable notion of equivalence is a *property-based* one where we simply require the two systems to agree on the truth value they assign to their respective version of the original input property in the Lustre model, when given the same input. For $j = 1, 2$, let $\mathbf{i}_j$ be the subtuple of $\mathbf{x}_j$ that corresponds to the input variables of the input Lustre model. Then $\mathcal{S}_{\text{obs}} = (\mathbf{x}_{\text{obs}}, I_{\text{obs}}, T_{\text{obs}})$ and P_{obs} are defined as follows:

$$\begin{aligned} \mathbf{x}_{\text{obs}} &= \mathbf{x}_1, \mathbf{x}_2 & I_{\text{obs}} &= \mathbf{i}_1 \approx \mathbf{i}_2 \ \wedge\ I_1[\mathbf{x}_1] \ \wedge\ I_2[\mathbf{x}_2] \\ P_{\text{obs}} &= (P_1[\mathbf{x}_1] \Leftrightarrow P_2[\mathbf{x}_2]) & T_{\text{obs}} &= \mathbf{i}'_1 \approx \mathbf{i}'_2 \ \wedge\ T_1[\mathbf{x}_1, \mathbf{x}'_1] \ \wedge\ T_2[\mathbf{x}_2, \mathbf{x}'_2] \end{aligned}$$

where, for two tuples $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{b} = (b_1, \ldots, b_n)$, the expression $\mathbf{a} \approx \mathbf{b}$ denotes the formula $\bigwedge_{i=1,\ldots,n} a_i = b_i$. Note that the set of state variables of the observer system $\mathcal{S}_{\text{obs}}$ is the (disjoint) union of the variables of $\mathcal{S}_1$ and $\mathcal{S}_2$. The system itself is effectively the parallel composition of $\mathcal{S}_1$ and $\mathcal{S}_2$ after their corresponding input variables have been pairwise identified.

One possible problem with this approach is the small likelihood that the property P_{obs} is k-inductive for $\mathcal{S}_{\text{obs}}$ and for a small k (so as to be easily provable by Kind 2). We mitigate this by identifying heuristically pairs of corresponding state variables from $\mathbf{x}_1$ and $\mathbf{x}_2$ and suggesting their equality as a potential auxiliary invariant for Kind 2 to try. Some of these equalities may indeed be proven invariant and so they can potentially help in the proof of P_{obs}. Note that while this harks back to the stronger notion of observational equivalence we mentioned earlier, it is not the same since the equivalence between certain non-input variables is only suggested, not required.

Example 1. *Consider again the Lustre model and property of Figure 2. The systems $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively generated by JKind 2.1*[3] *and Kind 2 from that model are the following, in abstract syntax and modulo variable renaming:*

$\mathcal{S}_1$	$\mathcal{S}_2$
$\begin{aligned} \mathbf{x}_1 &= \{a_1, b_1, c_1, v_1\} \\ I_1 &= R[\top, \mathbf{x}_1, \mathbf{x}'_1] \\ T_1 &= R[\bot, \mathbf{x}_1, \mathbf{x}'_1] \\ R[g, \mathbf{x}_1, \mathbf{x}'_1] &= (v'_1 = a'_1 + b'_1 \wedge \\ &\quad c'_1 = ite(g, \tfrac{10}{10}, ite(c_1 > v'_1, c_1, v'_1))) \\ P_1 &= \left(a_1 > \tfrac{0}{10} \wedge b_1 > \tfrac{0}{10} \Rightarrow c_1 > \tfrac{0}{10}\right) \end{aligned}$	$\begin{aligned} \mathbf{x}_2 &= \{i, a_2, b_2, c_2, v_2\} \\ I_2 &= (i \ \wedge v_2 = a_2 + b_2 \wedge c_2 = 1) \\ T_2 &= (\neg i' \ \wedge v'_2 = a'_2 + b'_2 \wedge \\ &\quad c'_2 = ite(c_2 > v'_2, c_2, v'_2)) \\ P_2 &= (a_2 > 0 \wedge b_2 > 0 \Rightarrow c_2 > 0) \end{aligned}$

The equivalence observer $\mathcal{S}_{\text{obs}}$ is defined by

$$\begin{aligned} \mathbf{x}_{\text{obs}} &= \mathbf{x}_1, \mathbf{x}_2 & I_{\text{obs}} &= (a_1 = a_2 \wedge b_1 = b_2 \ \wedge\ I_1 \ \wedge\ I_2) \\ P_{\text{obs}} &= (P_1 \Leftrightarrow P_2) & T_{\text{obs}} &= (a'_1 = a'_2 \wedge b'_1 = b'_2 \ \wedge\ T_1 \ \wedge\ T_2) \end{aligned}$$

Suggested auxiliary invariants in this case will be the equalities $a_1 = a_2$, $b_1 = b_2$, $c_1 = c_2$, and $v_1 = v_2$ between corresponding state variables in the two systems. □

[3] We produce $\mathcal{S}_1$ by having JKind 2.1 write a dump file from which we can extract its internal representation.

The equivalence observer $\mathcal{S}_{\text{obs}}$ and the associated property P_{obs} constitute an intermediate certificate of Kind 2's translation from the input Lustre model and properties to Kind 2's internal representation. Checking it consists in proving that the property P_{obs} is invariant for $\mathcal{S}_{\text{obs}}$. Now, not only can this check be done by Kind 2 itself, it can also be provided with its own proof certificate, of the sort discussed in Section 2. We call the latter *front end certificates*. The reason this is both feasible and sensible is that $\mathcal{S}_{\text{obs}}$ and P_{obs} are not written in Lustre. Instead, they are generated, in SMT-LIB-like syntax, in a format that corresponds directly to the internal representation of transition systems and properties. This allows us then to bypass Kind 2's Lustre front end completely when model checking equivalence observers.

4 Experimental Evaluation

We evaluated our certificate generation and checking techniques on a set of academic benchmarks and a smaller set of industrial-grade benchmarks.[4] We selected only benchmark problems consisting of a Lustre model and one or more properties that Kind 2 could prove with a 5 minute timeout. Because the instrumentation of CVC4 to produce full LFSC proofs is still in progress, we focused on the certificates generated by Kind 2, without the lower-lever LFSC proofs generated by CVC4 to certify of its own reasoning. Instead, we used Z3, a widely used SMT solver with similar capabilities as CVC4, to corroborate CVC4's answers. We ran our tests on a Linux machine with two 12-core 64-bits AMD Opteron processors and 32GB of memory. We used a certifying version of Kind 2[5] based on Kind 2 `v0.8`. The CVC4 binary was from version `1.5-prerelease` (git master `230dbc7f`), while the Z3 binary was from version `4.3.2`. Each run of every tool (all calls to Kind 2 but also to the SMT solvers) was given a timeout of 5 minutes.

Certificate simplification. The plot in Figure 5 focuses on the effects of the certificate simplification techniques presented in Section 2. It shows how many problems a particular configuration can cumulatively process within a certain amount of time. We compare various measures: S measures the time needed by Kind 2 to solve the model checking problem and generate an initial safety certificate, i.e., before minimization;[6] mE measure the time to minimize the safety certificate using the *easy* simplification technique (*i.e.*, only `trim` in Algorithm 1); m is the time to do the full minimization (*i.e.* both `trim` and `cherry-pick`); finally, cvc4 measures the time necessary for CVC4 to check the safety certificate—we exclude front end certificates for now. We can see from the plot that without any minimization (S+cvc4) we can check a lot less certificates and take much more time than with minimization. We can also see that, even if the full minimization 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 minimization 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 a minimization overhead for a single benchmark, which is larger than the solving time. However, even with this outlier, after a while it becomes worth it to apply full minimization on the certificates.

Certification process. The plot in Figure 6 focuses on the complete certification chain. The measurements show the time necessary up to produce and check certificates for the safety property by CVC4 and by Z3 (C1cvc4 and C1z3) and the total time of the full certification process (C2cvc4 and C2z3). To recap, the latter includes the time to prove the input property; generate and fully minimize its safety certificate; construct the equivalence observer, including the time to call JKind and extract its transition

[4] All benchmarks are available at `https://github.com/kind2-mc/kind2-benchmarks`.

[5] Available at `https://github.com/kind2-mc/kind2/tree/certificates`.

[6] We do not show the time to just solve the problem because its difference with S is negligible.

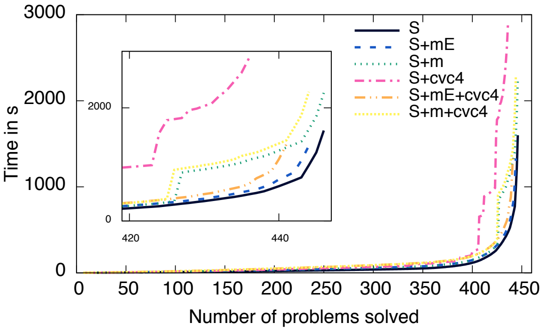

Figure 5: Overhead and improvements of minimization.

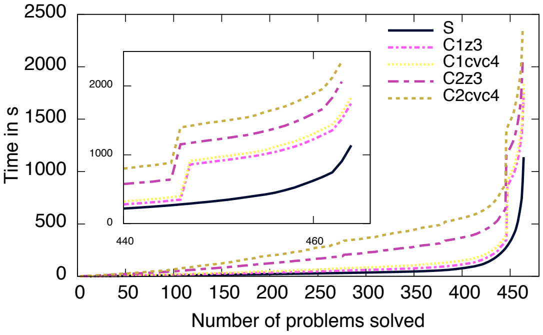

Figure 6: Experimental evaluation of certification chain on a large set of benchmarks.

system; model check the observer with Kind 2; and generate, minimize and check the front end certificate. Additional assurance is provided by the fact that in our tests CVC4 and Z3 agreed on every certificate.

The biggest bottlenecks are the generation of the equivalence observer and the simplification of certificates. A large percentage of this time, especially on simpler problems, is taken by the call to JKind, which incurs a constant overhead (of about 1s per problem) to start the JVM. 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.[7] We find this level of performance, which we think could be improved further, already rather good, especially considering that a lot of the benchmarks we used are non-trivial.

5 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.[8] One example of this kind of certification effort is the modern SAT solver `versat` which was developed and verified using the programming language Guru [24]. 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 an interactive proof assistants like Coq or Isabelle, and obtain an executable program from these tools through a refinement process or a code extraction mechanism. Although the first formal verification of a model checker in Coq for the modal μ-calculus [34] 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 package. Esparza *et al.* [13] have fully verified a version of the Spin model checker with the Isabelle theorem prover. Using successive refinements, they built a correct by construction model checker from high level specifications down to functional (SML) code.

Certifying Model Checkers. A number of techniques have been proposed for certifying model checking, the approach we advocate in this work, whereby a model checker produces an independently checkable

[7] As confirmed by the full numerical results available at `http://cs.uiowa.edu/~amebsout/smt16`.

[8] Some impact is possible if the tool has to be modified to facilitate its functional verification.

proof certificate for each property it claims to hold in the model. Some of them were developed for automata-based model checking, others for logic-based model checking. Earlier solutions (e.g., [21,23,25]) were limited to finite-state systems. The first certifying model checker for infinite-state systems was perhaps the C model checker BLAST [18], 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) [16], 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 [12], 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 parameterized model checking, Cubicle [10] generates certificates as Why3 files that can be independently checked by several SMT solvers and automated theorem provers [11]. In this work, trust is claimed through the redundant use of multiple solvers.

Proof-producing Solvers. Logic-based model checkers, which utilize SAT or SMT solvers as internal reasoning engines, can eliminate these large and complex tools from their trusted core by using proof-producing solvers. A recent approach for the certification of SAT and SMT solvers [2, 20] consists in having the solver produce a detailed certificate in which each rule is read and verified by the composition 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]. The SMT solvers CVC3 and CVC4 are able to produce proof objects in LFSC, an extension of LF with computational side conditions [35]. The computational power brought by the use of side conditions allows the efficient checking of very large proofs [35].

6 Conclusion and Future Work

We have presented a two-fold technique for generating and checking proof certificates for SMT-based model checkers and have applied it to the model checker Kind 2. Given a Lustre model as input and one or more safety properties, Kind 2 generates two certificates for properties that it is able to prove for the system. The first one attests that the model and the property are encoded correctly in Kind 2's internal representation format. It does that by proving the observational equivalence over the input property between the internal system and another one produced from the same Lustre model by an independent third-party tool. The second certificate certifies that the encoded property is an invariant of the internal transition system encoding the Lustre model. Certificates are initially generated as (possibly combined) k-inductive invariants and simplified before being emitted, to facilitate their external verification.

We mentioned in the introduction that CVC4 has recently been instrumented to produce LFSC proofs of valid queries. This is currently limited to a subset of the background theories that CVC4 supports. Although CVC4's proof production does not extend to the full fragment used by Kind 2, for Lustre models with Boolean streams only Kind 2 is already capable of reducing its own certificates to a single LFSC proof object by post-processing and assembling various proof objects returned by CVC4. Future work will consist in further reducing the trusted core in our approach by having LFSC proofs for the complete fragment of Lustre supported in Kind 2.

Kind 2 has the ability to perform compositional and modular analysis of Lustre models extended with assume-guarantee style contracts. Another line of future work will be to extend the current certificate generation techniques to apply to compositional proofs by incorporating their underlying abstraction mechanisms.

Acknowledgments.

We would like to thanks Lucas Wagner for his feedback on this work and Andrew Gacek for his assistance with JKind.

References

[1] H. Amjad. Programming a symbolic model checker in a fully expansive theorem prover. In *TPHOL*, pages 171–187. Springer, 2003.

[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Wener. Verifying SAT and SMT in Coq for a fully automated decision procedure. In *PSATTT*, 2011.

[3] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *CPP*, pages 135–150. Springer, 2011.

[4] T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Vardi, and L. Zuck. Formal verification of backward compatibility of microcode. In *Proceedings of CAV'04*, volume 3576 of *LNCS*, pages 185–198. Springer, 2005.

[5] H. Barendregt and F. Wiedijk. The challenge of computer mathematics. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 363(1835):2351–2375, 2005.

[6] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of CAV'11*, pages 171–177. Springer, 2011.

[7] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB standard: Version 2.5, 2015.

[8] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels. Model checking flight control systems: The Airbus experience. In *ICSE*, pages 18–27, May 2009.

[9] A. R. Bradley. SAT-based model checking without unrolling. In *Proceedings of VMCAI'11*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[10] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Cubicle: A parallel SMT-based model checker for parameterized systems. In *CAV*, pages 718–724, 2012.

[11] S. Conchon, A. Mebsout, and F. Zaïdi. Certificates for parameterized model checking. In *Proceedings of FM'15*, volume 9109 of *LNCS*, pages 126–142. Springer, June 2015.

[12] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. SLAB: A certifying model checker for infinite-state concurrent systems. In *TACAS*, volume 6015, pages 271–274. Springer, 2010.

[13] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A fully verified executable ltl model checker. In *CAV*, volume 8044, pages 463–478. Springer, 2013.

[14] L. Fix. Fifteen years of formal property verification in intel. In *25 Years of Model Checking - History, Achievements, Perspectives*, pages 139–144, 2008.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[16] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.

[17] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proceedings of CAV'02*, pages 526–538. Springer, 2002.

[19] A. Ivrii, A. Gurfinkel, and A. Belov. Small inductive safe invariants. In *Proceedings of FMCAD'14*, pages 115–122. IEEE, 2014.

[20] C. Keller. *A Matter of Trust: Skeptical Communication Between Coq and External Provers*. PhD thesis, École Polytechnique, June 2013.

[21] O. Kupferman and M. Y. Vardi. From complementation to certification. *Theor. Comput. Sci.*, 345:83–100, November 2005.

[22] S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, Feb. 2010.

[23] K. S. Namjoshi. Certifying model checkers. In *CAV*, pages 2–13. Springer, 2001.

[24] D. Oe, A. Stump, C. Oliver, and K. Clancy. versat: A verified modern SAT solver. In *VMCAI*, volume 7148, pages 363–378. Springer, 2012.

[25] D. Peled and L. Zuck. From model checking to a temporal proof. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 1–14. Springer, 2001.

[26] M. Petit-Doche, N. Breton, R. Courbis, Y. Fonteneau, and M. Güdemann. Formal verification of industrial critical software. In *FMICS*, pages 1–11. Springer, 2015.

[27] Prover Technology. Prover certifier. `http://www.prover.com/products/prover_certifier`.

[28] Prover Technology. Prover Plug-In. `http://www.prover.com/products/prover_plugin`.

[29] Rockwell Collins. JKind - a Java implementation of the KIND model checker. `https://github.com/agacek/jkind`.

[30] RTCA. DO-178C, Software considerations in airborne software, Dec. 2011.

[31] RTCA. DO-330, Software tool qualification considerations, Dec. 2011.

[32] N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR*, pages 1–16. Springer, 2000.

[33] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, pages 108–125, London, UK, 2000. Springer.

[34] C. Sprenger. A verified model checker for the modal μ-calculus in coq. In *TACAS*, TACAS '98, pages 167–183, London, UK, UK, 1998. Springer.

[35] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.