Certification of Model Checking Tools: Study Report

Alain Mebsout
Cesare Tinelli
The University of Iowa

March 26, 2015

Contents

1 Model Checking

1.1 Introduction

Automatic approach One of the most successful technique for the formal verification of programs is Model Checking. Its biggest strength compared to other approaches is that it is a largely automatic process that requires very little intervention from human experts. This makes it particularly adequate in settings where formal assurance is desirable but the resources allocated to the task are limited. Because of its "push-button" aspect, model checking can be easily deployed in a traditional software (or hardware) development process by engineers that are not experts in formal verification.

Industry standard Model checkers are already commonly used in the industry, particularly for the development of critical systems. This is the case for hardware vendors like Intel, who has been using model checking since 1995 [5,29] in particular the model checker Mur-φ [25] for software verification [24]. This is also the case for the avionics industry where model checking is used to verify *e.g.* flight control systems. Impressive industrial verification efforts have been achieved by industry leaders like NASA [35,36] with Java PathFinder and the Spin model checker [38], Rockwell Collins [46–48,65,66] with NuSMV [17] and Prover Plug-In [56], SRI with their own model checker SAL (Symbolic Analysis Laboratory) [22,57] or Airbus [14, 43,63] with the SCADE Design Verifier [1,28]. Another model checker that is widely used in avionics industries is Uppaal [9] for the analysis of real time systems [11,15].

In all these cases, model checking is used as a mean to ensure certain correctness properties of the system at hand. More generally, formal methods are usually introduced in the development process as a mean to replace costly (unit and black box) testing [40,43,67]. At the same time they provide a greater level of confidence. For example, model checkers are able to consider exhaustive program behaviors for their analysis contrary to traditional testing which only examines a limited number of inputs and states.

So as a tool, a model checker bears an important responsibility for the validation of critical pieces of software. If a model checker gave incorrect answers, defective software (or hardware) could go into production and be shipped in safety critical systems. However, a model checker will never generate code that will be executed in a critical system so their impact is considered to be relatively low (compared to *e.g.* a compiler).

Specialized tools The purpose for which a model checker is used, has a direct repercussion on the consequence of a wrong answer. A model checker can be used to find defects. Much like testing, the tool will only report erroneous states but is more likely to discover subtle bugs. In this case, a model checker will not be used to declare that a program is free of errors. The other mode of use is to rely on a model checker to verify that a system is correct with respect to a formal specification. We look in detail at what this means in terms of soundness in the following.

1.2 Model Checking Techniques

1.2.1 Enumerative

Enumerative model checking algorithms are limited to analyzing finite programs. They essentially traverse the whole graph representing transitions over the state space of the program. For this they use many different search techniques. They are said to be *enumerative* because they consider exact individual states one-by-one (or almost) as opposed to *symbolic* techniques which consider *sets of states* at once.

Explicit state

In the enumerative approach, explicit state (or *stateful*) search constructs and stores the states of the program to be analyzed in some sort of memory. Two main remarks can be made here. First, the actual states can be stored in various data-structures, usually a hash table for the set of visited states – essentially because a common operation is to query for membership – and a queue, stack, or heap for the states to visit. The search strategy is usually determined by this last structure. Secondly, instead of storing the entire state space upfront, model checkers normally construct the reachable state graph on the fly. This allows to only store the set of reachable states instead of the whole (and much larger) state space.

To trust an explicit state model checker, one has very little choice but to trust the whole model checker. This includes the exploration and search algorithm. In particular, we need to trust that all transitions are executed and represented correctly in the graph. We also need to trust that the data structures employed (hash table, queues, stacks, *etc.*) are correct. For instance, we don't want to miss states because they were incorrectly dropped by the queue, or because the hashing and membership functions of the hash table were wrong.

Verifying safety properties on a reachablity state graph is a linear process which constitutes in ensuring that every state satisfies the property. This check needs to be trusted, however to a lesser extent if we consider that the burden is on exploring the state space (and it usually is) rather that checking the property, something which can always be reverified afterwards.

For other more complex temporal properties, checks conducted after construction of the reachability graph can rarely be trivialized. For instance the model checker Spin [38] implements an enumerative algorithm based on an automata theoretic framework. LTL properties (or rather their negation) are converted to Büchi automata and checked for intersection with the reachability graph. A simplified version of this model checker has recently been entirely reimplemented within the Isabelle theorem prover [54] and proven correct [27, 51].

These tools generally use special techniques to control the state explosion problem inherent to this approach. Among them are *reduction techniques* like partial order reduction or symmetry reduction and *compositional techniques* like assume guarantee [39]. These are usually implemented at the heart of the model checker to scale down the number of visited states but now we also need to trust the meta-theorems that govern these optimizations and their correct application.

Bitstate hashing

To cut down on the space requirements, some model checkers sacrifice their soundness by only storing *hashes* of states instead of the exact states. Because of the collisions that may appear in the hash function (two different states can have the same hash value), some states, and in consequence parts of the reachability graph, can be missed. To mitigate the negative effects of not storing the states, these techniques require the use of hash functions with particular properties so as to guarantee a probability that a state will be missed.

As such, bitstate hashing is only used for *bug-finding*.

Stateless

Another technique used in enumerative model checking is based on the observation that it is sometimes not necessary to even remember the states of the program. This is the case in some finite state concurrent programs where only the scheduler has an impact on the paths taken. In this case it is only necessary to explore all possible *interleavings* given as scheduler outputs. In between the calls to the scheduler, it suffices to execute the program and inspect its state at strategic points. The advantage of this technique is that programs can be instrumented for this framework independently of the language. This approach was pioneered by the model checker VeriSoft [31] and is also referred to as *execution based* model checking. The program must be restarted from scratch for each scheduler interleaving so there is no real sharing between traces. To counter this potential drawback, *partial order reduction* is very heavily used.

The main disadvantage of this technique is that only finite schedules can be explored and so generally only terminating programs can be handled.

To trust a model checker like VeriSoft one does not need to trust a translation

between different formalisms because the program is executed directly. However it is vital to ensure that all sources of *non-determinism* are accurately accounted for in the use of the scheduler.

1.2.2 Symbolic

Symbolic model checking was conceived to counter the effects of state explosion traditionally encountered in enumerative techniques. Instead of storing individual states in memory, this approach is based on the idea of using a single object to represent a set of states. One of the challenges is to find appropriate compact representations for sets of states [16]. In this section, we describe a few symbolic model checking techniques and highlight some of their peculiarities in regards to certification.

BDD based

One such representation of the state transition relation is based on BDDs (Binary Decision Diagrams). This particular data-structure is used to represent compactly and reason on *Boolean functions*. McMillan reports on this technique in his thesis [45] and gives several graph based algorithms for the language of μ-calculus.

The use of compact data-structures to represent *large* sets of states also allows to grasp regularities and symmetries that naturally appear in circuits or other hardware systems. This and the fact that circuits are Boolean by essence has made symbolic BDD based model checking a widely used technique for the analysis of hardware.

In addition to the algorithms, one must now trust the implementation provided by the BDD library (be it external or *ad hoc*) used in the model checker. Note that there has been attempts at providing certified BDD libraries [64] in Coq but most industrial model checkers like NuSMV 2 [17] do not make use of such libraries. However, because the reachable state space can be represented more compactly than with traditional enumerative approaches, it is easier to dump it for future re-verification. The main drawback of BDD based approaches is that the size of the constructed diagram is directly (and very sensitively) impacted by the ordering chosen on Boolean variables. In the unfavorable case, the explosion can be exponential in the number of variables, negating the improvements brought by the approach while still being more complex than a simple enumeration.

Bounded Model Checking

Biere *et al.* describe in [12] a technique which sacrifices soundness in favor of efficient bug finding called Bounded Model Checking (BMC). The idea of BMC is to look for counterexamples among program executions whose size is bounded by a number of steps k. Its main strength is to take advantage of modern SAT solvers to handle all the propositional reasoning. The original problem can be efficiently encoded in

a propositional formula whose satisfiability directly yields a counterexample. If no error is found for traces of length k, the value of the bound k is incremented until a bug is found, or the problem becomes too hard.

Because the encoding to propositional constraints captures precisely the semantics of circuits, model checkers of this family have been used to uncover subtle implementation bugs. BMC is most successful in exposing complex bugs and this is how it is traditionally used. As such, it does not pose a trust challenge because we are only interested in error traces. These usually take the form of initialization values for variables and a sequence of steps to reach an error state. They are small and can be replayed.

In some cases, an upper bound on k is known (the completion threshold) which allows to affirm that the system is safe with respect to a property. In this case, several meta arguments need to be trusted – or their correctness justified – if BMC is to be used as a *proof* technique.

k-Induction

Another extension of BMC for proof purposes is called k-induction and adds an induction check at each step (*i.e.* for each k). It differs from a classical induction scheme by the following point: when asked to check if the property is preserved by one step, we assume it to be true in the k-previous steps instead of only the single previous step [23, 58].

A lot of the techniques mentioned above have been extended to handle systems with an *infinite* number of states. This is the case for instance when the variables of the program are in infinite domains (*e.g.* mathematical integers) or the data-structures are infinite (*e.g.* buffers, wait queues, unbounded memory). To handle these systems, we have the choice between two alternatives: directly manipulating representations of infinite sets of states, or constructing a finite abstraction of the system. For the software model checking we consider in this report, we will focus on the first approach. In this technique, an appropriate representation is first-order logic formulas. Previous techniques based on SAT solvers (only capable of handling finite domains encoded in propositional logic) were lifted to SMT solvers (Satisfisbility Modulo Theories). These solvers have a propositional engine at their core (a SAT solver) paired with methods for combining theories. Their power comes from the large number of built-in theories they support, such as linear arithmetic (on mathematical integers, or rational numbers), equality over uninterpreted functions, arrays, bitvectors, enumerated data-types, *etc.* Some of these solvers even support quantifiers (existential and universal) natively.

SMT-based k-induction for infinite systems is one of the model checking engines implemented in Kind 2, the model checker taken as a case study for this report. Trusting the algorithm implemented in this model checkers demands that we trust in particular:

- BMC, used for the base cases
- the k-induction scheme used
- the SMT solver(s)
- *etc.*

This list is not exclusive but we can already see that the complexity of modern model checkers demands that we trust increasingly complicated components and all their interactions.

To make things worse, a lot of these model checkers also have parallel architectures with processes running different engines, each of them exchanging information of the fly.

PDR

One of the other engines used in Kind 2 is a Property Directed Reachability (PDR) analysis. PDR is a process that tries to "connect" the unsafe states with the initial states by generalizing and keeping only what is necessary to discard spurious traces (blocking). It can also be viewed as a process that constructs an *inductive invariant* of the system, *i.e.* a subset of the state space:

1. for which the property is verified
2. that contains the initial states
3. that is closed by the transition relation.

In particular (2) and (3) imply that all reachable states verify this invariant.

There exists may variants of PDR. The first one was called IC3 and used a SAT solver at its core. It was tailored to the verification of hardware. The one implemented in Kind 2 lifts this algorithm and many of its optimizations to more expressive logics, and uses an SMT solver at its core.

The same remarks as k-induction can be made of PDR, except that PDR has an interesting property from a validation / certification standpoint. It constructs a *witness* of the safety of the system, *i.e.* a small object from which the safety can be easily inferred without the need for any kind of search.

1.2.3 Conclusion

★ Some comments + find numbers for state spaces

Technique	Category	Mode	State space	Input	Tools
Explicit state	Enum.	Proof	smaller	FSA	Murφ, Spin
Bitstate hashing	Enum.	Bug	small	FSA	
Stateless	Enum.	Proof	medium	real program	VeriSoft
BDD based	Sym.	Proof	10^{120}	FSA	SMV, NuSMV 2
BMC	Sym.	Bug	finite / ∞	circuit / Boolean program	CBMC
k-induction	Sym.	Proof	∞	transition system	Kind 2, SAL
PDR	Sym.	Proof	finite / ∞	circuit / transition system	IC3, Kind 2

Table 1.1: Model checking techniques

1.3 Anticipated Usage

In practice, model checking is effective both as a bug finding tool and as a verification tool. Contrary to test and simulation, model checking is able to explore the most intricate scenarios. This is especially true for concurrent asynchronous programs, which are notoriously difficult to grasp and reason about for the human mind. This is also true of synchronous programs which have many input parameters in possibly unbounded domains.

Cofer and Miller report in a case study [20] that they found model checking in a DO-178C setting to be useful both to find errors of the designs of a flight guidance system (FGS), and to verify the absence of errors once the designs were corrected. They also give a more detailed account of the errors exposed by Kind in a NASA contractor report [21]. Out of the sixteen errors found by Kind 2, the authors estimate that only five of them were likely to be discovered by traditional verification, which clearly shows the interest of model checking. They especially stress the usefulness of the error traces returned by the model checker[1].

We can then expect model checkers to be used in both modes. In the following,

[1]Though not perfect: some traces can contain noise that do not pertain to the actual defect. However, modern model checkers are able to return *smooth* traces to limit this noise.

we give a brief overview of each mode and look at potential issues.

1.3.1 Error detection

Finding a defect, *i.e.* a state that exhibits a behavior that escapes what the designers expected, is something for which model checkers are very useful. Their exhaustive nature makes them particularly apt at exploring intricate behaviors and corner cases that were not accounted for. Moreover, when such a defect is discovered, they traditionally expose an *error trace*, *i.e.* input values and program paths that falsify one of the desired property. This trace is very helpful in practice for designers and developers as it allows the defective behaviors to be easily reproduced, and possibly corrected.

Potential Issues

Suppose now that we have a model checker that returns erroneous results. In this case, there are two possible mistakes.

The first one is to find a defect that does not exist and return a *spurious* error trace. Usually this will be easily detected when a human tries to reproduce the bug and finds out that the trace is not possible. Even if no further checks were conducted, a program could be mistakenly declared as defective and as so would never be shipped. The impact of a *completeness* bug in a model checker that is only used for bug finding is relatively low and harmless.

The second possible mistake is to *not* report a defect that is actually present in the program. It could seem that this is a bigger problem, and it can be. If we have no guarantee on the result of the model checker but it is only used as a safeguard in addition to other techniques then it is not very problematic. We already know that the model checker will not report all bugs of the program. If, on the contrary, some coverage guarantees are obtained through the use of a model checker, then a *soundness* bug can be harmful. For example, a model checker could claim that there is no bug for program executions of less that ten instructions. Other analyses could rely on this fact and claim the program to be safe when in fact it is not.

1.3.2 Proof

Because a lot of model checkers essentially conduct an *exhaustive* exploration of a program's behaviors pertaining to some properties, they are also extensively used to *prove* that programs meet their specification. In this case, the user relies on the model checker to find defects, but also to show the absence of defects in the program. Their ability to consider all behaviors in exhaustive ways constitute a big advantage of model checking over testing. Theoretically, when such a model checker is able to verify the program, then we can be sure it satisfies its specification. Of course we loose this guarantee if the model checker is itself buggy.

Potential Issues

Suppose that the model checker declares that the program has an error on a spurious trace. As before, this kind of mistake can be easily detected by simple *a posteriori* checks, *e.g.* by trying to replay the error trace in the real system. A *completeness* error is still relatively harmless because it would only prevent error-free software to be shipped. However, when used as a proving tool, it is perfectly reasonable to assume that a model checker is potentially the only tool used to verify a program. This means that completeness error could potentially block the development process.

Now, if an error of the analyzed program is not reported then the use of a model checker for proof purposes becomes dangerous. A critical error could go unnoticed because of a *soundness* bug, which could have potentially disastrous consequences depending on the severity of the error and the criticality of the system. For instance a safety error in the flight controller of an aircraft could result in the loss of lives.

1.3.3 Conclusion

Nature of the bug	Usage	Severity	Impact on safety
Completeness	Bug finding	harmless	no
Completeness	Proof	relatively harmless	no
Soundness	Bug finding	potentially harmful	**yes**
Soundness	Proof	harmful	**yes**

Table 1.2: Severity of bugs in a model checker

We sum up this section with the table 1.2 that gives the severity and impacts of bugs in the model checker depending on its mode of use. The configurations are given in ascending order of severity. We will concentrate on bugs that can be harmful and can impact the safety of the system. They are given below the red line in the table.

In other words we will focus in soundness bugs of the model checker, independently of its mode of use.

1.4 Trust Proposition

Definition 1.4.1 (Trust proposition for a model checker)**.**

1. ***Trusted Logic*** *The logics used by the model checker are trusted.*

2. ***Valid Model*** *The model and properties accurately depict the system under scrutiny in the semantic given by the logic of the model checker.*

3. ***Correct Translation*** *The input system is correctly translated to its internal representation.*

4. ***Adequate Mode*** *The mode (error detection, or proof) is adequately selected and identified for the desired analysis.*

5. ***Correct Algorithms*** *The model checking algorithms are sound for the models and properties expressible in the supported logic.*

6. ***Correct Implementation*** *The model checking algorithms are correctly implemented.*

7. ***Trusted Components and Libraries*** *If the model checker makes use of external libraries, their implementation is trusted, and if the model checker uses external tools, they are trusted to be correct.*

8. ***Correct Compilation*** *The model checker and its components are correctly compiled to executable machine code.*

9. ***Correct Execution*** *The machine correctly runs the executable.*

10. ***Trusted IO*** *The parsing and output of the model checker are trusted and interpreted correctly.*

The rest of this section describes in detail into each point of the trust proposition. We also try to evaluate (based on our own experience) the risk of an error that could compromise the soundness of the tool. The risk is given as a number between 0 and 5, without any particular meaning excepted that a higher number depicts a higher risk (in our opinion) and identical risk values mean that risks of a soundness error are somewhat similar. When pertinent, we give an example for the logic based symbolic model checker Kind 2 at the end.

1.4.1 Trusted Logic

The logics used by the model checker are trusted.

Risk: 1

Most formal methods tools rely on some underlying logics. SAT or SMT based tools, like a lot of model checkers, use classical logic and a limited set of base theories. Those have been intensively studied, are widely used and very well defined. For instance, SAT based model checkers only rely on the correctness of propositional logic. SMT based tools use combinations of propositional logic, and background

theories like equality over uninterpreted functions, linear (or non-linear) integer (and real) arithmetic, functional arrays, *etc.* Some tools offer the user the possibility to constraint some under-specified symbols by adding axioms in their models. Such axiomatizations are reflected in the logic and can render the base theories unsound.

Example (Kind 2)**.** *Kind 2 is an SMT based model checker. It does not allows the use of all theories supported by the underlying SMT solvers but only a subset. At the moment, only quantifier free combinations of propositional logic, equality over uninterpreted functions as well as linear and non-linear real and integer arithmetic. It does not allow the user to further specify some symbols by adding logic axioms.*

1.4.2 Valid Model

The model and properties accurately depict the system under scrutiny in the semantic given by the logic of the model checker.

Risk: 5

The term *model checking* was originally coined by Clarke [18, 19] to describe the technique of verifying if a program P is a model of a formula φ ($P \models \varphi$). In model checking, the formula φ is usually expressed in a special modal logic called a temporal logic [53, 55]. These logics allow to specify several kind of properties, among which:

- safety: a bad configuration never happens, or
- liveness: an expected behavior will eventually happen.

Different temporal logics allow for different properties to be expressible. Generally, model checkers only support one logic, and can verify a subset of all expressible properties. For instance, some model checkers only allow the verification of *safety properties* and *deadlock freedom*.

The term *model* in the expression *model checking* can also refer to the idea that programs are commonly represented by an abstract model. Traditional models for programs include transition systems, finite-state automatons, Petri nets, *etc.*

To define the analysis we want to conduct with model checking, one must first be able to define (1) what are the models, and (2) what properties of these models are to be verified.

Models

The *models* manipulated by model checkers are (low or high level) representations of systems and depend on the kind of program to be analyzed. For example, it is possible to translate RTL to finite-state machines. Contrary to programs, models are conceptually simple and have a well defined semantic. Finite-state machines can

be represented by *e.g.* Kripke structures. A Kripke structure over a set of predicates P is simply a tuple (S, I, R, L) where S is a finite set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, and $L : S \to 2^P$ is a labeling function (over the predicates P). A behavior is then defined in terms of states and transitions (*i.e.* a run or computation).

When doing model checking, it is of paramount importance to know what models are handled and how they are obtained. One possibility is for source programs or system descriptions to be translated (and sometime abstracted) to models. This allows for real implementations of systems to be verified. The other possibility is that the model is created early in the design process using some high-level modeling language. Verification is performed on the designs prior to implementation as a way to ensure that the design is sound. Traditionally in this setting, the real system is then implemented with the aid of the abstract model. In all cases, one must keep in mind that the model checking phase is always conducted on the model and that all properties are verified only for this model.

The limits imposed by the model are generally obvious. For instance, using finite-state automatons restricts the systems that can be analyzed to be themselves finite-state or to have a finite-state abstraction. On the other hand, a model that provides built-in support for clocks or continuous time will be particularly adapted to represent timed systems.

Properties

Properties that can be analyzed depend essentially on the logic that is supported by the model checker. For reactive systems, it is important to reason on its *temporal* behaviors. In particular, concurrent systems with their intricate behaviors greatly benefit from temporal specifications.

For instance LTL (Linear Temporal Logic) allows to represent infinite computations. Probably the most basic form of temporal property is the class of safety properties. They characterize properties which say that a dangerous state can never happen. It is also one of the most commonly used kind of property. Assertions written in source code essentially correspond to safety properties. Invariants, which are properties that must be true in every state of the program, are also safety properties (the safety property is that an invariant can never be violated). For instance a safety property of a concurrent system might be "It is not possible for two processes to have a simultaneous access to a shared variable."

Another class of temporal properties is liveness properties which encompass the notion of termination. In an elevator control mechanism, one such liveness property could be that "If a user presses the elevator button then the elevator will eventually arrive." Otherwise said, there is no computation in which the elevator indefinitely avoids the request of the user.

Example (Kind 2)**.** *The model checker Kind 2 uses as models reactive programs*

expressed in the synchronous language Lustre [?]. It also accepts arbitrary transition systems in a subset of first-order logic. In the later, the user has to make sure that the transition system accurately represents the artifact to be analyzed.

Kind 2 only supports safety properties. In the case of Lustre, they are usually expressed with the addition of an observer *node that monitors the outputs of other nodes. On the other hand, the native input allows any (acceptable) formula to be given as safety property.*

1.4.3 Correct Translation

The input system is correctly translated to its internal representation.

Risk: 3

When the model is constructed by hand, as is the case with if model checking is used as an aid during the design phase, then it falls upon the user to make sure that it is accurate for his verification needs. In particular, if under-approximations are used to avoid encumbering the model checker with details that do not pertain to the property, they should be documented. For example, it is sometimes necessary to replace unbounded buffers by *e.g.* buffers of size 3, or an unknown number of concurrent processes by *e.g.* two processes if the tool does not support parameterized reasoning.

In other scenarios, model checkers embed translation facilities in order to directly analyze artifacts in other description formalisms. A program expressed in a high level programming language like Java, can be represented by a model that can be understood and analyzed by a state-of-the-art model checker like Spin. This is the general idea at the core of the tool Java PathFinder. Other tools like Slam [6, 7] or Blast [10] perform a Boolean abstraction of a C program. Translations between different representations which do not have the same semantic expressive power often demands that approximations be employed. Some approximations are always safe because the abstracted system has more behaviors than the real system. These are over-approximations. On the other hand, under-approximations remove original behaviors to simplify the system. Some are precise; behaviors removed have no impact on the property to be verified. This is the case for *property-directed slicing*. Others like *data reduction* rely on meta-arguments and their correctness is more difficult to ensure.

Some model checkers will only accept already abstracted systems. The user has then the responsibility to ensure that his abstraction is correct. More so when performing under-approximation like program slicing.

Example (Kind 2)**.** *The model checker Kind 2 takes as input reactive programs expressed in the synchronous language Lustre [?]. This program is then simplified and transformed to successive intermediate representations and finally translated to an internal transition system expressed in a subset of first order logic. Kind 2 can*

also be used with its native input format (a direct textual representation of its first order transition system), in which case there are no preprocessing or translation phase.

1.4.4 Adequate Mode

The mode (error detection, or proof) is adequately selected and identified for the desired analysis.

Risk: 1

★ This is more of the user's responsibility, is it something we want to include in the trust proposition?

Example (Kind 2)**.** *Kind 2 uses many engines in parallel that can be enabled or disabled at will (all engines are enabled by default). In particular one of these engine performs bounded model checking (BMC) to find bugs. If it is the only engine activated (with the option* `--enable BMC`*) then Kind 2 will not terminate. If given a bound for the analysis, Kind 2 will never say that a property is true with only BMC but instead will print a message like:*

```
P1: true up to 561 steps
```

1.4.5 Correct Algorithms

The model checking algorithms are sound for the models and properties expressible in the supported logic.

Risk: 2

A basic requirement of a model checker is to use sound algorithms. The underlying approach should never report that a property is true when it may not be true.

Example (Kind 2)**.** *For instance, the induction [30] and k-induction [58] principles for asserting invariance of properties — as used in Kind 2 — are known to be sound.*

1.4.6 Correct Implementation

The model checking algorithms are correctly implemented.

Risk: 4

The implementation of a model checking algorithm in a tool should be free of defects that could compromise its soundness. The implementation can however use heuristics or have internal resource limitations that could make the model checker fail to report an answer or diverge on some inputs. If the algorithms are sound and the implementation is correct, then one is guaranteed that positive results reported by the tool are true of the given model in the underlying logic.

Verifying that an implementation of a model checker is correct is a very heavy task. These tools are usually very large and have multiple intricate optimizations and implementation details that make them effective on industrial size problems. Moreover most of them use external libraries and other reasoning engines (like SAT or SMT solvers) that are often even more complex than model checkers. There exists formal verification efforts of full model checkers using interactive theorem provers but they remain research prototypes [27, 60]. Another possibility of dealing with the complexity of these tools is to make them produce certificates (much like some theorem provers can export proof objects) to be independently verified by a correct but small external checker. This is discussed in Section 1.7.

Example (Kind 2)**.**

1.4.7 Trusted Components and Libraries

If the model checker makes use of external libraries, their implementation is trusted, and if the model checker uses external tools, they are trusted to be correct.

Risk: 3

Most model checkers use external libraries in their implementation. They can provide useful data-structures (like GMP [32] for arbitrary precision arithmetic or CUDD [59] for decision diagrams) or features (like ∅MQ [37] for parallel and distributed message passing in Kind 2).

Some model checkers even call external tools. These can range from translators (*e.g.* CIL [50] for transformation and normalization of C programs) to reasoning engines like SMT solvers or even other model checkers[2].

Some of these external libraries and tools need not be trusted if the implementation of the model checker itself is not trusted (*e.g.* in the case where the model checker produces certificates, see Section 1.7).

Example (Kind 2)**.** *Kind 2 calls external SMT solvers to decide the validity of first order logical formulas, but also to perform simplification by quantifier elimination.*

It uses external libraries like ∅MQ, Menhir, Camlp4 and the standard library of OCaml. Moreover its backend solvers themselves embark other libraries (e.g. many of them use GMP).

1.4.8 Correct Compilation

The model checker and its components are correctly compiled to executable machine code.

Risk: 0

[2]This is the case for model checkers that extend or combine other model checkers.

Compilers for well established languages (like C, Java, OCaml, *etc.*) are traditionally "*certified by usage*". However, problems can still arise when using high levels of optimization. All compilers in their default setting provide some sort of optimization, and these are never disabled when compiling verification tools. The benefit brought by these optimizations is often non negligible for computationally heavy automated tools like model checkers. Moreover, an incorrect compilation that could trigger in turn a soundness bug in the model checker, while possible, is rather unlikely. The (low) risk of trusting off-the-shelf compilers could nonetheless be reduced by disabling optimizations, and even eliminated by using certified compilers such as CompCert [44].

Example (Kind 2)**.** *Kind 2 relies on the correctness of the standard OCaml compiler for its core but also on the system's default C/C++ compiler for its external dependencies (plus all compilers used to compile the chosen SMT solvers).*

1.4.9 Correct Execution

The machine correctly runs the executable.

Risk: 0

This potential problem is common to all verification techniques supported by software tools. However, a hardware error is even more unlikely to impact the soundness of an analysis than an incorrect optimization of a compiler.

1.4.10 Trusted IO

The parsing and output of the model checker are trusted and interpreted correctly.

Risk: 2

While errors can be present in the parsing and printing facilities of the model checker, they can also come from an incorrect usage of the tool. This is why formal methods tools need to be adequately documented. In particular, the user manual should be extensive in providing details on each operating mode and what it entails for the verification of a program. For instance some model checkers provide facilities to execute or interpret the model / program while performing checks. In this case the absence of errors detected by the model checker should not be interpreted as meaning the program is safe. We briefly review the potential user and IO errors in this section. ★ Maybe this discussion should be postponed to the section on practical sources of errors (if we want to have one).

Input

The first task someone has to do to use a model checker is to give it a model or a program to analyze. The program should be in the fragment that is supported

by the model checker (which should be clear from the documentation). If instead a model is given as a direct input, the user should make sure that the models indeed captures the behavior of what he is trying to verify. For instance, if using scalarset (a symmetrical subrange for which operations do not depend on the ordering of its elements) variables, he should make sure that the feature is appropriate for his verification needs. Also, if using uninterpreted symbols / functions as a way to do under-specification, it should be clear what are the assumptions on these symbols. For example, is their domain always assumed to be infinite? Inhabited? Can an error arise when a domain only has one element?

The next task is to give a formal specification of the system to be analyzed. This specification can be a simple safety property, or a more complex behavioral property. It's important to make sure that the properties to be verified correspond exactly to what the user is trying to accomplish. More often than not, a program is proven to have a property but the property is wrong, trivial or does not capture the behaviors one wants to ensure. Let's take a function that given a list of integers returns a sorted version of this list. It's not sufficient to state that the output list is sorted to verify the functional correctness of this function. We must also ensure that the output list is a permutation of the input list. ★ Put this in Valid Model?

Usage

The use and purpose of a model checking tool should be clear to a user from the start. This is achieved mainly by providing adequate documentation, and extensive examples that cover all the possible uses of the tool. When a user has determined that a particular model checker is the appropriate tool for the verification task he is attempting then he must make sure he is using it in the correct way.

For example, some model checkers for timed systems offer the possibility to switch between discrete and continuous time. It's important to first select the mode that is appropriate for the verification effort and to use the corresponding options of the tool.

Interpretation of Results

Let's take the example of a model checker which has an option to perform a bounded analysis (Kind 2 does this when given the only option `--enable BMC`). It is incorrect to interpret the result saying that no bugs were found as meaning that the system is safe. It is however correct to interpret this result as meaning that the system is safe for executions of at most k steps where k is the bound reached by the analysis.

If a model checker reports that a program is safe, it does not mean that it is completely free from static bugs and runtime errors. It simply mean that there are no executions which violate the *given* property. Even if this last point is clear, this can be true of systems which do nothing. If the user did not specify progress as one

of the properties to be verified then it should not assume that the absence of errors reported by the model checker means that the program is bug free.

If the model is not precise enough, then assumptions are hidden inside the very specification. For instance a set of three instructions can be assumed to be executed atomically, while a violation of the property can happen after the execution of the first instruction, but not the third. So again, one should interpret the results of the analysis according to the model *and* its specification.

Interface Problems

Sometimes model checkers are used as components inside a larger process. When this is the case, it is important to ensure that the interface through which the interaction takes place is well defined.

If a model checker is called as an external tool in a single run that does not require back and forth interaction, then it does not bring new particular soundness concerns. It is still essential to follow a correct usage mode and to make sure that results are automatically parsed and interpreted in a correct way.

On the other hand, if the model checker is called through an interactive session it is vital to ensure the interface is expressive enough and correctly documented. For instance, commands can be sent to the model checker through a textual interface (*e.g.* trough standard input), results should be processed in the correct order and errors or warning should not be ignored. The same things apply if the model checker is used through API calls. Interactive tools can also maintain an internal state between calls and offer facilities to reset them. In this case, it is important to reset the model checker when needed and to understand what is removed from the context. An incorrect usage could lead to unsound results with respect to what is expected.

1.5 Limitation of Analyses

1.6 Practical Sources of Errors

In this section, we review and discuss the possible sources of errors in the practical use of model checkers.

1.6.1 Erroneous Output

The basic task of a model checker is to verify if a given model expressed in a formal specification language verify one or multiple properties. The main answer returned by these tools is by consequence of the form *yes* / *no*. An erroneous output is returning *no* when we should have said *yes* (a *false positive*) or returning *yes* when we should have said *no* (a *false negative*).

False Positives

We have already discussed the impact of such errors in Section 1.1. They can appear because of bugs in the model checker called *completeness bugs* and they have no impact on the soundness of the analyze. These errors are not specific to model checkers and are a recurring concern of automatic formal verification techniques and tools. For instance tools for which termination is not guaranteed (and even some for which it is guaranteed but where execution times can be astronomical) might have an internal bound after which they will report the answer *unknown*.

Tools that perform *abstraction* can also be subject to this kind of errors even if the tool does not contain any bug. The abstraction mechanisms at play can inherently make the tool *incomplete* for certain classes of problems. Usually, these are documented and model checkers can emit warnings when they escape their complete fragment.

Other verification methods like testing or simulation are generally not prone to false positive errors due to their animating nature (they execute a program or a system).

False Negatives

We have previously referred to false negative errors in Section 1.1 as *soundness bugs*. These bugs can manifest for a very large number of reasons. We give a few examples below in no particular order.

- The algorithm in itself may not be sound
- The algorithm makes assumptions which are not correct
- There can be a bug in the implementation
- Errors discovered during the search can be ignored
- Errors are incorrectly reported
- Undocumented limits of the tool make certain analyses wrong
- One of the component of the model checker (like an external tool) is unsound itself
- One component is incomplete but the soundness of the analysis depends on the correctness of this component
- Interactions between different parts of the tool are buggy
- Translation steps between different formalisms are unsound

Other

1.6.2 Potential User Errors

1.6.3 Model Construction

1.7 Techniques for Verifying Model Checkers

Establishing the correctness of tools like model checkers is a difficult task, mostly due to the sheer size and complexity of these programs. In any case, peer review and testing should be part of the quality process used in the development of a model checker.

One way to ensure that a model checker (or any other formal tool) returns sound results is to *certify* it. This consists in providing strong arguments which demonstrate that the output of a specific tool is correct and can be trusted for assurance cases. For this, it is sometimes possible to show with mathematical arguments (proofs) that the tool is correct by construction. This means that it will only return correct results. For some other tools, it is possible to construct versions of them that augment their results with *evidence*. In this case the evidence needs to be evaluated each time the tool is executed so as to ensure the result is correct.

1.7.1 Peer Review and Traditional Testing

Traditional testing, be it unit testing and/or integration testing, is a relatively easy way to eliminate simple and trivial implementation errors during the development process. Black box testing, when performed as a traditional verification technique, is more difficult. Indeed, it is very complicated to construct input problems for the model checker that will stress all of its components. Moreover, constructing a test that evaluates a particular feature of the model checker often demands to have an extensive knowledge of the internals of the tool.

Constructing relevant tests is still possible and vital, especially ones that are unsafe (with respect to their properties). They are a way to somewhat safeguard the implementation against the introduction of *soundness* bugs.

1.7.2 Formally Verified Model Checkers

Two different lines of work coexist for the certification of verification tools. One approach focuses on verifying the program (here the model checker) correct once and for all. In this category, there exists several different approaches for proving a program correct. For some programming languages, it is possible to prove the code directly (*e.g.* using ESC Java, Frama-C, VCC, F* etc.), though this is a very tough job because such programs are often very complex, the proofs rapidly become convoluted and are unlikely to be automated. One advantage is that the performances

of such programs can be close to the ones of their non certified counterparts. One example of this kind of certification effort is the modern SAT solver `versat` which was developed and verified using the programming language Guru [52]. We are however not aware of similar results for model checkers.

Another possibility is to prove the algorithm correct in a descriptive language adapted to verification (*e.g.* interactive proof assistants like Coq, PVS or Isabelle) and obtain an executable program through a refinement process or a code extraction mechanism. In the recent years, certified software of this category have gained interest. Worth mentioning is the C compiler CompCert [44] or the operating system micro-kernel seL4 [42]. CompCert is written entirely in Coq and uses external oracles in some of the compilation passes. These oracles provide solutions (*e.g.* a coloring of a graph) that can be verified by a certified checker.

Although the first formal verification of a model checker in Coq for the modal μ-calculus [60] goes back to 1998, only recently have *certified verification tools* started to emerge. Blazy *et al.* have verified a static analyzer for C programs [13] to be used inside CompCert. Although this static analyzer is not on par with the performances of commercial tools, it is sufficient to enable safely some of the optimizations of a compiler. The most relevant works concerning model checking are probably [2] and [27].

In [2], Amjad 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.* [27] 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.

Clearly, the advantage here is that the model checker is verified correct once and for all. Usually, a trade-off exists between an efficient program from a precise algorithm working on complex data structures, and a less concrete program from an algorithm where some data structures and operations are abstracted.

1.7.3 Proof-Emitting Model Checkers

The other approach consists in producing, in addition to the final answer, a proof of the result also called a *certificate*. The first application of this technique to model checking is the work described in [49]. However, this is a very heavy task since model checkers are very optimized programs with a large number of components. In the second approach, certificates have to be checked after each run. Its advantage is to be far less intrusive, the only necessity is to instrument an already existing model checker. However, this approach is only applicable if certificates or proof objects are small and / or simple enough to be checked in a reasonable time after the fact.

An approach for the certification of SAT and SMT solvers is the work by Keller

et al. [3,41]. Their idea consists in having the solver produce a detailed certificate in which each rule is read and verified by the composition of several small certified (in Coq) checkers. This approach also allows to import proof terms from SMT solvers inside Coq [4].

In the world of SMT solvers, CVC4 [8] is also able to produce proof trees in a variant of the Edinburgh Logical Framework (LF) [34] extended with side conditions called LFSC [61]. The computational power brought by the use of side conditions allows to verify efficiently very large proofs [62].

One recent of such application to model checking is Slab [26] which produces certificates in the form of inductive verification diagrams to be checked by SMT solvers. While less intrusive, this approach is only applicable if the certificates or proof objects are small and simple enough to be checked in a reasonable time after the fact.

Certifying model checker trust proposition

1. **Trusted Logic** The logics used by the model checker are trusted.
2. **Valid Model** The model and properties accurately depict the system under scrutiny in the semantic given by the logic of the model checker.
3. **Correct Certificate** The certificate produced by the model checker is sufficient to convince that the properties are true of the system under scrutiny.
4. **Correct Certificate Checker** The program that checks the certificates is sound and correctly implemented.
5. **Trusted IO** The parsing and output of the *certificate* checker are trusted.

In the trust proposition for a *certifying* model checker, we are not interested in the correctness of the tool itself (algorithms, implementation, compilation, IO) but we rely on its ability to produce a correct *certificate*. The trust is thus shifted from an extremely complex piece of software — the model checker — to a smaller and simpler one, the *certificate checker*.

2 Tool Study: The Kind 2 Model Checker

Let a program, for now be in the form of a transition system $\mathcal{S} = (I, T, \mathcal{Q})$ where I and T are quantifier-free first-order formulas and $\mathcal{Q}$ is a set of state variables. Let a property $P(X)$ where P is a quantifier-free first-order formula with free variables $X \subseteq Q$. We want to construct a certificate $C(\mathcal{S}, P)$, valid if P is an invariant of $\mathcal{S}$.

2.1 k-Induction

2.1.1 Certificates

For k-induction, properties are established by iteratively increasing k, until it becomes inductive with respect to k steps in the program. The k-induction scheme consists in proving the two sequents:

$$I(X_0) \wedge T(X_0, X_1) \wedge \ldots \wedge T(X_{k-2}, X_{k-1}) \models P(X_0) \wedge \ldots \wedge P(X_{k-1}) \quad (2.1)$$
base

$$T(X_0, X_1) \wedge \ldots \wedge T(X_{k-1}, X_k) \wedge P(X_0) \wedge \ldots \wedge P(X_{k-1}) \models P(X_k). \quad (2.2)$$
step

However, (I think) proving these sequents is not enough to guarantee that P is an invariant of the system. In particular when the program terminates or when the transition relation can become inconsistent, the base case can be rendered trivially valid even when a bug exists.

Example. Let's take the system (I, T) over the state variables $\{x\}$ where the initial states are described by the formula $I(x) \equiv x = 0$, and the transition relation is $T(x, x') \equiv x \neq 1 \wedge x' = 1$. In other words, the program changes the value of x to 1, but only once. Now we want to verify the property $P(x) \equiv x \neq 1$. It clear that P can be falsified after just one step. However, the following base sequent is valid:

$$I(x_0) \wedge T(x_0, x_1) \wedge T(x_1, x_2) \models P(x_0) \wedge P(x_1) \wedge P(x_2)$$

In particular $I(x_0) \wedge T(x_0, x_1)$ implies $x_0 = 0$ and $x_1 = 1$ while $T(x_1, x_2)$ requires $x_1 \neq 1$. Because the left hand side is inconsistent we obviously can't find a model in which one of the P is falsified. P is not an invariant because $I(x_0) \wedge T(x_0, x_1) \not\models P(x_1)$. For the same reason, the sequent step is valid because two applications of the transition relation yields an unsatisfiable context.

What we should verify instead to ensure k-induction is correct is the sequents:

$$I(X_0) \models P(X_0) \qquad \text{(base-0)}$$

$$I(X_0) \wedge T(X_0, X_1) \models P(X_1) \qquad \text{(base-1)}$$

$$\ldots$$

$$I(X_0) \wedge T(X_0, X_1) \wedge \ldots \wedge T(X_{k-2}, X_{k-1}) \models P(X_{k-1}) \qquad \text{(base-}k-1\text{)}$$

$$T(X_0, X_1) \wedge \ldots \wedge T(X_{k-1}, X_k) \wedge P(X_0) \wedge \ldots \wedge P(X_{k-1}) \models P(X_k). \qquad \text{(step)}$$

To keep the presentation more concise we can express it with two sequents as such:

$$\begin{array}{l} \quad I(X_0) \wedge \neg P(X_0) \\ \vee\ (I(X_0) \wedge T(X_0, X_1) \wedge \neg P(X_1)) \\ \vee\ \ldots \\ \vee\ (I(X_0) \wedge T(X_0, X_1) \wedge \ldots \wedge T(X_{k-2}, X_{k-1}) \wedge \neg P(X_{k-1})) \end{array} \models \bot \qquad (2.1)$$

base

$$T(X_0, X_1) \wedge \ldots \wedge T(X_{k-1}, X_k) \wedge P(X_0) \wedge \ldots \wedge P(X_{k-1}) \models P(X_k). \qquad (2.2)$$

step

The rules for k-induction are given in Figure 2.1. They use a function *unroll* which returns a formula characterizing an unrolling of k steps for the transition relation, each step satisfying the formula φ. In particular, if $\varphi = \top$ then no assumption is made on the unrolling. The function *unroll* is defined as follows:

$$unroll(k, T, \varphi) \triangleq \begin{cases} k = 0 & : \varphi(X_0) \\ k = \mathtt{S}\ m & : unroll(m, T, \varphi) \wedge T(X_m, X_k) \wedge \varphi(X_k) \end{cases}$$

2.1.2 From k-induction back to induction

If we trust the rules of k-induction shown in Figure 2.1 then we don't need more to check the certificates. However one might want to only keep *simple induction* in its trust base. We show here two possible ways to do this and discuss the feasibility and drawbacks of each.

History variables

k-induction intrinsically establishes properties that are preserved by chains of length k. To reduce a proof carried by k-induction to a simple induction proof it is necessary to have a way to represent these chains of length k in the system. One way to do this is to introduce additional auxiliary variables that act as history variables to store (up to) k previous states.

In this section, we define an auxiliary system $\hat{\mathcal{S}} = (\hat{I}, \hat{T}, \hat{Q})$ together with an auxiliary property $\hat{P}$ such that P is *k-inductive* in S *iff* $\hat{P}$ is *inductive* $\hat{\mathcal{S}}$.

$$\textsc{K-Ind}\ \frac{\mathcal{S} = (I, T, Q) \qquad base_k(I, T, P) \qquad step_k(T, P)}{C(\mathcal{S}, P)}$$

$$\textsc{Base-0}\ \frac{I(X) \models P(X)}{base_0(I, T, P)}$$

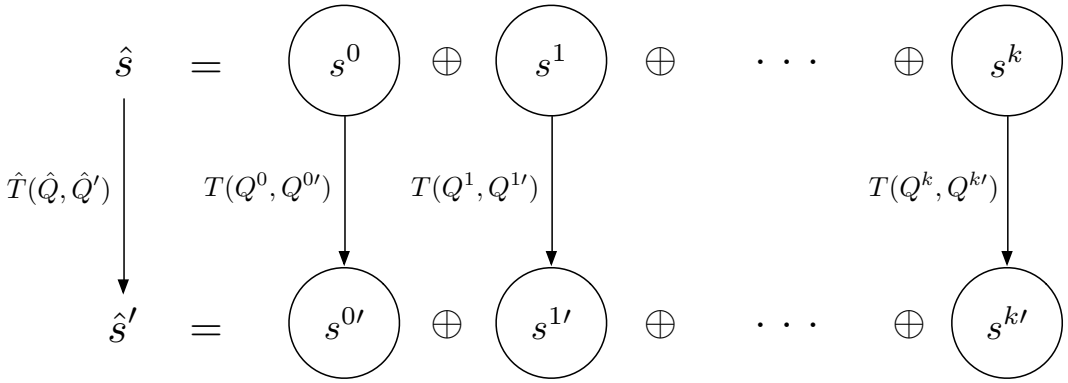
$$\textsc{Base-k}\ \frac{I(X_0) \wedge unroll(k, T, \top) \models P(X_k) \qquad base_{k-1}(I, T, P)}{base_k(I, T, P)}$$

$$\textsc{Step-k}\ \frac{unroll(k-1, T, P) \wedge T(X_{k-1}, X_k) \models P(X_k)}{step_k(T, P)}$$

Figure 2.1: Rules for k-induction

- $\hat{Q}$ is a set containing k distinct "copies" of the variables of Q, $\hat{Q} = Q^0 \uplus Q^1 \uplus \ldots \uplus Q^k$ (Q^i is a renaming of the variables of Q).
- $\hat{I}(\hat{Q}) \equiv I(Q^0) \vee (I(Q^0) \wedge T(Q^0, Q^1)) \vee \ldots \vee (I(Q^0) \wedge T(Q^0, Q^1) \wedge \ldots \wedge T(Q^{k-1}, Q^k))$
- $\hat{T}(\hat{Q}, \hat{Q}') \equiv T(Q^0, Q^{0\prime}) \wedge \ldots \wedge T(Q^k, Q^{k\prime})$
- $\hat{P}(\hat{Q}) \equiv P(Q^0) \wedge \ldots \wedge P(Q^k)$.

One way to see $\hat{\mathcal{S}}$ is to consider its states as the partition of $k+1$ states of systems $\mathcal{S}$, each of them as one step intervals.



- $\hat{P}$ is true in $\hat{\mathcal{S}}$ *iff* P is true in $\mathcal{S}$.

Suppose that the property P is violated in the system $\mathcal{S}$, then there exists a sequence of length m such that $I(X_0) \wedge T(X_0, X_1) \wedge \ldots \wedge T(X_{m-1}, X_m) \wedge \neg P(X_m)$ is *satisfiable*. Because $\hat{\mathcal{S}}$ intuitively contains the behaviors of $\mathcal{S}$, the same violation exists in $\hat{\mathcal{S}}$ by taking $Q^0 = X$.

Suppose now that the property $\hat{P}$ is violated in the system $\hat{\mathcal{S}}$, then there exists a sequence of length m such that $\hat{I}(\hat{Q}_0) \wedge \hat{T}(\hat{Q}_0, \hat{Q}_1) \wedge \ldots \wedge \hat{T}(\hat{Q}_{m-1}, \hat{Q}_m) \wedge \neg \hat{P}(\hat{Q}_m)$ is *satisfiable*. Let's call its model $\mathcal{M}$. $\neg \hat{P}(\hat{Q}_m)$ is a disjunction so at least one of its disjunct is satisfied by $\mathcal{M}$, $\neg P(Q^p_m)$. We know that $\hat{Q}$ is partitioned in k subsets, and Q^p is one of its components.

$$\begin{array}{rll} & (I(Q^0_0) \wedge T(Q^0_0, Q^1_0) \wedge \ldots \wedge T(Q^{p-1}_0, Q^p_0)) \wedge & \leftarrow \text{part of } \hat{I} \\ \mathcal{M} \models & T(Q^p_0, Q^p_1) \wedge \ldots \wedge T(Q^p_{m-1}, Q^p_m) \wedge & \leftarrow \text{part of } \hat{T} \\ & \neg P(Q^p_m) & \leftarrow \text{part of } \hat{P} \end{array}$$

So there exists a sequence of $m + p$ transitions from an initial state that violate the property P in $\mathcal{S}$.

- P is *k-inductive* in S *iff* $\hat{P}$ is *inductive* $\hat{\mathcal{S}}$.

This follows directly from the definition of $\hat{\mathcal{S}}$ and $\hat{P}$, using the same k for which P is k-inductive.

Discussion. While it is nice to be able to reduce a proof by k-induction to expose an inductive invariant ($\hat{P}$), we need to modify the original system $\mathcal{S}$. This allows to use this inductive invariant as a certificate, but it requires an extra justification for the transformation of $\mathcal{S}$ to $\hat{\mathcal{S}}$. Moreover, a checker for this certificate (extracted this way) would have to check the inductiveness property for $\hat{P}$ which contains redundant information *w.r.t.* the hypothesis (the part from 0 to $k-1$). All in all, this technique demands more from a certificate and its verification than what would a proof by k-induction.

Inductive Strengthening?

Let's concentrate on the k-inductive step (at $k+1$) and assume that it is valid:

$$P(X_0) \wedge T(X_0, X_1) \wedge P(X_1) \wedge \ldots \wedge P(X_k) \wedge T(X_k, X_{k+1}) \models P(X_{k+1})$$

Let $B(X_k) \equiv \exists X_0, \ldots, X_{k-1}.\ P(X_0) \wedge T(X_0, X_1) \wedge P(X_1) \wedge \ldots \wedge T(X_{k-1}, X_k) \wedge P(X_k)$, then we can rewrite the step as:

$$B(X_k) \wedge T(X_k, X_{k+1}) \models P(X_{k+1})$$

we can also remark that $B(X_k) \wedge T(X_k, X_{k+1}) \wedge P(X_{k+1}) \models B(X_{k+1})$ so

$$B(X_k) \wedge T(X_k, X_{k+1}) \models B(X_{k+1})$$

We know that

$$P(X_0) \wedge T(X_0, X_1) \wedge \ldots \wedge T(X_{k-1}, X_k) \wedge P(X_k) \wedge T(X_k, X_{k+1}) \wedge \neg P(X_{k+1}) \models \bot$$

so

$$B(X_k) \wedge T(X_k, X_{k+1}) \wedge \neg P(X_{k+1}) \models \bot$$

We want to show that

$$B(X_k) \wedge T(X_k, X_{k+1}) \wedge \neg B(X_{k+1}) \models \bot \tag{2.3}$$

Let's have a closer look at the definition of B:

$$B(X_k) \equiv \exists X_0, \ldots, X_{k-1}.\ \underbrace{P(X_0) \wedge T(X_0, X_1) \wedge P(X_1) \wedge \ldots \wedge P(X_{k-1})}_{U(X_0,\ldots,X_{k-1})} \wedge T(X_{k-1}, X_k) \wedge P(X_k)$$

We note U the part that mentions the assumption on the $k-1$ first steps.

Now we can rewrite (2.3) with U:

$$\begin{array}{ll} \exists & X_0, \ldots, X_{k-1}. \\ & U(X_0, \ldots, X_{k-1}) \wedge T(X_{k-1}, X_k) \wedge P(X_k) \wedge T(X_k, X_{k+1}) \wedge \\ & \neg(\exists Y_0, \ldots, Y_{k-1}.\ U(Y_0, \ldots, Y_{k-1}) \wedge T(Y_{k-1}, X_{k+1}) \wedge P(X_{k+1})) \\ \models & \bot \end{array}$$

In the next step we skolemize the existential prenex variables $X_0, \ldots, X_{k-1}$. For simplicity we give the new Skolem constants the same name as the existential variables. At the same time we instantiate the variables $Y_0, \ldots, Y_{k-1}$ with the following substitution:

$$\begin{cases} Y_0 \mapsto X_1 \\ Y_1 \mapsto X_2 \\ \vdots \\ Y_{k-2} \mapsto X_{k-1} \\ Y_{k-1} \mapsto X_k \end{cases}$$

We now need to show

$$\begin{array}{ll} U(X_0, \ldots, X_{k-1}) \wedge T(X_{k-1}, X_k) \wedge P(X_k) \wedge T(X_k, X_{k+1}) \wedge & \\ \neg(U(X_1, \ldots, X_k) \wedge T(X_k, X_{k+1}) \wedge P(X_{k+1})) & \models \bot \end{array}$$

Equivalently, we can show both (2.4) and (2.5)

$$\begin{array}{ll} U(X_0, \ldots, X_{k-1}) \wedge T(X_{k-1}, X_k) \wedge P(X_k) \wedge T(X_k, X_{k+1}) \wedge & \\ \neg(U(X_1, \ldots, X_k) \wedge T(X_k, X_{k+1})) & \models \bot \end{array} \tag{2.4}$$

$$U(X_0, \ldots, X_{k-1}) \wedge T(X_{k-1}, X_k) \wedge P(X_k) \wedge T(X_k, X_{k+1}) \wedge \neg P(X_{k+1}) \models \bot \tag{2.5}$$

The second one, (2.5) holds trivially because this is exactly the $k+1$-inductive case. By expanding the definition of U in (2.4) we get

$$P(X_0) \wedge T(X_0, X_1) \wedge P(X_1) \wedge \ldots \wedge P(X_{k-1}) \wedge T(X_{k-1}, X_k) \wedge P(X_k) \wedge T(X_k, X_{k+1}) \wedge \\ \neg(P(X_1) \wedge T(X_1, X_2) \wedge P(X_2) \wedge \ldots \wedge P(X_k) \wedge T(X_k, X_{k+1})) \models \bot$$

Let's introduce $F \equiv P(X_1) \wedge T(X_1, X_2) \wedge P(X_2) \wedge \ldots \wedge P(X_k) \wedge T(X_k, X_{k+1})$, we are left with

$$P(X_0) \wedge T(X_0, X_1) \wedge F \wedge \neg F \models \bot$$

which trivially holds. □

In addition, we also assume the base case of k-induction to hold which means that in particular $I(X) \models P(X)$ from which we can deduce that $I(X) \models B(X)$. All this makes B an *inductive* invariant of the system.

The problem now is that B contains (existential) quantifiers. B appears on the right hand side of the sequents which we must verify, so this demands to use an SMT solver that is able to handle *universal* quantification. Even though this is feasible, B will contain potentially many unrollings, be very large, and contain a prohibitive number of quantified variables. One solution is to use quantifier elimination techniques to find a formula equisatisfiable with B that does not contain quantifiers. These techniques normally require very expensive computations even when the number of quantified variables is small, so they are inadequate here. Instead of exact quantifier elimination (which would have to be run repeatedly for every variable to eliminate) we can use an approximate quantifier elimination that returns an under-approximation of B. The problem now is that we have no guarantee that this approximation is inductive.

2.2 Property Directed Reachability

The algorithm that is implemented in Kind 2 for property directed reachability (or PDR) is based on IC3 (for Incremental Construction of Inductive Clauses for Indubitable Correctness). In other words, it constructs a set of inductive clauses which is an inductive invariant of the system. When a property P is proven, it is easy to extract a witness ϕ in the form of an inductive invariant subsuming P.

All that is necessary is for it to verify the two following conditions:

$$I(X) \models \phi(X) \qquad (2.1)$$
initialization

$$\phi(X) \wedge T(X, X') \models \phi(X'). \qquad (2.2)$$
preservation

The base case (2.1) says that the invariant ϕ must be true in the initial states of the system and the inductive case (2.2) says that the invariant must be preserved by the transition relation. If additionally, we have

$$\phi(X) \models P(X) \qquad \text{property} \tag{2.3}$$

then the property P is an invariant (not necessarily inductive) of the system.

The rules to express the inductive principle are easier, there is only one.

$$\textsc{Ind}\ \frac{\mathcal{S} = (I, T, Q) \qquad I(X) \models \phi(X) \qquad \phi(X) \wedge T(X, X') \models \phi(X) \qquad \phi(X) \models P(X)}{C(\mathcal{S}, P)}$$

Figure 2.2: Induction principle for invariance

2.3 Invariants Generation

An important feature of Kind 2 is its ability to generate invariants on the fly. This is done by a parallel k-induction process that, given a set of candidates, returns a subset of invariant (by k-induction) formulas. The k for which these invariants are verified is independent of the k used for the main k-induction loop, or the number of frames computed by PDR.

A false invariant can render the whole model checking process unsound so it is important to ensure that they are correct. The question here is two-fold: how do we integrate invariants and their proof in the certificate, and how does it affect the certification effort?

Not all discovered invariants are useful for the proof of the original properties. By consequence, it is important to first identify which invariant is used by each part of the model checker and for which purpose:

- Proof of base case?
- Proof of preservation by step?
- Pruning in PDR?

Some bookkeeping might be necessary to maintain dependencies between the invariants. For instance, the proof of an invariant ϕ_1 might necessitate to know that a previous ϕ_0 is invariant. If ϕ_1 is used for the proof of property P (let's say by

k-induction step preservation), then both ϕ_0 and ϕ_1 with their proofs are needed in the certificate.

Other times, not an invariant of the global system but an *instance* of an invariant of a subnode might be necessary. Here, we would need to decide if we want to re-do the proof of the instance of the invariant (probably not because the k may become arbitrarily large – unless it can be statically computed), or if we want to use the modularity allowed by Kind 2 and keep the proofs of the local invariants at the subnode level.

Update: Some invariants which are k-inductive for a subnode, might not be k-inductive for its instance in a node call. It is particularly true for the case of the Scade activation condition operators (*condacts*):

```
z = condact(b, n(x), i)
```

is equivalent to

```
z = i -> if b then current n(x when b) else pre z
```

so a property ϕ for the node `n` is lifted as `b` $\implies \phi$ for the condact. This need not be inductive! What we propose here is to keep the input problem in its modular version, *i.e.* with the different nodes expressed as predicates in FOL, and to state the invariants of each subnode.

In the spirit of re-running Kind 2 in the hope of reducing the proof, maybe the invariant can be inserted in the set of properties. Let's assume that P is k_1-inductive and an invariant ϕ is k_2-inductive, then is $P \wedge \phi$ $max(k_1, k_2)$-inductive?

Yes.

These are questions that will need to be addressed when invariants are necessary for the verification. Because of their usefulness this might be rather sooner than later.

2.4 Format of Certificates

Whether we construct a certificate which encodes the k-induction rule (for runs of k-induction in Kind 2) or a certificate which consists of an inductive invariant (for runs of PDR in Kind 2), some considerations apply regardless.

We consider two possibilities for the format of the certificate returned by Kind 2. In both cases we would like that the inputs do not appear directly in the certificate but be instead gathered from the same source as what Kind 2 uses. This is to prevent errors in case the input system/property is modified by the model checker during its search.

Our plan is to have a final certificate expressed as a derivation in the LFSC format. For this we can devise a process in which an *intermediate* certificate is produced,

whose validity is later verified by an external tool which in turn produces a proof (in LFSC) that the intermediate certificate is correct. The alternative is to produce directly a proof in LFSC that contains the proof of k-induction (or induction) as well as the proof of the leaves of the induction tree (given by an SMT Solver, *i.e.* CVC4).

2.4.1 Intermediate Certificate

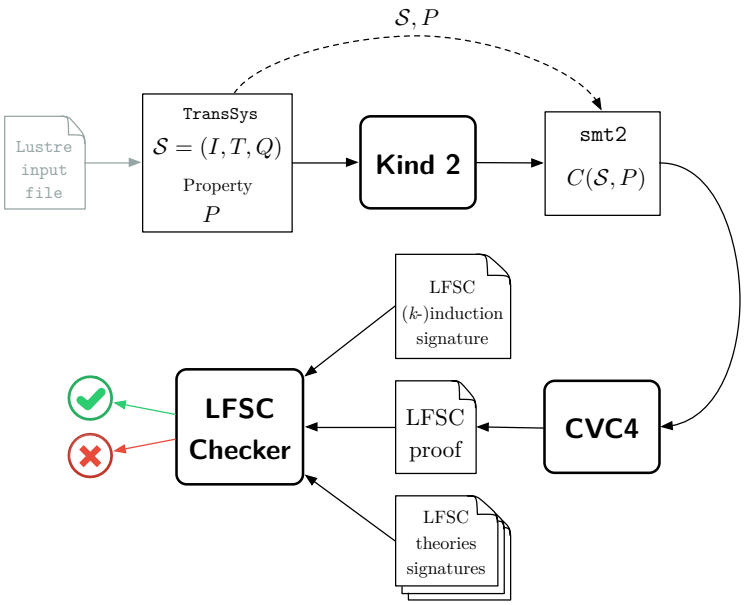


Figure 2.3: Intermediate SMT certificate

From one run of Kind 2, we can extract a certificate that corresponds to the sequents of the k-induction (or induction) rule. These are expressed in first-order logic and can be produced as SMT-LIB 2 compliant files. This first certificate can then be verified by CVC4 in proof producing mode (*cf.* Figure 2.3). This will generate a proof in LFSC, which can be augmented by a proof of (k-)inductiveness. The proof, together with the corresponding rules, can then be machine checked by the LFSC checker.

2.4.2 LFSC Certificate

In this approach, the certificate contains both the proof tree of the k-induction (or induction) reasoning as well as the proofs of the SMT calls given by CVC4. For this, we probably want to run Kind 2 twice because the proof producing version of CVC4 is a lot slower. The first run of Kind 2 will perform the search and identify a

k for which the property is k-inductive, or construct an inductive invariant ϕ. An optional second run will attempt to reduce k (unlikely) or generalize ϕ as much as possible. Finally a last run will use the proof producing version of CVC4 to generate LFSC proofs corresponding to Kind 2's SMT calls. In the case of k-induction, only the last step will be replayed, and in the case of PDR a single induction will be performed (because the inductive invariant is already identified).

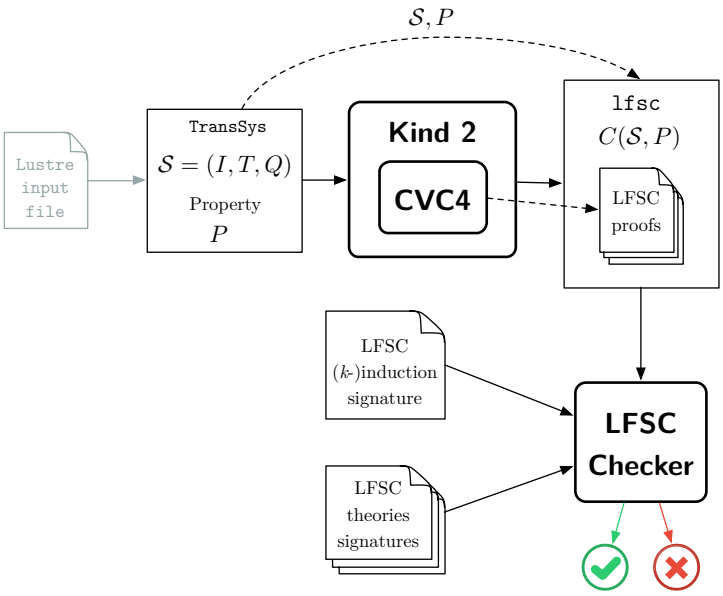


Figure 2.4: LFSC certificate

2.4.3 Proposition

What we would like is a common certificate regardless of the engine used by Kind 2. So they must be suitable to synthesize the work performed by k-induction and the work performed by PDR.

For k-induction, the simplest form of certificate consists of the single natural number k for which the property is k-inductive. With only this knowledge and the input system, it is possible to check the rules of k-induction.

For PDR, an inductive invariant can be extracted. Using a second pass which acts as a safeguard to ensure of its inductiveness, we can recover the useful part of the inductive invariant with an unsat-core.

★ Not sure. Suppose the inductive invariant is a conjunction of clauses $\phi \equiv C_1 \wedge \ldots \wedge C_n$ then the check for the preservation would be

$$\begin{aligned} C_1 \wedge \ldots C_n \wedge T \wedge \neg(C'_1 \wedge \ldots \wedge C'_n) &\models \bot \\ \textit{i.e. } C_1 \wedge \ldots C_n \wedge T \wedge (\neg C'_1 \vee \ldots \vee \neg C'_n) &\models \bot \end{aligned}$$

Getting the unsat core from this query amounts to doing the union of the unsat cores for following n SMT queries

$$\begin{aligned} C_1 \wedge \ldots C_n \wedge T \wedge \neg C'_1 &\models \bot \\ &\vdots \\ C_1 \wedge \ldots C_n \wedge T \wedge \neg C'_n &\models \bot \end{aligned}$$

Most SMT solvers return unsat cores in terms of a subset of the input clauses. So if T is not inconsistent (and assuming that ϕ is indeed inductive), atoms of C'_i will appear in the unsat core for its respective query. We cannot just drop these unused atoms form C_i because we want to keep both C_i and C'_i to be identical modulo renaming. In other words, simplifying ϕ can make it not inductive. In conclusion, it is unlikely that we will be able to simplify (what we want here is an over-approximation or a generalization) the inductive invariant constructed by PDR. ★ maybe use some sort of fixpoint computation.

In fact, ϕ is k-inductive for $k = 1$. A possible certificate for the problem of verifying the property P of a system $\mathcal{S}$, that is appropriate for both k-induction and PDR, is a tuple (k, ϕ) such that ϕ is k-inductive in $\mathcal{S}$ and that $\phi \models P$. **Remark.** At the end of k-induction, we can construct a certificate that satisfy this requirement if we take $\phi \equiv P$, and as well at the end of PDR if we take $k = 1$.

Now, we need to find a way to incorporate invariants (possibly with their respective certificate) into this format. Because the invariants discovered (and used) are not necessarily k-inductive (see Section 2.3), we cannot simply add them to P and try to find an appropriate k. Instead we must provide the predicates for each subnodes together with their invariants. For the same reason, these might not be k-inductive because of invariants of other node calls that appear inside that node. What we really need is a recursive certificate:

$$C := (k, \phi, [n \mapsto (Q, C), \ldots, n \mapsto (Q, C)])$$

where k is a natural number and ϕ is a k-inductive property when the following invariants are assumed. The map that follows associates each element in a subset of the nodes (n is the name of a predicate which encodes a node) to an invariant of this node with a certificate. When the list in the last component is empty, this is a certificate for k-induction as described above.

To check this certificate, we do it recursively assuming the invariants given for each subnode at each level. For this we need to trust the k-induction rule that essentially states that if a formula Q is k-inductive, then it is an invariant of this node. We also want to express and trust what it means to be an invariant.

$$\textsc{Inv}\ \frac{C \text{ is a valid certificate for the invariant } Q \text{ of node } n}{\forall \bar{x}.\ \forall i \in \mathbf{N}.\ \ n(\bar{x}(i), \bar{x}(i+1)) \ \models\ Q(\bar{x}(i), \bar{x}(i+1))}$$

★ Remark. This looks more like a function summary, is it correct to have this if n is not abstract? **No**! cannot quantify over $\bar{x}$ which are not input variables. This can be inconsistent if instantiated with variables that characterize an unreachable state.

While checking the certificate, we can add the quantified formulas that expresses that Q is an invariant as an axiom. But we can also add all useful instances for our proof, *i.e.* up to the k given by the parent certificate. (What justification for the axioms introduced by the instances?)

2.5 Questions on LFSC

- Can proofs be modular?
- Can we prove lemmas or subgoals?
- Can we reuse proof terms in other proofs
- Does CVC4 generate correct proofs?

If we take the following problem expressed in purely propositional form:

```
(set-logic QF_SAT)

(declare-fun A () Bool)
(declare-fun B () Bool)
(declare-fun C () Bool)

(assert (or A B C))
(assert (not A))
(assert (not B))
(assert (not C))

(check-sat)
```

And we run CVC4 (compiled with proof generation support) with
`cvc4 --lang smt2 --proof --check-proofs --dump-proofs file.smt`
we get the following proof on the standard output:

```
(check
 ;; Declarations
(% A (term Bool)
(% B (term Bool)
(% C (term Bool)
(% A0 (th_holds true)
(% A1 (th_holds (not (p_app C)))
(% A2 (th_holds (not (p_app B)))
(% A3 (th_holds (not (p_app A)))
(% A4 (th_holds (or (p_app A) (or (p_app B) (p_app C) )))
```

```
 ;; Proof of empty clause follows
(: (holds cln)
 ;; Clauses
(satlem _ _ (ast _ _ _ atom2 (\ lit5 (clausify_false trust))) (\ pb3
(satlem _ _ (ast _ _ _ atom3 (\ lit7 (clausify_false trust))) (\ pb4
(satlem _ _ (asf _ _ _ atom4 (\ lit8
            (asf _ _ _ atom3 (\ lit6
            (asf _ _ _ atom2 (\ lit4 (clausify_false trust)))))))) (\ pb2
(satlem _ _ (ast _ _ _ atom4 (\ lit9 (clausify_false trust))) (\ pb5
 ;; Theory Lemmas
(satlem_simplify _ _ _ (R _ _ (R _ _ pb2 pb4 var3) pb3 var2) (\cl6
(satlem_simplify _ _ _ (Q _ _ pb5 cl6 var4) (\empty empty))))))))))))))))))))))
```

In fact the following proof would type check:

```
(check (: (holds cln) (clausify_false trust)))
```

because `trust` introduces a trivial inconsistency, being defined as

```
(declare trust (th_holds false))
```

2.6 Examples

★ write it

2.7 Translations

2.7.1 Lustre Input

The input program and its properties are expressed in the synchronous language Lustre. However, the certificate we produce considers the program expressed in first order logic. In Kind 2, the Lustre file is first parsed and a number of optimizations and simplifications (slicing, propagation, *etc.*) are performed on this input. Only after is it translated to an internal representation which is essentially a transition relation in FOL.

The certification process needs to argue that these optimizations and translations are correct. This can be done by proving once that this translator is correct. For instance, in Coq, this would require to formalize the semantic of Lustre and to show that it is preserved by all simplifications and translations. However, we want to certify the actual program Kind 2. With Coq we could extract some purely functional code to replace the current frontend of Kind 2. This will likely yield not very efficient code, but more importantly it would turn maintaining and improving this frontend into a convoluted process.

Another possibility which we envision is to have this translation phase generate certificates of its own. Most likely, they will relate the semantic of the actual Lustre

program with the semantic of its translation. The kind and format of these certificates still need to be determined. We should however keep in mind that we want to check them *automatically*.

2.7.2 LFSC format

Once the certificate is output in its intermediate format, CVC4 verifies its validity and generates in turn a certificate in LFSC format. The problem is that this certificate contains the input system expressed in a different format. Optimizations in CVC4 can also make it difficult to recognize this input problem. What we suggest here is to have an external program whose task is can be one of these two:

- recover the original input system and properties (or the intermediate certificate) from the LFSC file
- translates the original input problem in LFSC syntax and check that it appears correctly in the LFSC file.

2.8 Actions

This is a list of tasks that will need to be undertaken so as to have a first prototype of a certifying version of Kind 2. The order in which they are given is not necessarily final but some prerequisite clearly exists and I believe some issues would be better addressed sooner.

1. Identify the kind of certificate for each engine. Do we really want only one kind of certificate (inductive invariants)?
2. Decide which granularity we need for the certificate. (Several layers?)
3. Figure out what is the better way to reduce and/or simplify the proofs in Kind 2. Do we need several passes?
4. Instrument Kind 2 to do the necessary bookkeeping for proofs.
5. Instrument Kind 2 to produce the certificate in the desired format. This might be more or less complicated depending on the granularity.
6. Evaluate the whole chain on a supported fragment (for now SAT + EUF).

More ...

Bibliography

[1] P. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund. Designing safe, reliable systems using Scade. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods*, volume 4313 of *Lecture Notes in Computer Science*, pages 115–129. Springer Berlin Heidelberg, 2006.

[2] H. Amjad. Programming a symbolic model checker in a fully expansive theorem prover. In *Theorem proving in higher order logics*, pages 171–187. Springer, 2003.

[3] 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'11: International Workshop on Proof-Search in Axiomatic Theories and Type Theories*, 2011.

[4] 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 *Certified Programs and Proofs*, pages 135–150. Springer, 2011.

[5] 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 K. Etessami and S. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 185–198. Springer Berlin Heidelberg, 2005.

[6] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Communications of the ACM*, 54(7):68–76, 2011.

[7] T. Ball and S. K. Rajamani. The slam toolkit. In *Computer aided verification*, pages 260–264. Springer, 2001.

[8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *CAV*, pages 171–177. Springer, 2011.

[9] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. *UPPAAL—a tool suite for automatic verification of real-time systems*. Springer, 1996.

[10] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[11] S. Bhattacharyya, S. Miller, J. Yang, S. Smolka, B. Meng, C. Sticksel, and C. Tinelli. Verification of quasi-synchronous systems with uppaal. In *Digital Avionics Systems Conference (DASC), 2014 IEEE/AIAA 33rd*, pages 8A4–1. IEEE, 2014.

[12] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[13] S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a c value analysis based on abstract interpretation. In *SAS*, volume 7935, pages 324–344. Springer, 2013.

[14] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels. Model checking flight control systems: The Airbus experience. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 18–27, May 2009.

[15] J. Boudjadar, K. G. Larsen, J. H. Kim, and U. Nyman. Compositional schedulability analysis of an avionics system using uppaal. In *International Conference on Advanced Aspects of Software Engineering*, 2014.

[16] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, 1990. LICS'90*, pages 428–439, June 1990.

[17] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.

[18] E. M. Clarke. 25 years of model checking. chapter The Birth of Model Checking, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2008.

[19] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[20] D. Cofer and S. Miller. Do-333 certification case studies. In *NASA Formal Methods*, pages 1–15. Springer, 2014.

[21] D. Cofer and S. P. Miller. Formal methods case studies for do-333. Technical Report NASA/CR-2014-218244, NASA Langley Research Center, NASA Langley Research Center Hampton, Virginia 23681, USA, 2014.

[22] L. De Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. Sal 2. In *Computer aided verification*, pages 496–500. Springer, 2004.

[23] L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proceedings of the 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455. Springer-Verlag, July 2002.

[24] D. L. Dill. A retrospective on murφ. In Grumberg and Veith [33], pages 77–88.

[25] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *ICCD*, pages 522–525, 1992.

[26] 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.

[27] 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.

[28] Esterl Technologies. SCADE Design Verifier. `http://www.esterel-technologies.com/products/scade-suite/verify/design-verifier/`.

[29] L. Fix. Fifteen years of formal property verification in intel. In Grumberg and Veith [33], pages 139–144.

[30] R. W. Floyd. Assigning meanings to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.

[31] P. Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 476–479, London, UK, UK, 1997. Springer-Verlag.

[32] T. Granlund et al. The gnu multiple precision arithmetic library. *TMG Datakonsult, Boston, MA, USA*, 2(2), 1996.

[33] O. Grumberg and H. Veith, editors. *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008.

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

[35] K. Havelund, M. Lowry, and J. Penix. Formal analysis of a space-craft controller using spin. *Software Engineering, IEEE Transactions on*, 27(8):749–765, 2001.

[36] 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.

[37] P. Hintjens. *ZeroMQ: Messaging for Many Applications*. "O'Reilly Media, Inc.", 2013.

[38] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[39] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):1–54, Oct. 2009.

[40] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, et al. Replacing testing with formal verification in intel^{\ scriptsize\ circledR} coretm i7 processor execution engine validation. In *Computer Aided Verification*, pages 414–429. Springer, 2009.

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

[42] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *ACM SIGOPS*, SOSP, pages 207–220, New York, NY, USA, 2009. ACM.

[43] O. Laurent, P. Michel, and V. Wiels. Using formal verification techniques to reduce simulation and test effort. In *FME 2001: Formal Methods for Increasing Software Productivity*, pages 465–477. Springer, 2001.

[44] X. Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, Dec. 2009.

[45] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[46] S. Miller, E. Anderson, L. Wagner, M. Whalen, and M. Heimdahl. Formal verification of flight critical software. *Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit*, pages 15–18, 2005.

[47] S. P. Miller, A. C. Tribble, and M. P. Heimdahl. Proving the shalls. In *FME 2003: Formal Methods*, pages 75–93. Springer, 2003.

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

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

[50] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction*, pages 213–228. Springer, 2002.

[51] R. Neumann. Using promela in a fully verified executable ltl model checker. In *Verified Software: Theories, Tools and Experiments*, pages 105–114. Springer, 2014.

[52] D. Oe, A. Stump, C. Oliver, and K. Clancy. versat: A verified modern sat solver. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7148, pages 363–378. Springer, 2012.

[53] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, July 1982.

[54] L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[55] A. Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the International Symposium on Semantics of Concurrent Computation*, pages 1–20, London, UK, UK, 1979. Springer-Verlag.

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

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

[58] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In W. A. Hunt Jr. and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 127–144. Springer Berlin Heidelberg, 2000.

[59] F. Somenzi. Cudd: Cu decision diagram package-release 2.4. 0. *University of Colorado at Boulder*, 2009.

[60] C. Sprenger. A verified model checker for the modal μ-calculus in coq. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 167–183, London, UK, UK, 1998. Springer-Verlag.

[61] A. Stump. Proof checking technology for satisfiability modulo theories. *Electronic Notes in Theoretical Computer Science*, 228:121–133, 2009.

[62] 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.

[63] P. Traverse, I. Lacaze, and J. Souyris. Airbus fly-by-wire: a process toward total dependability. In *25th Int. Congress of the Aeronautical Sciences*, 2006.

[64] K. N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting bdds in coq. In H. Jifeng and M. Sato, editors, *Advances in Computing Science — ASIAN 2000*, volume 1961 of *Lecture Notes in Computer Science*, pages 162–181. Springer Berlin Heidelberg, 2000.

[65] M. Whalen, D. Cofer, S. Miller, B. H. Krogh, and W. Storm. Integration of formal analysis into a model-based software development process. In *Formal Methods for Industrial Critical Systems*, pages 68–84. Springer, 2008.

[66] M. W. Whalen, J. D. Innis, S. P. Miller, and L. G. Wagner. Adgs-2100 adaptive display and guidance system window manager analysis. 2006.

[67] V. Wiels, R. Delmas, D. Doose, P.-L. Garoche, J. Cazin, and G. Durrieu. Formal verification of critical aerospace software. *AerospaceLab Journal, Issue*, 2012.