Certificates for Parameterized Model Checking

Sylvain Conchon[1,2], Alain Mebsout[2,3], and Fatiha Zaïdi[1]

[1] LRI, Université Paris-Sud, CNRS, Orsay F-91405
[2] INRIA Saclay – Île-de-France, Orsay cedex, F-91893
[3] The University of Iowa

Abstract. This paper presents a technique for the certification of Cubicle, a model checker for proving safety properties of parameterized systems. To increase the confidence in its results, Cubicle now produces a proof object (or certificate) that, if proven valid, guarantees that the answer for this specific input is correct. The main challenges addressed in this paper are (1) the production of such certificates without degrading the performances of the model checker and (2) the construction of these proof objects so that they can be independently and efficiently verified by an SMT solver. Since the burden of correctness insurance now relies on this external solver, a stronger guarantee is obtained by the use of multiple backend automatic provers for redundancy. Experiments show that our approach does not impact Cubicle's performances and that we were able to verify certificates for challenging parameterized problems. As a byproduct, these certificates allowed us to find subtle and critical implementation bugs in Cubicle.

1 Introduction

Multi-core architectures or distributed systems usually rely on protocols (such as mutual exclusion, cache coherence or fault-tolerance) which are designed for an arbitrary number of components. These protocols are critical and known as being notoriously difficult to design essentially because of their highly asynchronous and fine-grained concurrent nature. As a result, their validation by simulation is risky because some race conditions appear scarcely and are unlikely to be reproduced. Consequently, the formal verification of these protocols is a necessity.

One of the most successful formal technique for verifying concurrent systems is model checking which automatically determines if a model, usually described by a transition system, meets a specification expressed as temporal properties. When the model is defined independently of the number of components, its verification is known as the *parameterized model checking problem*.

Being parameterized or not, the answer produced by a model checker is usually simply "yes" or "no". When the result is negative, a counterexample (in the form of a sequence of transitions) can also be easily returned (and checked by the user). On the contrary, model checkers rarely return a proof evidence for a positive answer. So, should we trust a model checker when it simply returns "yes"? From our experience, given the high complexity of the implementation of these tools, the answer is clearly no.

To be sure of the correctness of these answers, we can either use a certified model checker [12] or a model checker that produces in addition a proof of its result, also called a *certificate* [21]. The advantage of the first approach is that the model checker is verified correct once and for all. However, this is a very heavy task since model checkers are profoundly 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 simple enough to be checked in a reasonable time after the fact.

The aim of this work is to bring a higher level of confidence in the results produced by Cubicle [6], an SMT-based model checker for proving safety properties of parameterized systems[4]. Cubicle represents states as logical formulas (expressed in a fragment of first-order logic) and checks that unsafe states are not reachable using a backward analysis. In that framework, it is far simpler to produce and check a certificate than to certify the model checker itself. Indeed, Cubicle is a very complex piece of software combining higher order functional programming style with efficient imperative data structures and concurrency. As far as we know, there are no framework for certifying such a program as is. Furthermore, we demonstrate through a set of experiments that checking proof objects can be done efficiently, even for industrial size protocols.

The content of the paper, our contributions and the originality of our approach are as follows:

- Extract an inductive invariant ϕ from the backward reachability loop and generate proof obligations (POs) whose validity guarantees that ϕ is an inductive invariant subsuming the original safety property (Section 3). These POs are first order formulas which are sent to an automatic theorem prover (Section 4).
- A set of algorithmic techniques to enrich and simplify the certificates in order to handle more complex and larger problems (Section 5). A stronger guarantee on the certification process is achieved by redundancy: each PO is independently proven by several tools (SMT solvers, automatic theorem provers, *etc.*).

We illustrate the general approach with a running example: a simple cache coherence protocol. We show the merit of our approach in Section 5.2 through a set of benchmarks for which the certification process is conducted entirely automatically. These notably include two industrial parameterized cache coherence protocols: FLASH and a new protocol developed at Intel and Duke.

Last but not least, the certificates allowed us to find several bugs in Cubicle whose severity can be classified as harmless to critical with a direct impact on its correctness. Some of these bugs have been found by testing but others have escaped all traditional debugging techniques because they only appear very rarely and are related to tricky implementation details.

[4] Developed conjointly between Université Paris-Sud and Intel.

2 Array Based Transition Systems

Cubicle is based on the theoretical foundation of *Model Checking Modulo Theories* (MCMT) [14] by Ghilardi and Ranise. This is a declarative framework for parameterized systems in which transitions and properties are expressed in a particular fragment of first order logic. Systems expressible in this framework are called array based transition systems because their state can be seen as a set of unbounded arrays whose indexes range over elements of the parameterized domain.

Definition 1. *An* array based transition system *is a tuple* $\mathcal{S} = (Q, I, \tau)$ *where* Q *is a set of function symbols (also called arrays) representing the state variables,* I *is a formula which characterizes the* initial states *of the system (in which variables of* Q *can appear free) and* τ *is a transition relation.*

In the following, the formula I is universally quantified. The relation τ is expressed in the form of a disjunction of existentially quantified (by zero, one, or several variables of the parameterized domain) formulas. Each component of this disjunction is called a transition and is said to be parameterized by its existential variables. Following usual notations, we note x' the value of $x \in Q$ after executing the transition. Transitions relate values of primed and un-primed variables and arrays, and are of the form:

$$t(Q, Q') \equiv \exists \bar{i}.\ \underbrace{\gamma(\bar{i}, Q)}_{\text{guard}} \wedge \underbrace{\bigwedge_{x \in Q} \forall \bar{j}. x'(\bar{j}) = \delta_x(\bar{i}, \bar{j}, Q)}_{\text{action}}$$

where γ is a quantifier free formula called the *guard* of t and δ_x is a quantifier free formula called the *update* of x.

Safety properties are expressed by characterizing unsafe states. An unsafe formula must be in a special form called a *cube*, *i.e.* a conjunction of literals existentially quantified by distinct variables:

$$\Theta \equiv \exists (\bar{i}).\ \mathit{distinct}(\bar{i}) \wedge l_1(\bar{i}) \wedge \ldots \wedge l_n(\bar{i}).$$

Running Example. We illustrate this framework on a simplified version of the directory based cache coherence protocol proposed by German [24]. In Figure 1, we give a high-level view of the evolution of a single cache as a state diagram. The protocol consists of a global directory which maintains the consistency of a shared memory between a parameterized number of cache clients. The status of each cache i is indicated by a variable `Cache`$[i]$ which can be in one of the three states: (`E`)xclusive (read and write accesses), (`S`)hared (read access only) or (`I`)nvalid (no access to the memory). Clients send requests to the directory when cache misses occur: `rs` for a shared access (read miss), `re` for an exclusive access (write miss). The directory has four variables: a boolean flag `Exg` indicates whether a client has an exclusive access to the main memory, a boolean array `Shr`, such that `Shr`$[i]$ is true when a client i is granted (read or write) access to

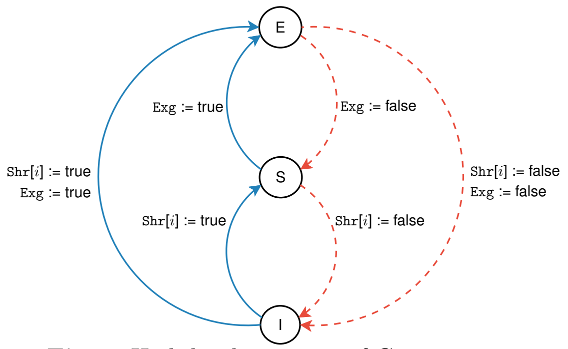


Fig. 1. High level overview of German-*esque*

the memory, Cmd stores the current request (ϵ stands for the absence of request), and Ptr contains the emitter of the current request.

The array based transition system for this protocol is described by its initial states, represented by the following logical formula (caches are invalid, no access has been given and there is no request to be processed)

$$I \;\equiv\; \forall i.\ \mathtt{Cache}[i] = \mathsf{I} \;\wedge\; \mathtt{Shr}[i] = \mathsf{false} \;\wedge\; \mathtt{Exg} = \mathsf{false} \;\wedge\; \mathtt{Cmd} = \epsilon$$

and by its transition relation given below (an horizontal line separates guards from actions, depicted in blue when they modify variables while the ones that don't change values are light gray). For instance, transition t_6 should read as: if there exists a process i such that the current pointer (Ptr) is i, the command to be processed is a request to exclusive access (re), the flag Exg is not set and the array Shr contains false for *all processes*, then erase the command, set the flag Exg, register the process i in Shr and change the cache state of i to exclusive (E).

$$\tau \;\equiv\; t_1 \;\vee\; t_2 \;\vee\; t_3 \;\vee\; t_4 \;\vee\; t_5 \;\vee\; t_6$$

$$t_1 : \exists i.\ \begin{array}{l} \mathtt{Cache}[i] = \mathsf{I} \;\wedge\; \mathtt{Cmd} = \epsilon \;\wedge \\ \hline \mathtt{Ptr}' = i \;\wedge\; \mathtt{Cmd}' = \mathsf{rs} \;\wedge \\ \mathtt{Exg}' = \mathtt{Exg} \;\wedge \\ \forall j.\ \mathtt{Shr}[j] = \mathtt{Shr}'[j] \;\wedge \\ \quad\quad \mathtt{Cache}[j] = \mathtt{Cache}'[j] \end{array}$$

$$t_2 : \exists i.\ \begin{array}{l} \mathtt{Cache}[i] \neq \mathsf{E} \;\wedge\; \mathtt{Cmd} = \epsilon \;\wedge \\ \hline \mathtt{Ptr}' = i \;\wedge\; \mathtt{Cmd}' = \mathsf{re} \;\wedge \\ \mathtt{Exg}' = \mathtt{Exg} \;\wedge \\ \forall j.\ \mathtt{Shr}[j] = \mathtt{Shr}'[j] \;\wedge \\ \quad\quad \mathtt{Cache}[j] = \mathtt{Cache}'[j] \end{array}$$

$$t_3 : \exists i.\ \begin{array}{l} \mathtt{Shr}[i] = \mathsf{true} \;\wedge\; \mathtt{Cmd} = \mathsf{re} \;\wedge \\ \hline \mathtt{Ptr}' = \mathtt{Ptr} \;\wedge\; \mathtt{Cmd}' = \mathtt{Cmd} \;\wedge \\ \mathtt{Exg}' = \mathsf{false} \;\wedge \\ \forall j.\ ite(i = j, \neg\mathtt{Shr}'[j], \\ \quad\quad\quad\quad \mathtt{Shr}'[j] = \mathtt{Shr}[j]) \;\wedge \\ \forall j.\ ite(i = j, \mathtt{Cache}'[j] = \mathsf{I}, \\ \quad\quad\quad\quad \mathtt{Cache}'[j] = \mathtt{Cache}'[j]) \end{array}$$

$$t_4 : \exists i.\ \begin{array}{l} \mathtt{Shr}[i] = \mathsf{true} \;\wedge\; \mathtt{Cmd} = \mathsf{rs} \;\wedge\; \mathtt{Exg} \;\wedge \\ \hline \mathtt{Ptr}' = \mathtt{Ptr} \;\wedge\; \mathtt{Cmd}' = \mathtt{Cmd} \;\wedge \\ \mathtt{Exg}' = \mathsf{false} \;\wedge\; \forall j. \mathtt{Shr}'[j] = \mathtt{Shr}[j] \;\wedge \\ \forall j.\ ite(i = j, \mathtt{Cache}'[j] = \mathsf{S}, \\ \quad\quad\quad\quad \mathtt{Cache}'[j] = \mathtt{Cache}'[j]) \end{array}$$

$$t_5 : \exists i.\ \begin{array}{l} \mathtt{Ptr} = i \;\wedge\; \mathtt{Cmd} = \mathsf{rs} \;\wedge\; \neg\mathtt{Exg} \;\wedge \\ \hline \mathtt{Ptr}' = \mathtt{Ptr} \;\wedge\; \mathtt{Cmd}' = \epsilon \;\wedge \\ \mathtt{Exg}' = \mathtt{Exg} \;\wedge \\ \forall j.\ ite(i = j, \mathtt{Shr}'[j], \\ \quad\quad\quad\quad \mathtt{Shr}'[j] = \mathtt{Shr}[j]) \;\wedge \\ \forall j.\ ite(i = j, \mathtt{Cache}'[j] = \mathsf{S}, \\ \quad\quad\quad\quad \mathtt{Cache}'[j] = \mathtt{Cache}'[j]) \end{array}$$

$$t_6 : \exists i.\ \begin{array}{l} \mathtt{Ptr} = i \;\wedge\; \mathtt{Cmd} = \mathsf{re} \;\wedge\; \neg\mathtt{Exg} \;\wedge \\ \forall j.\ \neg\mathtt{Shr}[j] \;\wedge \\ \hline \mathtt{Ptr}' = \mathtt{Ptr} \;\wedge \\ \mathtt{Cmd}' = \epsilon \;\wedge\; \mathtt{Exg}' = \mathsf{true} \;\wedge \\ \forall j.\ ite(i = j, \mathtt{Shr}'[j], \\ \quad\quad\quad\quad \mathtt{Shr}'[j] = \mathtt{Shr}[j]) \;\wedge \\ \forall j.\ ite(i = j, \mathtt{Cache}'[j] = \mathsf{E}, \\ \quad\quad\quad\quad \mathtt{Cache}'[j] = \mathtt{Cache}'[j]) \end{array}$$

This protocol ensures that when a cache client is in an exclusive state then no other process has (read or write) access to the memory. Proving this safety property amounts to checking that states satisfying Θ are not reachable:

$$\Theta \;\equiv\; \exists i,j.\; i \neq j \;\wedge\; \texttt{Cache}[i] = \mathsf{E} \;\wedge\; \texttt{Cache}[j] \neq \mathsf{I}$$

3 Proof Evidence in Backward Reachability

In this section we explain how to get proof objects from the backward reachability analysis used by Cubicle to prove the safety of array based systems.

For a state formula φ and a transition $\tau \in \mathcal{T}$, let $pre(\tau, \varphi)$ be the formula describing the set of states from which a state satisfying φ can be reached in one τ-step. The pre-image *closure* of φ, denoted by $\text{Pre}^*(\varphi)$, is defined as follows

$$\begin{cases} \text{Pre}^0(\varphi) \triangleq \varphi \\ \text{Pre}^n(\varphi) \triangleq pre(\tau, \text{Pre}^{n-1}(\varphi)) \\ \text{Pre}^*(\varphi) \triangleq \bigvee_{k \in \mathbb{N}} \text{Pre}^k(\varphi) \end{cases}$$

and the pre-image of a set of formulas V is defined by $\text{Pre}^*(V) = \bigcup_{\varphi \in V} \text{Pre}^*(\varphi)$. We also write $\text{Pre}(\varphi)$ for $\text{Pre}^1(\varphi)$.

Definition 2. *A formula φ is said to be* reachable *iff* $\text{Pre}^*(\varphi) \wedge I$ *satisfiable. It is* unreachable *otherwise.*

The framework of MCMT gives sufficient conditions for which the reachability problem (Is the unsafe formula Θ reachable in the system $\mathcal{S}$?) is decidable. In particular, we consider that the pre-image of a cube by the transition relation τ (Pre_τ) is effectively computable. The interested reader is referred to [14] for more details. Under these conditions, safety can be checked by backward reachability analysis.

We give a standard backward reachability algorithm for this framework, as defined by the function `BWD` in Algorithm 1. Starting with an empty formula $\mathcal{V}$ of *visited nodes* (*i.e.* false) and a queue $\mathcal{Q}$ of *pending nodes* initialized with a formula Θ, `BWD` iteratively computes the backward reachability graph of $\text{Pre}^*_\tau(\Theta)$. The algorithm terminates when a node fails the *safety check* (consistency with the initial condition — line **6**), or when all nodes in $\mathcal{Q}$ are *subsumed* by $\mathcal{V}$ (line **8**). These logical checks are performed by an SMT solver.

In the case where the return value of the algorithm is ***unsafe***, it is easy to expose an error trace — from the initial states to one of the violated property — to the user. This trace can then be replayed afterwards to ensure the system is indeed unsafe with respect to its specification. In the case where the return value is ***safe***, a certificate can be produced. Because of the nature of the program at hand, any instrumentation for certification purposes could either diminish efficiency (in the worst case, prevent the verification of industrial size systems) or even badly interfere and compromise the correctness (this is however not

Algorithm 1. Backward reachability analysis

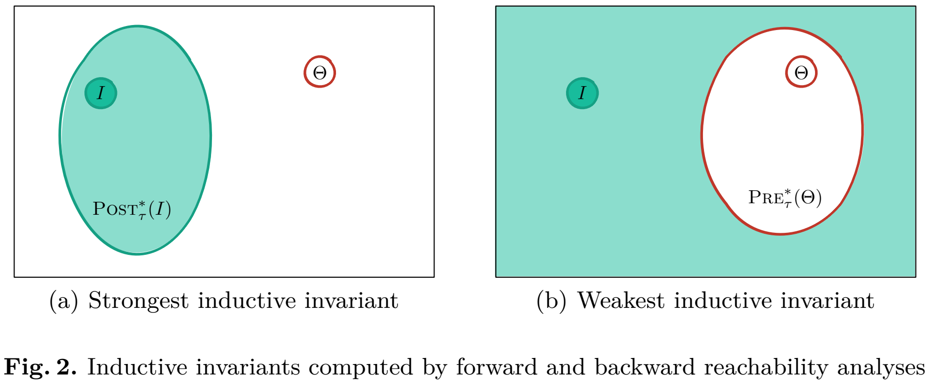
```
Input: an array based system $\mathcal{S} = (Q, I, \tau)$ and a cube $\Theta$
Variables:
    $\mathcal{V}$: visited cubes
    $\mathcal{Q}$: work queue

function BWD($\mathcal{S}, \Theta$) : begin
    $\mathcal{V}$ := $\emptyset$;
    push($\mathcal{Q}$, $\Theta$);
    while not_empty($\mathcal{Q}$) do
        $\varphi$ := pop($\mathcal{Q}$);
        if $\varphi \wedge I$ satisfiable then
            return unsafe
        else if $\varphi \not\models \mathcal{V}$ then
            $\mathcal{V}$ := $\mathcal{V} \vee \varphi$;
            push($\mathcal{Q}$, $\text{PRE}_\tau(\varphi)$);
    return safe
```

problematic from a certification standpoint if the results are to be independently checked, but it would nonetheless render the tool ineffective). In fact, there is no need for this certificate to contain or reflect all reasoning steps taken by the model checker, such as pre-images, fixpoint and safety checks, because $\mathcal{V}$ already contains enough information to guarantee the correctness.

Definition 3. *An* invariant *of a system is any property that holds in all reachable states of the system.*

The notion of safety is very closely related to the one of invariance. Checking the safety of a system essentially amounts to ensuring that a given property is an invariant of this system. In reality, for a system, the set of all *reachable* states constitutes the *strongest inductive invariant* (green-shaded area of Figure 2(a)). Dually, the set of states that *cannot reach* an unsafe state of the system constitutes the *weakest inductive invariant* (*w.r.t.* the unsafe states) of the system (green part of Figure 2(b)).



(a) Strongest inductive invariant (b) Weakest inductive invariant

Fig. 2. Inductive invariants computed by forward and backward reachability analyses

The set $\mathcal{V}$ computed by Cubicle (in Algorithm 1) is really the negation of this weakest inductive invariant. It forms in itself a proof or a certificate of safety of the system. Moreover, it is very simple to establish that a formula ϕ is an inductive invariant of a system $\mathcal{S} = (Q, I, \tau)$. All that is necessary is for it to verify the two following conditions:

$$I(X) \models \phi(X) \tag{1}$$
initialization

$$\phi(X) \wedge \tau(X, X') \models \phi(X'). \tag{2}$$
preservation

The base case (1) says that the invariant ϕ must be true in the initial states of the system and the inductive case (2) says that the invariant must be preserved by the transition relation. If additionally, we have

$$\phi(X) \models P(X) \tag{3}$$
property

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

If we take $\phi = \neg\mathcal{V}$ and $P = \neg\Theta$, where $\mathcal{V}$ is the disjunction of visited cubes in Algorithm 1 and Θ is the unsafe formula for the system, then these three conditions are verified. Indeed, we have by construction that $\mathcal{V} \models \textsc{Pre}^*_\tau(\Theta)$, $\mathcal{V}$ is closed by pre-image, *i.e.* $\mathcal{V}(X') \wedge \tau(X, X') \models \mathcal{V}(X)$ so $\neg\mathcal{V}(X) \wedge \tau(X, X') \models \neg\mathcal{V}(X')$. Because $\mathcal{V}$ contains Θ, the final condition (3) is also verified.

To certify that the result of Algorithm 1 implemented by Cubicle is correct, it suffices to independently make sure that $\phi = \neg\mathcal{V}$ and $P = \neg\Theta$ satisfy the three conditions (1), (2) and (3). In the sequel, we show how to do this automatically and efficiently.

4 A Certification Framework for Cubicle

We can prove the conditions identified at the end of the previous section with the aid of a proof assistant or directly with an automatic theorem prover if we desire to carry out the certification without human intervention. In the latter, we have to trust the prover we choose. To remedy this possible disadvantage, we have decided to use Why3 [13], a platform for deductive program verification. It provides a logical language called Why to describe formulas in a first order polymorphic logic with a translation mechanism to several automatic or interactive theorem provers. One big advantage of Why3 is that proof obligations can be described in a common language and can be discharged by a multitude of backend tools: SMT solvers like Alt-Ergo [5], CVC4 [3], Yices [11] or Z3 [8]; resolution based solvers like E [27], iProver [17], SPASS [30], Vampire [25]; or when necessary, even proof assistants like Coq [9] or PVS [23].

Redundancy as a tool is used in multiple contexts. For instance, control systems of avionics are physical entities which can fail (with known probabilities), and higher fault tolerance is achieved by having several identical redundant components and voting mechanisms. In formal methods, the use of different tools to independently corroborate results is a way to achieve a higher level of confidence. In our case, we trust our certification process when at least two independent solvers confirm the validity of our certificates.

Our certification process follows the diagram of Figure 3. The inductive invariant ϕ constitutes the essence of the certificate produced by Cubicle. It can then be fed directly to the checker (here Why3) or can be simplified and enriched with a set L of lemmas (box Simpl described in Section 5). Once the solvers used by the checker redundantly prove the conditions (1)–(3), the certificate is declared valid.

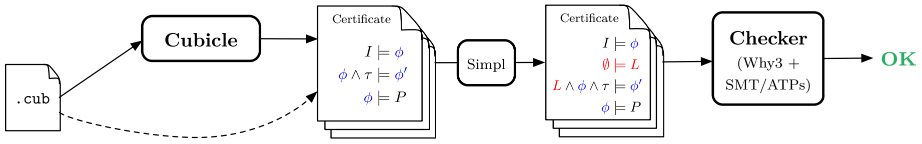


Fig. 3. Certification schema

Running Example. When Cubicle is executed (without any options) on the small protocol of Section 2, the certificate ϕ produced is composed of 15 quantified clauses. Now, proofs obligations are generated in Why3's input language to ensure that ϕ is indeed inductive.

$$\phi \equiv \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge \phi_5 \wedge \phi_6 \wedge \phi_7 \wedge \phi_8 \wedge$$
$$\phi_9 \wedge \phi_{10} \wedge \phi_{11} \wedge \phi_{12} \wedge \phi_{13} \wedge \phi_{14} \wedge \phi_{15}$$

$$\begin{aligned}
\phi_1 &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \mathtt{Exg} \wedge \mathtt{Cmd} = \mathsf{rs} \wedge \mathtt{Ptr} = z_1 \wedge \mathtt{Cache}[z_2] \neq \mathsf{I} \wedge \mathtt{Shr}[z_1] \wedge \neg\mathtt{Shr}[z_2])\\
\phi_2 &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \mathtt{Cache}[z_1] = \mathsf{E} \wedge \mathtt{Cache}[z_2] \neq \mathsf{I})\\
\phi_3 &\equiv \neg(\exists z_1, z_2, z_3.\ z_2 \neq z_3 \wedge z_1 \neq z_3 \wedge z_1 \neq z_2 \wedge\\
&\qquad \mathtt{Exg} \wedge \mathtt{Cmd} = \mathsf{rs} \wedge \mathtt{Ptr} = z_1 \wedge \mathtt{Cache}[z_2] \neq \mathsf{I} \wedge \neg\mathtt{Shr}[z_2] \wedge \mathtt{Shr}[z_3])\\
\phi_4 &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \mathtt{Cmd} = \mathsf{re} \wedge \mathtt{Ptr} = z_2 \wedge \mathtt{Cache}[z_1] = \mathsf{E} \wedge \neg\mathtt{Shr}[z_1] \wedge \mathtt{Shr}[z_2])\\
\phi_5 &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \neg\mathtt{Exg} \wedge \mathtt{Cmd} = \epsilon \wedge \mathtt{Cache}[z_1] \neq \mathsf{E} \wedge \mathtt{Cache}[z_2] \neq \mathsf{I} \wedge \neg\mathtt{Shr}[z_1] \wedge \neg\mathtt{Shr}[z_2])\\
\phi_6 &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \mathtt{Cmd} = \mathsf{re} \wedge \mathtt{Ptr} = z_1 \wedge \mathtt{Cache}[z_2] \neq \mathsf{I} \wedge \mathtt{Shr}[z_1] \wedge \neg\mathtt{Shr}[z_2])\\
\phi_7 &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \mathtt{Exg} \wedge \mathtt{Cmd} = \mathsf{rs} \wedge \mathtt{Cache}[z_1] = \mathsf{E} \wedge \mathtt{Shr}[z_2])\\
\phi_8 &\equiv \neg(\exists z_1, z_2, z_3.\ z_2 \neq z_3 \wedge z_1 \neq z_3 \wedge z_1 \neq z_2 \wedge\\
&\qquad \mathtt{Cmd} = \mathsf{re} \wedge \mathtt{Ptr} = z_1 \wedge \mathtt{Cache}[z_2] \neq \mathsf{I} \wedge \neg\mathtt{Shr}[z_1] \wedge \neg\mathtt{Shr}[z_2] \wedge \mathtt{Shr}[z_3])\\
\phi_9 &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \neg\mathtt{Exg} \wedge \mathtt{Cmd} = \mathsf{rs} \wedge \mathtt{Ptr} = z_2 \wedge \mathtt{Cache}[z_1] = \mathsf{E})\\
\phi_{10} &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \neg\mathtt{Exg} \wedge \mathtt{Cmd} = \mathsf{re} \wedge \mathtt{Ptr} = z_2 \wedge \mathtt{Cache}[z_1] = \mathsf{E} \wedge \neg\mathtt{Shr}[z_1] \wedge \neg\mathtt{Shr}[z_2])\\
\phi_{11} &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \neg\mathtt{Exg} \wedge \mathtt{Cmd} = \mathsf{re} \wedge \mathtt{Ptr} = z_1 \wedge \mathtt{Cache}[z_2] \neq \mathsf{I} \wedge \neg\mathtt{Shr}[z_1] \wedge \neg\mathtt{Shr}[z_2])\\
\phi_{12} &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \mathtt{Cmd} = \epsilon \wedge \mathtt{Cache}[z_1] \neq \mathsf{E} \wedge \mathtt{Cache}[z_2] \neq \mathsf{I} \wedge \mathtt{Shr}[z_1] \wedge \neg\mathtt{Shr}[z_2])\\
\phi_{13} &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \mathtt{Exg} \wedge \mathtt{Cmd} = \epsilon \wedge \mathtt{Cache}[z_1] = \mathsf{E} \wedge \mathtt{Cache}[z_2] = \mathsf{I} \wedge \mathtt{Shr}[z_2])\\
\phi_{14} &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \neg\mathtt{Exg} \wedge \mathtt{Cmd} = \mathsf{rs} \wedge \mathtt{Ptr} = z_1 \wedge \mathtt{Cache}[z_2] \neq \mathsf{I} \wedge \neg\mathtt{Shr}[z_2])\\
\phi_{15} &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \neg\mathtt{Exg} \wedge \mathtt{Cmd} = \epsilon \wedge \mathtt{Cache}[z_1] = \mathsf{E} \wedge \mathtt{Cache}[z_2] = \mathsf{I})
\end{aligned}$$

This certificate is immediate to extract from the set $\mathcal{V}$ computed by Cubicle so there is zero overhead. It is then fed directly to Why3 which in turn calls several automated theorem provers. The certificate contains quantifiers (both universal and existential) so we are limited to solvers that natively support them. Here we chose to have Why3 call seven different backend provers to discharge the proof obligations of our certificate. The results of this certificate's verification are given in table 1. Each prover was run with a timeout of five seconds. Times are given in seconds and bold numbers stand for a "valid" answer, barred text in red cells is for the answer "unkown" while T.O. denotes executions that did not end in the allocated time (120s). The PO for preservation is split in 15 subgoals (one for each conjunct of ϕ') and we can notice that each goal is discharged by at least three provers.

Remark. The input file describing both the system and the properties is given in the syntax of Cubicle. When generating the certificate, a translation phase is present to express the problem in the language of Why3 (*cf.* dashed line in figure 3). In order to trust completely our certification process, this translation should be proven correct (as semantics preserving). This is relatively easy because everything that is written in Cubicle is simply formulas in a fragment of first order logic, so there exists a one-to-one correspondence and the translation essentially consists in a pretty printing step. Ideally, we could even adopt the same input language (*i.e.* Why3's) to describe parameterized systems and thus dissipate all remaining doubts.

Table 1. Why3's output on certificate for German-*esque*

Proof obligations		Alt-Ergo (0.96)	CVC3 (2.4.1)	CVC4 (1.3)	Eprover (1.8-001)	Spass (3.5)	Yices (1.0.40)	Z3 (4.3.2)
initialisation	1.	**0.02**	**0.02**	**0.01**	**0.01**	**0.05**	**0.13**	**0.01**
property	1.	**0.01**	**0.01**	**0.01**	**0.01**	**0.02**	**0.00**	**0.00**
preservation	1.	~~0.01~~	~~0.85~~	**0.03**	**0.03**	**0.03**	~~1.01~~	**0.02**
	2.	~~0.01~~	~~0.69~~	**0.04**	**0.20**	T.O.	~~0.97~~	**0.02**
	3.	**0.02**	**0.03**	**0.03**	**0.02**	**0.04**	**0.35**	**0.01**
	4.	~~0.01~~	~~1.18~~	**0.03**	**0.03**	**0.36**	~~0.67~~	**0.02**
	5.	**0.03**	~~0.99~~	**0.04**	T.O.	T.O.	**1.08**	**0.01**
	6.	**0.03**	~~1.24~~	**0.04**	**0.04**	**3.65**	**0.91**	**0.01**
	7.	**0.02**	**0.03**	**0.03**	**0.04**	**0.05**	**0.61**	**0.01**
	8.	**0.06**	~~1.18~~	**0.03**	**0.07**	**59.6**	~~0.82~~	**0.01**
	9.	**0.02**	~~1.17~~	**0.03**	**0.01**	**0.06**	~~1.33~~	**0.01**
	10.	**0.03**	**0.03**	**0.02**	**0.04**	**0.81**	~~1.49~~	**0.01**
	11.	~~0.01~~	~~0.58~~	**0.02**	**0.03**	**0.18**	~~0.99~~	**0.02**
	12.	**0.03**	**0.03**	**0.03**	**0.05**	**0.45**	**0.78**	**0.01**
	13.	**0.03**	~~0.93~~	**0.02**	**0.01**	**0.08**	~~0.95~~	**0.01**
	14.	~~0.01~~	~~0.82~~	**0.20**	**0.21**	**4.60**	~~2.12~~	**0.01**
	15.	**0.02**	**0.03**	**0.02**	**0.02**	**0.07**	**0.83**	**0.01**

Certificates for toy examples like a simple atomic mutex can be verified by all seven provers, but here the protocol German-*esque*, while simply expressed, is far from trivial. POs related to the initialization (1) and property (3) conditions are easily and almost instantly discharged by all solvers. However POs that concern preservation are usually a lot more difficult. These rather disappointing performances can be attributed to the ubiquitous quantifiers of these goals, in particular in the representation of the transition relation. Most of these solvers use very sensitive heuristics for quantifiers so performances are often uneven and hard to predict. However results for this (small) benchmark are still satisfactory because all goals are proven independently several times.

This is an efficient and unintrusive way of generating correctness certificates. The remaining challenge is now to be able to automatically verify these certificates for problems whose size and complexity are orders of magnitude larger.

5 Simpler *and* Richer Certificates

One nice feature of extracting certificates in this manner is that the certification phase becomes completely independent of the model checking phase. In particular certificates are completely oblivious to any optimization — that preserves the transitive closure property of $\mathcal{V}$— used inside Cubicle. For instance, running a parallel reachability loop or changing the search strategy does not impair the ability to produce correct certificates. Even if one optimization was incorrect, a certificate could still be produced the same way, but would likely[5] not be verified by external solvers.

5.1 Invariants Inference

One crucial optimization used in Cubicle is a mechanism for automatically inferring quantified invariants [7]. Invariants found this way are particularly valuable because they allow our technique for parameterized verification to scale up effectively on industrial size protocols. Inevitably, it also speeds up the verification for small and medium size problems. It is a well known fact that invariants, if given or when found, will prune the search space and directly impact the time and space used by model checking algorithms. The number of visited nodes (in $\mathcal{V}$) is thus immediately diminished so this constitutes an effective way of *reducing the size* of the certificate. One particularity of the algorithm BRAB (Backward Reachability with Approximations and Backtracking) described in [7] and implemented in Cubicle is that inferred invariants are inserted and proved by the same backward reachability loop. From a certification standpoint, this ensures that $\neg\mathcal{V}$ will remain inductive at the

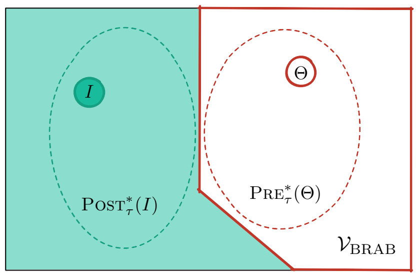

I
Θ
$\textsc{Post}^*_\tau(I)$
$\textsc{Pre}^*_\tau(\Theta)$
$\mathcal{V}_{\text{BRAB}}$

Fig. 4. Inductive invariant computed by BRAB

[5] If one optimization is incorrect but the resulting certificate is verified for one particular benchmark then it simply means that the model checker gave a correct answer by incorrect means.

end of the search, meaning that the technique described in section 3 is still applicable.

In fact the inductive invariant computed by BRAB is halfway between the strongest and the weakest invariant (see Figure 4). $\phi \equiv \neg\mathcal{V}_{\mathrm{BRAB}}$ is a good candidate for a certificate, being expressible more easily that either of those extremes. The underlying reason is that the "internal proof" constructed by the model checker is much shorter.

Running Example. By now running BRAB instead of traditional backward reachability, the certificate extracted at the end of the search is only composed of *four* quantified clauses (*cf.* below) and Table 2 shows that it is proved in totality by all seven solvers we used.

$$
\begin{aligned}
\phi &\equiv \phi_1 \;\wedge\; \phi_2 \;\wedge\; \phi_3 \;\wedge\; \phi_4 \\
\phi_1 &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \\
&\qquad \texttt{Cache}[z_1] = \mathsf{E} \wedge \texttt{Cache}[z_2] \neq \mathsf{I}) \\
\phi_2 &\equiv \neg(\exists z_1.\ \neg\texttt{Exg} \wedge \texttt{Cache}[z_1] = \mathsf{E}) \\
\phi_3 &\equiv \neg(\exists z_1, z_2.\ z_1 \neq z_2 \wedge \\
&\qquad \texttt{Cache}[z_1] = \mathsf{E} \wedge \texttt{Shr}[z_2]) \\
\phi_4 &\equiv \neg(\exists z_1.\ \texttt{Cache}[z_1] \neq \mathsf{I} \wedge \\
&\qquad\qquad \neg\texttt{Shr}[z_1])
\end{aligned}
$$

Table 2. Why3's output on certificate generated by BRAB for German-*esque*

Proof obligations		Alt-Ergo (0.96)	CVC3 (2.4.1)	CVC4 (1.3)	Eprover (1.8-001)	Spass (3.5)	Yices (1.0.40)	Z3 (4.3.2)
initialisation	1.	**0.01**	**0.01**	**0.03**	**0.01**	**0.02**	**0.00**	**0.00**
property	1.	**0.01**	**0.00**	**0.01**	**0.01**	**0.02**	**0.00**	**0.00**
preservation	1.	**0.02**	**0.02**	**0.03**	**0.08**	**0.12**	**0.01**	**0.00**
	2.	**0.02**	**0.02**	**0.04**	**0.11**	**0.11**	**0.05**	**0.01**
	3.	**0.02**	**0.02**	**0.03**	**0.04**	**0.07**	**0.01**	**0.01**
	4.	**0.03**	**0.02**	**0.03**	**0.13**	**0.08**	**0.26**	**0.01**

Experiments. By running Cubicle with the BRAB algorithm we are able to prove the safety of a selected set of benchmarks and we are able to generate certificates small enough for most of them so that they can be verified automatically and independently. To obtain the results depicted in Table 3 we executed Cubicle version 1.0.2 and Why3 0.83 (with the backend solvers used previously) on a laptop with a dual core Intel i7 processor (1.7 GHz) and 8GB of memory. Szymanski_* is a mutual exclusion protocol given in an atomic (at) and non-atomic (na) version. Ricart-Argrwala is a distributed timed mutual exclusion algorithm [26]. These benchmarks also include cache coherence protocols: several versions of the academic protocol German and two industrial size problems: FLASH [18] and an even larger hierarchical protocol Hirr_PV [20]. The numbers given in the column **∀-clauses** correspond to the number of quantified formulas composing the certificate ϕ. The size is for the resulting Why3 file. We say for each certificate if it has been verified and the shortest amount of time to carry the entire proof by one prover. The column **Level** denotes the minimum number of solvers that were able to independently discharge each proof obligation. It morally depicts the level of confidence we get with this certificate.

We can see that the certificates for academic problems can be verified in just a couple seconds but larger certificates are out of reach of all solvers, mostly due to their size.

Table 3. Result for the verification of certificates generated with BRAB

Benchmark	**$\forall$-clauses**	**Size**	**Verified**	**Level**	**Time**
Szymanski_at	31	18 kB	Yes	3	0.96s
Szymanski_na	38	28 kB	Yes	2	1.45s
Ricart_Agrawala	30	39 kB	Yes	2	1.26s
German_Baukus	48	44 kB	Yes	2	1.58s
German.CTC	69	83 kB	Yes	2	2.73s
German_pfs	51	50 kB	Yes	3	1.79s
Flash_nodata	41	123 kB	Yes	2	2.99s
Flash	733	650 kB	No	0	-
Hirr_PV_nodata	2704	1.9 MB	No	0	-
Hirr_PV	2815	1.9 MB	No	0	-

5.2 Intermediate Lemmas

While certificates can be simplified, they can also be enriched to ease the tasks of the automated solvers. The hardest part for these solvers to handle preservation proof obligations is to find good instances of the quantified formulas that appear in their context. If we take a look at the PO (4.) of preservation in the previous table 2, it takes the form $\phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge \tau \models \phi'_4$. With a closer inspection we can remark that we already have $\phi_4 \wedge \tau \models \phi'_4$ which directly implies the PO we want to prove. We call these pieces of additional information *intermediate lemmas* and show in the following how to enrich our certificates with them.

The information we need to infer these lemmas is actually already computed by Cubicle during fixpoint checks. Every time a cube φ_0 is added to $\mathcal{V}$, its pre-image $\textsc{Pre}_\tau(\varphi_0)$ is added to $\mathcal{Q}$. When part of this pre-image passes the fixpoint check, we can retrieve the necessary information of which elements of $\mathcal{V}$ were really useful. This can be done by asking for the *unsat core*[6] of this particular SMT check. The union of the unsat cores, for the fixpoints of all $\textsc{Pre}_\tau(\varphi_0)$, makes up the part of $\mathcal{V}$ that is sufficient to prove the preservation of $\neg\varphi_0$ by τ.

Some extra bookkeeping can be added to the reachability loop to gather this information during runtime. However it can also be reconstructed after the fact simply with $\mathcal{V}$. This has two advantages. First, it allows to keep the model checking phase and the certification phase separated and independent. Second, computing the reasons for the inductiveness of $\mathcal{V}$ once $\mathcal{V}$ is complete yields possibly smaller and simpler intermediate lemmas.

We denote by `UC` a function that returns the unsat core for a satisfiability check in the form of a set of formulas[7]. Our algorithm to extract intermediate lemmas is given by Algorithm 2. It uses the fact that at the end of the search $\mathcal{V}$ is closed under pre-image as shown in Figure 5. For each node φ of $\mathcal{V}$ `uc` is the

[6] The quality of the unsat core depends on the solver we use, but our goal here is only to trim the context. Because only a small portion of the context is necessary for the proof, most solvers will reflect this in their unsat cores.

[7] If the check is satisfiable then `UC` fails, though in our case, if the certificate is correct this should never happen. This amounts to a pre-verification of inductiveness of the certificate by Cubicle itself.

subset of $\mathcal{V}$ that makes the pre-image of φ be in $\mathcal{V}$. Now, going the other way around, if we start in a state of the conjunction Γ, then we necessarily end up in φ after one step of τ. This is what is stated by the lemma added to $\mathcal{L}$ line **6**.

When the intermediate lemmas are assumed by the solvers, the proof of preservation is trivial by a simple propositional reasoning. The burden of verification is shifted to the proof of these intermediate lemmas instead but they are much smaller than the original POs arising from the proof of preservation. For instance, the largest premise of a lemma for the protocol FLASH (see section 5.2) is composed of 41 quantified formulas while the majority has less than 20 (instead of 742 originally). Because the lemmas extraction shown in algorithm 2 is only a series of fixpoint checks, the time spent for the construction of the certificate is always strictly less that the time spent for the model checking phase. This overhead is in our sense acceptable.

Algorithm 2. Intermediate lemmas extraction

```
  Input: $\mathcal{V}$: visited cubes
  Variables: $\mathcal{L}$ : a set of intermediate lemmas
1 $\mathcal{L}$ := $\emptyset$;
2 foreach $\varphi \in \mathcal{V}$ do
3     let uc = UC($\textsc{Pre}_\tau(\varphi) \models \mathcal{V}$) \ $\textsc{Pre}_\tau(\varphi)$ in
4     (* uc is a subset of $\mathcal{V}$ *)
5     let $\Gamma$ = $\bigwedge_{\psi \in \text{uc}} \psi$ in
6     $\mathcal{L}$ := "$\Gamma \wedge \tau \models \varphi'$" $\cup \mathcal{L}$;
7 return $\mathcal{L}$
```

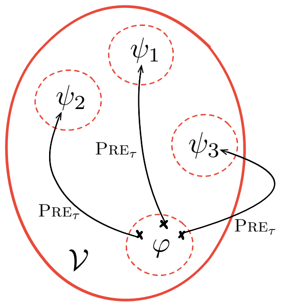


Fig. 5. Finding intermediate lemmas

Table 4. Result for the verification of certificates with intermediate lemmas

Benchmark	MC.	Gen.	∀-clauses	Size	Verified	Level	Time
Szymanski_at	0.04s	0.01s	31	21 kB	Yes	3	0.66s
Szymanski_na	0.06s	0.03s	38	30 kB	Yes	2	1.79s
Ricart_Agrawala	0.05s	0.02s	16	36 kB	Yes	2	0.52s
German_Baukus	0.10s	0.03s	48	40 kB	Yes	**3**	1.16s
German.CTC	0.14s	0.07s	69	62 kB	Yes	**4**	1.98s
s German_pfs	0.11s	0.04s	48	43 kB	Yes	3	1.42s
Flash_nodata	0.11s	0.09s	41	133 kB	Yes	**3**	2.68s
Flash	1m09s	35.8s	733	1.1 MB	**Yes**	**1**	4m7s
Hirr_PV_nodata	4m51s	1m13s	2704	3.4 MB	**Yes**	**1**	42m
Hirr_PV	4m54s	1m25s	2815	3.5 MB	**Yes**	**1**	53m

Experiments. We give experimental results in Table 4 for certificates enriched with intermediate lemmas. The Cubicle systems and corresponding Why3 certificates are available at http://cubicle.lri.fr/certificates. We use the same set of benchmarks as in the previous section. The column **MC.** gives the time spent by Cubicle for model checking the problem, whereas the column **Gen.** gives the time that was necessary to generate the certificate (essentially compute the intermediate lemmas). We can see that it is always faster to generate the certificate than to do the model checking phase. The number of clauses does not change compared to Table 3 but the files are now larger because they include all the extra intermediate lemmas. We can see that it is far more advantageous to pay the price for including these hints in the certificate. Some of the certificates for easier (academic) protocols are now entirely proven by more solvers independently and in a shorter time. Notably, we are now able to verify (albeit only with confidence level 1) the certificates for industrial size protocols FLASH and Hirr_PV. They are significantly larger – a few megabytes instead of kilobytes – and only one SMT solver (Z3) was able to completely discharge all POs. Only a few subgoals are problematic for other solvers, likely due to some inappropriate heuristic for quantifiers instantiation. For instance, 718 of the 736 POs for FLASH were proven by at least two solvers. We would still like to increase the level of confidence brought by the certificates for these large problems.

Exposing Bugs in Cubicle. Cubicle has itself directly benefited from the generation of certificates. During our experiments on our various benchmarks, we were at first not able to verify the certificates for Hirr_PV_nodata and Hirr_PV but only a few (approximately a dozen) of obligations for the intermediate lemmas failed to prove. This allowed us to uncover a bug of Cubicle that was present in its optimized *ad-hoc* instantiation mechanism. Some substitutions were ill-formed in the computation of relevant permutations which would cause the fixpoint check (line **8** of Algorithm 1) to answer incorrectly in some specific cases when multiply nested if-then-else constructs were present in the original system. The Hirr_PV benchmarks are some of the only ones that triggered this bug which had escaped our testing process so far.

6 Related work

Two different lines of work coexist for the certification of verification tools. One approach focuses on verifying the program 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 [22]. 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 [19] or the operating system micro-kernel seL4 [16]. 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. Our oracles, on the other hand, do not even need to provide correct results because they only suggest potential invariants.

Although the first formal verification of a model checker in Coq for the modal μ-calculus [28] goes back to 1998, only recently have *certified verification tools* started to emerge. Blazy *et al.* have verified a static analyzer for C programs [4] 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 [1] and [12]. Amjad [1] shows how to embed BDD based symbolic model checking algorithms in the HOL theorem prover so that results are returned as theorems. This approach relies on the correctness of the backend BDD package. Esparza *et al.* [12] 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.

Usually in these approaches, 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.

The other approach consists in relying on tools that produce traces or certificates to be checked afterwards. This is the approach which is adopted in our work. An approach for the certification of SAT and SMT solvers is the work by Keller *et al.* [2] whose 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. CVC4 is also able to produce full proof trees in a variant of the Edinburgh Logical Framework extended with side conditions [29].

One recent of such application to model checking is Slab [10] which produces certificates in the form of inductive verification diagrams to be checked by SMT solvers.

7 Conclusion

We have presented a technique for certifying the parameterized model checker Cubicle. We showed how to extract certificates from runs of backward reachability analysis in the form of inductive invariants. This approach is minimally intrusive and works with most optimizations. It even directly benefits from the algorithm BRAB to reduce the size and complexity of these certificates. The aim of this work was to bring a higher level of confidence in the results of a parameterized model checker such as Cubicle. We think this is a success because we were able to automatically verify large certificates for industrial size cache coherence protocols. This progress was made possible essentially by computing intermediate lemmas to help and guide the automated theorem provers.

So far our certification framework demands that we trust three of its components:

1. Our translation of Cubicle's systems in Why3's first order logic. An immediate next step for our work would be to unify these two input specification languages.
2. The logic part of the deductive platform Why3. We don't use any advanced programming features of Why3 so this reduces our trust base. A possibility would be to use a certified version of Why3 [15].
3. The automated theorem provers. It would not be unreasonable to place our trust in *e.g.* one of the SMT solvers, but our technique makes use of redundancy by using multiple solvers. This allows to not trust any single prover.

To further this effort, an interesting approach would be to remove all quantifiers from the certificates. This is feasible because the unsat cores of Algorithm 2 can be easily refined to include useful instances. It would allow to use solvers that do not support quantifiers and reduce the burden on the ones who do.

Acknowledgment

This work was partially supported by the French ANR project ANR-12-INSE-0007 Cafein.

References

1. 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.
2. M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Wener. Verifying sat and smt in coq for a fully automated decision procedure. In *PSATTT'11: International Workshop on Proof-Search in Axiomatic Theories and Type Theories*, 2011.
3. 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.
4. 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.
5. F. Bobot, S. Conchon, É. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The alt-ergo automated theorem prover, 2008.
6. S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems - Tool Paper. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 718–724. Springer, 2012.
7. S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Invariants for finite instances and beyond. In *FMCAD*, pages 61–68. IEEE, 2013.
8. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963, pages 337–340. Springer, 2008.
9. G. Dowek, A. Felty, H. Herbelin, G. Huet, B. Werner, C. Paulin-Mohring, et al. The coq proof assistant user's guide: Version 5.6. 1991.

10. 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.
11. B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI International, 2006.
12. 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.
13. J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792, pages 125–128. Springer, Mar. 2013.
14. S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *LMCS*, 6(4), 2010.
15. P. Herms, C. Marché, and B. Monate. A certified multi-prover verification condition generator. In *VSTTE*, volume 7152, pages 2–17. Springer, 2012.
16. 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.
17. K. Korovin. iprover–an instantiation-based theorem prover for first-order logic (system description). In *Automated Reasoning*, pages 292–298. Springer, 2008.
18. J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Computer Architecture*, pages 302–313, Apr. 1994.
19. X. Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, Dec. 2009.
20. L. Matthews. personal communication.
21. K. S. Namjoshi. Certifying model checkers. In *CAV*, pages 2–13. Springer, 2001.
22. 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.
23. S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In *CADE*, pages 748–752. Springer, 1992.
24. A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, pages 82–97. Springer, 2001.
25. A. Riazanov and A. Voronkov. Vampire. In *CADE*, volume 1632, pages 292–296. Springer, 1999.
26. G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, Jan. 1981.
27. S. Schulz. System description: E 1.8. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 735–743. Springer, 2013.
28. 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.
29. A. Stump. Proof checking technology for satisfiability modulo theories. *Electronic Notes in Theoretical Computer Science*, 228:121–133, 2009.
30. C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. Spass version 3.5. In *CADE*, pages 140–145. Springer, 2009.